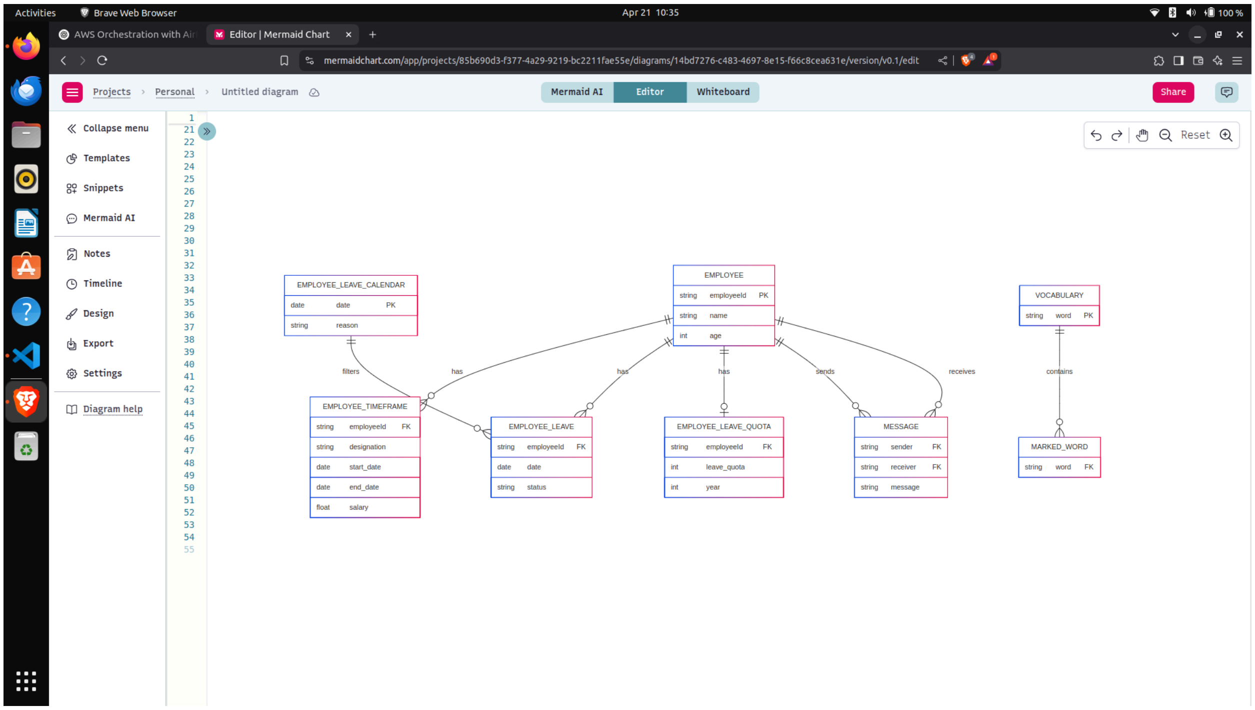
# PROJECT

## ER DIAGRAM




```
erDiagram

    EMPLOYEE {
        string employeeId PK
        string name
        int age
    }

    EMPLOYEE_TIMEFRAME {
        string employeeId FK
        string designation
        date start_date
        date end_date
        float salary
    }

    EMPLOYEE_LEAVE_QUOTA {
        string employeeId FK
        int leave_quota
        int year
    }
```

# PROJECT

```
EMPLOYEE_LEAVE_CALENDAR {
    date date PK
    string reason
}

EMPLOYEE_LEAVE {
    string employeeId FK
    date date
    string status
}

MESSAGE {
    string sender FK
    string receiver FK
    string message
}

VOCABULARY {
    string word PK
}

MARKED_WORD {
    string word FK
}

EMPLOYEE ||--o{ EMPLOYEE_TIMEFRAME : has
EMPLOYEE ||--o{ EMPLOYEE_LEAVE : has
EMPLOYEE ||--o| EMPLOYEE_LEAVE_QUOTA : has
EMPLOYEE_LEAVE_CALENDAR ||--o{ EMPLOYEE_LEAVE : filters
EMPLOYEE ||--o{ MESSAGE : sends
EMPLOYEE ||--o{ MESSAGE : receives
VOCABULARY ||--o{ MARKED_WORD : contains
```

