Employee Data Management System

Private (https://app.clickup.com/9016770133/docs/8cq1zjn-716/8cq1zjn-756)

Good [morning/afternoon], everyone.

Hi everyone, I'm [Your Name], and today my team and I are going to walk you through the Employee Data Management System we've built.

In today's data-driven workplaces, timely decisions require integrated insights from multiple sources—employee records, leave history, communication logs, and policy violations. Our solution uses AWS services, PySpark, and Kafka to build a robust system that not only processes this data efficiently but also generates intelligent reports and real-time alerts.

At a high level, the system supports:

- Daily ingestion and transformation of employee, leave, and holiday data.
- Real-time monitoring of employee messages using Kafka and Spark Structured Streaming.
- Flagging of employees who exceed leave thresholds or violate communication policies.
- Strike tracking, salary deductions, and cooldown logic—all stored and audited via PostgreSQL.

We've also implemented daily and monthly automated workflows using Apache Airflow, with each layer—bronze, silver, and gold—ensuring data traceability and quality.

With that, I'll now hand it over to [Team Member Name], who will walk you through the system components and deep-dive into specific modules like leave quota tracking, cooldown enforcement, and Kafka stream processing.

Workflow:

1. employee_data.csv — Basic Employee Master Table

Columns:

• emp_id: Unique identifier

• name: Employee name

• age: Employee's age

@ Purpose:

This is your **base employee registry**—used to track the latest version of every employee's identity.

🔁 Ingestion Strategy:

- Frequency: Daily at 00:00 UTC
- Nature: Append-only (no updates to old data)
- "Append-only (no updates to old data)" means that once a record is written to the dataset or table, it is never modified or deleted. All changes, corrections, or additions are handled by appending new rows rather than updating existing ones.
- Target Table: Append-only table with columns: emp_id , name , age , ingest_date

X ETL Notes:

- Remove duplicates by emp_id, keeping latest if needed.
- Add ingest_date = current_date() column for tracking.
- Good candidate for partitioning by ingest date in S3 or Delta Lake.

2. metalen employee_timeframe_data.csv + employee_timeframe_data_*.csv — Designation History &

Status

Columns:

- emp_id : Unique employee ID
- start_date , end_date : Employment period (can be UNIX timestamps)
- designation: Role during that period
- salary: Salary for that role

Purpose:

Tracks **employee role and compensation over time**. This is foundational for timelines, current status, and salary-based reporting.

Processing Logic:

Temporal Continuity:

- Must ensure no gaps or overlaps in dates per employee.
- Enforce: prev.end_date == curr.start_date
- For open-ended rows (missing end_date): treat as ACTIVE.

🔀 Status Derivation:

```
• If end_date IS NULL → status = 'ACTIVE'
```

Else → status = 'INACTIVE'

EMPLOYEE DATA PIPELINE

Goal

Create an **append-only table** that captures the latest snapshot of employees with fields:

emp_id

age

name

Schedule

• Run Time: Every day at 07:00 UTC

Source File: employee_data.csv (delivered daily via S3)

EMPLOYEE TIMEFRAME DATA PIPELINE

Goal

Create an **incremental table** capturing role history with fields:

emp_id

start_date

end_date

designation

salary

status → derived field (ACTIVE or INACTIVE)

Run Time: Every day at 07:00 UTC

Source Files:

Historical base: employee_timeframe_data.csv

Daily incrementals: employee_timeframe_data_*.csv

continuity means that the time periods for an employee's roles follow each other without gaps or overlaps. This ensures a complete, gap-free timeline of their employment history.

CONTINUITY: Core Concept

In the context of employee history tracking:

Continuity means that the employment time periods for an employee are connected seamlessly, without missing days (gaps) or repeated days (overlaps) between one role and the next.

This is important to build a **clean, chronological employment timeline** for each employee.

WHAT IS A GAP?

A **gap** means there are **missing dates** between the end of one employment period and the start of the next.

Example of a Gap:

emp_id	start_date	end_date	designation	salary
101	2023-01-01	2023-06-30	Analyst	50000
101	2023-07-02	2023-12-31	Sr Analyst	60000

Between 2023-06-30 and 2023-07-02, July 1st is missing \rightarrow 1-day gap.

Why Gaps Are a Problem:

- Timeline is incomplete we don't know what happened on the missing day.
- Makes it unclear whether the employee was active or left temporarily.
- Breaks systems like:
 - Salary continuity
 - Leave entitlement

Designation progression

WHAT IS AN OVERLAP?

An **overlap** means that two employment periods **share one or more dates** — creating ambiguity.

Example of an Overlap:

emp_id	start_date	end_date	designation	salary
102	2023-01-01	2023-06-30	Analyst	50000
102	2023-06-25	2023-12-31	Sr Analyst	60000

The second period starts on 2023-06-25, before the first one ends on 2023-06-30 \rightarrow 6-day overlap.

Why Overlaps Are a Problem:

- Two conflicting designations for the same date(s)
- Inconsistent salary if both rows are considered valid
- Confuses logic that checks who held what position on a specific day

WHAT IS A CONTINUOUS TIMELINE?

A **continuous** timeline means each period ends exactly when the next one begins — **no gaps**, **no overlaps**.

Ideal Continuous Example:

emp_id	start_date	end_date	designation	salary
103	2023-01-01	2023-06-30	Analyst	50000
103	2023-06-30	2023-12-31	Sr Analyst	60000
103	2023-12-31	NULL	Manager	80000

Every end_date matches the next $start_date$ \rightarrow clean, unbroken chain.

Imports and Initialization

import sys
import boto3

from datetime import datetime

```
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from pyspark.sql.functions import col
```

- Imports necessary modules:
 - sys : for exiting the script
 - boto3 : AWS SDK for interacting with S3
 - datetime: to get the current date
 - GlueContext / SparkContext : for initializing AWS Glue and Spark
 - col : Spark SQL function for column operations

Spark & Glue Setup

```
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
```

Initializes Spark and Glue execution contexts

SparkContext is the main entry point to any Spark functionality.

It connects your Python program to a **Spark cluster** (your processing engine). When you write Spark code, it needs a **SparkContext** to start doing anything.

Think of it as:

The engine that lets you run Spark code and manage your jobs.

What is **GlueContext**?

GlueContext is a wrapper built on top of SparkContext. It's specific to AWS Glue.

It extends Spark's features by adding things Glue-specific:

- Working with AWS Glue Data Catalog
- Reading/writing to S3 in a structured way
- Connecting to Glue tables, crawlers, and jobs
- Generating dynamic frames (a Glue data structure)

Think of it as:

A special Spark environment tailored for AWS Glue.

Registers PostgreSQL JDBC driver to Spark

· Data Cati

```
Date Setup
```

```
today = datetime.utcnow().strftime("%Y-%m-%d")
```

• Captures today's UTC date in the format YYYY-MM-DD for timestamped output paths

S3 & JDBC Configurations

```
bucket = "poc-bootcamp-capstone-group1"
raw_prefix = "poc-bootcamp-group1-bronze/emp_data_qus1/raw/"
processed_output_path = f"s3://{bucket}/poc-bootcamp-group1-
bronze/emp_data_qus1/processed/data_processed_{today}.csv"
```

- bucket : your S3 bucket name
- raw_prefix : S3 folder where raw CSVs are dropped
- processed_output_path : final merged + cleaned CSV for today

```
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"

db_properties = {
    "user": "test_user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}

output_table = "employee_db"
```

- Config for writing to PostgreSQL: JDBC URL, credentials, and target table
- "Connect to the test_topic database hosted at IP 3.221.182.234 on port 5432 using the PostgreSQL JDBC protocol."

S3 Client + File Listing

```
s3 = boto3.client('s3')
raw_files = s3.list_objects_v2(Bucket=bucket, Prefix=raw_prefix).get("Contents", [])
csv_files = [obj["Key"] for obj in raw_files if obj["Key"].endswith(".csv")]
```

- Lists objects under the raw prefix
- Let's explain this block line by line in a clear and structured way:

```
s3 = boto3.client('s3')
```

- What it does:
- Creates a low-level client object for AWS S3 using the boto3 library.

 This object allows you to programmatically interact with S3, including listing, reading, uploading, and deleting files.

```
raw_files = s3.list_objects_v2(Bucket=bucket, Prefix=raw_prefix).get("Contents", [])
```

- What it does:
- Calls List_objects_v2, which lists **all objects (files/folders)** under a specific path in your S3 bucket.
- You provide:
 - Bucket=bucket → the name of the S3 bucket (e.g., "poc-bootcamp-capstone-group1")
 - Prefix=raw_prefix → the folder path inside the bucket (e.g., "emp_data/raw/")
- .get("Contents", []) safely extracts the list of files.
 - If there are no files, it returns an **empty list** instead of None.
- Output:
- raw_files will be a **list of dictionaries**, each describing an S3 object, like:

```
[
    {"Key": "emp_data/raw/file1.csv", "LastModified": ..., "Size": ...},
    {"Key": "emp_data/raw/file2.csv", ...},
]
csv_files = [obj["Key"] for obj in raw_files if obj["Key"].endswith(".csv")]
```

- What it does:
- This is a **list comprehension**.
- It filters the list of files to **only include** .csv **files**.
- It extracts the "Key" field, which is the S3 path to each file, like:

```
[
  "emp_data/raw/file1.csv",
  "emp_data/raw/file2.csv"
]
```

- s3 = boto3.client('s3')
- Initializes the S3 client so your script can access files
- s3.list_objects_v2(...)
- Fetches all files under a given path inside the bucket
- .get("Contents", [])

Prevents errors by returning an empty list if no files found

```
csv files = [...]
```

- Filters and lists only csv files for further processing
- Filters only .csv files

```
if not csv_files:
    print(" ↑ No CSV files found in raw folder.")
    sys.exit(0)
```

Exits early if no input files are found

Step 2: Merge Raw CSVs

```
raw_paths = [f"s3://{bucket}/{key}" for key in csv_files]
df_raw = spark.read.option("header", "true").csv(raw_paths)
```

- Builds full S3 paths
- Reads all raw CSVs into a single Spark DataFrame with headers

```
df_raw.coalesce(1).write.mode("overwrite").option("header", "true").csv(processed_output_path)
print(f"Merged raw CSV backed up to: {processed_output_path}")
```

- Writes the merged DataFrame as 1 output file to a processed folder
- Acts as a backup + intermediate clean zone
- By default, when Spark writes data (e.g., to CSV or Parquet), it writes 1 file per partition.
- For example:
- If the DataFrame has 200 partitions, you'll get 200 part-xxxxx files in S3, thats why coalesce for 1 partition only

Step 2: Merging Raw CSVs from S3

Code Block

```
raw_paths = [f"s3://{bucket}/{key}" for key in csv_files]
df_raw = spark.read.option("header", "true").csv(raw_paths)
```

Line 1: Build S3 paths to raw CSV files

```
raw_paths = [f"s3://{bucket}/{key}" for key in csv_files]
```

What's happening?

- You're constructing full S3 URLs for each CSV file.
- csv_files contains relative paths like:

```
emp_data/raw/file1.csv
emp_data/raw/file2.csv
```

bucket is something like:

poc-bootcamp-capstone-group1

Result:

Each item in raw paths becomes:

```
s3://poc-bootcamp-capstone-group1/emp_data/raw/file1.csv
s3://poc-bootcamp-capstone-group1/emp_data/raw/file2.csv
```

These are valid, absolute S3 paths Spark can read from.

Line 2: Read all raw CSV files into one DataFrame

```
df raw = spark.read.option("header", "true").csv(raw paths)
```

What's happening?

- You instruct Spark to read all the CSV files at once, combining them into a single DataFrame.
- .option("header", "true") tells Spark:
 - The first row in each CSV file contains column names, not data.
- .csv(raw_paths) reads all files in raw_paths as one dataset.

Example:

If file1.csv:

```
emp_id, name, age
101, John, 30
102, Alice, 27
```

and file2.csv:

```
emp_id, name, age
103, Bob, 32
```

Then df raw will look like:

emp_id	name	age
101	John	30

102	Alice	27
103	Bob	32

% Key Points:

- Spark merges the schema automatically based on headers.
- This works **only if all CSV files have the same schema** (which they should in production).

🕺 Now Saving the Merged Data

Code Block

```
df_raw.coalesce(1).write.mode("overwrite").option("header", "true").csv(processed_output_path)
print(f"Merged raw CSV backed up to: {processed output path}")
```

Line 1: Write the merged DataFrame to a backup location

df_raw.coalesce(1).write.mode("overwrite").option("header", "true").csv(processed_output_path)

🧠 What's happening?

- coalesce(1):
 - Reduces all Spark output partitions into a single partition, so the output is a single
 CSV file.
 - Spark typically writes 1 file per partition without this, you'd get many small files.
- . write.mode("overwrite") :
 - If a file already exists at the processed_output_path, it will be deleted and replaced.
 - This ensures today's run always creates a fresh backup.
- .option("header", "true"):
 - Ensures the output CSV includes column names in the first row.
- .csv(processed_output_path):
 - Saves the file to S3 in the "processed" folder with today's date:

s3://{bucket}/poc-bootcamp-group1-bronze/emp data qus1/processed/data processed YYYY-MM-DD.csv/

Why it saves to a *folder* not a *file*:

• Spark always writes CSVs as a **folder containing part files**, even with coalesce(1).

So you'll see:

�� Line 2: Log the output location

```
print(f"Merged raw CSV backed up to: {processed_output_path}")
```

• Logs the full S3 location of the processed/merged CSV file for traceability.

Delete Raw Files

```
for key in csv_files:
    s3.delete_object(Bucket=bucket, Key=key)
    print(f" Deleted raw file: {key}")
```

• Deletes raw input files from S3 after backup

Step 3: Load Merged CSV

```
df_processed = spark.read.option("header", "true").csv(processed_output_path)
```

Reads the processed CSV (written above) into a new DataFrame for cleaning

Step 4: Clean + Deduplicate

```
df_cleaned = df_processed.select(
    col("emp_id").cast("string"),
    col("age").cast("int"),
    col("name").cast("string")
).dropna().filter(col("age") > 0).dropDuplicates(["emp_id", "age", "name"])
```

- Casts columns to correct types
- Removes:
 - Rows with nulls
 - Rows where age ≤ 0
 - Duplicates based on emp_id , age , and name

```
.dropDuplicates(["emp_id", "age", "name"])
```

This is a **Spark transformation** that removes rows that have the **exact same values** in the specified columns: <code>emp_id</code>, <code>age</code>, and <code>name</code>.

What does "duplicates based on emp_id, age, and name" mean?

It means:

If multiple rows in the DataFrame have the same combination of values for <code>emp_id</code>, <code>age</code>, and <code>name</code>, only one is kept. All others are dropped.

Example Before Deduplication

emp_id	name	age
101	Alice	30
102	Bob	25
101	Alice	30
103	Charlie	28
102	Bob	25

After .dropDuplicates(["emp_id", "age", "name"])

emp_id	name	age
101	Alice	30
102	Bob	25
103	Charlie	28

Only **one copy** of each (emp_id, age, name) trio remains.

tep 5: Write to PostgreSQL

```
if not df_cleaned.rdd.isEmpty():
    df_cleaned.write.mode("append").jdbc(url=jdbc_url, table=output_table, properties=db_properties)
    print(f"Final cleaned employee data written to PostgreSQL table: {output_table}")
else:
    print("No valid rows to write after cleaning.")
```

- If valid rows exist, write to PostgreSQL table in **append** mode
- If not, skip writing and log that no data was available

- What is .mode("append")?
- append is one of the write modes supported by Spark's .write API.
- It tells Spark:

"Insert these new rows into the existing table, without deleting or replacing what's already there."Final Message

print("Glue job completed successfully.")

Signals successful end of the job

Summary

Phase	Purpose
File listing	Gather input from raw S3
Merge + backup	Consolidate into 1 processed CSV
Cleanup	Remove raw files
Clean/Transform	Remove nulls, bad ages, and duplicates
Load to DB	Write cleaned data to PostgreSQL

Here is a **complete summarized workflow** of your **employee_data Glue job**, step-by-step:



🔄 Workflow Overview

This Glue job ingests daily raw employee CSV files from S3, cleans and deduplicates them, stores a backup, and inserts cleaned records into a PostgreSQL database.



Step-by-Step Summary

🔽 1. Initialize Spark, Glue, and JDBC

- Sets up Spark and Glue contexts.
- Registers the PostgreSQL JDBC driver.
- Prepares connection settings and today's date for dynamic file naming.

2. List Raw CSV Files in S3

- Uses boto3 to list all .csv files in the S3 raw input folder:
- s3://<bucket>/<prefix>/raw/
- If no files are found, the job exits early.

🔽 3. Merge Raw CSVs into a Single DataFrame

- Reads all raw CSVs into Spark using .csv([...paths]).
- Assumes files have the same header (emp_id , name , age).
- Result: one unified DataFrame (df_raw) for today's ingestion.

🔽 4. Backup the Merged Data

• Saves the merged data to a processed S3 path with today's date:

```
s3://<bucket>/.../processed/data_processed_YYYY-MM-DD.csv/
```

• Uses .coalesce(1) to output a single CSV file.

🔽 5. Delete Raw Files After Backup

- Deletes all files listed in Step 2 from the raw folder.
- This cleans up the landing zone and prevents reprocessing tomorrow.

6. Reload the Processed Backup

- Reads the newly written single CSV from the processed path into df_processed.
- Serves as the cleaned source for transformation.

7. Clean and Deduplicate Data

• Selects and casts columns: emp_id (string), age (int), name (string)

- Removes:
 - Rows with nulls
 - Rows with non-positive ages
 - Duplicate rows based on (emp_id, age, name)

8. Write to PostgreSQL

- If cleaned data exists:
 - Appends it into the employee_db table in the PostgreSQL database.
- If not:
 - Logs that there's nothing to write.

9. Job Completion

Logs success message at the end.

•

```
import sys
import boto3
from datetime import datetime
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from pyspark.sql.functions import col
# Initialize Spark and Glue
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark session
spark. jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
# Date setup
today = datetime.utcnow().strftime("%Y-%m-%d")
# S3 Paths
bucket = "poc-bootcamp-capstone-group1"
raw_prefix = "poc-bootcamp-group1-bronze/emp_data_qus1/raw/"
processed output path = f"s3://{bucket}/poc-bootcamp-group1-
bronze/emp data qus1/processed/data processed {today}.csv"
```

```
jdbc url = "jdbc:postgresql://3.221.182.234:5432/test topic"
db properties = {
    "user": "test user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}
output_table = "employee_db"
# S3 Client
s3 = boto3.client('s3')
# Step 1: List raw CSV files
raw_files = s3.list_objects_v2(Bucket=bucket, Prefix=raw_prefix).get("Contents", [])
csv files = [obj["Key"] for obj in raw files if obj["Key"].endswith(".csv")]
if not csv files:
    print("No CSV files found in raw folder.")
    sys.exit(0)
print(f" Found {len(csv files)} CSV files.")
# Step 2: Merge CSVs
raw paths = [f"s3://{bucket}/{key}" for key in csv files]
df_raw = spark.read.option("header", "true").csv(raw_paths)
# Save processed backup to processed folder
df raw.coalesce(1).write.mode("overwrite").option("header", "true").csv(processed output path)
print(f" Merged raw CSV backed up to: {processed_output_path}")
# Delete raw files after backup
for key in csv files:
    s3.delete object(Bucket=bucket, Key=key)
    print(f" Deleted raw file: {key}")
# Step 3: Load from processed CSV
df_processed = spark.read.option("header", "true").csv(processed_output_path)
# Step 4: Clean and transform
df cleaned = df processed.select(
    col("emp id").cast("string"),
    col("age").cast("int"),
    col("name").cast("string")
).dropna().filter(col("age") > 0).dropDuplicates(["emp_id", "age", "name"])
# Step 5: Write to PostgreSQL
if not df cleaned.rdd.isEmpty():
    df cleaned.write.mode("append").jdbc(url=jdbc url, table=output table, properties=db properties)
```

JDBC Config

```
print(f"Final cleaned employee data written to PostgreSQL table: {output_table}")
else:
    print("No valid rows to write after cleaning.")
print(" Glue job completed successfully.")
```

Private (https://app.clickup.com/9016770133/docs/8cq1zjn-716/8cq1zjn-776)

Private (https://app.clickup.com/9016770133/docs/8cq1zjn-716/8cq1zjn-716/8cq1zjn-836) Private (https://app.clickup.com/9016770133/docs/8cq1zjn-716/8cq1zjn-816)

Question/Answer

is the **Amazon Web Services (AWS) SDK for Python**. It allows Python developers to interact programmatically with AWS services such as S3 (Simple Storage Service), EC2 (Elastic Compute Cloud), DynamoDB, Lambda, Glue, and many more.

What is it used for?

boto3 is typically used to:

- Upload/download files to/from S3
- Manage EC2 instances (start, stop, terminate)
- Query and manipulate DynamoDB tables
- Invoke Lambda functions
- Start, stop, or monitor Glue jobs
- Manage IAM users and permissions
- Automate AWS infrastructure as part of data pipelines or applications

You're at a **laundromat**.

- client is like using the **raw washing machine panel** yourself you press every button manually (e.g., water level, soap type, spin speed).
- resource is like asking the **attendant**: "Please wash my clothes." They know the buttons to press for you.

Both will **get the job done**, but one gives you **more control**, the other gives you **more convenience**.

boto3, client - The Raw Remote

```
python
CopyEdit
import boto3

s3 = boto3.client('s3')

s3.upload_file('local.txt', 'my-bucket', 'data/local.txt')
```

You are saying:

"Hey AWS S3, here's the exact command: upload this file to this bucket and this path."

```
python
CopyEdit
import boto3
s3 = boto3.resource('s3')
bucket = s3.Bucket('my-bucket')
bucket.upload_file('local.txt', 'data/local.txt')
```

You are saying:

"Hey S3 bucket, please upload this file."

It's a bit more **object-oriented** — the bucket is like a Python object.

boto3.client('s3')

This gives you **low-level access** to AWS.

Everything you do maps **directly** to AWS service APIs.

It returns responses as dictionaries.

. boto3.resource('s3')

This gives you a **high-level**, **object-oriented** interface.

Here, AWS entities like **Buckets** and **Objects (files)** are treated like Python **objects**, with their own methods and attributes.

Great, let's break this down clearly:



What is hadoopConfiguration()?

In the line:

spark. jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")

- spark._jsc. hadoopConfiguration() accesses the underlying Hadoop configuration object used by Spark.
- It comes from Spark's integration with the Hadoop FileSystem API (which Spark uses to read/write files on S3, HDFS, etc.).
- It lets you **set runtime configurations** such as access credentials, file formats, and paths to resources like JARs.
- Imagine Spark is a **delivery van** that moves data.
- The van (Spark) knows how to drive around (compute).
- But if you want it to go on certain roads (like S3), or carry certain packages (like connect to PostgreSQL), you need to give it **maps and instructions**.

