**Code:**

# === Import Required Modules === #

# Core PySpark modules for session management, schema definitions, and transformations

from pyspark.sql import SparkSession, DataFrame

from pyspark.sql.functions import col, to\_timestamp, date\_format, regexp\_like, broadcast

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

from pyspark import StorageLevel # To cache intermediate DataFrames

# General-purpose Python libraries

from dataclasses import dataclass

from typing import List, Dict, Optional

import datetime

import json

import os

# AWS and Airflow integrations

import boto3

from botocore.exceptions import ClientError

from airflow.models import Variable # For reading Airflow variables

# === Configuration Classes === #

@dataclass

class PipelineConfig:

"""

Stores configuration settings for the data pipeline.

Supports AWS/S3, local paths, JDBC configs, file inputs, AQE, and skew handling.

"""

aws\_access\_key: str = None

aws\_secret\_key: str = None

bucket\_name: str = "default-bucket"

local\_path: str = "/tmp/data"

s3\_directory: str = "sensor\_data/"

db\_credentials\_path: str = "/etc/db\_creds.json"

input\_files: List[str] = None

sensor\_patterns: List[str] = None

default\_start\_date: str = "2024-01-01"

lookback\_days: int = 30

jdbc\_fetch\_size: int = 10000

jdbc\_num\_partitions: int = 8

write\_mode: str = "overwrite"

use\_aqe: bool = True

enable\_skew\_handling: bool = True

class ConfigManager:

"""

Handles loading configuration from either Airflow variables or AWS Secrets Manager.

"""

@staticmethod

def load\_from\_airflow(var\_name: str = "sensor\_pipeline\_config") -> PipelineConfig:

config = Variable.get(var\_name, deserialize\_json=True)

return PipelineConfig(\*\*config)

@staticmethod

def load\_from\_aws\_secrets(secret\_name: str) -> PipelineConfig:

try:

client = boto3.client("secretsmanager")

response = client.get\_secret\_value(SecretId=secret\_name)

return PipelineConfig(\*\*json.loads(response["SecretString"]))

except ClientError as e:

raise ValueError(f"AWS Secrets Error: {str(e)}")

@classmethod

def load\_config(cls, source: str = "airflow", \*\*kwargs) -> PipelineConfig:

"""

Main config loader entrypoint.

:param source: 'airflow' or 'aws'

"""

if source == "airflow":

return cls.load\_from\_airflow(kwargs.get("var\_name"))

elif source == "aws":

return cls.load\_from\_aws\_secrets(kwargs.get("secret\_name"))

else:

raise ValueError("Invalid config source")

# === Schema Definitions === #

class PipelineSchemas:

"""

Defines PySpark schemas for input and output data.

"""

TAGS = StructType([

StructField("id", IntegerType(), False),

StructField("tagpath", StringType(), False),

StructField("description", StringType(), True),

StructField("unit", StringType(), True)

])

SENSOR\_RAW = StructType([

StructField("tagid", IntegerType(), False),

StructField("t\_stamp", LongType(), False),

StructField("value", DoubleType(), False),

StructField("dataintegrity", IntegerType(), False)

])

OUTPUT = StructType([

StructField("datetime", TimestampType(), False),

StructField("sensor\_value", DoubleType(), True)

])

# === Data Loading Logic === #

class DataLoader:

"""

Loads existing data from local or S3 paths.

"""

def \_\_init\_\_(self, spark: SparkSession, config: PipelineConfig):

self.spark = spark

self.config = config

def try\_load(self, paths: List[str], schema: StructType) -> Optional[DataFrame]:

"""

Tries loading data from multiple locations (local/S3).

"""

for path in paths:

try:

df = self.spark.read.schema(schema).parquet(path)

if df.rdd.getNumPartitions() > 1:

df = df.coalesce(1)

return df

except Exception:

continue

return None

def load\_existing\_data(self) -> Dict[str, DataFrame]:

"""

Loads input files and caches them for reuse.

"""

data = {}

for filename in self.config.input\_files:

paths = [

os.path.join(self.config.local\_path, filename),

f"s3a://{self.config.bucket\_name}/{self.config.s3\_directory}{filename}"

]

if df := self.try\_load(paths, PipelineSchemas.OUTPUT):

data[filename] = df.persist(StorageLevel.MEMORY\_AND\_DISK)

return data

# === JDBC Database Access === #

class DatabaseManager:

"""

Handles JDBC connections and SQL queries to a PostgreSQL-compatible source.