# PROJECT

STEP 1 CODE

```
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

# Spark + Glue setup
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session

# Config
today = datetime.utcnow().strftime("%Y-%m-%d")
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_pr
ocessed_{today}.csv"
silver_output_path =
f"s3://{bucket}/poc-bootcamp-group1-silver/employee_data/"

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 and save raw files to processed zone
raw_paths = [f"s3://{bucket}/{key}" for key in csv_files]
```

# PROJECT

```
df_raw = spark.read.option("header", "true").csv(raw_paths)

# Write merged raw as CSV into processed folder (coalesced to 1 file)
df_raw.coalesce(1).write.mode("overwrite").option("header",
"true").csv(processed_output_path)
print(f"📦 Raw merged CSV copied 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: Read processed file back for transformation
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 silver zone (partitioned by folder, NOT ingestion
column)
df_cleaned.write.mode("append").parquet(silver_output_path)
print(f"✅ Processed & clean data written to: {silver_output_path}")

print("🎉 Glue job completed successfully.")
```

# PROJECT

STEP 2
PYSPARK CODE

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_unixtime, to_date, when,
row_number, lit, min as min_
from pyspark.sql.window import Window
import os
import shutil

# === Spark Session ===
spark = SparkSession.builder.appName("SCD2 Incremental").getOrCreate()
spark.sparkContext.setLogLevel("ERROR")

# === Folder paths ===
RAW_DIR = '/home/himanshu/Learning/bootcamp_project/QUS_02/data/raw'
STAGING_DIR =
'/home/himanshu/Learning/bootcamp_project/QUS_02/data/processed'
SILVER_PATH =
'/home/himanshu/Learning/bootcamp_project/QUS_02/data/silver/employees_scd
2.parquet'
PROCESSED_RAW_DIR =
'/home/himanshu/Learning/bootcamp_project/QUS_02/data/processed_raw'

os.makedirs(STAGING_DIR, exist_ok=True)

# === Step 1: Read raw file ===
raw_files = sorted([f for f in os.listdir(RAW_DIR) if f.endswith('.csv')])
if not raw_files:
   print("📭 No raw file to process. Exiting.")
   exit()

file_name = raw_files[0]
raw_path = os.path.join(RAW_DIR, file_name)
print(f"\n📥 Processing file: {file_name}")

# === Step 2: Load and preprocess new file ===
df = spark.read.option("header", True).csv(raw_path)
df = df.withColumn("start_date",
to_date(from_unixtime(col("start_date").cast("long"))))
```

# PROJECT

```
df = df.withColumn("end_date",
to_date(from_unixtime(col("end_date").cast("long"))))
df = df.withColumn("salary", col("salary").cast("double"))
print("\n📄 Raw DataFrame after timestamp conversion:")
df.show()

# === Step 3: Deduplicate within file ===
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")
print("\n🧹 After deduplication within raw data:")
df.show()

# === Step 4: Mark ACTIVE based on latest salary/null end ===
df = df.withColumn("status", when(col("end_date").isNull(),
"ACTIVE").otherwise("INACTIVE"))
print("\n✅ After status marking (ACTIVE/INACTIVE):")
df.show()

# === Step 5: Save processed version to staging
staging_path = os.path.join(STAGING_DIR, file_name.replace(".csv",
".parquet"))
df.write.mode("overwrite").parquet(staging_path)
print(f"\n💾 Saved processed file to staging at: {staging_path}")

# === Step 6: IF silver does not exist — First-time load
if not os.path.exists(SILVER_PATH):
    print("🆕 First-time load — writing data directly to silver.")
    df.write.mode("overwrite").parquet(SILVER_PATH)
    print("\n🪙 Silver Table (Initial Load):")
    df.show()
else:
    # === Step 7: SCD2 CONTINUATION LOGIC
    print("\n🔁 Silver table exists — continuing with SCD2 logic.")
    silver_df = spark.read.parquet(SILVER_PATH)
    processed_df = spark.read.parquet(staging_path)

    print("\n📁 Silver Table:")
    silver_df.show()
```

# PROJECT

```
    print("\n📁 Processed DataFrame:")
    processed_df.show()

    active_df = silver_df.filter(col("status") == "ACTIVE")
    inactive_df = silver_df.filter(col("status") == "INACTIVE")
    print("\n🟢 ACTIVE Records:")
    active_df.show()
    print("\n🔴 INACTIVE Records:")
    inactive_df.show()