That's what this hadoopConfiguration().set(...) call is doing:

It's giving the van new **rules** and **tools** to do the job.

What is happening here?

```
spark._jsc.hadoopConfiguration()
```

- This accesses Spark's internal configuration system that it uses to interact with file systems, like:
- S3 (Amazon storage)
- HDFS (Hadoop storage)
- Local file systems
- Spark itself is built on top of **Hadoop APIs**, even if you're not using Hadoop directly.
- Why does Spark use this?
- Because Spark needs to read and write files, and Hadoop provides a common API for accessing file systems (even S3 is treated like a file system in Spark).
- Apache Spark is a general-purpose data processing engine, but it reuses a lot of tools
 originally built for Hadoop. One of the most important is the Hadoop FileSystem API.

Why use it to set "spark.jars"?

This sets the path to a **JDBC driver JAR** so that Spark can:

- Connect to PostgreSQL using .jdbc(...)
- Load the correct driver: org.postgresql.Driver

Without this, Spark would raise an error like:

ClassNotFoundException: org.postgresql.Driver

Where did this file come from?

/home/ubuntu/postgresql-42.7.2.jar

This JAR file is the **PostgreSQL JDBC Driver**, version 42.7.2.

You can download it from:

jdbc.postgresql.org

https://jdbc.postgresql.org/download.html

Install instructions:

1. Download the JAR:

wget https://jdbc.postgresql.org/download/postgresql-42.7.2.jar

1. Move it to a known path:

```
mv postgresql-42.7.2.jar /home/ubuntu/
```

Then refer to it using:

```
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
```

Alternatively: if you're running on EMR or Glue with PySpark, you can pass the JAR via -- jars in the job submission or Glue UI.

Purpose of bucket and raw_prefix

Variable	Purpose	Example Value
bucket	Holds just the bucket name	"poc-bootcamp-capstone-group1"
raw_prefix	Holds the folder path inside bucket	"poc-bootcamp-group1-bronze/emp_data_qus1/raw/"

Together, they form the full S3 path:

```
python
CopyEdit
s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp_data_qus1/raw/
```

Why split them?

1. Clarity and Modularity

- You can reuse the bucket name across many datasets (raw, processed, archived).
- Each dataset or phase can have its own prefix (e.g., raw/, processed/, archive/).

Employee TimeFrame

```
import sys
import shutil
from io import StringIO
import os
import boto3
import re
import psycopg2
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from pyspark.sql import SparkSession, Window
from pyspark.sql.functions import col, from_unixtime, to_date, when, row_number, lit, min as min_, lead
from pyspark.sql.types import StructType, StructField, StringType, DateType, DoubleType
args = getResolvedOptions(sys.argv, ['JOB NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark session
spark. jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
# Register PostgreSQL JDBC driver JAR from S3 if not already passed via job parameter
# NOTE: The correct way in Glue is to use --extra-jars while submitting the job
# Example: --extra-jars s3://your-bucket/jars/postgresql-42.7.2.jar
spark.sparkContext.setLogLevel("ERROR")
POSTGRES URL = "idbc:postgresql://3.221.182.234:5432/test topic"
POSTGRES_PROPERTIES = {"user": "test_user", "password": "test_user", "driver": "org.postgresql.Driver"}
PSYCOPG2 CONN = dict(dbname="test topic", user="test user", password="test user", host="3.221.182.234",
port="5432")
STAGING TABLE = "staged employees"
FINAL TABLE = "employee db salary"
BACKUP TABLE = "employee db salary backup"
TEMP LOCAL PATH = "/tmp/final pg upload"
# S3 Paths for raw and archive
RAW PATH = "s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp timeframe data qus2/raw/"
PROCESSED RAW PATH = "s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-
bronze/emp timeframe data qus2/processed raw/"
# Manual Schema
schema = StructType([
    StructField("emp id", StringType(), True),
    StructField("designation", StringType(), True),
```

```
StructField("start_date", DateType(), True),
    StructField("end date", DateType(), True),
    StructField("salary", DoubleType(), True),
    StructField("status", StringType(), True)
])
# Read Raw File
df = spark.read.option("header", True).csv(RAW PATH)
df = df.withColumn("start_date", to_date(from_unixtime(col("start_date").cast("long"))))
df = df.withColumn("end_date", to_date(from_unixtime(col("end_date").cast("long"))))
df = df.withColumn("salary", col("salary").cast("double"))
# Deduplication
w1 = Window.partitionBy("emp_id", "start_date", "end_date").orderBy(col("salary").desc())
df = df.withColumn("rn1", row_number().over(w1)).filter(col("rn1") == 1).drop("rn1")
w2 = Window.partitionBy("emp id").orderBy(col("salary").desc())
df_with_nulls = df.filter(col("end_date").isNull()).withColumn("rn2",
row number().over(w2)).filter(col("rn2") == 1).drop("rn2")
df non nulls = df.filter(col("end date").isNotNull())
df = df_with_nulls.unionByName(df_non_nulls)
df = df.withColumn("status", when(col("end_date").isNull(), "ACTIVE").otherwise("INACTIVE"))
# Write to staging
df.write.mode("overwrite").jdbc(POSTGRES URL, STAGING TABLE, properties=POSTGRES PROPERTIES)
# Load existing data
tables df = spark.read.jdbc(POSTGRES URL, "information schema.tables", properties=POSTGRES PROPERTIES)
exists = tables df.filter(col("table name") == FINAL TABLE).count() > 0
if not exists:
    df.write.mode("overwrite").jdbc(POSTGRES_URL, FINAL_TABLE, properties=POSTGRES PROPERTIES)
else:
    silver_df = spark.read.jdbc(POSTGRES_URL, FINAL_TABLE, properties=POSTGRES PROPERTIES)
    processed df = spark.read.jdbc(POSTGRES URL, STAGING TABLE, properties=POSTGRES PROPERTIES)
    active df = silver df.filter(col("status") == "ACTIVE")
    inactive df = silver df.filter(col("status") == "INACTIVE")
    continuity_dates = processed_df.groupBy("emp_id").agg(min_("start_date").alias("new_start_date"))
    updated_old = active_df.alias("old") ¥
        .join(continuity_dates.alias("new"), "emp_id") \(\xi$
        .filter(col("new.new start date") >= col("old.start date")) ¥
        .withColumn("end date", col("new.new start date")) ¥
        .withColumn("status", lit("INACTIVE")) ¥
        .select("old.emp id", "old.designation", "old.start date", "end date", "old.salary", "status")
    untouched active = active df.join(continuity dates.select("emp id"), "emp id", "left anti")
    new_emps = processed_df.join(silver_df.select("emp_id").distinct(), "emp_id", "left_anti")
```

```
columns = ["emp_id", "designation", "start_date", "end_date", "salary", "status"]
    final df = inactive df.select(columns) ¥
        .union(updated_old.select(columns)) ¥
        .union(untouched active.select(columns)) ¥
        .union(processed df.select(columns)) ¥
        .union(new emps.select(columns)) ¥
        .dropDuplicates(columns)
    w_dedupe = Window.partitionBy("emp_id", "start_date").orderBy(col("salary").desc())
    final_df = final_df.withColumn("row_num", row_number().over(w_dedupe)).filter(col("row_num") ==
1).drop("row num")
    w_desc = Window.partitionBy("emp_id").orderBy(col("start_date").desc())
    final_df = final_df.withColumn("is_latest", row_number().over(w_desc) == 1)
    w_ordered = Window.partitionBy("emp_id").orderBy("start_date")
    final_df = final_df.withColumn("next_start_date", lead("start_date").over(w_ordered))
    final df = final df.withColumn("end date", when(col("is latest"),
None).otherwise(col("next_start_date"))) \(\xi$
                       .drop("next_start_date", "is_latest")
    final df = final df.withColumn("status", when(col("end date").isNull(),
"ACTIVE").otherwise("INACTIVE"))
    # Backup table
    conn = psycopg2.connect(**PSYCOPG2_CONN)
    cur = conn.cursor()
    cur.execute("DROP TABLE IF EXISTS employee db salary backup;")
    cur.execute("CREATE TABLE employee_db_salary_backup AS TABLE employee_db_salary;")
    # Delete matching rows using temp table
    keys = final df.select("emp id", "start date").distinct().collect()
    key_tuples = [(row["emp_id"], row["start_date"].strftime("%Y-%m-%d")) for row in keys]
    if key tuples:
        cur.execute("DROP TABLE IF EXISTS delete_keys;")
        cur.execute("CREATE TEMP TABLE delete keys (emp id TEXT, start date DATE);")
        key_buf = StringIO("".join([f"{e}, {d}*n" for e, d in key_tuples]))
        key buf.seek(0)
        cur.copy_expert("COPY delete_keys FROM STDIN WITH CSV DELIMITER',' NULL'';", key_buf)
        cur.execute("""
            DELETE FROM employee_db_salary
            USING delete keys
            WHERE employee db salary.emp id = delete keys.emp id
              AND employee db salary.start date = delete keys.start date;
        """)
    # Copy data into PostgreSQL
    final_df.coalesce(1).write.option("header", False).option("delimiter",
",").mode("overwrite").save(TEMP_LOCAL_PATH)
```

```
# Convert Spark DataFrame to Pandas and write directly to buffer
    pandas df = final df.toPandas()
    buffer = StringIO()
    pandas df.to csv(buffer, index=False, header=False)
    buffer.seek(0)
    cur.copy_expert("""
        COPY employee_db_salary (emp_id, designation, start_date, end_date, salary, status)
        FROM STDIN WITH CSV DELIMITER ',' NULL '';
    """, buffer)
    conn.commit()
    cur.close()
    conn.close()
    print("Final table updated using psycopg2 (delete + insert).")
# Archive raw file
s3 = boto3.client("s3")
raw_bucket = RAW_PATH.split("/")[2]
raw prefix = "/".join(RAW PATH.split("/")[3:])
archive prefix = raw prefix.replace("raw/", "processed raw/")
objs = s3.list objects v2(Bucket=raw bucket, Prefix=raw prefix).get("Contents", [])
for obj in objs:
    key = obj["Key"]
    if not key endswith ('/'):
        s3.copy object(Bucket=raw bucket, CopySource={'Bucket': raw bucket, 'Key': key},
Key=archive_prefix + key.split('/')[-1])
        s3.delete object(Bucket=raw bucket, Key=key)
        print(f"Archived {key}")
```

Private (https://app.clickup.com/9016770133/docs/8cq1zjn-716/8cq1zjn-796)

Here is a **complete**, **clear**, **and structured explanation** of the **workflow** behind your employee_timeframe_data Glue job — covering what it does, why each step exists, and how the pieces fit together.



To maintain a **clean, continuous, and up-to-date table** of each employee's designation history (start & end dates, salary, status), with:

- Deduplication
- Status marking (ACTIVE / INACTIVE)
- . Continuity enforcement
- Safe and incremental updates to PostgreSQL

S OVERALL WORKFLOW: STEP-BY-STEP

1. Read Raw Data from S3

- Source: employee_timeframe_data.csv files in a raw S3 folder.
- These contain daily snapshots or incremental records.
- Data includes: emp_id , start_date , end_date , designation , salary .

2. Normalize and Clean the Data

- Convert UNIX timestamps to proper date format (yyyy-MM-dd).
- Deduplicate records where (emp_id, start_date, end_date) are the same.
 - Keep the highest salary version for those duplicates.
- Classify rows as:
 - ACTIVE → if end_date IS NULL
 - INACTIVE → otherwise

🔽 3. Write to PostgreSQL Staging Table

- Overwrite a temporary staging table in PostgreSQL:
 - employee_db_salary_stg
- This acts as a buffer for validating and comparing before updating the real target.

🔽 4. Compare with Final Table

- Read both:
 - Final table (employee_db_salary)
 - New staging table
- Detect:
 - New employees
 - Employees whose current roles have changed (active status updated)
 - Updated salary or designation entries

☑ 5. Build Updated Final Table

- Break down current data:
 - active_df: Employees marked ACTIVE in final table
 - new_emps: Employees only in the staging table
 - untouched_active: Employees who remain unchanged
- For those being updated:
 - Mark their old record INACTIVE
 - Set a proper end_date based on the next start_date (enforces continuity)

6. Enforce Continuity

- Use lead(start_date) to compute the next time period per employee.
- Adjust end_date of each record so:

```
end_date = next_record.start_date
```

• Ensures there are **no gaps** or **overlaps** in employment timelines.

GOAL

When an employee has multiple job records over time, you need to:

- 1. Mark the older record as **INACTIVE**
- 2. Set its end_date to the start_date of the next record → This enforces temporal continuity

WHY?

Because:

- Raw data may have an open-ended record (no end_date)
- But the employee has a new role starting at a later date
- So the **old record must end where the new one begins** (no overlap or gap)

EXAMPLE

Raw Data (staging)

emp_id	start_date	end_date	designation	salary
101	2023-01-01	NULL	Analyst	50000

|--|

Problem: both records are open-ended \rightarrow conflict and overlap.

What We Want

emp_id	start_date	end_date	designation	salary	status
101	2023-01-01	2023-06-01	Analyst	50000	INACTIVE
101	2023-06-01	NULL	Sr Analyst	60000	ACTIVE

The first record now ends on the start of the next, and is marked as INACTIVE.

Step 1: Sort records per employee by start_date

```
from pyspark.sql.window import Window
from pyspark.sql.functions import lead, when, col
window_spec = Window.partitionBy("emp_id").orderBy("start_date")
```

Step 2: Use lead() to get the next start date

```
df = df.withColumn("next start", lead("start date").over(window spec))
```

This gives us:

emp_id	start_date	designation	next_start
101	2023-01-01	Analyst	2023-06-01
101	2023-06-01	Sr Analyst	NULL

Step 3: Set end_date using next_start

```
df = df.withColumn("end_date", when(col("next_start").isNotNull(), col("next_start")).otherwise(None))
```

Step 4: Set status

```
df = df.withColumn("status", when(col("next_start").isNull(), "ACTIVE").otherwise("INACTIVE"))
```

Final Output

emp_id	start_date	end_date	designation	status
101	2023-01-01	2023-06-01	Analyst	INACTIVE
101	2023-06-01	NULL	Sr Analyst	ACTIVE

Result

This enforces:

- No overlap
- No gap
- One current ACTIVE record
- Historical records properly ended

STEP 7: BACKUP EXISTING FINAL TABLE

What it does:

It **copies the current state** of the final employee salary table (<code>employee_db_salary</code>) into a backup table (<code>employee_db_salary_bkp</code>) **before making any changes**.



To protect yourself.

If something goes wrong during the update process — e.g., bad records, incomplete insert, wrong deletions — you can **restore from the backup**.

Example:

Before update:

emp_id	start_date	designation	salary
101	2023-01-01	Analyst	50000

After creating the backup, the same row exists in:

- employee_db_salary 🗸
- employee_db_salary_bkp (safety copy)



STEP 8: DELETE CONFLICTING ROWS FROM FINAL TABLE

What it does:

Deletes rows in employee_db_salary that are **about to be reinserted/updated**, to prevent duplicates or conflicts.

® Based on which key?

It creates a temp table (delete_keys) with just:

```
(emp_id, start_date)
```

Then it runs:

```
DELETE FROM employee_db_salary
USING delete_keys
WHERE employee_db_salary.emp_id = delete_keys.emp_id
   AND employee_db_salary.start_date = delete_keys.start_date;
```

Why?

You're doing a **refresh of rows** for specific employees and time periods — to ensure no conflict on re-insertion, you **delete them first**.

Example:

If your new batch contains:

emp_id	start_date
101	2023-01-01

You remove that same row (if it exists) from employee_db_salary before inserting a new version of it.

STEP 9: BULK INSERT NEW DATA (COPY)

What it does:

It loads the cleaned, deduplicated, and status-marked final DataFrame into PostgreSQL using:

```
cur.copy_expert("COPY employee_db_salary FROM STDIN ...", buffer)
```

Why copy_expert()?

Because:

- It is **much faster** than .jdbc() writes (can load millions of rows efficiently)
- It's the same method that tools like psql use under the hood

Example:

The cleaned final rows like:

emp_id	start_date	end_date	designation	salary	status
101	2023-01-01	2023-06-01	Analyst	50000	INACTIVE
101	2023-06-01	NULL	Sr Analyst	60000	ACTIVE

Are written directly into PostgreSQL in bulk.



STEP 10: ARCHIVE RAW FILES

What it does:

Moves raw S3 files from:

s3://.../raw/

to:

s3://.../processed_raw/



Why?

- Keeps the raw folder clean for tomorrow's new data
- Prevents reprocessing the same files again
- Creates a traceable history of what was already processed

Here's a **line-by-line breakdown** of your **employee_timeframe_data Glue job**, explained in an understandable and structured way:

1. Imports & Setup

import sys, shutil, os, boto3, re, psycopg2
from io import StringIO

Standard libraries for system control, file operations, AWS access (boto3), SQL interaction (psycopg2), and streaming strings.

```
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from pyspark.sql import SparkSession, Window
from pyspark.sql.functions import ...
from pyspark.sql.types import ...
```

Initializes Spark, Glue, and imports key functions for transformations, windows, and schema definitions.

💢 2. Glue + Spark Context Setup

```
args = getResolvedOptions(sys.argv, ['JOB NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark session
spark. jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
spark.sparkContext.setLogLevel("ERROR")
```

Initializes Glue and Spark context, loads PostgreSQL JDBC JAR for .jdbc() operations.

🔐 3. Database Config

```
POSTGRES URL = ...
POSTGRES PROPERTIES = ...
PSYCOPG2 CONN = ...
```

Connection settings for Spark JDBC and psycopg2 (used for advanced PostgreSQL operations like COPY, DELETE).

4. Paths and Schema

```
RAW PATH = "s3://.../raw/"
PROCESSED RAW PATH = "s3://.../processed raw/"
schema = StructType([...])
```

Specifies input S3 path and schema for the timeframe data (with manually defined columns).



蝽 5. Read and Normalize Raw Data

```
df = spark.read.option("header", True).csv(RAW_PATH)
df = df.withColumn("start date", ...)
```

Reads raw CSVs from S3, converts start_date and end_date from UNIX to yyyy-MM-dd, and casts salary to double.

6. Deduplicate and Status Derivation

```
w1 = Window.partitionBy("emp id", "start date", "end date")...
```

 Deduplicates records based on (emp_id, start_date, end_date) using the highest salary as tiebreaker.

```
df_with_nulls = ... ; df_non_nulls = ...
df = df_with_nulls.unionByName(df_non_nulls)
```

- Ensures that only one "ACTIVE" record per employee (null end_date) is kept.
- Adds a status column: 'ACTIVE' if end_date IS NULL, else 'INACTIVE'.

7. Write to Staging Table

```
df.write.mode("overwrite").jdbc(POSTGRES_URL, STAGING_TABLE, ...)
```

Writes the cleaned + deduplicated data to a staging table in PostgreSQL.

2 8. Check for Final Table Existence

```
tables_df = spark.read.jdbc(...); exists = ...
```

- If the final table (employee_db_salary) doesn't exist, create it.
- If it **does exist**, proceed to merging logic.

9. Merging Logic (if table exists)

a. Load Existing Final + Staged Data

```
silver_df = ... ; processed_df = ...
```

b. Determine What Needs to Be Updated

- Identify:
 - active_df : currently active records
 - inactive_df : closed records
 - continuity_dates : new start_date s for employees in staged data

c. Create Updated Timeline

```
updated_old = ...  # Truncate active records
untouched_active = ...  # Employees who aren't updated
new_emps = ...  # New employees in current batch
```

d. Union & Deduplicate Final DataFrame

```
final_df = union(...) # Merge all categories
final_df = dropDuplicates(...)
```

e. Ensure Continuity

```
# Fix overlapping or missing end_dates
final_df = withColumn("is_latest", ...)
final_df = withColumn("end_date", when(...))
final_df = withColumn("status", when(...))
```

Û

10. Backup Current Final Table

```
conn = psycopg2.connect(...); cur = conn.cursor()
cur.execute("DROP TABLE IF EXISTS ...")
cur.execute("CREATE TABLE ... AS TABLE ...")
```

Creates a backup of the current final table before overwriting.

X 11. Delete Matching Records (by emp_id + start_date)

```
key_tuples = [(emp_id, start_date), ...]
# Create temp table `delete_keys` and bulk load keys into it
cur.copy_expert("COPY delete_keys FROM STDIN...", buffer)
cur.execute("DELETE FROM final_table USING delete_keys WHERE ...")
```

• Removes existing rows that conflict with the new batch, ensuring **idempotency**.

12. Write Final Data with psycopg2 COPY

```
pandas_df = final_df.toPandas()
buffer = StringIO(); pandas_df.to_csv(buffer, ...)
cur.copy_expert("COPY ... FROM STDIN WITH CSV ...", buffer)
conn.commit()
```

- Uses fastest bulk load method into PostgreSQL (COPY FROM STDIN).
- Bypasses Spark JDBC write, giving **better control + speed**.

13. Archive Raw Files

```
s3.copy_object(...); s3.delete_object(...)
```

- Moves processed raw files from /raw/ to /processed_raw/ in S3.
- Ensures raw zone is clean and archival is retained.

End Result

Step	Output
Deduplication	1 record per (emp_id, start_date, end_date)
Status marking	ACTIVE or INACTIVE set properly
Continuity enforced	Gaps/overlaps adjusted via <pre>lead() + end_date</pre> logic
PostgreSQL staging	Used for validation before final merge
Final table written cleanly	via DELETE + COPY, with backup and overwrite safety
Raw data archived	To ensure traceability and avoid reprocessing

Question/Answer

In PySpark, Lead() is a **window function** that lets you **access the next row's value** within a partition, without changing the order of your data.

What does lead() do?

It **looks ahead** a certain number of rows and gives you a value from a future row in the same partition.

Use Case Example: Compare Current and Next Salary

Imagine a DataFrame of employees' salaries by month:

You want to compare each month's salary with the **next month's** for the same employee.