    # === Step 8: Get min start_date from new incoming rows
    continuity_dates =
processed_df.groupBy("emp_id").agg(min_("start_date").alias("new_start_dat
e"))
    print("\n📆 Continuity Dates:")
    continuity_dates.show()

    # === Step 9: Update previous ACTIVE rows to INACTIVE
    updated_old = active_df.alias("old") \
        .join(continuity_dates.alias("new"), "emp_id") \
        .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")
    print("\n✏️ Updated old ACTIVE rows:")
    updated_old.show()

    # === Step 10: Keep untouched active rows
    untouched_active = active_df.join(continuity_dates.select("emp_id"),
"emp_id", "left_anti")
    print("\n✅ Unchanged ACTIVE rows:")
    untouched_active.show()

    # === Step 11: Get truly new employees
    new_emps = processed_df.join(silver_df.select("emp_id").distinct(),
"emp_id", "left_anti")
    print("\n🆕 Truly new employees (not seen before):")
    new_emps.show()
```

# PROJECT

```
    # === Step 12: Merge final result
    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)
    print("\n📊 Final merged Silver Table to write:")
    final_df.show()

    # === Step 13: Write final output to silver
    final_df.write.mode("overwrite").parquet(SILVER_PATH)
    print("\n💽 Final data written to Silver!")


# === Step 14: Cleanup: Move raw to processed_raw, delete staging
new_raw_path = os.path.join(PROCESSED_RAW_DIR, file_name)
shutil.move(raw_path, new_raw_path)
print(f"\n📦 Moved raw file to archive: {new_raw_path}")

shutil.rmtree(staging_path)
print(f"🧹 Cleaned staging folder: {staging_path}")

print(f"\n✅ Done. Silver updated and raw file archived: {file_name}")
```

# PROJECT

GLUE JOB

```
from awsglue.utils import getResolvedOptions
from awsglue.context import GlueContext
from pyspark.context import SparkContext
from pyspark.sql import SparkSession, functions as F
from pyspark.sql.window import Window
from py4j.protocol import Py4JJavaError
from pyspark.sql.types import StructType, StructField, StringType,
DateType, DoubleType
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_unixtime, to_date, when
import pyspark.sql.functions as F
import sys
import boto3
import re

# === Glue Setup ===
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session

# === Define Schema Manually ===
manual_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)
])

# === S3 Paths ===
RAW_PATH =
"s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp_timefram
e_data_qus2/raw/"
STAGING_PATH =
"s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp_timefram
e_data_qus2/processed/"
```

# PROJECT

```
PROCESSED_RAW_PATH =
"s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-bronze/emp_timefram
e_data_qus2/processed_raw/"
SILVER_PATH =
"s3://poc-bootcamp-capstone-group1/poc-bootcamp-group1-silver/emp-timefram
e-data/"

# === Step 1: Read raw CSV data
print(f"\n📥 Reading raw data from: {RAW_PATH}")
df = spark.read.option("header", True).csv(RAW_PATH)

# === Step 2: Cast salary and convert timestamps
df = df.withColumn("salary", F.col("salary").cast("double"))
df = df.withColumn("start_date",
F.to_date(F.from_unixtime(F.col("start_date").cast("long"))))
df = df.withColumn("end_date",
F.to_date(F.from_unixtime(F.col("end_date").cast("long"))))
print("\n📄 Raw DataFrame after cleaning and casting:")
df.show()

# === Step 3: Deduplicate
w1 = Window.partitionBy("emp_id", "start_date",
"end_date").orderBy(F.col("salary").desc())
df = df.withColumn("rn1", F.row_number().over(w1)).filter(F.col("rn1") ==
1).drop("rn1")

# === Step 4: Mark ACTIVE
df = df.withColumn("status", when(col("end_date").isNull(),
"ACTIVE").otherwise("INACTIVE"))
print("\n✅ After status marking (ACTIVE/INACTIVE):")

# === Step 5: Save to staging
df.write.mode("overwrite").parquet(STAGING_PATH)
print(f"\n💾 Saved processed file to staging at: {STAGING_PATH}")

# === Step 6: Check if silver path exists
s3 = boto3.client("s3")
bucket = "poc-bootcamp-capstone-group1"
key_prefix = "poc-bootcamp-group1-silver/emp-timeframe-data/"
response = s3.list_objects_v2(Bucket=bucket, Prefix=key_prefix)
```

# PROJECT

```
silver_exists = "Contents" in response

if not silver_exists:
      print("🆕 First-time load — writing to silver.")
      df.write.mode("overwrite").parquet(SILVER_PATH)
else:
      print("\n🔁 Silver table exists — continuing with SCD2 logic.")

      try:
      silver_df = spark.read.schema(manual_schema).parquet(SILVER_PATH)
      except Py4JJavaError as e:
      if "UNABLE_TO_INFER_SCHEMA" in str(e):
            print("⚠️ Silver path is empty. Creating empty DataFrame.")
            silver_df = spark.createDataFrame([], schema=manual_schema)
      else:
            raise

      try:
      processed_df =
spark.read.schema(manual_schema).parquet(STAGING_PATH)
      except Py4JJavaError as e:
      if "UNABLE_TO_INFER_SCHEMA" in str(e):
            print("⚠️ Staging path is empty. Creating empty DataFrame.")
            processed_df = spark.createDataFrame([], schema=manual_schema)
      else:
            raise

      active_df = silver_df.filter(F.col("status") == "ACTIVE")
      inactive_df = silver_df.filter(F.col("status") == "INACTIVE")

      continuity_dates =
processed_df.groupBy("emp_id").agg(F.min("start_date").alias("new_start_da
te"))

      updated_old = active_df.alias("old") \
      .join(continuity_dates.alias("new"), "emp_id") \
      .filter(F.col("new.new_start_date") >= F.col("old.start_date")) \
      .withColumn("end_date", F.col("new.new_start_date")) \
      .withColumn("status", F.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)

    final_df.write.mode("overwrite").parquet(SILVER_PATH)
    print("\n💽 Final data written to Silver!")

# === Step 7: Move raw → processed_raw, delete only files from processed
def move_raw_files_to_archive(src_bucket, src_prefix, dst_prefix):
    print(f"\n📦 Moving raw files from s3://{src_bucket}/{src_prefix} to
s3://{src_bucket}/{dst_prefix}")
    response = s3.list_objects_v2(Bucket=src_bucket, Prefix=src_prefix)
    if "Contents" in response:
    for obj in response['Contents']:
        key = obj['Key']
        if not key.endswith('/'):
            copy_source = {'Bucket': src_bucket, 'Key': key}
            new_key = key.replace(src_prefix, dst_prefix, 1)
            s3.copy_object(Bucket=src_bucket, CopySource=copy_source,
Key=new_key)
            s3.delete_object(Bucket=src_bucket, Key=key)
            print(f"  - Moved: {key} → {new_key}")
    else:
    print("⚠️ No files in raw folder.")

def delete_files_only(bucket, prefix):
```

# PROJECT

```
    print(f"\n🧹 Deleting files in s3://{bucket}/{prefix}")
    response = s3.list_objects_v2(Bucket=bucket, Prefix=prefix)
    if "Contents" in response:
    files = [obj['Key'] for obj in response['Contents'] if not 
obj['Key'].endswith('/')]
    if files:
        s3.delete_objects(Bucket=bucket, Delete={'Objects': [{'Key': 
k} for k in files]})
        for key in files:
            print(f"  - Deleted: {key}")
    else:
        print("⚠️ No file objects found.")
    else:
    print("⚠️ Nothing found under prefix.")

def extract_bucket_prefix(s3_path):
    match = re.match(r's3://([^/]+)/(.+)', s3_path)
    return match.groups() if match else (None, None)

raw_bucket, raw_prefix = extract_bucket_prefix(RAW_PATH)
staging_bucket, staging_prefix = extract_bucket_prefix(STAGING_PATH)
_, processed_raw_prefix = extract_bucket_prefix(PROCESSED_RAW_PATH)

if raw_bucket:
    move_raw_files_to_archive(raw_bucket, raw_prefix, 
processed_raw_prefix)

if staging_bucket:
    delete_files_only(staging_bucket, staging_prefix)