Code Example:

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import lead, col

# Initialize Spark
spark = SparkSession.builder.appName("LeadExample").getOrCreate()

# Sample data
data = [
    (101, "2024-01", 5000),
    (101, "2024-02", 5100),
    (101, "2024-03", 5200),
    (102, "2024-01", 6000),
    (102, "2024-02", 6000)
]
columns = ["emp_id", "month", "salary"]
```

```
df = spark.createDataFrame(data, columns)

# Define window: partition by employee, ordered by month
window_spec = Window.partitionBy("emp_id").orderBy("month")

# Add next month's salary using lead()
df_with_lead = df.withColumn("next_month_salary", lead("salary").over(window_spec))

df_with_lead.show()
```

Output:

Notes:

- The last row in each group has null as there is no "next" row.
- You can also use lead(column, offset, default) to look multiple rows ahead or provide a fallback value.

GOAL

When an employee has multiple job records over time, you need to:

- 1. Mark the older record as **INACTIVE**
- 2. Set its end_date to the start_date of the next record → This enforces temporal continuity

WHY?

Because:

- Raw data may have an open-ended record (no end_date)
- But the employee has a new role starting at a later date
- So the **old record must end where the new one begins** (no overlap or gap)

EXAMPLE

Raw Data (staging)

emp_id	start_date	end_date	designation	salary
101	2023-01-01	NULL	Analyst	50000
101	2023-06-01	NULL	Sr Analyst	60000

Problem: both records are open-ended \rightarrow conflict and overlap.

What We Want

emp_id	start_date	end_date	designation	salary	status
101	2023-01-01	2023-06-01	Analyst	50000	INACTIVE
101	2023-06-01	NULL	Sr Analyst	60000	ACTIVE

The first record now ends on the start of the next, and is marked as INACTIVE.

Step 1: Sort records per employee by start_date

```
from pyspark.sql.window import Window
from pyspark.sql.functions import lead, when, col
window_spec = Window.partitionBy("emp_id").orderBy("start_date")
```

Step 2: Use lead() to get the next start date

```
df = df.withColumn("next_start", lead("start_date").over(window_spec))
```

This gives us:

emp_id	start_date	designation	next_start
101	2023-01-01	Analyst	2023-06-01
101	2023-06-01	Sr Analyst	NULL

Step 3: Set end_date using next_start

```
df = df.withColumn("end_date", when(col("next_start").isNotNull(), col("next_start")).otherwise(None))
```

Step 4: Set status

df = df.withColumn("status", when(col("next start").isNull(), "ACTIVE").otherwise("INACTIVE"))

Final Output

emp_id	start_date	end_date	designation	status
101	2023-01-01	2023-06-01	Analyst	INACTIVE
101	2023-06-01	NULL	Sr Analyst	ACTIVE

Result

This enforces:

- No overlap
- No gap
- One current ACTIVE record
- Historical records properly ended

You're absolutely right to ask — yes, your original data already has an end_date column, and you're right to question why we seem to overwrite it with the next_start.

Let me explain what's really happening here, and why this step makes sense only in a specific situation.

🔽 What You Already Have

Your raw data (or staged data) might already include an end_date field:

emp_id	start_date	end_date	designation	salary
101	2023-01-01	2023-06-01	Analyst	50000
101	2023-06-01	NULL	Sr Analyst	60000

In this case:

- First row has an end_date: V Good
- Second row has **NULL**: assumed to be the current active record

? Then Why Recompute end_date?

We only recompute end_date like this:

```
df = df.withColumn("end_date", when(col("next_start").isNotNull(), col("next_start")).otherwise(None))
```

...if:

- 1. You are receiving data with missing or broken end_date s
- 2. Or, you **truncate** end_date s for currently active employees and want to **rebuild** them properly
- 3. Or, you're merging **new records** into an existing table and need to adjust end_date of the **older ACTIVE rows**

Real-World Justification (From Your Pipeline)

In your Glue job:

- You detect changes in ACTIVE rows.
- You **truncate** their end_date to NULL and mark them ACTIVE temporarily.
- Then, you run the lead() logic to:
 - Mark them as INACTIVE
 - Set their new end_date to the next role's start_date

So yes:

You had end_date, but you deliberately reset it to NULL for ACTIVE records to re-enforce temporal continuity.

Example of Your Use Case

Input (After Truncating ACTIVE rows):

emp_id	start_date	end_date	designation	status
101	2023-01-01	NULL	Analyst	ACTIVE
101	2023-06-01	NULL	Sr Analyst	ACTIVE

Then you apply:

```
df = df.withColumn("next_start", lead("start_date").over(...))
df = df.withColumn("end_date", when(col("next_start").isNotNull(), col("next_start")).otherwise(None))
df = df.withColumn("status", when(col("next_start").isNull(), "ACTIVE").otherwise("INACTIVE"))
```

Final Output:

emp_id	start_date	end_date	designation	status
101	2023-01-01	2023-06-01	Analyst	INACTIVE
101	2023-06-01	NULL	Sr Analyst	ACTIVE

Summary

Situation	Should you reset end_date ?
You already have clean, continuous dates	X No need to recompute
You're refreshing or replacing ACTIVE rows	Yes, recompute using lead()

So — you're not doing this blindly. You do it *because* your pipeline **deliberately truncates active records before merging**, and you need to **rebuild continuity** cleanly.

Yes — there **are alternatives** to using psycopg2.cur.copy_expert(...), but whether they're better
depends on your use case, scale, and control needs.

Let's walk through the **main alternatives**, compare them, and explain **when to use which**:

✓ Pros:

- Fastest bulk insert method (streaming into DB using COPY)
- Handles large volumes of data efficiently
- Direct control of SQL (can customize with options)

X Cons:

- More complex requires formatting CSV buffer manually
- You must manage transactions, commits, errors, and encoding

2. DataFrame.write.jdbc(..., mode="append") 🔄 (Spark



JDBC write)

df.write.mode("append").jdbc(url, table, properties)

Pros:

- Very easy to use, native to Spark
- Integrates naturally with your DataFrame transformations

X Cons:

- **Slower** for large datasets (does row-wise inserts)
- JDBC writes are not optimized for batch/bulk
- Doesn't handle primary key conflicts or upserts without custom logic
- Use it for small to medium DataFrames, and when you want simplicity.

3. PostgreSQL INSERT ... ON CONFLICT (Upsert)

This lets you **insert and update in one shot** using native SQL logic:

```
INSERT INTO employee db salary (emp id, start date, ...)
VALUES (...)
ON CONFLICT (emp id, start date)
DO UPDATE SET salary = EXCLUDED.salary, ...;
```

Pros:

- Handles upserts (insert or update) cleanly
- Works well if your schema has unique constraints

Cons:

- Not directly usable via copy_expert()
- Only works with .execute() or .executemany() **slower** for large volumes
- Use this when data might already exist and you need to **merge** instead of insert.

4. pg_bulkload or COPY FROM PROGRAM (Advanced

tools)

• These tools go beyond COPY FROM STDIN and can load data even faster, directly from files or stdin programs.

✓ Pros:

- Extremely fast (used in production-scale systems)
- Optimized for raw performance

X Cons:

- Requires external tools or custom database access
- Not compatible with basic psycopg2 usage

Use this when you manage your own PostgreSQL and need to load **hundreds of millions of rows** regularly.

TL;DR — What's Best for You?

Method	Best For	Performance	Complexity
copy_expert	Large batches, full control, max speed	444	333
DataFrame.write.jdbc	Simple daily loads, Spark-native	•	3
INSERT ON CONFLICT	Small updates, merging/upserts	44	44
pg_bulkload	Extreme performance use cases	4444	1 Very High

Recommendation for You

Since your pipeline:

- Loads daily deduplicated employee timelines
- Requires safe deletion + controlled bulk insert
- Works with PostgreSQL from EC2 or Glue
- You're doing the right thing using copy_expert() for now.

But if:

- You start needing upserts,
- Or smaller frequent updates per employee,
- You may consider INSERT ... ON CONFLICT or even materialized views.

psycopg2 is a Python library used to connect and interact with PostgreSQL databases.

What is PostgreSQL?

PostgreSQL (or Postgres) is an open-source **relational database system**, similar to MySQL or Oracle, where you store structured data in **tables**.

What does psycopg2 let you do?

It allows Python programs to:

- Connect to a PostgreSQL database
- Run SQL queries (SELECT, INSERT, UPDATE, DELETE)
- Fetch results from a query
- Commit transactions (like saving changes)
- Handle errors and exceptions

Basic Example

```
import psycopg2
# Connect to the database
conn = psycopg2.connect(
    dbname="my_database",
    user="postgres",
    password="mypassword",
    host="localhost",
    port="5432"
)
# Create a cursor to run SQL commands
cur = conn.cursor()
# Run a query
cur.execute("SELECT * FROM employees;")
# Get results
rows = cur.fetchall()
for row in rows:
    print(row)
```

What is happening here?

- connect(...) sets up the connection to your database.
- cursor() gives you an interface to run SQL commands.
- execute(...) runs an SQL statement.
- fetchall() gets all rows from the query result.
- commit() (not shown above) is used for INSERT/UPDATE/DELETE to save changes.
- close() cleans up the connection.

When is psycopg2 used?

- Web applications using Flask or Django
- Data pipelines that read/write to PostgreSQL
- Scripts that automate database tasks
- ETL jobs (like with AWS Glue or Apache Airflow)

Key Concepts

Term	Meaning
connection	The link between Python and the DB
cursor	The object used to run SQL statements
execute()	Runs SQL commands
fetch*()	Pulls data from queries
commit()	Applies changes for write operations
rollback()	Undoes changes if an error happens

psycopg2.copy_expert() ?

copy_expert() is a function in the psycopg2 library that allows you to run the PostgreSQL COPY command manually, which is one of the fastest ways to insert large amounts of data into a table.

This is different from a normal INSERT INTO ... query — which inserts rows one at a time. COPY inserts many rows at once, very efficiently.



Real Use Case

You have a big DataFrame or a large CSV file, and you want to load it into a PostgreSQL table as fast as possible from Python.

Rather than looping through rows and inserting them individually (which is slow), you stream the entire data block into the database using COPY.

Why use copy_expert()?

Because it gives you full control over the COPY command.

Syntax:

```
python
CopyEdit
cursor.copy expert(sql, file or buffer)
```

- sql: A full COPY command string (e.g. "COPY my_table FROM STDIN WITH CSV HEADER").
- file_or_buffer: A file-like object (e.g. a CSV file or an in-memory buffer like StringIO) containing the data.

Great — you're asking why INSERT ... ON CONFLICT or materialized views are recommended for **smaller, frequent updates** per employee. Let's break this down.

THE CONTEXT

You're currently doing **bulk batch inserts** using COPY (via psycopg2. copy_expert), which is excellent for:

- Inserting large batches of new data
- Rebuilding tables from scratch Full refresh or "load and replace" logic

But when you're dealing with **frequent updates** — like small changes to a few employees **throughout the day** — COPY becomes less suitable.

WHY USE INSERT ... ON CONFLICT FOR SMALL/FREQUENT UPDATES

This pattern handles **upsert logic** — meaning:

Insert a new row if it doesn't exist; **otherwise update** the existing one.

Syntax:

```
INSERT INTO employee_db_salary (emp_id, start_date, salary, ...)
VALUES ('101', '2023-01-01', 60000, ...)
ON CONFLICT (emp_id, start_date)
DO UPDATE SET salary = EXCLUDED.salary, status = EXCLUDED.status;
```

Advantages:

Feature	Why It Helps for Small Updates
Atomic upsert	Avoids you having to manually DELETE then INSERT
✓ Minimal I/O	Only touches rows that changed
Conflict- safe	Won't error out on duplicates
Real-time compatibility	Works well for streaming/micro-batch pipelines

Example:

emp_id	start_date	salary	status
101	2023-01-01	50000	ACTIVE

→ You get an updated row for 101 with salary 60000.

Using ON CONFLICT:

- It doesn't reinsert
- It **updates** the salary in-place



WHY MATERIALIZED VIEWS FOR FREQUENT QUERYING

A materialized view is a query result that's stored and can be refreshed on demand.

Useful when:

- You **frequently query** transformed or aggregated employee timelines
- The underlying data changes, but not drastically

Advantages:

Feature	Benefit
Precomputed logic	Complex joins/aggregates stored as a table
Reduces compute load	Queries become faster
Can refresh selectively	Either whole or incremental refresh

Example:

```
CREATE MATERIALIZED VIEW active employees AS
SELECT * FROM employee_db_salary WHERE status = 'ACTIVE';
```

Then you query the view, and periodically:

REFRESH MATERIALIZED VIEW active_employees;



So, Why Not Just Use COPY All the Time?

Use Case	Best Method
Big batch load (daily)	COPY
Few rows updated hourly	ON CONFLICT UPSERT
Pre-aggregated reporting	✓ Materialized View
Real-time microbatch stream	✓ ON CONFLICT + index

TL;DR

You use:

- COPY → for **bulk loading** big daily batches (like you're doing now)
- ON CONFLICT → when you have **frequent**, **row-level updates**

 Materialized views → when you need query optimization for often-used aggregates or filters

Great — let's break down the buffer concept in psycopg2. copy_expert() from the ground up, so it's clear even if you're new to it.

The Problem It Solves

psycopg2.copy_expert() is a **fast way to bulk load data into PostgreSQL**, but it doesn't read rows one-by-one.

Instead, it expects data in the same format as a **CSV file** — and it wants that **as a stream**.

That's where the **buffer** comes in.

Imagine This:

You want to run:

```
COPY employee_db_salary FROM STDIN WITH CSV HEADER
```

But instead of reading from a file on disk like:

```
COPY ... FROM '/tmp/data.csv'
```

You want to **send the data directly from memory** — no intermediate file.

Enter: StringIO buffer

```
from io import StringIO
buffer = StringIO()
```

StringIO is like a **fake file in memory** — it behaves just like a file object, but everything is stored in RAM.

🔄 Typical Workflow with Buffer

🔽 1. Convert Spark or Pandas DataFrame to CSV in memory

```
pandas_df = final_df.toPandas()
buffer = StringIO()
pandas_df.to_csv(buffer, index=False, header=True)
buffer.seek(0)  # rewind to beginning
```

You now have:

• buffer = a virtual file containing CSV content like:

```
emp_id, start_date, end_date, designation, salary, status
101, 2023-01-01, 2023-06-01, Analyst, 50000, INACTIVE
```

2. Use copy_expert() to stream this into PostgreSQL

cur.copy_expert("COPY employee_db_salary FROM STDIN WITH CSV HEADER", buffer)

This line:

- Tells PostgreSQL to bulk load rows from CSV format
- Reads directly from your in-memory buffer (instead of disk)

Term	What it really is
buffer	An in-memory string that mimics a file
StringIO	Python's fake file system in RAM
Why needed	To feed bulk CSV data into COPY FROM STDIN

Where It's Used in Your Glue Job

You're using this block near the **end of your employee timeframe processing**:

```
pandas_df = final_df.toPandas()
buffer = StringIO()
pandas_df.to_csv(buffer, index=False, header=True)
buffer.seek(0)

cur.copy_expert("COPY employee_db_salary FROM STDIN WITH CSV HEADER", buffer)
```

What's Happening, Step by Step

1. Final DataFrame is ready

You've cleaned, deduplicated, and marked all rows as ACTIVE or INACTIVE. This DataFrame (final_df) contains your **final state** for the employee_db_salary table.

2. Convert to Pandas + CSV buffer

```
pandas_df = final_df.toPandas()
buffer = StringIO()
pandas_df.to_csv(buffer, index=False, header=True)
buffer.seek(0)
```

- You convert the Spark DataFrame to Pandas
- You serialize it into a CSV string held in memory (not a file on disk)
- buffer now holds something like:

```
emp_id,start_date,end_date,designation,salary,status
101,2023-01-01,2023-06-01,Analyst,50000,INACTIVE
101,2023-06-01,,Sr Analyst,60000,ACTIVE
```

3. Send buffer into PostgreSQL using copy_expert

```
cur.copy_expert("COPY employee_db_salary FROM STDIN WITH CSV HEADER", buffer)
```

This sends the data from the buffer directly into PostgreSQL using **PostgreSQL's fastest bulk load mechanism**.

🧠 Summary: Why You Did This

You use <u>StringIO</u> buffer + <u>copy_expert()</u> to **stream thousands of cleaned employee timeline rows into PostgreSQL** without writing intermediate files — all in memory, all efficiently, all at once.

An **upsert** is a database operation that **inserts a new row** if it doesn't exist — or **updates the existing row** if it does.

The word itself = "Update" + "Insert"

It answers this question:

"If this record is new, add it. If it already exists, update it."

Why do upserts matter?

Upserts are essential when:

- You get incremental data (same employee, updated salary or status)
- You want to avoid duplicate records
- You don't want to manually check: "Do I insert or update?"

Example (PostgreSQL)

Suppose you have this table:

```
CREATE TABLE employees (
emp_id INT PRIMARY KEY,
```

```
name TEXT,
  age INT
);
```

Now you run:

```
INSERT INTO employees (emp id, name, age)
VALUES (101, 'Alice', 30)
ON CONFLICT (emp id)
DO UPDATE SET name = EXCLUDED.name, age = EXCLUDED.age;
```

🔄 What happens:

- If emp_id = 101 doesn't exist, it gets inserted.
- If emp_id = 101 already exists, its name and age are updated.

Example Input

emp_id	name	age
101	Alice	28

→ You receive an update:

```
(101, 'Alice', 30)
```

→ Using an upsert, the existing row is updated without needing a DELETE or SELECT.

How to do upsert in PostgreSQL

```
INSERT INTO table (id, val)
VALUES (1, 'A')
ON CONFLICT (id)
DO UPDATE SET val = EXCLUDED.val;
```

EXCLUDED. val refers to the new incoming value for val.

Benefits of Upsert

Benefit	Why It's Important
፭ Idempotency	Re-running same data won't duplicate rows
Conflict-safe	Avoids primary key errors
	One command does insert or update

Simplifies pipelines



What does "archive" mean in your workflow?

You are **moving** the raw file from:

s3://<bucket>/raw/employee timeframe data 2024 05 07.csv

To:

s3://<bucket>/processed_raw/employee_timeframe_data_2024_05_07.csv

You're not deleting it, and you're not overwriting — you're preserving a copy of the exact file that was processed.



Why Not Just Delete or Overwrite?

X Option 1: Delete

If you just s3. delete_object(...) after processing...

Problem	Why It's Risky
No traceability	You lose all evidence of what data was processed
Debugging impossible	Can't audit if something went wrong in transformation
Compliance risks	Some orgs need 90-day/365-day data retention policies
Reprocessing impossible	You can't rerun if the pipeline fails downstream

X Option 2: Overwrite

If you overwrite the same filename in raw/...

Problem	Why It's Bad
You lose yesterday's input	Raw zone should be immutable (write once)
You can't recover past state	No versioning, no lineage
Makes pipeline non-idempotent	Same input, different behavior over time

Why ARCHIVE is Best Practice

Advantage	Description
Traceability	You can prove exactly what was processed and when
Debuggability	If something breaks downstream, you can re-run it safely
Reprocess-friendly	Enables idempotent re-runs without needing raw files again
Audit/Compliance	You retain history of data for audit trails
⋘ Lineage-friendly	Useful for building end-to-end lineage or versioning

Example

Without Archive:

- 1. You process emp_data_2024_05_07.csv
- 2. It causes a silent corruption in PostgreSQL
- 3. You go back the next day file is gone 🔀

With Archive:

- File is safely stored in /processed_raw/
- You can reprocess from that backup if needed 🔽
- You can even diff yesterday's vs today's file

Industry Rule of Thumb

Raw data is never deleted. It's only archived.

Processed zones (bronze/silver/gold) can be overwritten. Raw should be immutable.

You **manually defined the schema** for the employee_timeframe_data.csv in your Glue job to ensure:



🔽 1. Precise Column Types and Order

When using inferSchema, Spark tries to guess column types based on file content. This can cause:

Problem	Example
X Wrong type inference	Interprets salary as string
X Missing nulls	Assumes end_date always exists
X Column order mismatch	Breaks downstream joins or inserts

Manually defining schema like:

```
StructType([
    StructField("emp_id", StringType(), True),
    StructField("start_date", DoubleType(), True),
    StructField("end_date", DoubleType(), True),
    StructField("designation", StringType(), True),
    StructField("salary", DoubleType(), True)
])
```

forces Spark to **interpret each column exactly as expected**.

2. Ensure Nullability of Critical Columns

Some columns like end_date might be null (for ACTIVE records).

If inferred wrong (e.g., non-nullable), Spark throws errors.

By explicitly setting True in StructField(..., True), you're saying:

"I know this field might be missing sometimes — and that's okay."

3. Consistency Across Days and Files

Since you ingest this file **daily**, if one day salary is all integers and next day it has a decimal, inferSchema will conflict.

With a fixed schema:

- Every file, every day, is read into the same structure
- Downstream logic (casting, writing to DB) never breaks

4. Performance Boost

Manual schema:

- Skips the header scanning and sample inference phase
- Results in **faster job startup** especially on large files

Summary: Why Manual Schema?

Reason	Benefit
✓ Type safety	Avoids wrong type guesses by Spark
✓ Null handling	Prevents crashes on missing end_date
✓ Daily job stability	Ensures schema doesn't change daily
Clean DB mapping	Matches expected PostgreSQL table types
Faster reading	Skips inference step

Great question — you're asking:

Why are we deduplicating or using keys on all three columns: (emp_id, start_date, end_date), rather than just on emp_id or (emp_id, start_date)?

Let's break this down logically and practically.

The Goal of the Key:

To uniquely identify each **designation period** for an employee.

Why (emp_id, start_date, end_date) is used:

Column	Meaning
emp_id	Which employee this record belongs to
start_date	When this job/designation began
end_date	When this job ended (or NULL

if ongoing)

Together, this trio defines a unique period in an employee's timeline.

Example:

Let's say an employee got promoted, then demoted, then rehired.

emp_id	start_date	end_date	designation	salary
101	2023-01-01	2023-06-01	Analyst	50000
101	2023-06-01	2023-08-01	Sr Analyst	60000
101	2023-08-01	NULL	Analyst	52000

- All 3 rows have **same** <code>emp_id</code>
- Some have **same** designation
- But each has a unique start + end range

So:

(emp_id, start_date, end_date) → uniquely identifies a job period
While:

X emp_id alone → NOT unique

Could return all 3 records

X (emp_id, start_date) alone → still ambiguous

Imagine if someone was re-hired or corrected a timeline with the same start date but a different end_date .

Why Not Just Use emp_id or emp_id, start_date?

Key Used	What Happens	Risk
emp_id	All records for an employee conflict	Too broad

emp_id, start_date	Breaks if same start date appears again	Possible for corrections
<pre>emp_id, start_date, end_date</pre>	Safest, truly defines a unique record	Best for timeline data

Think of It Like a Time Interval

Each record defines a timeline segment:

```
[--- start date ---- end date ---)
```

Two records with the same start_date but different end_date are not the same segment.

Hence: you need all three to be sure you're deduplicating or updating the correct one.

Summary

You use (emp_id, start_date, end_date) because:

- It defines the **full identity** of a time period
- It avoids overwriting or deleting the wrong record
- It's safe for **deduplication**, **deletion**, and **conflict resolution**

Let me know if you want to enforce this as a primary key in PostgreSQL or model it using a surrogate key (record_id).

"Why would the same start_date appear twice for the same emp_id?" Isn't that unlikely or wrong?"

Let's dig into the **real-world scenarios** where this can happen — and why your pipeline needs to handle it defensively.

1. Correction of Previously Stored Data

Imagine an HR mistake:

A record is inserted with:

```
emp id = 101, start date = 2023-01-01, end date = 2023-06-01, salary = 50000
```

Later, a corrected row arrives:

emp_id = 101, start_date = 2023-01-01, end_date = 2023-07-01, salary = 52000

🧠 Same start_date, different end_date/salary/designation

These could be duplicates or corrections — you want to keep **only the latest or most accurate one**, usually with higher salary.

2. Late Arriving Data or Reprocessing

- A job rerun or re-send of older records might re-emit:
 - Rows from 2023-01-01 for $emp_id = 101$
 - But they differ slightly in designation, salary, or end_date

Your system should:

- Detect these as potential updates
- Avoid keeping both copies with the same start_date

🔽 3. Business Rules Change Over Time

Suppose:

- Original contract ends on 2023-06-01
- Later extended retroactively to 2023-09-01
- HR system resends:

```
emp_id = 101, start_date = 2023-01-01, end_date = 2023-09-01
```

You now have **two records for the same start_date** — but the second is a better version.

4. Data from Multiple Systems

You may ingest data from:

- An HR platform
- A payroll system
- A manual update tool

Each might send partial timelines — but with the same start_date, leading to:

```
emp_id,start_date,end_date,designation
101,2023-01-01,2023-05-01,Analyst
101,2023-01-01,2023-06-01,Sr Analyst
```

If your deduplication doesn't handle this correctly, you'll break timeline continuity.

Excellent catch — and your observation is 100% valid.

You're asking:

"If we deduplicate using (emp_id, start_date, end_date), then why do we only delete based on (emp_id, start_date) later?"

Let's break this down precisely:

What's Actually Happening

Q During Deduplication:

You use (emp_id, start_date, end_date) because:

- The **combination uniquely identifies** a historical designation period.
- There could be multiple records with the same start date but **different end dates** (e.g., a mistake, correction, or rehire).

During Deletion Before Re-Insertion:

```
DELETE FROM employee_db_salary
USING delete_keys
WHERE employee_db_salary.emp_id = delete_keys.emp_id
AND employee_db_salary.start_date = delete_keys.start_date;
```

You're **deleting only on (emp_id, start_date)**, not including end_date.

? Why Not Include end_date in the DELETE?

because you're replacing or refreshing that time slice.

When you prepare your final_df, you've already:

- Marked some older rows as INACTIVE (by truncating or editing their end_date)
- Created updated versions of existing records (perhaps with new salary, designation, or corrected end date)

So, to avoid leaving **stale/duplicate time blocks**, you delete:

```
All rows with the same <code>emp_id</code> and <code>start_date</code> regardless of what their old <code>end_date</code> was
```

This avoids:

- Keeping an outdated (emp_id, start_date, old_end_date) row
- Duplicating overlapping timelines

Example

Suppose the current row in your final table is:

emp_id	start_date	end_date	designation	salary
101	2023-06-01	2023-12-01	Analyst	50000

And now your new batch includes:

emp_id	start_date	end_date	designation	salary
101	2023-06-01	2023-10-01	Analyst	52000

You want to:

- 1. **Delete the old one** (same emp_id and start_date)
- 2. Insert the new one
- → So you delete using only (emp_id, start_date) to wipe any version of that time block.

Summary

Phase	Key Used	Why
Deduplication	<pre>(emp_id, start_date, end_date)</pre>	Ensures uniqueness of historical periods
Deletion before insert	<pre>(emp_id, start_date)</pre>	Removes any previous version of the time block you're replacing

© Think of it as: "I don't care what the old end_date was — if I'm re-inserting that start_date, wipe it first."

Yes — what you're building **is a form of Slowly Changing Dimension (SCD)** — specifically, it's closest to:

SCD Type 2 (with some Type 1 behavior)

Let's unpack that with clarity.

What Is a Slowly Changing Dimension (SCD)?

In data warehousing:

SCDs are ways to **track how dimension data (like employee role or salary) changes over time** without losing history.

SCD Type 1 – Overwrite

- Simply updates the existing row
- No history is kept
- Example:
 - Old: emp_id=101, designation=Analyst
 - New: emp_id=101, designation=Sr Analyst → just overwrites the old row

SCD Type 2 – Historical Rows

- Creates a **new row** when data changes
- Keeps old row by marking its validity period (using start_date / end_date)
- Typically uses:
 - start_date , end_date
 - is_current or status

What You Are Doing:

You:

- Insert a new row for every change in designation or salary
- Use start_date / end_date to indicate the period
- Mark current rows as status = 'ACTIVE'
- Mark old ones as status = 'INACTIVE'
- Recompute end_date using lead(start_date)

★ That is classic SCD Type 2 logic.

Why Some Type 1 May Exist

In some parts of your pipeline (e.g., PostgreSQL deletions):

- You delete and replace rows with the same start_date
- This acts like a **Type 1 correction**: overwrite when the business deems a row wrong (like fixing salary)

Example of Your Table After Processing

emp_id	start_date	end_date	designation	salary	status
101	2023-01-01	2023-06-01	Analyst	50000	INACTIVE
101	2023-06-01	NULL	Sr Analyst	60000	ACTIVE

This preserves full history, tracks employee lifecycle — hallmark of SCD Type 2.

TL:DR

Feature	Your Pipeline
Track history over time	✓ Yes
Use start_date/end_date or status	✓ Yes
Insert new rows instead of overwrite	✓ Mostly
Sometimes delete & refresh records	☑ (Type 1 hybrid)

🔽 So yes — your logic implements an SCD Type 2 model, with occasional Type 1 behavior for corrections.

Your employee_timeframe_data Glue job is thoughtfully designed to handle a number of **critical** edge cases that commonly occur in slowly changing dimension (SCD) pipelines. Here's a comprehensive breakdown of the edge cases you've handled — and how your job addresses them:

1. Duplicate Records (Same emp_id, start_date, end_date)

Why it's a problem:

Multiple identical rows can lead to redundant history and incorrect salary/designation counts.

How you handled it:

Window.partitionBy("emp_id", "start_date", "end_date").orderBy(col("salary").desc())

- You keep the row with the highest salary
- Discards noisy or redundant versions of the same time block

2. Conflicting ACTIVE Records (Multiple NULL end_dates)

Why it's a problem:

Only one ACTIVE record per employee should exist. Multiple open-ended rows confuse downstream logic.

How you handled it:

- Separated end_date IS NULL and end_date IS NOT NULL into two DataFrames
- Deduplicated ACTIVE records and preserved only the top one per emp_id

3. Overlapping/Incorrect end_dates for ACTIVE records

Why it's a problem:

A new record may arrive for an employee, but the old ACTIVE record doesn't have a proper end_date .

✓ How you handled it:

- You truncate old ACTIVE records (set end_date = NULL)
- Then use $lead(start_date)$ to set their $end_date \rightarrow enforces$ continuity

```
end_date = when(is_latest == False, lead("start_date").over(...))
```

4. Gap/Overlap in Employee Timeline

🧠 Why it's a problem:

If an employee's job timeline has gaps or overlaps, it breaks historical accuracy.

How you handled it:

- lead(start_date) ensures the next record fills the gap of the previous
- when(next_start_date IS NOT NULL, next_start_date) → smooth continuity

5. Incorrect or Missing Dates (e.g., UNIX timestamps, nulls)

Why it's a problem:

Raw CSVs might have inconsistent date formats or missing end dates

✓ How you handled it:

to date(from unixtime(col("start date").cast("double")))

- Explicitly convert timestamps to dates
- . Casted salary and all fields with known schema to avoid inference issues

6. Stale Rows in PostgreSQL Final Table

Why it's a problem:

New data may conflict with old rows — if not deleted, you'll get duplicates

✓ How you handled it:

- You created a delete_keys temp table
- Deleted rows from final table using (emp_id, start_date) before insert

7. Partial Pipeline Failures / Recovery Support

Why it's a problem:

Job may fail after deleting but before inserting — leading to data loss

How you handled it:

- Backed up the final table (employee_db_salary_bkp)
- Allows rollback in case anything goes wrong

🔽 8. Archival of Raw Files to Prevent Reprocessing

Why it's a problem:

If raw files aren't moved, the same data could be processed again tomorrow

How you handled it:

- Moved raw files from /raw/ → /processed_raw/
- Ensures idempotent, one-time processing

9. Non-Deterministic Write Performance

Why it's a problem:

. jdbc() writes are slow for large DataFrames

How you handled it:

- Used psycopg2.copy_expert() with CSV StringIO buffer
- Result: fast, memory-efficient, bulk write into PostgreSQL

10. Schema Drift or Inference Errors

Why it's a problem:

CSV files may differ slightly — Spark might misinterpret types or fail

✓ How you handled it:

- Manually defined schema using StructType
- Explicit nullability and casting ensures consistency

🔽 11. Idempotency for Re-runs

Why it's a problem:

If job is re-run on same data without delete, it causes duplicates

How you handled it:

- Deletes conflicting (emp_id, start_date) rows before insert
- Ensures the same batch can be re-run safely

12. Null-safe Handling for Final Status and end_date

✓ You avoided errors like:

when(col("next_start_date").isNotNull(), col("next_start_date")).otherwise(None)

Ensures that **NULL** logic doesn't break transformation or writes.

Bonus: Separation of Concerns

- Raw file reading
- Staging write
- Conflict resolution
- Final table merge
- Backup + archival

This modular separation makes your pipeline debuggable, testable, and extensible.

Would you like a checklist or unit test strategy to validate all these edge cases in staging before writes?

Kafka

Private (https://app.clickup.com/9016770133/docs/8cq1zjn-716/8cq1zjn-876)

Here's the **complete producer flow** for your <u>producer.py</u> script — step by step, in **logical execution order**, so you understand how it works end-to-end:

🔁 Kafka Producer Flow

1. Start the script

• Run the Python script: producer.py.

2. Import required modules

```
from kafka import KafkaProducer
import json
import time
```

- Load libraries:
 - KafkaProducer: to send messages to Kafka.
 - json: to read the message file and serialize data.
 - time: to delay message sending (simulate real-time).

3. Read message list from file

```
with open(".../messages.json", "r") as f:
    messages = json.load(f)
```

Loads a list of messages like:

```
"sender": "EMP001",
   "receiver": "EMP002",
   "message": "badword1 goodword2"
}
```

4. Initialize Kafka producer

```
producer = KafkaProducer(
    bootstrap_servers="localhost:9092",
    value_serializer=lambda v: json.dumps(v).encode("utf-8")
)
```

- Connects to the local Kafka broker.
- Tells the producer to:
 - Convert messages to JSON.
 - Encode them to UTF-8 bytes (Kafka requires byte messages).

5. Loop through messages

```
for msg in messages:
    producer.send("employee_messages", msg)
    print(f" Sent: {msg}")
    time.sleep(2)
```

- Iterates through each message.
- Sends it to Kafka topic employee_messages.
- Waits 2 seconds before sending the next to simulate live conversation.
- Logs every message sent.

6. Flush and close producer

```
producer.flush()
producer.close()
```

- flush(): Makes sure all messages were sent from buffer to Kafka.
- close(): Shuts down the producer cleanly.

Final Result

- Messages are sent one by one to the Kafka topic.
- The Kafka **consumer** (in your system) will pick them up and process for violations.

Flow Diagram Summary (in plain words)

```
messages.json 🛛 load into list 🖾 KafkaProducer (with serializer)

🖾 for each message:

🖾 send to 'employee_messages' topic
```

Would you like a visual diagram or code walkthrough for the consumer side as well?

Here's a **line-by-line breakdown** of the <u>producer.py</u> script that sends employee messages to a Kafka topic called <u>employee_messages</u>:

Imports

```
from kafka import KafkaProducer
import json
import time
```

- KafkaProducer: Imports the Kafka client from the kafka-python library to send messages to a Kafka topic.
- json: Used to parse JSON-formatted files and convert Python objects to JSON strings.
- time: Used to introduce a delay between message sends (simulating real-time messaging).

Load Messages from File

```
with open("/home/himanshu/Learning/bootcamp_project/KAFKA_QUS/messages.json", "r") as f:
    messages = json.load(f)
```

- Opens a JSON file (messages. json) that contains a list of employee messages.
- Loads the file into a Python list of dictionaries (messages), where each message looks like:

```
"sender": "EMP001",
"receiver": "EMP002",
"message": "some text"
}
```

🗱 Configure Kafka Producer

```
producer = KafkaProducer(
   bootstrap_servers="localhost:9092",
   value_serializer=lambda v: json.dumps(v).encode("utf-8")
)
```

• bootstrap_servers="localhost:9092": Connects the producer to a Kafka broker running on the local machine at port 9092.

• value_serializer: Converts Python dictionaries (msg) into JSON strings and encodes them to UTF-8 bytes before sending them over the network. Required because Kafka works with byte strings.

📤 Send Messages to Kafka Topic

```
for msg in messages:

producer.send("employee_messages", msg)

print(f" ✓ Sent: {msg}")

time.sleep(2) # simulate delay
```

- Iterates over each message in the messages list.
- producer.send("employee_messages", msg):
 - Sends the serialized message to the Kafka topic named employee_messages.
- print(f" Sent: {msg}"): Confirms in the terminal that the message was sent.
- time. sleep(2): Waits for 2 seconds between messages to simulate real-time message flow (as
 opposed to sending all at once).

🚿 Cleanup

```
producer.flush()
producer.close()
```

- flush(): Ensures all buffered messages are actually sent to the Kafka broker.
- close(): Closes the connection to the Kafka broker cleanly.

Summary

This script:

- Loads prewritten messages from a JSON file.
- Sends them one by one to the Kafka topic employee_messages every 2 seconds.
- Converts Python objects to JSON and encodes them before sending.
- Simulates a **real-time stream** of employee communication data.

Do you want to see what a messages. json file should look like for this to work?

CONSUMER

Here's the **complete flow of consumer.py**, step-by-step, in the **same logical order it executes**, so you can explain or remember it easily:

Kafka Consumer Flow (Checkpointed Strike Logger)

Step 1: Script starts

 Starts the consumer script (consumer.py) — runs continuously to consume employee messages from Kafka.

Step 2: Setup & Imports

- Loads essential modules: kafka, json, pyspark, datetime, etc.
- Sets Kafka and PostgreSQL configuration.
- Prepares the path for saving offsets (kafka_checkpoint. json).

🔌 Step 3: Spark + JDBC Setup

• Initializes SparkSession with the Postgres JDBC JAR:

```
spark = SparkSession.builder...
```

Reduces Spark logs with:

spark.sparkContext.setLogLevel("WARN")

Step 4: Load Marked Words

Loads marked_word.json into a set of lowercase bad words.

Step 5: Define Schema & Utility Functions

- Defines schema for flagged message rows:
 - sender_id , receiver_id , message , date , strike_count
- Adds helper functions:
 - count_bad_words() → returns 1 if any word is flagged.
 - Load_checkpoint() → loads last processed Kafka offsets.
 - save_checkpoint() \rightarrow writes current offsets to JSON.

🛰 Step 6: Kafka Consumer Initialization

Creates a KafkaConsumer (manual mode):

enable_auto_commit=False

- Assigns to all topic partitions.
- Uses load_checkpoint() to resume from previous offset, or earliest if not found.

Step 7: Prepare Buffering Mechanism

- Sets buffer flush rules:
 - Every 5 seconds or
 - After 10 flagged messages

🔄 Step 8: Main Kafka Message Loop

For each message:

- 1. Deserialize and validate:
 - Convert JSON \rightarrow Python dict.
 - Skip invalid/non-dict messages.
- 2. Extract fields:
 - sender, receiver, message.
- 3. Check for bad words:
 - If flagged:
 - Create a Spark Row with current timestamp.
 - Add to buffer.

💾 Step 9: Flush Buffer to Postgres

If:

- 5 seconds passed or
- 10 flagged messages buffered

Then:

- 1. Convert buffer to Spark DataFrame.
- 2. Write to PostgreSQL table flagged_messages using .write().mode("append").

Step 10: Save Offsets

- After successful flush:
 - Extract latest Kafka offsets from each message.
 - Save using save_checkpoint() to the JSON file.

X Step 11: Failure Handling

- If Postgres write fails:
 - Catch the exception.
 - Do **not** save offsets this ensures the same messages will be **reprocessed** next time.
 - Keeps the system resilient and idempotent.

🔁 Step 12: Loop continues forever

Keeps consuming and processing incoming Kafka messages.

Final Output

Each flagged message is:

- Stored in PostgreSQL (flagged_messages).
- Used later in cooldown logic to recalculate strike count and salary.
- Tracked with offset checkpoints so **no message is lost or duplicated**.

Here's a **complete**, **line-by-line explanation** of the **consumer.py** script for consuming Kafka messages, flagging violations, and writing flagged messages to PostgreSQL with **manual offset checkpointing** and **structured buffering** using PySpark:

IMPORTS & ENVIRONMENT SETUP

from kafka import KafkaConsumer, TopicPartition
import json, os
from datetime import datetime, timedelta
from pyspark.sql import SparkSession, Row
from pyspark.sql.types import StructType, StringType, DateType, IntegerType

- **KafkaConsumer** and **TopicPartition**: Used to manually control partition assignment and offset management.
- **json, os, datetime**: For parsing, filesystem access, and time logic.
- SparkSession, Row, StructType: For creating Spark DataFrames to persist flagged messages.

CONFIGURATION VARIABLES

```
CHECKPOINT_FILE = ".../kafka_checkpoint.json"
TOPIC NAME = "employee messages"
BOOTSTRAP SERVERS = "localhost:9092"
```

- File to persist consumed offsets.
- Kafka topic and broker info.

SPARK + JDBC SETUP

```
jdbc_jar_path = "/home/himanshu/postgresql-42.7.5.jar"
assert os.path.exists(jdbc_jar_path), ...
```

Verifies PostgreSQL driver exists.

```
spark = SparkSession.builder ¥
   .appName("KafkaCustomCheckpointConsumer") ¥
    .config("spark.jars", jdbc_jar_path) ¥
    .get0rCreate()
```

Sets up Spark session with JDBC jar for writing to Postgres.

```
spark.sparkContext.setLogLevel("WARN")
```

Reduces Spark logs.

LOAD MARKED WORDS

```
with open("/.../marked_word.json", "r") as f:
    bad words = set(word.strip().lower() for word in json.load(f))
```

Loads and lowercases all flagged (marked) words.

DATABASE CONFIG

```
pg url = "jdbc:postgresql://localhost:5432/test topic"
pg_table = "flagged_messages"
```

```
pg_user = "postgres"

pg_pass = "postgres"
```

PostgreSQL JDBC connection details.

SCHEMA DEFINITION

```
schema = StructType() ¥
    .add("sender_id", StringType()) ¥
    .add("receiver_id", StringType()) ¥
    .add("message", StringType()) ¥
    .add("date", DateType()) ¥
    .add("strike_count", IntegerType())
```

Defines schema for writing to PostgreSQL via Spark.

FLAGGED MESSAGE CHECK

```
def count_bad_words(message):
   words = message.lower().split()
   return 1 if any(word in bad_words for word in words) else 0
```

• Returns 1 if any bad word is present (1 strike per message logic).

P CHECKPOINTING UTILS

• Reads last known Kafka offsets from file and maps them to TopicPartition.

Saves latest offsets back to disk.

SAFE DESERIALIZATION

```
def safe_json_deserializer(m):
    try:
        return json.loads(m.decode("utf-8"))
    except Exception as e:
        ...
    return None
```

Converts Kafka message bytes into Python dict; handles malformed JSON gracefully.

SAFKA CONSUMER WITH MANUAL CONTROL

```
consumer = KafkaConsumer(
    bootstrap_servers=B00TSTRAP_SERVERS,
    auto_offset_reset="earliest",
    enable_auto_commit=False,
    value_deserializer=safe_json_deserializer
)
```

- No consumer group ID (so fully manual).
- Prevents auto commits, letting the script handle offset tracking.

PARTITION ASSIGNMENT & OFFSET RESTORATION

```
checkpoint = load_checkpoint()
partitions = [TopicPartition(TOPIC_NAME, p) for p in consumer.partitions_for_topic(TOPIC_NAME)]
consumer.assign(partitions)

for tp in partitions:
    if tp in checkpoint:
        consumer.seek(tp, checkpoint[tp])
    else:
    ...
```

• Manually assigns partitions and restores last known offset from checkpoint file.

CONSUMPTION + BUFFERING

```
buffer = []
flush_interval = timedelta(seconds=5)
flush_max_messages = 10
last_flush_time = datetime.now()
```

Set up message buffer and flushing conditions: flush every 5s or after 10 messages.

MAIN CONSUMPTION LOOP

```
for msg in consumer:
```

Infinite consumption loop.

MESSAGE VALIDATION

```
data = msg.value
if not isinstance(data, dict):
    continue
```

Skips any invalid Kafka message.

EXTRACT AND FLAG

```
sender = data.get("sender", "")
receiver = data.get("receiver", "")
message = data.get("message", "")
strike_count = count_bad_words(message)
```

Extracts fields and evaluates message for bad word strikes.

IF FLAGGED, ADD TO BUFFER

```
if strike_count > 0:
    ...
buffer.append((msg, row))
```

• Wraps flagged message into a Spark Row and stores original Kafka message with it for offset tracking.

H FLUSH TO POSTGRES IF NEEDED

```
if datetime.now() - last_flush_time >= flush_interval or len(buffer) >= flush_max_messages:
```

Checks if it's time to flush.

BUILD DATAFRAME AND WRITE

```
rows = [r for (_, r) in buffer]
df = spark.createDataFrame(rows, schema=schema)
df.show()
df.write ... .mode("append").save()
```

• Converts rows into DataFrame and writes to Postgres via JDBC.

CHECKPOINT AFTER SUCCESS

```
latest_offsets = {}
for m, _ in buffer:
    tp = TopicPartition(m.topic, m.partition)
    latest_offsets[tp] = m.offset + 1

save_checkpoint(latest_offsets)
```

• Tracks the last successfully written offset per partition and commits it to file.

X FAILOVER