print("\n✅ Glue Job Finished Successfully!")
```

# PROJECT

# Start Zookeeper
cd ~/kafka
bin/zookeeper-server-start.sh config/zookeeper.properties

# In a new terminal session, start Kafka
cd ~/kafka
bin/kafka-server-start.sh config/server.properties

# Create a topic named 'test-topic'
bin/kafka-topics.sh --create --topic test-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

# List available topics
bin/kafka-topics.sh --list --bootstrap-server localhost:9092

# Start a producer to send messages
bin/kafka-console-producer.sh --topic test-topic --bootstrap-server localhost:9092

# In a new terminal session, start a consumer to receive messages
bin/kafka-console-consumer.sh --topic test-topic --from-beginning --bootstrap-server localhost:9092

Producer running
(airflow_venv) ubuntu@ip-172-31-6-153:~$ python /home/ubuntu/kafka_codes/KAFKA_QUS/producer.py

Consumer running
(airflow_venv) ubuntu@ip-172-31-6-153:~$ ~/spark/bin/spark-submit   --master local[*]   --jars /home/ubuntu/postgresql-42.7.5.jar   --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.0,org.apache.kafka:kafka-clients:3.3.2 /home/ubuntu/kafka_codes/KAFKA_QUS/consumer_spark.py

Cooldown running
(airflow_venv) ubuntu@ip-172-31-6-153:~$ python /home/ubuntu/kafka_codes/KAFKA_QUS/cooldown_strike_level_spark.py

# PROJECT

```
pkill -f producer.py
pkill -f consumer_spark.py
pkill -f cooldown_strike_level_spark.py