```
except Exception as e:
    print(...)
    print("Retaining buffer, not committing offsets.")
```

 If writing to Postgres fails, offsets are not committed — so messages will be reprocessed on restart.

Summary

This consumer.py script:

- Consumes Kafka messages manually (with checkpointing).
- Flags messages containing marked words.
- Buffers and writes flagged messages to PostgreSQL via Spark.
- Uses custom **offset tracking** and **recovery** logic to ensure reliability.

Would you like a flowchart showing how messages flow from Kafka to PostgreSQL with checkpoints?

COOLDOWN MECHANISM

Q&A

Let's break down **Kafka** and its key terms — in a **clear, beginner-friendly way**.

What is Apache Kafka?

Apache Kafka is a **distributed system** designed for **real-time data streaming**.

In simple terms:

Kafka helps you **send, store, and receive** messages (data) between different parts of a system — like a **messaging bus** or a **real-time pipeline**.

Why is it useful?

- You have apps or services that **generate data continuously** (logs, metrics, events).
- You want to **process that data** right away or store it for later.
- Kafka lets you **connect producers (senders) and consumers (receivers)** in a scalable and fault-tolerant way.

Key Concepts

1. Message

A single piece of data — usually a small chunk like:

```
{"user_id": 123, "action": "click", "page": "home"}
```

This is what gets sent, stored, and consumed.

2. Producer

An application that **sends messages** to Kafka.

Example: a web server logs user activity and sends it to Kafka.

3. Consumer

An application that **reads messages** from Kafka.

Example: a Spark job that processes activity logs for analytics.

4. Topic

A **named channel** in Kafka where messages are written to and read from.

Like a folder where all messages on a particular subject go.

You can think of it like:

```
Topic: user_clicks

—— Message 1

—— Message 2

—— Message 3
```

5. Partition

A topic is split into **multiple parts (partitions)** for parallelism and speed.

Each partition:

- Stores messages in the order they arrive
- Can be read in parallel

6. Offset

Each message in a partition gets a **unique number** called an **offset**, which tracks its position.

Consumers use this to:

- Know where they are
- Resume from the right spot if they restart

How It All Works Together

Kafka acts as a middleman:

- Apps send messages to topics
- Other apps read from those topics

Summary Table

Term	Meaning
Kafka	Real-time data streaming platform

Message	Unit of data (e.g., event, log, record)
Producer	Sends messages to Kafka
Consumer	Reads messages from Kafka
Topic	Logical channel for grouping messages
Partition	A split of a topic to handle data in parallel
Offset	Position of a message within a partition

Great question — and this is a **key concept** in understanding how Kafka works.

We **send messages to a Kafka topic, not directly to a consumer.

Let's break it down:

✓ Producer → Topic → Consumer

1. Producer

Your app sends a message like:

```
{"user_id": 123, "action": "click"}
```

• It sends this message to a **topic** (e.g., "user_events"), not to a specific consumer.

2. Kafka Topic

- This topic **stores** the message.
- Kafka saves the message in a **partition** inside that topic.

3. Consumer

- A consumer **subscribes** to the topic (e.g., "user_events").
- It **reads** messages from the topic, using offsets to track progress.

Why this is better

Kafka **decouples** producers and consumers.

Without	With Kafka
Kafka	

Producer must know who the consumer is	Producer just sends to a topic
Messages are sent directly	Messages are stored until consumers read them
Tight coupling	Loose coupling (more scalable)

Analogy

Think of a **topic** as a **mailbox**.

- A producer **drops letters** (messages) into the mailbox (topic).
- Consumers are like **people checking the mailbox** they come and read when they're ready.

The sender (producer) doesn't know or care who picks it up — they just deliver the letter to the mailbox.

Conclusion

Messages go to Kafka topics.

Consumers pull messages from those topics.

Let me know if you want a quick code example to show this flow.

Apache Kafka is fundamentally a **real-time data streaming platform**, **not a batch processing system**.

Kafka is built for real-time (stream) processing

- Kafka is designed to handle continuous streams of data, such as:
 - Website activity logs
 - Financial transactions
 - Sensor data

Application logs

As data flows in, Kafka **immediately stores it** in topics and makes it available to consumers, often within **milliseconds**.

But... Kafka also supports delayed / batch-style consumption

This depends on how the **consumer** reads the data.

Туре	Description
Real-time	Consumers read messages as they arrive , continuously
Batch-like	Consumers can read messages in chunks or at intervals

So, **Kafka itself is real-time**, but **you can use it in a batch-like way** by controlling:

- Polling frequency
- Offsets
- Commit intervals

Real-time Example (Streaming Consumer):

A Spark Structured Streaming job reading from Kafka:

```
spark.readStream ¥
   .format("kafka") ¥
   .option("subscribe", "user_events") ¥
   .load()
```

This reads new messages **as they arrive**, in real-time micro-batches.

Batch-style Example:

A script that runs every hour:

```
df = spark.read ¥
    .format("kafka") ¥
    .option("startingOffsets", "earliest") ¥
    .option("endingOffsets", "latest") ¥
    .load()
```

This reads a **fixed chunk** of Kafka messages — like a batch process.

Summary

Aspect	Kafka Behavior
Core Design	Real-time streaming
Message Delivery	As soon as it's produced
Consumer Control	Can simulate batch or real-time
Processing Approach	Real-time first, but batch is possible

Yes — let's walk through how **Apache Kafka works with Spark Structured Streaming**, both for **real-time streaming** and **batch-like processing**.

Spark + Kafka = Real-Time Stream Processing

Apache Spark Structured Streaming provides a high-level API to **consume Kafka data as a stream**.

Core Idea:

- Kafka is the source (live data).
- Spark reads messages as soon as they arrive.
- You define logic to transform/process/write the data continuously.

Real-Time Kafka Stream with Spark Example

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr

spark = SparkSession.builder ¥
```

```
.appName("KafkaStreamExample") ¥
    .get0rCreate()
# Read stream from Kafka
df = spark.readStream ¥
    .format("kafka") ¥
    .option("kafka.bootstrap.servers", "localhost:9092") ¥
    .option("subscribe", "user events") ¥
    .load()
# Kafka returns key and value as binary, decode them
decoded_df = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)", "timestamp")
# Write output to console (or could be S3, PostgreSQL, etc.)
query = decoded df.writeStream ¥
    .format("console") ¥
    .option("truncate", False) ¥
    .start()
query.awaitTermination()
```

What this does:

- Subscribes to Kafka topic user_events
- Streams records **as they arrive**
- Prints them continuously

Batch-style Processing from Kafka (micro-batch or one-time)

You can also configure Spark to read a chunk of Kafka messages, like a batch process.

Example: One-time read

• This reads all data from the beginning to the latest.

It doesn't run continuously — just one batch.

•

When to Use Each

Use Case	Use Streaming (readStream)	Use Batch (read)
Real-time dashboards	V	×
Continuous ETL	V	×
One-time migration	×	
Hourly reports	X (or use trigger)	(controlled batch)
Low-latency alerts	V	×

That line:

```
time.sleep(2)
```

means:

"Pause the program for 2 seconds before continuing."

In context: Simulating real-time flow

When you're sending messages to Kafka in a loop — for example:

```
import time
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='localhost:9092')

messages = ["event1", "event2", "event3"]
for msg in messages:
    producer.send("my-topic", msg.encode())
    time.sleep(2) # <--- wait for 2 seconds before sending the next</pre>
```

Without time. sleep(2), all 3 messages would be sent **instantly** — like a burst.

With time.sleep(2):

- event1 is sent
- wait 2 seconds
- event2 is sent
- wait 2 seconds

event3 is sent

This simulates **real-world data** (like user actions or IoT sensor readings) which happen **over time**, not all at once.

Summary

time. sleep(2) introduces a **delay** between each message, which helps in:

- Simulating live data streams
- Preventing system overload
- Making it easier to debug or visualize real-time processing

```
producer.flush()
```

means:

"Force all unsent messages in the Kafka producer buffer to be sent immediately."

Why is this needed?

Kafka producers **don't send messages instantly** — they often **buffer** messages to send in **batches** (for performance).

- When you call producer.send(...), it **queues** the message in memory.
- The actual sending happens **asynchronously** (in the background).
- If your program exits before the messages are flushed, they might never be sent.

So what does .flush() do?

- It waits until all buffered messages are sent to Kafka.
- It's like saying: "I'm done writing. Send everything now."

Example:

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='localhost:9092')

producer.send("my-topic", b"hello world")

# Ensure it's really sent before quitting producer.flush()
```

Without .flush(), the message might still be sitting in the buffer when the program exits.

When should you use it?

Situation	Use .flush() ?
Before shutting down the app	✓ Yes
In small scripts/test cases	✓ Yes
Inside long-running producers	X Not usually

In long-running systems, Kafka's background thread sends data continuously, so you don't need flush() unless you're shutting down or debugging.

Summary

• flush() = force all pending messages to be sent to Kafka now.

Good — let's go deeper into **how Kafka improves performance by buffering and batching messages** in the producer.

What actually happens?

When you call:

producer.send(topic, message)

Kafka does **not immediately** open a network connection and send that message.

Instead:

- 1. It puts the message into an **in-memory buffer**.
- 2. When the buffer is full or a timeout is reached, Kafka sends the whole **batch** at once.

Why does this improve performance?

1. Fewer network calls

- Without batching: each message = one network request = expensive.
- With batching: multiple messages = one network request = cheaper.

Result: much lower **network overhead** and **higher throughput**.

2. Better compression

- Compressing many messages together is more effective than compressing them one at a time.
- Kafka supports compression codecs like snappy, gzip, lz4.

3. Improved I/O efficiency

- Disk and network operations perform better when data is sent in **chunks**, not individually.
- Kafka's batching aligns with how hardware and OS caches operate.



Controlled by Producer Settings

Kafka producer config parameters that affect this behavior:

Setting	Meaning
batch.size	Max bytes to batch before sending (default ~16KB)
<u>linger.ms</u>	Max time to wait before sending even if batch isn't full
buffer.memory	Total memory Kafka can use to buffer messages
compression.type	Compress messages for network and disk efficiency

Example:

```
KafkaProducer(
 batch_size=32768,
                          # 32 KB
                         # wait up to 10ms before sending
 linger ms=10,
 compression type='gzip' # compress batches
```

Kafka is streaming, even though it uses batching internally

Yes, Kafka **batches messages** internally (for performance), but that does **not** make it a batch processing system.



Key Differences

Feature	Kafka (streaming)	Batch processing systems
Data arrival	Continuous, unbounded	One-time, fixed-size input

Processing style	Ongoing, record-by-record or micro- batch	Processed after all data is collected
Output timing	Immediate / near real-time	Delayed until full batch is processed
Examples	Kafka + Spark Streaming, Flink	Apache Spark (batch mode), Hadoop MapReduce

🗱 Why Kafka uses batching, but isn't batch processing

Kafka batches only for transport efficiency:

- Messages still arrive in real-time.
- Consumers can read them immediately, as soon as they're published.
- The batching is invisible to them unless they want to tune it.

Kafka itself:

- Does **not wait** for the whole batch to finish before making data available.
- Streams each message to consumers as soon as it's ready.

🔄 In contrast: Batch systems

A true batch system like Spark (batch mode) or Hadoop:

- Waits until a complete dataset is available
- Then processes it all together
- Only outputs after the batch job finishes

Analogy

Kafka batching is like:

You're on a train that leaves every 1 minute, even if it has only a few passengers. It picks up passengers continuously.

Batch processing is like:

A bus that only leaves after it fills up completely — you wait until it's full.

Summary

Kafka is a real-time streaming system that uses batching internally for efficiency, not as a fundamental model of processing.

The phrase "Minimizes Spark log verbosity" means:

Reduces the amount of detailed logs (like debug and info messages) that Spark prints while running.

Why this matters

By default, Apache Spark generates a **lot of log output**, such as:

- Task progress
- Stage scheduling
- Executor heartbeats
- Internal debug messages

This can **clutter your console or log files**, especially in large jobs or notebooks.

When do you want less verbosity?

Situation	Why you'd minimize logs
In notebooks (like Jupyter)	To keep output clean and readable
In production scripts	To avoid huge log files
During testing	Focus only on errors/warnings

How do you minimize verbosity?

By setting the **log level** for Spark.

Example:

spark.sparkContext.setLogLevel("WARN")

Log levels (in increasing order of verbosity):

Level	Description
ERROR	Only critical failures
WARN	Warnings and errors
INFO	General progress messages (default)
DEBUG	Detailed debug logs

TRACE

Everything (very verbose)

So if you're getting too much noise, setting WARN or ERROR can help.

Example before:

```
INFO SparkContext: Starting job...
INFO DAGScheduler: Submitting stage...
DEBUG ShuffleBlockFetcherIterator...
...
```

Example after:

```
WARN Executor: Memory usage is high ERROR Task failed: ...
```

Summary

Minimizing Spark log verbosity means adjusting logging to show only what you care about (e.g., errors or warnings), instead of flooding your output with technical internals.

Let me know if you want to configure this in <u>log4j.properties</u> or suppress logs in a PySpark notebook.

In your consumer.py, when we say:

"Assigns to all topic partitions,"

we're referring to this part of the code:

```
partitions = [TopicPartition(TOPIC_NAME, p) for p in consumer.partitions_for_topic(TOPIC_NAME)]
consumer.assign(partitions)
```

What is being assigned?

You're assigning the consumer manually to all partitions of the topic employee_messages.

Let's break it down:

- consumer.partitions_for_topic(TOPIC_NAME)
 - Queries Kafka to get **all partition IDs** for the topic.
 - Example result: {0, 1} for a topic with 2 partitions.
- 2. [TopicPartition(TOPIC_NAME, p) for p in ...]
 - Creates TopicPartition objects like:

```
[TopicPartition('employee_messages', 0), TopicPartition('employee_messages', 1)]
```

- consumer.assign(partitions)
 - Assigns those partitions explicitly to your Kafka consumer.
 - This disables Kafka's group-based auto-assignment and gives you full control.

Why you do this:

- You're using manual offset management via load_checkpoint() and seek(...).
- To do this, you must manually assign partitions using assign() otherwise, you can't call seek().

Final Result

```
consumer.assign([TopicPartition('employee_messages', 0), TopicPartition('employee_messages', 1)])
```

Now you can do:

```
consumer.seek(tp, saved_offset)
```

...to start consuming from an exact offset in each partition.

Let me know if you'd like to see how this changes when using group-based auto-commits instead.

Great — let's break down why you set up a buffering mechanism with:

- Flush every 5 seconds
- OR flush when 10 flagged messages are in the buffer

Why buffering?

If you write to PostgreSQL every time you find 1 flagged message, you'll:

- Hammer the database with too many tiny writes
- Increase network and JDBC overhead
- Slow down your streaming job
- Risk data inconsistency on partial failures

Buffering solves all of that.

10 Why 5 seconds **OR** 10 messages?

1. Flush after 10 messages

- This ensures you're writing **efficient batches** to the database.
- JDBC performs better when writing a batch instead of row-by-row.
 - ♀ If flagged messages are coming in fast, you don't wait too long once 10 are ready, you flush.

🕒 2. Flush every 5 seconds

- This avoids **delaying writes too long** when message volume is low.
- Even if only 2-3 messages came in during that time, you still persist them so your system stays real-time.
 - 💡 It's a timeout guard: flush even if batch size is small.

In short:

Goal	Solved By
Reduce database load	Batch up to 10 messages
Avoid long delays	Flush every 5 seconds
Improve reliability	Commit offsets only after successful batch write
Handle variable traffic	Combines size-based and time-based triggers

Summary

You do this to balance performance and timeliness:

- Size-based flushing keeps writes efficient.
- Time-based flushing keeps the system responsive and real-time.

ould you like to parameterize this so flush intervals or batch size can be controlled via cor LI?	ıfig or

Leave Data/ Leave Quota/ Leave Calender

Private (https://app.clickup.com/9016770133/docs/8cq1zjn-716/8cq1zjn-856)



- 1. employee_leave_quota_data.csv
- Purpose: Yearly leave allocation per employee
- columns:
- Usage:
 - Used to calculate leave usage percentage in the monthly 80% quota report
- Pipeline:
 - Append-only table by emp_id and year

2. employee_leave_calendar_data.csv

- Purpose: List of holidays for the year (non-working days)
- columns:
- Usage:
 - Exclude holidays from working day calculations in both daily and monthly reports
- Pipeline:
 - Append-only table, refreshed yearly (e.g., every Jan 1)

3. employee_leave_data.csv

- Purpose: Tracks all leave applications (one row per leave day)
- Key Fields:
 - emp_id , leave_date , leave_status (active or cancelled)
- Usage:
 - For both daily (8%) and monthly (80%) leave analysis
- Pipeline:

- Daily updated table
- Deduplicated and latest-status tracked (one row per emp_id + date)

PIPELINES AND REPORTS

A. Yearly Tables (Append-only)

- 1. Leave Quota Table
 - Tracks each employee's yearly leave quota
 - Only added to once per year

2. Leave Calendar Table

- Tracks holidays
- Updated yearly only

📆 B. Daily Updated Leave Table (at 07:00 UTC)

- Ingest employee_leave_data.csv every day
- Deduplicate by (emp_id, leave_date)
 - Keep the latest status (based on file timestamp or load order)
- Ensure only valid, non-cancelled, non-duplicate leave records persist

📊 C. Active Employees by Designation Table (daily at 07:00 UTC)

- Join current employee status with timeframe table
- Count active employees by designation:

```
designation | no_of_active_emp
```

🚨 D. Daily Report: Leaves > 8% of Remaining Working Days

- Run daily at 07:00 UTC
- Logic:
 - a. From tomorrow to Dec 31
 - b. Remove:
 - Weekends (Saturday, Sunday)

- Public holidays (from leave calendar)
- c. Count working days remaining
- d. Count active leave applications in that period
- e. If (leave_days / working_days) > 8% → flag the employee

Output:

```
emp_id | upcoming_leaves
```

- Edge Cases Handled:
 - Exclude cancelled leaves
 - Ignore weekends/holidays
 - Deduplicate repeated leave applications
- Perfect let's break down this **8% leave abuse detection logic** step-by-step, explain the formula in detail, and show you what the **final output** looks like.
- @ Objective

Identify employees who have applied for leave for **more than 8% of the remaining working days** in the current year (from tomorrow onward).

Formula

```
leave_days / working_days_remaining > 0.08
```

- Where:
- Leave_days = Number of active (not cancelled) leave applications between tomorrow and
 Dec 31
- working_days_remaining = Total number of working days from tomorrow to Dec 31
 - Exclude weekends (Sat/Sun)
 - Exclude public holidays from the holiday calendar

. 1. Define Time Window

```
start_date = today + 1 day
end_date = 31st Dec of current year
```

2. Build Working Day Calendar

```
Start from start_date to end_date
```

- Exclude:
 - date.weekday() in (5, 6) (Sat = 5, Sun = 6)
 - Dates in employee_leave_calendar_data.csv
- This gives you a list of working days remaining in the year.
- 3. Filter Leave Data
- From employee_leave_data.csv:
- Only keep:
 - leave_status = 'active'
 - leave_date >= tomorrow
 - leave_date in working_day_list (i.e., not on holiday or weekend)
- Deduplicate: (emp_id, leave_date) → keep only latest status
- Count total future leave days per employee
- 4. Apply the Formula
- For each emp_id:

```
if leave_days / working_days_remaining > 0.08:
    flag employee
```

╻ 🔽 Final Output Format

- Where:
- upcoming_leaves = number of future leave days the employee has applied for
- All of these violate the 8% rule
- Let's say today is April 19.
- Tomorrow: April 20
- End of year: Dec 31
- Total days in this range = 256

- Exclude weekends/holidays → 180 working days remaining
- Employee A applies for:
- 20 leave days on working days (active, not cancelled)
- Apply the formula:

```
20 / 180 = 11.1% \rightarrow > 8% \boxed{\checkmark} \rightarrow Employee A is flagged
```

📆 E. Monthly Report (on 1st at 07:00 UTC): Leaves > 80% of Quota

- For each employee:
 - a. From Jan 1 to current day, count active leave days
 - b. Compare with leave quota from employee_leave_quota_data
 - c. If leave taken > 80% of quota → flag

Output:

- Generate a **text file** for each flagged employee (as if manager was being notified)
- Ensure no duplicate alerts even if the job fails mid-run

• Edge Cases Handled:

- Cancelled leaves are ignored
- Leave quota defaults or missing values need handling
- Duplicate file suppression (e.g., using job run logs or checkpointing)

© Objective

Detect employees who have already used **more than 80%** of their **annual leave quota** for the current year.

This report is generated monthly, on the 1st of every month at 07:00 UTC.

Formula

```
leave_days_used / leave_quota > 0.80
```

Where:

 Leave_days_used: number of active (not cancelled) leave days taken by the employee from Jan 1 to today leave_quota: total leave allocated to that employee for the year (from employee_leave_quota_data.csv)

Step-by-Step Workflow

1. Fix the Date Range

- Start = 2025-01-01
- End = today (e.g., May 1 if it's the May run)

2. Filter and Clean Leave Data

From employee_leave_data.csv:

- . Include only:
 - leave_date >= Jan 1 AND <= today</pre>
 - status = active
- Deduplicate by (emp_id, leave_date) → keep the latest status
- Cancelled or duplicate entries are removed

3. Join with Leave Quota

- From employee_leave_quota_data.csv , get:
 - emp_id , year , quota (e.g., 30 days)

4. Apply the Formula

For each emp_id, calculate:

```
if leave_days_used / quota > 0.80:
    flag employee
```

Final Output Format

These employees are **flagged** and you generate a **text file** (e.g., "Manager_101.txt") to notify them.

Real-World Example

Let's say today is **May 1** and you're processing leave for **2025**.

Employee A:

- Quota: 30 leaves (from leave_quota_data.csv)
- Has taken 26 leave days from Jan 1 to May 1

Apply formula:

 $26 / 30 = 86.7\% > 80\% \rightarrow \checkmark$ Flag

? Why Do We Deduplicate on emp_id, leave_date and Not Just emp_id?

Same reason as the 8% logic:

- Each leave is tracked per day
- An employee may have multiple leave days over the year
- You must count all unique active leave days, not just "is this employee on leave?"

So:

- Deduplicate by (emp_id, leave_date)
- Group and count leave days per employee
- Then join with leave_quota on emp_id and year

Why Not Just Use emp_id as Key?

Key Used	Why It's Bad
emp_id only	You lose all individual leave days — not accurate
emp_id,	Each row represents a distinct day of leave — correct 🔽

import sys
import traceback
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext

```
from awsglue.job import Job
from pyspark.sql.types import StructType, StructField, IntegerType
from pyspark.sql.functions import col
# Glue job args
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
# Initialize Glue + Spark
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
# Input S3 path
INPUT_PATH = "s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp_leave_quota/"
# Schema
schema = StructType([
    StructField("emp_id", IntegerType(), True),
    StructField("leave_quota", IntegerType(), True),
    StructField("year", IntegerType(), True)
])
# JDBC connection info
jdbc url = "jdbc:postgresql://3.221.182.234:5432/test topic"
table name = "leave quota data"
db properties = {
    "user": "test_user",
    "password": "test user",
    "driver": "org.postgresql.Driver"
}
try:
    # Read data
    df = spark.read.option("header", True).schema(schema).csv(INPUT PATH)
    df = df.filter(col("emp_id").isNotNull())
    if df.rdd.isEmpty():
        print("No data found. Skipping write.")
    else:
        # Clean data
        df = df.dropDuplicates(["emp_id", "leave_quota", "year"])
        df = df.filter(
            (col("emp_id").isNotNull()) &
            (col("leave quota").isNotNull()) &
            (col("year").isNotNull())
```

```
# Write to PostgreSQL table
    df.write \( \)
        .mode("append") \( \)
        .jdbc(url=jdbc_url, table=table_name, properties=db_properties)

print(f"Successfully written to table '{table_name}'.")

except Exception:
    print("Job failed due to error:")
    print(traceback.format_exc())
    raise

finally:
    job.commit()
```

Leave_Quota

Here's a **line-by-line explanation** of your Glue job for ingesting **employee leave quota data**, with a review of **whether it's correct**, and notes on **improvements or validation where needed**.

📋 Script Purpose

Load **yearly employee leave quota data** from $S3 \rightarrow validate + deduplicate \rightarrow write to$ **PostgreSQL**table leave quota data.

🔽 Step-by-Step Analysis

Imports & Setup

```
import sys
import traceback
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql.types import StructType, StructField, IntegerType
from pyspark.sql.functions import col
```

- Correct imports all necessary modules.
- traceback is used for error logging
- col for column-wise filtering
- StructType for manual schema definition

Glue Arguments & Context

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

- Correct initializes the job properly.
- Loads JDBC driver for PostgreSQL
- Registers Glue context
- Job name comes from Glue argument

S3 Input Path

```
INPUT_PATH = "s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp_leave_quota/"
```

Correct — path to read leave quota CSV files (append-only yearly)

Manual Schema Definition

```
schema = StructType([
    StructField("emp_id", IntegerType(), True),
    StructField("leave_quota", IntegerType(), True),
    StructField("year", IntegerType(), True)
])
```

Correct & Recommended

- Manually defined schema prevents Spark from inferring wrong types
- Allows nullability for better filtering/validation

PostgreSQL Connection Info

```
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
table_name = "leave_quota_data"
db_properties = {
    "user": "test_user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}
```

Correct

Credentials and table name set

Driver matches the JAR

Record Note: In production, credentials should be securely stored in Glue connection or Secrets Manager.

📘 Try Block: Read Data

```
df = spark.read.option("header", True).schema(schema).csv(INPUT_PATH)
df = df.filter(col("emp_id").isNotNull())
```

Correct logic

- Reads CSV from S3 using given schema
- Removes records with null emp_id
- Suggestion:
 - Combine all 3 column null checks **here** instead of repeating in the next block

If Data Exists

```
if df.rdd.isEmpty():
    print("No data found. Skipping write.")
```

Correct — avoids writing an empty DataFrame

Clean & Deduplicate

```
df = df.dropDuplicates(["emp_id", "leave_quota", "year"])
df = df.filter(
     (col("emp_id").isNotNull()) &
     (col("leave_quota").isNotNull()) &
     (col("year").isNotNull())
```

Mostly correct.

However:

- You are filtering emp_id twice (earlier and again here). You can remove the first one.
- Deduplicating on all 3 fields may be too strict. You likely want:

```
dropDuplicates(["emp_id", "year"])
```

Because for the same employee and year, there should only be **one leave_quota**.

Write to PostgreSQL

```
df.write \( \)
.mode("append") \( \)
.jdbc(url=jdbc_url, table=table_name, properties=db_properties)
```

Correct

- Appends to an existing table
- Suitable for yearly incremental loads

Consider:

- Adding uniqueness constraint in PostgreSQL on (emp_id, year) to avoid logical duplication
- Add audit column like ingest_date if not already present in schema

Exception Handling

```
except Exception:

print("Job failed due to error:")

print(traceback.format_exc())

raise
```

Good practice

- Catches and logs the full traceback
- Raises the error so the job fails visibly in Glue UI

V Final Commit

```
finally:
    job.commit()
```

Needed for Glue to cleanly register the job run as complete.

Leave_calender

```
import sys
import traceback
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from awsglue.job import Job
from pyspark.sql.types import StructType, StructField, StringType, TimestampType
from pyspark.sql.functions import col, year

# Get job arguments
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
```

```
# Initialize contexts
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark session
# Set path to PostgreSQL JDBC JAR
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
# S3 source
bronze path = "s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp leave calender/"
# Define schema
schema = StructType([
    StructField("reason", StringType(), True),
    StructField("date", TimestampType(), True)
])
# PostgreSQL config
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
table_name = "leave_calendar_data"
db properties = {
    "user": "test user",
    "password": "test user",
    "driver": "org.postgresql.Driver"
}
try:
    # Read from S3
    df = spark.read ¥
        .format("csv") ¥
        .schema(schema) ¥
        .option("header", True) ¥
        .load(bronze path)
    if df.rdd.isEmpty():
        print("No data found in input. Skipping write.")
    else:
        # Clean and transform
        df = df.dropDuplicates()
        df = df.withColumn("year", year(col("date")))
        # Write to PostgreSQL
        df.write ¥
            .mode("append") ¥
            .jdbc(url=jdbc_url, table=table_name, properties=db_properties)
        print(f"Successfully written {df.count()} records to table '{table_name}'.")
```

```
except Exception as e:
    print("Job failed due to error:")
    print(traceback.format_exc())
    raise

finally:
    job.commit()
```

Here's a **line-by-line explanation and correctness review** of your AWS Glue job that ingests **employee leave calendar data** from S3 and writes it to a PostgreSQL table.

o Job Purpose

Ingest yearly public/mandatory holidays (leave calendar), clean & enrich them, and store in PostgreSQL for use in downstream logic (like excluding non-working days from leave calculations).

Imports and Setup

```
import sys
import traceback
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from awsglue.job import Job
from pyspark.sql.types import StructType, StructField, StringType, TimestampType
from pyspark.sql.functions import col, year
```

- Correct
- All necessary Glue + PySpark modules are imported
- traceback helps log full errors in case of job failure

Job Argument Setup

```
args = getResolvedOptions(sys.argv, ['JOB NAME'])
```

- Correct
- Allows Glue to access job parameters from AWS

Glue + Spark Context Initialization

```
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
```

• V Standard initialization

JDBC Driver Setup

spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")

- Works if run on EMR or EC2 with the JAR at the given location
- Consider using --extra-jars in Glue job parameters in production instead of hardcoding

Glue Job Init

```
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

• Required to track job lifecycle in Glue

Input S3 Path

bronze_path = "s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp_leave_calender/"

■ Correct path for raw bronze input

Schema Definition

```
schema = StructType([
    StructField("reason", StringType(), True),
    StructField("date", TimestampType(), True)
])
```

- Manually defining schema is good
- date as TimestampType is acceptable will work with year() transformation
- If you expect only date (no time), consider changing to DateType

PostgreSQL Configuration

```
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
table_name = "leave_calendar_data"
db_properties = {
    "user": "test_user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}
```

- Standard and correct
- Suggest storing secrets securely using Glue connections or Secrets Manager in production

Try Block for Main Logic

Read CSV from S3

```
df = spark.read ¥
    .format("csv") ¥
    .schema(schema) ¥
    .option("header", True) ¥
    .load(bronze_path)
```

- Correctly reads CSV with the defined schema
- . option("header", True) assumes first row is column names

Check for Empty Input

```
if df.rdd.isEmpty():
    print("No data found in input. Skipping write.")
```

Good safety check — avoids unnecessary PostgreSQL writes

Data Cleaning and Enrichment

```
df = df.dropDuplicates()
df = df.withColumn("year", year(col("date")))
```

- V Drops duplicate holiday entries (e.g., same date & reason)
- Extracts year from date to allow easy filtering/grouping later

Write to PostgreSQL

```
df.write \( \)
.mode("append") \( \)
.jdbc(url=jdbc_url, table=table_name, properties=db_properties)
```

- Appends the new year's holiday calendar
- Assumes target table leave_calendar_data exists

Logging Success

```
print(f"Successfully written {df.count()} records to table '{table_name}'.")
```

• V Gives post-job visibility on how many rows were inserted

Exception Handling

```
except Exception as e:
    print("Job failed due to error:")
    print(traceback.format_exc())
    raise
```

• Correct — logs detailed error, lets job fail visibly in Glue

Commit Job

```
finally:
job.commit()
```

Required for Glue job lifecycle management

Final Output Example (PostgreSQL Table)

reason	date	year
Independence Day	2025-08-15	2025
Christmas	2025-12-25	2025

Leave_Data

```
import sys
from datetime import datetime
import traceback
from pyspark.sql.types import StructType, StructField, StringType, DateType, LongType
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from awsglue.job import Job
from pyspark.sql.functions import (
        col, lit, current_timestamp, row_number, when, sum, year, month, dayofweek
)
from pyspark.sql import Window
# Initialize Glue job
```

```
args = getResolvedOptions(sys.argv, ['JOB NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
job.init(args['JOB NAME'], args)
# S3 Input Path
bucket = "poc-bootcamp-capstone-group1"
bronze_path = f"s3://{bucket}/poc-bootcamp-group1-bronze/emp_leave_data/"
# PostgreSQL Configuration
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
db properties = {
    "user": "test user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}
emp salary table = "employee db salary"
final output table = "leave data"
# Schema for leave data
leave schema = StructType([
    StructField("emp id", StringType(), True),
    StructField("date", DateType(), True),
    StructField("status", StringType(), True)
])
try:
    # Step 1: Read leave data from S3
    leave_data = spark.read.schema(leave_schema).option("header", True).csv(bronze_path)
    leave data = leave data.filter(~dayofweek(col("date")).isin([1, 7])) # Remove weekends
    # Step 2: Read ACTIVE employees from PostgreSQL
    emp time df = spark.read.jdbc(url=jdbc url, table=emp salary table, properties=db properties) ¥
                           .filter(col("status") == "ACTIVE") \u2204
                           .select("emp id").distinct()
    print("Active employee count:", emp_time_df.count())
    # Step 3: Join leave with active employee list
    leave data = leave data.join(emp time df, on="emp id", how="left semi")
    # Step 4: Add metadata
    today str = datetime.utcnow().strftime('%Y-%m-%d')
    leave_data = leave_data.withColumn("ingest_date", lit(today_str)) ¥
                           .withColumn("ingest_timestamp", current_timestamp())
    # Step 5: Status resolution
```

```
status_df = leave_data.withColumn("is_active", when(col("status") == "ACTIVE", 1).otherwise(0)) \( \)
                          .withColumn("is cancelled", when(col("status") == "CANCELLED", 1).otherwise(0))
    status summary = status df.groupBy("emp id", "date").agg(
        sum("is_active").alias("active_count"),
        sum("is_cancelled").alias("cancelled_count")
    )
    final_status = status_summary.withColumn(
        "final status",
        when(col("cancelled count") >= col("active count"), lit("CANCELLED")).otherwise(lit("ACTIVE"))
    )
    # Step 6: Keep only final rows
    filtered_df = leave_data.join(
        final_status.select("emp_id", "date", "final_status"),
        on=["emp id", "date"]
    ).filter(col("status") == col("final_status"))
    # Step 7: Deduplicate
    window_spec = Window.partitionBy("emp_id", "date").orderBy(col("ingest_timestamp").desc())
    deduped df = filtered df.withColumn("row num", row number().over(window spec)) ¥
                            .filter(col("row num") == 1) ¥
                            .drop("row_num", "final_status") ¥
                            .withColumn("year", year(col("date"))) ¥
                            .withColumn("month", month(col("date"))) ¥
                            .withColumn("emp_id", col("emp_id").cast(LongType()))
    # Step 8: Write to PostgreSQL
    if not deduped df.rdd.isEmpty():
        deduped df.write ¥
            .mode("overwrite") ¥
            .jdbc(url=jdbc_url, table=final_output_table, properties=db_properties)
        print(f"Leave data written to PostgreSQL table '{final output table}'.")
    else:
        print("No leave data to write. Skipping DB write.")
except Exception:
    print("Job failed due to error:")
    print(traceback.format exc())
    raise
# Finalize
job.commit()
```



Ingest daily employee leave data from S3, deduplicate records, resolve status (active/cancelled), filter out invalid rows, and load the clean result into a PostgreSQL table.

🔽 Imports & Initialization

```
import sys
from datetime import datetime
import traceback
from pyspark.sql.types import StructType, StructField, StringType, DateType, LongType
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from awsglue.job import Job
from pyspark.sql.functions import (
    col, lit, current_timestamp, row_number, when, sum, year, month, dayofweek
)
from pyspark.sql import Window
```

- All necessary imports are here
- dayofweek , row_number , and Window are especially important for filtering weekends and deduplication

🔽 Job Setup

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

- Correct and standard Glue job initialization
- Loads the PostgreSQL JDBC JAR

Path and JDBC Configuration

```
bucket = "poc-bootcamp-capstone-group1"
bronze_path = f"s3://{bucket}/poc-bootcamp-group1-bronze/emp_leave_data/"
```

Image: Value of the second of t

```
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
db_properties = {
```

```
"user": "test_user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}
emp_salary_table = "employee_db_salary"
final_output_table = "leave_data"
```

- PostgreSQL configs look good
- Consider using secrets manager for credentials in production

Schema Setup

```
leave_schema = StructType([
    StructField("emp_id", StringType(), True),
    StructField("date", DateType(), True),
    StructField("status", StringType(), True)
])
```

- Manually defining schema is best practice
- DateType ensures Spark can use year , month , and filter properly

Step 1: Read and Filter Leave Data

```
leave_data = spark.read.schema(leave_schema).option("header", True).csv(bronze_path)
leave_data = leave_data.filter(~dayofweek(col("date")).isin([1, 7]))
```

- Reads CSV and excludes weekends (Sunday=1, Saturday=7 correct per Spark convention)
- Optional: Also check for null dates or malformed rows

Step 2: Read Active Employees

```
emp_time_df = spark.read.jdbc(...).filter(col("status") == "ACTIVE").select("emp_id").distinct()
```

- You only include ACTIVE employees
- Good for filtering out terminated/inactive users

Step 3: Join to Filter Leave Data

```
leave_data = leave_data.join(emp_time_df, on="emp_id", how="left_semi")
```

■ Efficient "semi-join" to retain only matching employees (used correctly)

Step 4: Add Ingestion Metadata

- Z Adds helpful audit fields
- ingest_timestamp is crucial for deduplication windowing

Step 5: Status Resolution

Creates binary indicators for aggregation

```
status_summary = status_df.groupBy("emp_id", "date").agg(
    sum("is_active").alias("active_count"),
    sum("is_cancelled").alias("cancelled_count")
)
```

Summarizes how many times a date has been marked active or cancelled

```
final_status = status_summary.withColumn(
    "final_status",
    when(col("cancelled_count") >= col("active_count"), lit("CANCELLED")).otherwise(lit("ACTIVE"))
)
```

• Correct resolution logic: cancel overrides active if tied or higher

Step 6: Keep Only Final Rows

```
filtered_df = leave_data.join(
    final_status.select("emp_id", "date", "final_status"),
    on=["emp_id", "date"]
).filter(col("status") == col("final_status"))
```

What you're doing here:

You've already aggregated leave records across batches to determine the **final status** (ACTIVE or CANCELLED) for each (emp_id, date).

Now, you're using this logic to **filter out** any outdated or overridden versions of the leave for that day.

Why this is needed:

- Employees may cancel and re-apply for leave on the same date multiple times.
- Your data might look like this:

emp_id	date	status	ingest_timestamp
101	2025-05-10	ACTIVE	2025-05-05 07:00:00
101	2025-05-10	CANCELLED	2025-05-06 08:00:00
101	2025-05-10	ACTIVE	2025-05-07 09:00:00

You previously ran a groupBy-aggregation (Step 5) to resolve status as:

"The final status is **ACTIVE**, because ACTIVE count > CANCELLED count."

Now in Step 6, you:

- Join Leave_data with the resolved final_status table (from Step 5)
- Only keep rows where status == final_status

This ensures:

- You retain only the correct leave version (according to your rules)
- X You discard intermediate or outdated updates

Output after Step 6:

Only the row(s) where the status matches your resolved final outcome — one logical leave per (emp_id, date).

Step 7: Deduplication & Enrichment

```
window_spec = Window.partitionBy("emp_id", "date").orderBy(col("ingest_timestamp").desc())

deduped_df = filtered_df.withColumn("row_num", row_number().over(window_spec)) \( \)
    .filter(col("row_num") == 1) \( \)
    .drop("row_num", "final_status") \( \)
    .withColumn("year", year(col("date"))) \( \)
    .withColumn("month", month(col("date"))) \( \)
    .withColumn("emp_id", col("emp_id").cast(LongType()))
```

What you're doing here:

Even after status resolution, you might still have **duplicate rows** per (emp_id, date) if:

- The same status is repeated multiple times
- Same leave was updated repeatedly, even with no status change

You now:

- 1. **Partition** the data by emp_id and date
- 2. **Order** by <code>ingest_timestamp</code> DESC most recent update comes first
- 3. Use row_number() to assign ranks
- 4. Keep only the **first row per group** (i.e., the latest one)

This guarantees:

Exactly one most recent row per leave date per employee

X What else happens in Step 7:

- year() and month() are extracted for partitioning and reporting
- emp_id is cast to LongType for compatibility with PostgreSQL (which stores IDs as long integers)

★ Final Output: deduped_df

emp_id	date	status	ingest_timestamp	year	month
101	2025-05-10	ACTIVE	2025-05-07 09:00:00	2025	5

Exactly one authoritative record per employee per leave day, clean, status-resolved, and ready for database writing.

Let me know if you'd like to track multiple changes over time instead of just keeping the final row (e.g., Type 2 SCD style history).

Step 8: Write to PostgreSQL

```
if not deduped_df.rdd.isEmpty():
    deduped_df.write.mode("overwrite").jdbc(...)
```

```
else:
print("No leave data to write. Skipping DB write.")
```

- Prevents unnecessary DB operations
- Risk: Using .mode("overwrite") overwrites entire table every day, which may not be ideal
- Suggestion: Use .mode("append") with partition overwrite or upsert logic if this table is to be cumulative

Exception Handling + Commit

```
except Exception:
    print("Job failed due to error:")
    print(traceback.format_exc())
    raise

job.commit()
```

Value Logs traceback for debugging and commits the Glue job

8 %

```
import sys
from datetime import datetime
import traceback
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql.functions import *
from pyspark.sql.types import DateType

# Init Glue job
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
```

```
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
job.init(args['JOB NAME'], args)
# JDBC config
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
db properties = {
    "user": "test_user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
calendar_table = "leave_calendar_data"
leave_table = "leave_data"
output_table = "eight_percent"
# Get parameters
args = getResolvedOptions(sys.argv, [
    'JOB_NAME', 'today', 'start_of_year', 'end_of_year', 'CURRENT_YEAR'
])
CURRENT YEAR = int(args['CURRENT YEAR'])
today = datetime.strptime(args['today'], "%Y-\m-\m')
JAN 1 = datetime.strptime(args['start of year'], "%Y-%m-%d")
end_of_year = datetime.strptime(args['end_of_year'], "%Y-%m-%d")
today str = today.strftime("%Y-%m-%d")
end_of_year_str = end_of_year.strftime("%Y-\m-\mathbb{d}")
print(f"Running for date range: {today str} to {end of year str}")
try:
    # Read from PostgreSQL
    calendar_df = spark.read.jdbc(url=jdbc_url, table=calendar_table, properties=db_properties) ¥
                          .withColumn("date", col("date").cast(DateType()))
    leave df = spark.read.jdbc(url=jdbc url, table=leave table, properties=db properties) ¥
                        .withColumn("leave_date", col("date").cast(DateType()))
    # Clean leave data
    leave df = leave df.filter((col("status") == 'ACTIVE') & (col("leave date") >= lit(JAN 1))) ¥
                       .select("emp_id", "leave_date", "status").dropDuplicates(["emp_id", "leave_date"])
    print("Loaded leave entries:", leave_df.count())
    # Generate date range
    date range = spark.sql(f"SELECT explode(sequence(to date('{today str}'),
to_date('{end_of_year_str}'))) AS date")
    weekends = date range.withColumn("day of week", dayofweek("date")).filter(col("day of week").isin([1,
7])).select("date")
    non working days = calendar df.select("date").union(weekends).distinct()
    working days = date range.join(non working days, on="date", how="left anti")
```

```
total_working_days = working_days.count()
    print("Total working days left:", total working days)
    if total working days == 0:
        print("No valid working days. Exiting.")
    else:
        # Future valid ACTIVE leaves
        future leaves = leave df ¥
            .filter((col("leave_date") >= lit(today_str)) & (col("leave_date") <= lit(end_of_year_str)))</pre>
¥
            .join(working days, leave df.leave date == working days.date, "inner") \(\xi$
            .select("emp_id", "leave_date")
        print("Valid future ACTIVE working-day leaves:", future_leaves.count())
        # Leave % Flagging
        leave counts = future leaves.groupBy("emp id") ¥
            .agg(countDistinct("leave_date").alias("upcoming_leaves_count"))
        flagged = leave counts.withColumn(
            "leave_percent", (col("upcoming_leaves_count") / lit(total_working_days)) * 100
        ).withColumn(
            "flagged", when(col("leave percent") > 8, "Yes").otherwise("No")
        )
        # Final flagged employees
        final_flagged = flagged.filter(col("flagged") == "Yes") ¥
                               .select("emp_id", "upcoming_leaves_count")
        print("Flagged employee count:", final_flagged.count())
        final flagged.show()
        # Write to PostgreSQL
        if not final flagged.rdd.isEmpty():
            final flagged.write.mode("overwrite").jdbc(
                url=jdbc_url, table=output_table, properties=db_properties
            print("Written flagged report to PostgreSQL table:", output table)
        else:
            print("No flagged employees found. Skipping write.")
except Exception:
    print("Job failed:")
    print(traceback.format exc())
    raise
job.commit()
```

Here's a complete **line-by-line explanation** of your **"8% Leave Abuse Detection" Glue job**, including:

- Correctness
- Potential issues
- Suggestions for improvement



o Job Purpose

Identify employees who have applied for leave on **more than 8% of remaining working days** in the current year — excluding weekends and holidays — and write them to a PostgreSQL table.