rm -f /tmp/80_leave_report_*.flag
rm -f /tmp/leave_quota_done_*.flag
```

# PROJECT

[Spark DataFrame] → [Pandas DataFrame] → [CSV Buffer using StringIO] → [PostgreSQL COPY]

↑ ↑ ↑ ↑

Massive data processing | Converts for COPY (Pandas supports it) | Acts like a file without disk I/O | Fast bulk insert

## 🔍 Why Use Spark → Pandas → CSV → PostgreSQL?

### 🔹 Reason: Spark can't `INSERT` into PostgreSQL efficiently

Spark can write to PostgreSQL via `.jdbc()` — but **only in full overwrite or append mode**, not row-by-row control. You needed:

1. **Precise DELETE** of old rows (via `psycopg2`)
2. **Efficient INSERT** of new rows — but PostgreSQL doesn't support high-performance inserts directly via `.jdbc()` for bulk rows.

That's why you:

- Used **Spark** for all data processing and transformation
- Then **converted Spark → Pandas → CSV buffer**
- And used **PostgreSQL's** `COPY FROM STDIN` for fast bulk insert

---

## 🔧 Why `COPY FROM STDIN` is better?

| Method | Speed | Control | Use Case |
|---|---|---|---|
| `.jdbc().write()` | Slower | None | Overwrite or append entire table |
| `INSERT INTO ... VALUES` | Very slow | Yes | Fine-grained control, but slow |
| `COPY FROM STDIN` (via psycopg2) | Very fast | Yes | Bulk insert millions of rows efficiently |

`COPY FROM STDIN` works like loading from a file — but uses an in-memory `StringIO` buffer.

# PROJECT

Great question. Here's a clear and practical comparison to explain **why AWS Glue is more suitable than EMR** for your project.

---

## ✅ Why You Used AWS Glue Instead of EMR

| Reason | AWS Glue Advantage | Why EMR Falls Short for Your Use Ca |
|---|---|---|
| **1. Fully managed** | No need to provision or manage clusters — serverless Spark | You need to configure and maintain EC2-based Spark/Hadoop clusters |
| **2. Cost-efficient for ETL** | Pay only per job execution + short duration = low cost | EMR runs on EC2; costs accumulate even if the cluster is idle |
| **3. Optimized for ETL** | Built-in connectors for S3, PostgreSQL, Redshift, JDBC, etc. | EMR supports these, but setup and integration take more work |
| **4. Easy to maintain** | Auto-handles scaling, retries, metadata (via Glue Data Catalog) | You handle all failure recovery, scaling, and setup |
| **5. Native integration** | Easily integrates with boto3, Athena, Lake Formation, and more | Integration is possible, but not as seamless |
| **6. Trigger-based execution** | Supports event, schedule, and dependency-based job triggers | You would need to set this up manually via Step Functions or Airflow |
| **7. Ideal for small-to-medium pipelines** | Perfect for jobs that run in minutes (like yours) | EMR is more suitable for long-running or massively parallel jobs |

# PROJECT

## 🔧 When EMR Would Be a Better Fit

Use EMR **instead of Glue** if:

- You need **custom big data engines** (like Hive, Presto, Flink, or HBase)
- You run **long-running Spark streaming** or batch jobs over TBs of data
- You require **fine-grained control** over hardware, memory, and cluster scaling
- You're doing **real-time processing** or **ML model training** at scale

## 🧩 Your Project Profile

| Feature | Characteristic |
|---|---|
| Data size | Small to moderate (CSV files, PostgreSQL writes) |
| Job frequency | Daily or event-based |
| Latency requirement | Low (completes in minutes) |
| User management effort | You want minimal overhead |

✅ **Glue is a perfect match** for this scenario.

# PROJECT

## ✅ What Already Makes Your Pipeline Fault-Tolerant

| Feature | Status | Why it helps |
|---|---|---|
| **Try–Except blocks** | ✅ Present | Catches runtime errors and logs stack trace for debugging |
| **Glue Job Commit** | ✅ Present | Ensures the Glue job marks itself complete only when successful |
| **Deduplication & null checks** | ✅ Present | Prevents data corruption from duplicates or nulls |
| **S3 and PostgreSQL separation** | ✅ Good design | Avoids tight coupling of storage and compute |
| **JDBC write modes (`overwrite`)** | ⚠️ Used safely | Works for single-target writes, but can overwrite all data if misused |
| **S3 report duplication check** | ✅ Present | Avoids redundant writes and reruns |

## ⚠️ Where Fault Tolerance Is Missing / Needs Improvement

| Missing Feature | Impact |
|---|---|
| ❌ **Retries on failure** | If a job fails due to a transient issue (e.g., network), it won't retry |
| ❌ **Atomic PostgreSQL writes** | `.jdbc()` writes are not transaction-controlled — can result in partial writes |
| ❌ **Idempotency** for data writes | Some `.write()` operations can duplicate or overwrite previous good data |
| ❌ **Audit logs / checkpoints** | There's no tracking of what data was ingested, when, or by which run |
| ❌ **Upserts/Merge for final tables** | Current overwrite/append logic may lead to duplicate entries |
| ⚠️ **Failure notifications** | There's no alert system for failed jobs (like SNS, email, or CloudWatch Alarms) |

# PROJECT

## 🛠️ Recommendations to Make Your Pipeline Fully Fault-Tolerant

### 1. Enable Glue Job Retries

- Set `MaxRetries` in the job configuration or use workflows to auto-retry failed steps.

### 2. Use Atomic Writes or Staging Tables

- Instead of direct `.overwrite()` on final tables, write to a temp table and use SQL `MERGE` to update.
- Commit PostgreSQL transactions manually (via `psycopg2`) for large writes.

### 3. Idempotent Design

- Include a `run_id` or `ingest_batch_id` to avoid duplicate processing.
- Use `.dropDuplicates()` or `row_number()` in all staging logic.

### 4. Logging & Audit

- Write logs or checkpoints to a Glue table or S3 log file with `job_run_id`, `timestamp`, and status.

### 5. Monitoring & Alerts

- Add:
  - CloudWatch Alarms for failed Glue jobs
  - SNS alerts on exceptions
  - Logging to S3 for audit trails

# PROJECT

## ✅ 1. Where You Use Optimizations (Well Done)

| Optimization | Applied? | Notes |
| --- | --- | --- |
| **Broadcast join for small tables** | ✅ | You used `broadcast()` when joining `leave_df` and `holiday_df`, which is ideal |
| **Filtering early (predicate pushdown)** | ✅ | You filter `status`, `year`, and `date` before joins or writes (good practice) |
| `.dropDuplicates()` **and** `.distinct()` | ✅ | Reduces write I/O and ensures data sanity |
| **Read with Schema** | ✅ | You define schema for CSVs (prevents full scan and incorrect type inference) |
| **Use of** `coalesce(1)` **for small writes** | ✅ | Ensures single output file during S3 saves — useful for small datasets |
| **Working day calendar logic** | ✅ | Smart use of `dayofweek` and anti-joins to filter weekends/holidays efficiently |
| **Avoiding unnecessary columns** | ✅ | You often `.select()` only what you need before aggregation or writing |

# PROJECT

## ❌ 2. Where Optimizations Are Missing or Skipped

| Skipped Optimization | Impact |
|---|---|
| **Partitioning when reading/writing large data** | Reads entire tables instead of using filters like `pushDownPredicate` or `partitionColumn` with `.jdbc()` |
| **No caching for reused DataFrames** | Expensive joins or filters (like on leave data) are repeated across steps |
| **No `.persist()` before multiple actions** | E.g., multiple `.count()` and `.write()` on same `DataFrame` triggers recomputation |
| **.jdbc() writes are row-by-row** | JDBC writes can be slow if not batched or written via `COPY` using `psycopg2` for large data |
| **No `bucketBy()` or `partitionBy()` on writes** | Output tables aren't optimized for downstream queries |
| **Data catalog partitioning** (Glue Tables) | Not leveraged for faster S3 scans via Athena or Glue crawlers |
| **No compression when writing to S3** | You're writing raw CSVs — compressed formats like Parquet/ORC would save space and I/O |

# PROJECT

## 🛠️ 3. How to Improve Optimization

- **For Reads (Especially JDBC)**
- Use:

```python
.option("fetchsize", "10000")
.option("partitionColumn", "emp_id")
.option("lowerBound", "1")
.option("upperBound", "999999")
.option("numPartitions", "10")
```

This parallelizes JDBC reads.

- **For Writes**
- Avoid `.jdbc(mode="overwrite")` for big tables. Instead:
  - Use **staging table** → then `MERGE INTO`
  - Use `COPY FROM STDIN` via `psycopg2` for **bulk inserts**
- **DataFrame Caching**
- Use `.cache()` or `.persist()` on intermediate DataFrames accessed more than once (like filtered `leave_df`)
- **Partitioning / Bucketing**
- Use `.repartition("year")` or `.partitionBy("year")` when writing large S3 datasets
- **Format Optimization**
- Write to **Parquet** instead of CSV for S3:

```python
df.write.option("compression", "snappy").parquet(path)
```

- **Reduce Actions on Large DataFrames**
- Avoid calling `.count()`, `.collect()` unless necessary
- **Use Glue Job Bookmarks (if doing incremental loads)**
- This allows Glue to **only pick new data** since last successful job

## 🧠 4. Optimization Strategy for Your Use Case

| Pipeline Area | Optimization Goal | Recommendation |
|---|---|---|
| Leave Analysis (8%, 80%) | Low latency + efficient filters | Use caching, pushdown filters, avoid recompute |
| S3 writes | Reduce cost + increase scan speed | Use Parquet + `partitionBy("year")` |
| PostgreSQL I/O | Improve throughput | Use batching or `COPY` with psycopg2 |
| Report generation | Avoid recomputation for multiple employees | Cache filtered and joined DataFrame before `.collect()` |
| SCD2 updates | Efficient incremental updates | Avoid full overwrite — use `MERGE INTO` |

# PROJECT

## 📈 Final Scorecard: Optimization Maturity

| Area | Score (✅=Good, ⚠️=Some, ❌=Needs work) |
|---|---|
| Read efficiency | ✅ |
| Join strategies | ✅ |
| Memory management | ⚠️ |
| Write performance | ❌ |
| File formats | ⚠️ |
| Incremental logic | ⚠️ |
| Reuse / caching | ❌ |
| Compression | ❌ |

## 🔍 Bonus: Smart One-Liners (Use These in Answers)

- "We used `left_anti` joins to exclude holidays dynamically during working day filtering."
- "To prevent duplicate reports on S3, we compare keys before generating new TXT files."
- "For high-volume writes, I'd prefer `COPY FROM STDIN` using `psycopg2` over `.jdbc()`."
- "We used `row_number()` and `Window.partitionBy()` for deduplicating leave records."
- "I tuned `.jdbc()` reads using `partitionColumn`, `lowerBound`, and `numPartitions`."

# PROJECT

## 🔍 What is Predicate Pushdown?

**Predicate pushdown** is an optimization technique where **filters (`WHERE` clauses)** are **pushed down to the data source level**, so **only necessary rows** are loaded into Spark from a data source like PostgreSQL, S3 (Parquet), or JDBC.

---

## 📌 Why It's Important

Without predicate pushdown:

- Spark loads **entire table/data file** into memory, then applies filters.
- ❌ Slower, more memory-heavy

With predicate pushdown:

- Only filtered data is fetched from the **source**.
- ✅ Faster reads, reduced I/O, more efficient memory use

# PROJECT

## ✅ How You Can Use It

🔹 **In JDBC Reads**