Initialization and Configuration

```
import sys
from datetime import datetime
import traceback
```

Standard imports for parameter parsing and error handling

```
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql.functions import *
from pyspark.sql.types import DateType
```

■ V All Spark/Glue functions needed for SQL, windowing, and date logic

Glue Job Setup

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

- Initializes Spark & Glue context
- V Loads PostgreSQL JDBC driver

PostgreSQL Config

```
jdbc_url = ...
db_properties = {...}
```

```
calendar_table = "leave_calendar_data"
leave_table = "leave_data"
output_table = "eight_percent"
```

• Correct — connects to three tables: holidays, leave data, output

📅 Date Parameter Setup

```
args = getResolvedOptions(sys.argv, ['JOB_NAME', 'today', 'start_of_year', 'end_of_year',
'CURRENT_YEAR'])
CURRENT_YEAR = int(args['CURRENT_YEAR'])
today = datetime.strptime(args['today'], "%Y-%m-%d")
JAN_1 = datetime.strptime(args['start_of_year'], "%Y-%m-%d")
end_of_year = datetime.strptime(args['end_of_year'], "%Y-%m-%d")
```

- V Dynamically sets up the date range
- Good practice to allow override of date inputs

```
today_str = today.strftime("%Y-%m-%d")
end_of_year_str = end_of_year.strftime("%Y-%m-%d")
```

• V Formats for use in SQL

4

🛓 Step 1: Read Calendar + Leave Data from PostgreSQL

```
calendar_df = spark.read.jdbc(...).withColumn("date", col("date").cast(DateType()))
leave_df = spark.read.jdbc(...).withColumn("leave_date", col("date").cast(DateType()))
```

- Reads calendar and leave data
- Casts date columns properly

Step 2: Clean Leave Data

- Filters only ACTIVE leave records
- ▶ Deduplicates by emp_id , leave_date (✔ correct logic)

📆 Step 3: Generate Remaining Working Days

```
date_range = spark.sql(f"SELECT explode(sequence(to_date('{today_str}'), to_date('{end_of_year_str}')))
AS date")
```

■ Dynamically builds list of all dates from tomorrow to Dec 31

■ Identifies Saturdays and Sundays (correct in Spark: Sunday=1, Saturday=7)

```
non_working_days = calendar_df.select("date").union(weekends).distinct()
working_days = date_range.join(non_working_days, on="date", how="left_anti")
```

✓ Joins to exclude weekends and public holidays → gets true working days left

```
total_working_days = working_days.count()
```

Counts the number of working days

Step 4: Handle Edge Case — No Working Days Left

```
if total_working_days == 0:
    print("No valid working days. Exiting.")
```

• V Prevents divide-by-zero or incorrect output

Step 5: Filter Future Leave Applications

```
future_leaves = leave_df \u2204
    .filter((col("leave_date") >= lit(today_str)) & (col("leave_date") <= lit(end_of_year_str))) \u2204
    .join(working_days, leave_df.leave_date == working_days.date, "inner") \u2204
    .select("emp_id", "leave_date")</pre>
```

- Filters only leave dates:
 - From today onward
 - That are working days
- Removes leave taken on holidays or weekends (important business rule)

📊 Step 6: Flag Leave Abuse

Counts number of future leave days per employee

```
flagged = leave_counts.withColumn(
    "leave_percent", (col("upcoming_leaves_count") / lit(total_working_days)) * 100
```

```
).withColumn(
    "flagged", when(col("leave percent") > 8, "Yes").otherwise("No")
```

- Applies the 8% rule
- Converts to percentage, not just count

```
final flagged = flagged.filter(col("flagged") == "Yes") ¥
                       .select("emp_id", "upcoming_leaves_count")
```

Keeps only the employees that violate the 8% threshold

Let's go through this "Future Leaves" logic properly and comprehensively — including what it means, why it's needed, and how it works in the context of the 8% leave abuse detection job.

What Does "Future Leaves" Mean?

"Future leaves" refer to approved leave dates that are:

- **In the future** (from today to **December 31**)
- On working days (i.e., not on weekends or public holidays)
- **Marked as ACTIVE** in the system (i.e., not cancelled)

These are leaves that an employee has already applied for and that will cause future absences from work.

Example

Let's say today is **2025-05-01**. An employee has applied for:

emp_id	leave_date	status
101	2025-05-03	ACTIVE
101	2025-05-05	ACTIVE
101	2025-05-15	CANCELLED
101	2025-06-01	ACTIVE
101	2025-06-03	ACTIVE

Resulting "future leaves" for this employee:

emp_id	leave_date
101	2025-05-05

Step 5: Filter Future Leave Applications

```
future_leaves = leave_df \u2214
    .filter((col("leave_date") >= lit(today_str)) & (col("leave_date") <= lit(end_of_year_str))) \u2214
    .join(working_days, leave_df.leave_date == working_days.date, "inner") \u2214
    .select("emp_id", "leave_date")</pre>
```

What it does:

Step	Why it matters
leave_date >= today_str	Only keep leave dates from today onward
leave_date <= end_of_year_str	Only keep leave dates up to Dec 31
.join(working_days,, "inner")	Excludes weekends and public holidays — only working-day leaves remain
.select("emp_id", "leave_date")	Finalize the set of future , valid , active leave dates per employee

At this point, future_leaves contains only those future leave days that will affect upcoming workdays.

🗱 Step 6: Flag Leave Abuse

What it does	Why it's important
groupBy("emp_id")	Work per employee
<pre>countDistinct("date")</pre>	Count how many unique working days they are on leave

Apply the 8% Rule

```
flagged = leave_counts.withColumn(
    "leave_percent", (col("upcoming_leaves_count") / lit(total_working_days)) * 100
).withColumn(
    "flagged", when(col("leave_percent") > 8, "Yes").otherwise("No")
)
```

- You already calculated total_working_days in the date range
- This computes:
- leave_percent = future leave days / total working days * 100

If the result is > 8%, they are flagged

Final Filter: Keep Only Violators

Only keeps employees who **violate** the rule and reports how many leave days they have already planned in the future.

Step 7: Write Output to PostgreSQL

```
if not final_flagged.rdd.isEmpty():
    final_flagged.write.mode("overwrite").jdbc(...)
```

- Correct write logic
- But overwrites entire table this may be risky unless it's fully regenerated daily
 - Suggest using .mode("overwrite").option("truncate", "true") or use partition overwrite per date in scalable setups

X Exception Handling

```
except Exception:
    print("Job failed:")
    print(traceback.format_exc())
    raise
```

Good logging for failure trace



job.commit()

Final Output Table: eight_percent

emp_id	upcoming_leaves_count
101	23
202	18

80 %

```
import sys
import boto3
import traceback
from datetime import datetime, date
from dateutil.relativedelta import relativedelta
from awsglue.utils import getResolvedOptions
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.context import SparkContext
from pyspark.sql.functions import col, to_date, countDistinct, lit
from pyspark.sql.types import StringType
# Resolve Glue job parameters
args = getResolvedOptions(sys.argv, ['JOB NAME', 'CURRENT YEAR', 'today', 'start of year'])
CURRENT_YEAR = int(args['CURRENT_YEAR'])
today = datetime.strptime(args['today'], "%Y-%m-%d").date()
start_of_year = datetime.strptime(args['start_of_year'], "%Y-%m-%d").date()
run_date_str = today.strftime("%Y-%m-%d")
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
```

```
job = Job(glueContext)
job.init(args['JOB NAME'], args)
bucket = "poc-bootcamp-capstone-group1"
output_prefix = "poc-bootcamp-group1-gold/emp_80_percent_flagged/"
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
db properties = {
    "user": "test_user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
leave table = "leave data"
quota_table = "leave_quota_data"
holiday_table = "leave_calendar_data"
final_output_table = "eighty_percent"
s3_client = boto3.client("s3")
try:
    print("Step 1: Read Leave, Quota & Holiday Data from PostgreSQL")
    leave df = spark.read.jdbc(url=jdbc url, table=leave table, properties=db properties) ¥
        .withColumn("date", to_date("date"))
    quota df = spark.read.jdbc(url=jdbc url, table=quota table, properties=db properties) ¥
        .withColumnRenamed("year", "quota_year") ¥
        .withColumn("emp id", col("emp id").cast(StringType()))
    holiday_df = spark.read.jdbc(url=jdbc_url, table=holiday_table, properties=db_properties) ¥
        .withColumn("holiday_date", to_date("date")) ¥
        .filter(col("year") == CURRENT YEAR) ¥
        .select("holiday_date").dropDuplicates()
    print("Step 2: Filter ACTIVE leave records for current year and exclude holidays")
    valid leaves = leave df.filter(
        (col("status") == "ACTIVE") &
        (col("date") >= lit(start of year)) &
        (col("date") <= lit(today))</pre>
    ).select("emp_id", "date").distinct().withColumn("emp_id", col("emp_id").cast(StringType()))
    valid_leaves = valid_leaves.join(broadcast(holiday_df), valid_leaves.date == holiday_df.holiday_date,
"left anti")
    print("Step 3: Count distinct leave days")
    leaves taken = valid leaves.groupBy("emp id").agg(countDistinct("date").alias("leaves taken"))
    print("Step 4: Join with quota and calculate usage")
    leave usage = leaves taken.join(quota df, on="emp id", how="inner") ¥
        .filter(col("quota year") == CURRENT YEAR) ¥
```

```
.filter(col("leave quota") > 0) \neq
   .withColumn("leave percent", (col("leaves taken") / col("leave quota")) * 100) ¥
   .withColumn("flagged", lit("Yes")) ¥
   .filter(col("leave percent") > 80)
flagged count = leave usage.count()
print("Total flagged employees:", flagged_count)
if flagged_count == 0:
   print("No employees exceeded 80% usage. Skipping report generation.")
else:
   print("Step 5: Avoid duplicates and write TXT reports")
   existing_keys = set()
    response = s3_client.list_objects_v2(Bucket=bucket, Prefix=output_prefix)
   for obj in response.get("Contents", []):
        key = obj["Key"]
        if key.endswith(".txt"):
            emp = key.split("/")[-1].split("_")[0]
            existing keys.add(emp)
   for row in leave usage.collect():
        emp id = row["emp id"]
        if emp id in existing keys:
            continue
        content = (
            f"Employee ID: {emp id}\u00e4n"
            f"Leave Taken: {row['leaves taken']}\f"
            f"Leave Quota: {row['leave quota']}\u00e4n"
            f"Usage: {row['leave percent']:.2f}%\f"
            f"Report Date: {run_date_str}\n"
        key = f"{output_prefix}{emp_id}_report.txt"
        s3_client.put_object(Bucket=bucket, Key=key, Body=content.encode("utf-8"))
        print(f"Report written for: {emp id}")
   print("Step 6: Write final DataFrame to PostgreSQL")
   final_df = leave_usage.select(
        "emp_id",
        "leaves_taken",
        "leave_quota",
        col("quota_year").alias("year"),
        "leave percent",
        "flagged"
   )
   final_df.write.mode("overwrite").jdbc(
        url=jdbc url,
        table=final_output_table,
```

```
properties=db_properties
)

print(f"Final summary written to PostgreSQL table: {final_output_table}")

except Exception as e:
    print("Job failed with error:")
    print(traceback.format_exc())
    raise

job.commit()
```

Here's a **line-by-line explanation and correctness review** of your **Glue job for the 80% leave quota usage check**. This job identifies employees who have used **more than 80% of their annual leave quota** and generates **text reports per employee** + updates a summary table.

6 Job Objective

Identify employees whose leave usage has exceeded 80% of their yearly quota and write:

- A report per employee (as a .txt file in S3)
- A summary row to a PostgreSQL table

Imports and Job Setup

```
import sys, boto3, traceback
from datetime import datetime, date
from dateutil.relativedelta import relativedelta
```

■ Brings in required modules for AWS, time logic, and error handling

```
from awsglue.utils import getResolvedOptions
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.context import SparkContext
from pyspark.sql.functions import col, to_date, countDistinct, lit
from pyspark.sql.types import StringType
```

Spark/Glue modules for job logic and SQL functions

K Glue and Spark Context

```
args = getResolvedOptions(sys.argv, ['JOB_NAME', 'CURRENT_YEAR', 'today', 'start_of_year'])
CURRENT_YEAR = int(args['CURRENT_YEAR'])
today = datetime.strptime(args['today'], "%Y-%m-%d").date()
start_of_year = datetime.strptime(args['start_of_year'], "%Y-%m-%d").date()
run_date_str = today.strftime("%Y-%m-%d")
```

- Value
 Pulls in date parameters
- Converts them into usable Python date objects

```
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
```

Initializes Spark and sets the PostgreSQL JDBC driver

```
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

╻ 🔽 Glue job initialization

Configuration

```
bucket = "poc-bootcamp-capstone-group1"
output_prefix = "poc-bootcamp-group1-gold/emp_80_percent_flagged/"
```

Output prefix in S3 for text reports

```
jdbc_url = "jdbc:postgresql://..."

db_properties = {...}

leave_table = "leave_data"

quota_table = "leave_quota_data"

holiday_table = "leave_calendar_data"

final_output_table = "eighty_percent"
```

IDBC configs and table references

```
s3_client = boto3.client("s3")
```

AWS client for writing . txt files

Step 1: Read Tables

```
leave_df = spark.read.jdbc(...).withColumn("date", to_date("date"))
quota_df = spark.read.jdbc(...).withColumnRenamed("year", "quota_year")...
holiday_df = spark.read.jdbc(...).withColumn("holiday_date", to_date("date"))...
```

- Reads and cleans data from PostgreSQL
- Converts date columns to DateType
- ╻ 🔽 Filters holidays to current year

Step 2: Filter Leave Data

```
valid_leaves = leave_df.filter(
    (col("status") == "ACTIVE") &
    (col("date") >= lit(start_of_year)) &
    (col("date") <= lit(today))
).select("emp_id", "date").distinct()...</pre>
```

- Filters only current-year ACTIVE leaves
- Keeps unique leave dates (per employee)

```
valid_leaves = valid_leaves.join(broadcast(holiday_df), valid_leaves.date == holiday_df.holiday_date,
  "left_anti")
```

- Removes any leave taken on holidays
- broadcast() is used properly (holiday table is small)

Step 3: Count Leaves Taken

```
leaves_taken = valid_leaves.groupBy("emp_id").agg(countDistinct("date").alias("leaves_taken"))
```

• 🔽 Counts how many working (non-holiday) leave days each employee took

Step 4: Join with Quota & Filter >80%

```
leave_usage = leaves_taken.join(quota_df, on="emp_id", how="inner") \u224
    .filter(col("quota_year") == CURRENT_YEAR) \u224
    .filter(col("leave_quota") > 0) \u224
    .withColumn("leave_percent", (col("leaves_taken") / col("leave_quota")) * 100) \u224
    .withColumn("flagged", lit("Yes")) \u224
    .filter(col("leave_percent") > 80)
```

- Correctly:
 - Matches leave data with quota
 - Filters for >80% usage
 - Handles divide-by-zero risk with leave_quota > 0

Step 5: Avoid Duplicates & Write TXT Files

```
response = s3_client.list_objects_v2(Bucket=bucket, Prefix=output_prefix)
for obj in response.get("Contents", []):
    key = obj["Key"]
    if key.endswith(".txt"):
```

```
emp = key.split("/")[-1].split("_")[0]
existing_keys.add(emp)
```

- Scans for already-written . txt reports
- Prevents re-generating reports if the job is retried

```
for row in leave_usage.collect():
    emp_id = row["emp_id"]
    if emp_id in existing_keys:
        continue

    content = f"""Employee ID: {emp_id}
Leave Taken: {row['leaves_taken']}
Leave Quota: {row['leave_quota']}
Usage: {row['leave_percent']:.2f}%
Report Date: {run_date_str}
"""

    key = f"{output_prefix}{emp_id}_report.txt"
    s3_client.put_object(Bucket=bucket, Key=key, Body=content.encode("utf-8"))
    print(f"Report written for: {emp_id}")
```

- Generates per-employee . txt file only if not already present
- Writes to S3 using put_object()

Great — here's a **comprehensive explanation** of the full code block you're asking about, so you can confidently explain it to others during code reviews, demos, or interviews.

What Is This Code For?

This block handles **per-employee** .txt report generation for employees who have **used more** than 80% of their leave quota.

But — to avoid re-generating duplicate reports on retries or re-runs, it first **checks which reports already exist in S3**, and **only creates new ones** for employees who don't have a file yet.

Step-by-Step Breakdown

Step 1: List Existing .txt Files in S3

```
response = s3_client.list_objects_v2(Bucket=bucket, Prefix=output_prefix)
```

- This uses the boto3 S3 client to list all objects (files) **starting with the prefix** (i.e., the S3 folder).
- It returns a dictionary-like response containing file metadata like Key, Size, etc.

Step 2: Extract Employee IDs from Already Generated Files

```
for obj in response.get("Contents", []):
    key = obj["Key"]
    if key.endswith(".txt"):
        emp = key.split("/")[-1].split("_")[0]
        existing_keys.add(emp)
```

What it's doing:

- Loops through each file found in the folder
- Checks if the file ends in . txt (to filter only the report files)
- Extracts the emp_id from the file name:
- E.g., "emp_101_report.txt" \rightarrow emp_id = "101"
- Adds that emp_id to a Python set() called existing_keys

Why this is important:

- So we know which employee reports have already been written
- Prevents duplicate files in case the Glue job is rerun

Step 3: Generate a .txt Report for New Employees Only

```
for row in leave_usage.collect():
    emp_id = row["emp_id"]
    if emp_id in existing_keys:
        continue
```

- Loops over each flagged employee in the final leave usage DataFrame
- Skips the employee if a report already exists (based on S3 check)

Step 4: Build the Report Content

```
content = f"""Employee ID: {emp_id}
Leave Taken: {row['leaves_taken']}
Leave Quota: {row['leave_quota']}
Usage: {row['leave_percent']:.2f}%
Report Date: {run_date_str}
"""
```

- This is a simple, readable text report containing:
 - Employee ID
 - Leave statistics

Step 5: Write the Report to S3

```
key = f"{output_prefix}{emp_id}_report.txt"
s3_client.put_object(Bucket=bucket, Key=key, Body=content.encode("utf-8"))
print(f"Report written for: {emp_id}")
```

- Sets the S3 object key (i.e., path + file name)
- Writes the .txt content into the file using put_object
- Encodes the string as bytes before writing
- Logs a success message

📌 Final Takeaway: How to Explain It in a Few Sentences

"This code checks which employee reports have already been written to S3, and skips them to prevent duplicates. It only creates new . txt files for employees flagged in this run. Each report is uploaded to S3 with the employee ID as part of the file name. This ensures idempotent, traceable report generation per employee."

Let me know if you want to adapt this to generate one combined manager-wise file instead.

Step 6: Save Summary Table to PostgreSQL

```
final_df = leave_usage.select(
    "emp_id", "leaves_taken", "leave_quota",
    col("quota_year").alias("year"),
    "leave_percent", "flagged"
)
```

╻ 🔽 Prepares clean summary format

```
final_df.write.mode("overwrite").jdbc(...)
```

- overwrite works here since this job is **monthly** but be cautious:
 - It replaces entire table
 - Prefer partition overwrite if you're appending monthly results

X Error Handling

```
except Exception as e:
    print("Job failed with error:")
    print(traceback.format_exc())
    raise
```

■ Logs full traceback and fails visibly

V Job Commit

job.commit()

Required in Glue jobs

Final Output Summary

S3:

s3://poc-bootcamp-group1-gold/emp_80_percent_flagged/101_report.txt

PostgreSQL (eighty_percent table):

emp_id	leaves_taken	leave_quota	year	leave_percent	flagged
101	25	30	2025	83.33	Yes

countbyDesignation

```
import sys
import traceback
from pyspark.context import SparkContext
from pyspark.sql.functions import col, lower, count, when, lit
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from awsglue.job import Job

# Step 1: Glue job args
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")
job = Job(glueContext)
```

```
job.init(args['JOB_NAME'], args)
# Step 2: JDBC source and target info
jdbc url = "jdbc:postgresql://3.221.182.234:5432/test topic"
input_table = "employee_db_salary"
output_table = "count_by_designation"
db properties = {
    "user": "test user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}
try:
    print("Reading input data from PostgreSQL table...")
    df = spark.read.jdbc(url=jdbc_url, table=input_table, properties=db_properties)
    print("Filtering active employees and grouping by designation...")
    active employees df = (
        df.filter(lower(col("status")) == "active")
        .groupBy(
            when(col("designation").isNull(), lit("UNKNOWN"))
            .otherwise(col("designation"))
            .alias("designation")
        .agg(count("emp id").alias("active emp count"))
        .orderBy("active emp count", ascending=False)
    )
    if active employees df.count() > 0:
        print("Writing designation-wise active employee count to database...")
        active employees df.write ¥
            .mode("overwrite") ¥
            .jdbc(url=jdbc_url, table=output_table, properties=db_properties)
        print("Write successful to table 'count by designation'.")
    else:
        print("No active employees found. Nothing written.")
except Exception:
    print("Job failed due to error:")
    print(traceback.format exc())
    raise
# Step 3: Commit job
job.commit()
```

Here's a **line-by-line breakdown** and validation of your Glue job that **counts active employees by designation**, including correctness, intent, and improvement suggestions.

6 Job Objective

Connect to PostgreSQL, count the number of active employees per designation, and write the results to a new table (count_by_designation).

🔽 Imports & Initialization

```
import sys
import traceback
```

Used for argument parsing and error tracing

```
from pyspark.context import SparkContext
from pyspark.sql.functions import col, lower, count, when, lit
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from awsglue.job import Job
```

- Imports necessary Glue and Spark functionality
- col , lower , when , etc., are used for transformations

🔧 Glue Job Setup

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
spark._jsc.hadoopConfiguration().set("spark.jars", "/home/ubuntu/postgresql-42.7.2.jar")

job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

- Initializes Spark, Glue, and loads PostgreSQL driver
- Standard boilerplate for any Glue job

Database Config

```
jdbc_url = "jdbc:postgresql://3.221.182.234:5432/test_topic"
input_table = "employee_db_salary"
output_table = "count_by_designation"
db_properties = {
    "user": "test_user",
    "password": "test_user",
    "driver": "org.postgresql.Driver"
}
```

- V PostgreSQL connection setup
- Table names clearly indicate input and output purpose

Main Job Logic

```
print("Reading input data from PostgreSQL table...")

df = spark.read.jdbc(url=jdbc_url, table=input_table, properties=db_properties)
```

- Loads the entire employee_db_salary table
- Consider filtering at JDBC-level if the table is very large:

```
.option("dbtable", "(SELECT * FROM employee_db_salary WHERE status = 'ACTIVE') as sub")
```

Filtering & Grouping Logic

```
print("Filtering active employees and grouping by designation...")
active_employees_df = (
    df.filter(lower(col("status")) == "active")
```

- V Filters only status = 'ACTIVE' (case-insensitive)
- Correct to ensure no false exclusions due to casing (Active , ACTIVE , etc.)

```
.groupBy(
    when(col("designation").isNull(), lit("UNKNOWN"))
    .otherwise(col("designation"))
    .alias("designation")
)
```

- W Handles NULL designations safely by labeling them "UNKNOWN"
- Prevents null-grouping, which could cause downstream issues or confusion

```
.agg(count("emp_id").alias("active_emp_count"))
.orderBy("active_emp_count", ascending=False)
```

- Counts active employees per designation
- V Sorts from highest to lowest count

Example Output:

designation	active_emp_count
Engineer	120
Manager	45

UNKNOWN

3

Conditional Write

```
if active employees df.count() > 0:
```

- V Avoids writing empty DataFrame
- Efficient for job stability

- V Overwrites previous output
- If historical results matter, consider appending with a date column

X Error Handling

```
except Exception:
    print("Job failed due to error:")
    print(traceback.format_exc())
    raise
```

■ Logs full error and lets Glue UI capture failure trace

V Final Commit

job.commit()

Required for Glue jobs to finalize and notify AWS

Final Verdict

Section	Status	Comments
Glue initialization	✓	Fully correct and standard
JDBC read		Works fine; filtering could optionally be

		pushed to SQL
Filtering/grouping logic	V	Handles nulls and case sensitivity well
PostgreSQL write		Works as expected; overwrite is fine unless you need time history
Error handling	V	Standard and safe

Enhancements (Optional)

- Add run_date column to track daily snapshots
- Store output in S3 Parquet for querying with Athena/Glue Catalog
- Add DAG dependency (e.g., run after employee status refresh)

Let me know if you want this job parameterized to run daily and output to partitioned tables.

Question/Answer

Goal of Deduplication

You want to ensure one clean record per employee per leave date, reflecting the most current leave status (either active or cancelled).

Why Only emp_id and leave_date?

Because:

- Each row in the dataset represents a single day of leave
- If an employee updates or cancels a leave later, you'll receive another row for the same emp_id and leave_date
- So you only need to deduplicate based on:
 - Who took the leave → emp_id
 - On which day → leave_date

All other fields (e.g., status) are values that may change, not keys that define uniqueness.

Example

Suppose you have the following records for the same employee:

emp_id	leave_date	status
101	2025-05-01	active
101	2025-05-01	cancelled
101	2025-05-01	active

- All three refer to the same person, same date
- Only the status changes over time
- The **latest** record is the one you want to keep

So you **deduplicate by** (emp_id, leave_date), and then use window.orderBy(ingest_timestamp desc) to:

Keep the most recent status update for that day.

Why Not Include status or ingest_time in Deduplication Keys?

Because they **change**, and including them in .dropDuplicates([...]) would:

- Treat each version as unique
- Prevent correct tracking of the latest change

What You Actually Do

In Spark:

This:

- Keeps only the latest version of leave status
- Ensures your leave table is accurate and up-to-date

Summary

Why these columns?	Explanation
emp_i d	Identifies the employee
leave_date	Identifies the day of leave
Exclude status, timestamp	These change over time — you want only latest version kept

Excellent follow-up — you're asking:

"Why not deduplicate using just emp_id instead of emp_id + leave_date?"

Here's a clear explanation with examples:



Because an employee can apply for multiple different leave dates.

Each leave record represents a **specific day**, not a single application.

So deduplicating on just emp_id would collapse all their leave days into one — and you'd lose all the others.

Example:

Raw data from daily incremental files:

emp_id	leave_date	status
101	2025-05-01	active
101	2025-05-02	active
101	2025-05-03	cancelled
101	2025-05-02	cancelled

Now, if you do:



.dropDuplicates(["emp_id"])

You only keep **one row for emp_id 101** — all the leave day-level data is **lost**.

You'd end up with something like:

```
101 | 2025-05-01 | active
```

This is incorrect — 101 applied for **multiple distinct leave days**.

.dropDuplicates(["emp_id", "leave_date"])

This gives you the **correct logic**:

- You'll retain one row per employee per leave day
- You can then use a window or orderBy(ingest_time desc) to get the latest status for each day

Result:

```
101 | 2025-05-01 | active
101 | 2025-05-02 | cancelled ← most recent
101 | 2025-05-03 | cancelled
```

Summary

Key Used	Result
emp_id only	Keeps one row per employee, loses

	multiple leave dates (X)
<pre>emp_id , leave_date</pre>	Keeps one row per employee per leave day (correct)

Because leave is **tracked per day**, and employees can have **many leave dates**, you must use both emp_id and leave_date to preserve correctness.

left-semi

Returns only the rows from the left DataFrame where a match exists in the right DataFrame.

But — unlike other joins — it does not return any columns from the right DataFrame.

Think of it as:

"Filter the left DataFrame to keep only rows that have a matching key in the right DataFrame."

Great question — you're asking:

"Why do we need ingest_date and ingest_timestamp in the leave data?"

Let's break this down by **what each field is used for** and **why it's critical** in data pipelines — especially for leave tracking.

ingest_date: Human-readable batch metadata

What it is:

leave_data = leave_data.withColumn("ingest_date", lit(today_str))

- This marks which day this data was processed/ingested
- Comes from the Glue job's run date (e.g., 2025-05-07)

ingest_timestamp: Precise technical time for

deduplication

What it is:

leave_data = leave_data.withColumn("ingest_timestamp", current_timestamp())

- Captures the exact timestamp (to the second or millisecond) when the record was loaded
- Unlike ingest_date , it's precise and continuous

Example Scenario

Raw records for the same (emp_id, leave_date):

emp_id	date	status	ingest_timestamp
101	2025-05-10	ACTIVE	2025-05-06 07:00:00
101	2025-05-10	CANCELLED	2025-05-06 07:30:00

With ingest_timestamp, you can keep the most recent status using a window:

row_number().over(Window.partitionBy("emp_id", "date").orderBy(col("ingest_timestamp").desc()))

"Why do I need both ingest_date and ingest_timestamp in the leave data? Can't I just use one?"

Let's answer this comprehensively by comparing the two fields and their distinct purposes.

What Are They?

Column	Туре	Example	Purpose
ingest_date	yyyy-MM-dd	2025-05-07	Job-level batch identifier
ingest_timestamp	full datetime	2025-05-07 07:02:45.123	Record-level precision for sorting

Why You Need Both

- 1. ingest_date = To Daily Partition or Run Tag
- Helps identify which day's file or batch this record came from.
- Useful for:
 - Partitioning in data lake (e.g., year=2025/month=05/day=07)
 - Daily audit and tracking: "How many records did I process today?"
- Vou keep this for traceability and partitioning.
- 2. ingest_timestamp = ingest_timestamp

- Records can be updated within the same day.
- You need a high-resolution timestamp to decide which version is the latest.
- You use this for:
- row_number().over(partitionBy("emp_id", "date").orderBy(ingest_timestamp.desc()))
- Ensures **correct status is retained** per employee per leave date.

Real-World Example

Let's say you run the job twice on the same day:

emp_id	date	status	ingest_date	ingest_timestamp
101	2025-05-10	ACTIVE	2025-05-07	2025-05-07 07:00:00.000
101	2025-05-10	CANCELLED	2025-05-07	2025-05-07 08:00:00.123

Now, if you only had ingest_date, both would look like they came from the same run, and you
couldn't tell which is newer.

With ingest_timestamp, you can resolve the correct version — CANCELLED is newer, so that row wins.

Conclusion

You need:

- **ingest_date** for tracking job runs, partitioning, and human-readable filters.
- **ingest_timestamp** for **technical ordering**, deduplication, and final status selection.

Think of ingest_date as "what day's data is this?" and ingest_timestamp as "what version of this row is latest?"

Let me know if you'd like to version these rows with a row_version column too.

In Spark, a **left semi join** is a special type of join that:

Returns only the rows from the left DataFrame where a match exists in the right DataFrame.

But — unlike other joins — it does not return any columns from the right DataFrame.

Think of it as:

"Filter the left DataFrame to keep only rows that have a matching key in the right DataFrame."

Syntax:

```
df_left.join(df_right, on="emp_id", how="left_semi")
```

Example:

Left DataFrame (employees)

emp_id	name
1	John
2	Alice
3	Bob

Right DataFrame (active_ids)

emp_id
1
3

Result of left_semi join:

employees.join(active_ids, on="emp_id", how="left_semi")

emp_id	name
1	John
3	Bob

Only rows from employees where emp_id exists in active_ids are returned.

What is a **Left Anti Join** in Spark?

A **left anti join** returns all **rows from the left DataFrame that do not** have a match in the right DataFrame based on the join condition.

It's the **opposite** of a left semi join.

Syntax:

Real-world analogy:

"Give me all employees who are **not active**."

Example:

Left DataFrame (employees)

emp_id	name
1	John
2	Alice
3	Bob

Right DataFrame (active_ids)

emp_id
1
3

Code:

employees.join(active_ids, on="emp_id", how="left_anti")

Output:

emp_id	name
2	Alice

Only returns Alice — she is in employees but **not** in active_ids.

Join Type	Returns left rows?	Returns right rows?	Condition
left_semi	Yes (if matched)	No	Match

left_anti	Yes (if NOT	No	No match
	matched)		

Great — let's break down the **date range setup** section in your Glue job and explain **exactly what** it's **doing, why it's done this way**, and **how it fits into the leave percentage logic**.

Q Code in Focus

```
args = getResolvedOptions(sys.argv, ['JOB_NAME', 'today', 'start_of_year', 'end_of_year',
'CURRENT_YEAR'])

CURRENT_YEAR = int(args['CURRENT_YEAR'])
today = datetime.strptime(args['today'], "%Y-%m-%d")
JAN_1 = datetime.strptime(args['start_of_year'], "%Y-%m-%d")
end_of_year = datetime.strptime(args['end_of_year'], "%Y-%m-%d")

today_str = today.strftime("%Y-%m-%d")
end_of_year_str = end_of_year.strftime("%Y-%m-%d")
```

What This Does (Line-by-Line)

1. Get Runtime Parameters

```
args = getResolvedOptions(sys.argv, [...])
```

- You pass today, start_of_year, and end_of_year as **dynamic inputs** when triggering the Glue job (either from script or scheduler).
- This makes the job **re-runnable and testable** for different dates without code changes.

2. Parse the Dates

```
today = datetime.strptime(args['today'], "%Y-%m-%d")

JAN_1 = datetime.strptime(args['start_of_year'], "%Y-%m-%d")

end_of_year = datetime.strptime(args['end_of_year'], "%Y-%m-%d")
```

- Converts string inputs to actual Python datetime. date objects.
- These variables will define your calculation window.

3. Format for SQL and Logging

```
today_str = today.strftime("%Y-%m-%d")
end_of_year_str = end_of_year.strftime("%Y-%m-%d")
```

• These are used where Spark SQL or string-formatting is needed (e.g., sequence(...) or S3 logs).

📆 What the Date Range Represents

Variable	Meaning
start_of_year → JAN_1	Jan 1 of the current year (used to filter leave taken so far)
today	The current run date (job execution date or simulated)
end_of_year	Dec 31 of the same year (used to calculate remaining working days)

Where They're Used

▼ To filter already-taken leaves (for 80% report):

(leave.date >= JAN 1) & (leave.date <= today)

▼ To calculate future working days (for 8% abuse detection):

sequence(to date(today str), to date(end of year str))

Generates a list of all dates from tomorrow through Dec 31 for abuse prediction.

Example: If today is April 19, 2025

Parameter	Value
today	2025-04-19
start_of_year	2025-01-01
end_of_year	2025-12-31

Usage:

- 80% logic: Count leaves from Jan 1 to April 19
- 8% logic: Check leave applications between April 20 and Dec 31

The Code in Focus

```
leave_df = leave_df.filter(
     (col("status") == 'ACTIVE') &
     (col("leave_date") >= lit(JAN_1))
).select("emp_id", "leave_date", "status") \u224
     .dropDuplicates(["emp_id", "leave_date"])
```

Step 1: Filter only ACTIVE leave records

```
filter(col("status") == 'ACTIVE')
```

Removes any rows with status 'CANCELLED', 'REJECTED', or others.

▼ Step 2: Exclude Past Years

```
col("leave_date") >= lit(JAN_1)
```

- Filters records to include only this year's leave data.
- JAN_1 is typically '2025-01-01', based on the job parameter.

Step 3: Deduplicate leave rows

```
.dropDuplicates(["emp id", "leave date"])
```

- Removes multiple entries for the same person on the same day, even if they exist.
- Keeps only one version (usually the first encountered).

🗪 Real Example: Input Leave Data

emp_id	leave_date	status
101	2025-03-01	ACTIVE
101	2025-03-01	ACTIVE
101	2024-12-30	ACTIVE
102	2025-04-01	CANCELLED
103	2025-04-01	ACTIVE

After Applying Step 2:

emp_id	leave_date	status
101	2025-03-01	ACTIVE
103	2025-04-01	ACTIVE

- Leaves from previous years are gone
- ✓ Only ACTIVE rows remain
- ✓ Duplicate ACTIVE entries removed

Why This Approach Works

Action	Result
Filtering on status = 'ACTIVE'	Ensures you're only counting <i>currently valid</i> leaves
Filtering on <pre>leave_date >= JAN_1</pre>	Prevents old leave usage from affecting this year's report
Deduplication by (emp_id, leave_date)	Prevents double-counting same-day leave applications

Are There Alternatives?

Option 1: Use latest leave status resolution

Instead of filtering on status = 'ACTIVE' , you could:

- Collect all leave actions (ACTIVE , CANCELLED , etc.)
- Keep only the latest status per emp_id + leave_date based on a timestamp

 \mathbb{Z} This is more robust — especially if someone reapplies after cancelling.

🔁 Option 2: Group + Count logic

If you're not resolving status but just counting days:

```
leave_df.groupBy("emp_id", "leave_date").agg(first("status"))
```

But this may be dangerous unless you're **sure** only one status per day exists.

Absolutely — let's walk through the **"Generate Remaining Working Days"** section **step by step**, with **clear examples** and explanations of **why it's done this way**.

@ Goal of this block

Build a list of **all valid working days** from **today onward to December 31** of the current year, excluding:

- Weekends (Saturday & Sunday)
- Gazetted/Public holidays from the leave_calendar table

This is used in your **8% leave abuse check**, where you calculate:

```
(leave_days_applied / working_days_remaining) > 0.08
```

🔢 Step-by-Step Breakdown

Step 1: Generate All Calendar Dates in the Range

```
date_range = spark.sql(f"""
    SELECT explode(sequence(to_date('{today_str}'), to_date('{end_of_year_str}'))) AS date
""")
```

What it does:

- Uses Spark SQL's sequence(start_date, end_date) to create a full list of dates between today
 and end_of_year
- explode(...) turns that list into one row per date

Example:

If today_str = '2025-05-01' and end_of_year_str = '2025-12-31', this creates:

date
2025-05-01
2025-05-02
2025-05-03
•••
2025-12-31

This is your base calendar.

Step 2: Identify Weekends

```
weekends = date_range \times 
   .withColumn("day_of_week", dayofweek("date")) \times 
   .filter(col("day_of_week").isin([1, 7])) \times 
   .select("date")
```

What it does:

- Adds a column for day of week (1=Sunday, 7=Saturday)
- Filters out rows where the date is a **Saturday or Sunday**

Example:

From the full date_range, this extracts:

date	day_of_week
2025-05-03	7 (Saturday)
2025-05-04	1 (Sunday)
•••	•••

These are non-working days due to weekends.

Step 3: Combine Weekends with Public Holidays

```
non_working_days = calendar_df.select("date") \u2204
    .union(weekends) \u2204
    .distinct()
```

What it does:

- Takes:
 - Public holidays from calendar_df
 - Weekends from Step 2
- Combines both into one list of **non-working days**

Example Output:

date	
2025-05-01	

2025-05-03

2025-05-04

Step 4: Subtract Non-Working Days from Calendar

working_days = date_range.join(non_working_days, on="date", how="left_anti")

What it does:

• Performs a left anti join = "Keep only dates from date_range that do not exist in non_working_days"

Leaves only working days

✓ Final Output: working_days

date
2025-05-02
2025-05-06
2025-05-07

ightarrow These are the dates you will later compare against **leave applications**.

Why This Design Is Good

Feature	Why It Works
sequence()	Dynamically builds any range of dates
dayofweek()	Efficiently filters weekends
left_anti	Accurately removes all

	non-working days
Reusable logic	Can be reused for SLA checks, payroll cycles, or staffing

? Could You Use Alternatives?

Option A: Use a pre-generated date dimension table

- If you have one in your warehouse
- X But adds dependency and reduces portability

Option B: Use Python datetime range generation

- X Can't scale to Spark distributed processing
- X Doesn't work in Spark DataFrames

Great — you're asking:

"What do you mean by sequence(...) needs today_str and end_of_year_str?"

Let's explain this clearly with code, purpose, and example.

What is sequence(...) in Spark?

In Spark SQL and PySpark, sequence(start_date, end_date) generates an **array of dates** between the two dates (inclusive). You then use explode() to turn this into a list of rows.

Why do we need today_str and end_of_year_str?

Because sequence() needs **string literals or to_date(...)** types, and we're building SQL strings dynamically inside this line:

```
date_range = spark.sql(f"""
    SELECT explode(sequence(to_date('{today_str}'), to_date('{end_of_year_str}'))) AS date
""")
```

So you need today_str and end_of_year_str to insert values into this SQL string:

- today_str → '2025-05-01'
- end_of_year_str → '2025-12-31'

That makes the SQL turn into:

```
SELECT explode(sequence(to_date('2025-05-01'), to_date('2025-12-31'))) AS date
```

Example Output

This generates a DataFrame like:

date
2025-05-01
2025-05-02
2025-05-03
2025-12-31

Why Not Use Python datetime?

Because this needs to run inside Spark, distributed over worker nodes.

Python's datetime won't scale to Spark transformations.

Hence, we use sequence() inside **Spark SQL**, and that's why we pass string versions of the date.

Summary

- You use today_str and end_of_year_str to insert dates into a SQL query string.
- This enables dynamic generation of a date list using:

```
sequence(to_date('YYYY-MM-DD'), to_date('YYYY-MM-DD'))
```

Absolutely — let's clearly explain the difference between strptime() and strftime(), especially in the context of this line:

```
today = datetime.strptime(args['today'], "%Y-%m-%d").date()
run_date_str = today.strftime("%Y-%m-%d")
```

What's Happening Here?

1. strptime() → String to Python Date

today = datetime.strptime(args['today'], "%Y-%m-%d").date()

- args['today'] is a string, e.g., "2025-05-07" (passed into the Glue job)
- datetime.strptime(...) parses the string into a Python datetime object
- .date() strips the time portion and gives a clean date object

Converts:

"2025-05-07" (string) \rightarrow datetime.date(2025, 5, 7)

2. strftime() → Date to Formatted String

run_date_str = today.strftime("%Y-%m-%d")

- today is a Python date object (datetime. date (2025, 5, 7))
- strftime() formats that back into a string in "YYYY-MM-DD" format

Converts:

datetime.date(2025, 5, 7) \rightarrow "2025-05-07" (string)

★ Why Use Both?

Task	Function Used
Convert input string to date object	strptime()
Format date back to string (for SQL, filenames, etc.)	strftime()

In your Glue job:

strptime() is used to understand and compare dates

strftime() is used to output clean strings for SQL injection or logging

🧪 Example:

```
input_str = "2025-12-25"
date_obj = datetime.strptime(input_str, "%Y-%m-%d").date()
print(date_obj) #  2025-12-25 (as date object)

formatted = date_obj.strftime("%d-%b-%Y")
print(formatted) #  "25-Dec-2025"
```

Summary

Function	Direction	Used For
strptime()	String → datetime/date	Parsing input args, filtering logic
strftime()	datetime/date → String	Logging, file naming, SQL filters

Great — let's break down this line thoroughly:

```
leaves_taken = valid_leaves.groupBy("emp_id").agg(countDistinct("date").alias("leaves_taken"))
```

What this line is doing

- 1. valid_leaves is a DataFrame that contains future, valid, active leave dates after filtering.
 - Each row = one leave date applied by an employee on a working day.
- 2. You are now:
 - Grouping by emp_id
 - Counting how many distinct leave days the employee has applied for
 - Storing that count in a new column called "leaves_taken"

→ Output will look like:

emp_id	leaves_taken
101	5

102	2
103	0

Why use groupBy("emp_id").agg(countDistinct("date"))?

Because:

- An employee might have multiple entries for the same leave day (e.g., due to re-application, status flip)
- You only want to count each day once per employee
- countDistinct("date") ensures you don't double-count leaves on the same day

Example:

emp_id	leave_date
101	2025-06-01
101	2025-06-01
101	2025-06-02
102	2025-06-01
102	2025-06-01

Result:

- emp_id 101 → 2 distinct dates
- emp_id 102 → 1 distinct date

? Was There Any Other Way?

Yes — but they are less optimal:

X Option A: count("date")

.groupBy("emp_id").agg(count("date").alias("leaves_taken"))

- X Counts all rows, including duplicates
- X Will **inflate leave counts** if a date appears multiple times per employee

Option B: Drop duplicates first, then count

```
deduped = valid_leaves.dropDuplicates(["emp_id", "date"])
leaves_taken = deduped.groupBy("emp_id").count()
```

- V Functionally similar and correct
- Slightly less efficient than using countDistinct() directly, because:
 - You create a new intermediate DataFrame
 - You still group and count

Great — you're asking:

"Why do we need to encode the string as bytes before writing to \$3 using put_object?"

Let's break it down clearly:

Context: What you're doing

You have a report that's a **string**:

```
content = f"""Employee ID: 101
Leave Taken: 25
Leave Quota: 30
Usage: 83.33%
Report Date: 2025-05-07
"""
```

Then you do:

```
s3 client.put object(Bucket=bucket, Key=key, Body=content.encode("utf-8"))
```

Why You Need to .encode("utf-8")

- 1. S3 put_object() expects bytes, not strings
- The Body parameter in put_object() must be **binary data** (bytes or byte stream).
- Python str is a Unicode text string it's not raw bytes.
- .encode("utf-8") converts the string to a binary format.

For example:

```
"Hello".encode("utf-8") → b'Hello'
```



If you try:

```
s3_client.put_object(Bucket=..., Key=..., Body=content)
```

You'll get:

TypeError: string argument without an encoding

Because **S3 requires a byte stream**, and a raw Python string won't work.

✓ Why "utf-8"?

- It's the most common, standard encoding format
- Supports all languages, symbols, and is highly portable

You could use others ("ascii", "utf-16") — but "utf-8" is safest and default across cloud tools.