```python
spark.read.jdbc(
    url=jdbc_url,
    table="(SELECT * FROM employee_db WHERE year = 2024) AS sub",
    properties=props
)
```

or better:

```python
df = spark.read.jdbc(
    url=jdbc_url,
    table="employee_db",
    predicates=["year = 2024"],
    properties=db_properties
)
```

- Here, Spark sends `WHERE year = 2024` **to PostgreSQL**, reducing load

🔹 **With Parquet / ORC on S3**

If you:

- Use **columnar formats** like Parquet or ORC
- Apply `df.filter(col("year") == 2024)` **early**
- Glue or Spark can **read only the relevant row groups**

🔹 **With CSV/JSON → ⚠️ Limited**

Predicate pushdown **does not work well** with plain CSV or JSON.

- Entire file still gets scanned.

# PROJECT

## 🧠 How It Applies to Your Project

In your pipeline:

- ✅ **Good**: You apply filters **early** (e.g., `status == 'ACTIVE'`, `date >= X`) before joins or writes
- ❌ **Missed**: You could improve JDBC reads like this:

**Before (less optimal):**

python Copy Edit

```python
df = spark.read.jdbc(url=jdbc_url, table="leave_data", properties=props)
df = df.filter(col("year") == CURRENT_YEAR)
```

**After (optimized):**

python Copy Edit

```python
df = spark.read.jdbc(
    url=jdbc_url,
    table="leave_data",
    predicates=[f"year = {CURRENT_YEAR}"],
    properties=props
)
```

# PROJECT

## 🧠 What is Adaptive Query Execution (AQE)?

**Adaptive Query Execution (AQE)** is a **runtime optimization feature** in Spark that **dynamically adjusts the execution plan** of a query **based on actual data statistics** gathered **during execution**.

✅ Introduced in **Spark 3.0+**

---

## ⚙️ Why Do We Need AQE?

Spark's Catalyst optimizer generates **a static execution plan before running**, based on **estimated statistics** (like table size, partition size, etc.).

But:

- Estimates can be wrong
- Joins can be inefficient
- Shuffles can be skewed

✅ **AQE fixes this** by adapting the plan **at runtime**, when actual data stats are known.

## 🚀 Key Features of AQE

| Feature | Description |
|---|---|
| **1. Dynamic Join Selection** | Switches join strategy at runtime (e.g., from sort-merge to broadcast) |
| **2. Skew Join Handling** | Detects skew and splits large partitions across reducers |
| **3. Dynamically Coalescing Shuffle Partitions** | Reduces number of output partitions based on actual shuffle size |
| **4. Dynamically Switch Partition Pruning** | Skips irrelevant partitions even during query runtime |

---

## ✅ How to Enable AQE

python Copy Edit

```python
spark.conf.set("spark.sql.adaptive.enabled", "true")
```

# PROJECT

## 🔁 `repartition()` vs `coalesce()` – Quick Summary

| Feature | `repartition()` | `coalesce()` |
|---|---|---|
| **Purpose** | Increases or decreases number of partitions | Primarily **reduces** number of partitions |
| **Shuffle** | ✅ Full **shuffle** | ⚠️ Avoids shuffle when reducing partitions |
| **Use Case** | When you want **even distribution** or increase partitions | When you want to **reduce partitions** efficiently |
| **Performance Impact** | Costly due to full data movement | Fast (no shuffle), but can create skew |
| **Typical Use** | Before joins, groupBy, wide shuffles | Before writing to disk (e.g., to S3/DB) |

### 💧 What is Watermarking in Spark?

**Watermarking** is a technique used in **Structured Streaming** to **handle late-arriving data** and define **event time-based windows** with a **tolerance period** for lateness.

## 🧠 Why Watermarking is Needed?

In streaming data:

- Data doesn't always arrive in order.
- Some records come **late** (due to network delays, retries, etc.)
- You need a way to **allow some lateness**, but also **garbage collect** old state.

✅ **Watermarking** balances this:

- Waits for **late events up to a threshold** (e.g., 10 minutes)
- Drops data that arrives **after that threshold**

# PROJECT