"""

def \_\_init\_\_(self, spark: SparkSession, config: PipelineConfig):

self.spark = spark

self.config = config

with open(config.db\_credentials\_path) as f:

self.db\_config = json.load(f)["YourDB"]

def get\_connection\_params(self) -> Dict:

"""

Builds JDBC options dictionary.

"""

return {

"url": f"jdbc:postgresql://{self.db\_config['host']}/{self.db\_config['dbname']}",

"user": self.db\_config["user"],

"password": self.db\_config["password"],

"fetchSize": str(self.config.jdbc\_fetch\_size),

"numPartitions": str(self.config.jdbc\_num\_partitions),

"partitionColumn": "tagid",

"lowerBound": "1",

"upperBound": "100000"

}

def query\_table(self, query: str, schema: StructType = None) -> DataFrame:

"""

Executes a SQL query and returns the result as a DataFrame.

"""

reader = self.spark.read.format("jdbc").options(\*\*self.get\_connection\_params())

if schema:

reader = reader.schema(schema)

return reader.option("query", query).load()

# === Data Transformation Logic === #

class DataProcessor:

"""

Transforms raw sensor readings to structured output format.

"""

def \_\_init\_\_(self, config: PipelineConfig):

self.config = config

def filter\_tables(self, tables: List[str], cutoff\_date: datetime.datetime) -> List[str]:

"""

Filters DB table names based on cutoff year and month in their name.

Expected table naming convention: `table\_YYYY\_MM`

"""

return [t for t in tables

if (parts := t.split('\_')) and len(parts) > 2 and parts[-2].isdigit() and parts[-1].isdigit()

and (int(parts[-2]) > cutoff\_date.year or (int(parts[-2]) == cutoff\_date.year and int(parts[-1]) >= cutoff\_date.month))]

def process\_sensor\_data(self, df: DataFrame, tags\_df: DataFrame) -> DataFrame:

"""

Joins raw sensor data with tag metadata, filters by integrity, converts timestamps.

"""

df = self.\_validate\_schema(df, PipelineSchemas.SENSOR\_RAW)

tags\_df = self.\_validate\_schema(tags\_df, PipelineSchemas.TAGS)

processed = (

df.join(broadcast(tags\_df), df.tagid == tags\_df.id, "left")

.filter(col("dataintegrity") != 0)

.withColumn("datetime", to\_timestamp(col("t\_stamp") / 1000))

)

return self.\_validate\_schema(processed, PipelineSchemas.OUTPUT)

def \_validate\_schema(self, df: DataFrame, expected\_schema: StructType) -> DataFrame:

"""

Casts each column to expected type and ensures required columns exist.

"""

for field in expected\_schema:

if field.name in df.columns:

df = df.withColumn(field.name, col(field.name).cast(field.dataType))

elif not field.nullable:

raise ValueError(f"Missing required field: {field.name}")

return df.select([field.name for field in expected\_schema])

# === S3 Output Writer === #

class S3Writer:

"""

Writes processed DataFrame to S3 in optimized form.

"""

def \_\_init\_\_(self, spark: SparkSession, config: PipelineConfig):

self.spark = spark

self.config = config

def write\_optimized(self, df: DataFrame, path: str) -> None:

"""

Writes partitioned parquet files with record limits for optimized reads.

"""

(df.repartition(max(1, df.count() // 100000))

.write

.mode(self.config.write\_mode)

.option("maxRecordsPerFile", 100000)

.parquet(f"s3a://{self.config.bucket\_name}/{path}"))

# === Orchestration Logic === #

class SensorDataPipeline:

"""

Main orchestration class combining all components.

"""

def \_\_init\_\_(self, config: PipelineConfig):

self.config = config

self.spark = self.\_init\_spark()

self.loader = DataLoader(self.spark, config)

self.db\_manager = DatabaseManager(self.spark, config)

self.processor = DataProcessor(config)

self.writer = S3Writer(self.spark, config)

def \_init\_spark(self) -> SparkSession:

"""

Initializes Spark with AQE, skew handling, and AWS credentials (if provided).

"""

builder = (SparkSession.builder

.appName("SensorDataPipeline")

.config("spark.sql.adaptive.enabled", self.config.use\_aqe)

.config("spark.sql.adaptive.skewJoin.enabled", self.config.enable\_skew\_handling)

.config("spark.dynamicAllocation.enabled", "true"))

if self.config.aws\_access\_key:

builder = (builder

.config("spark.hadoop.fs.s3a.access.key", self.config.aws\_access\_key)

.config("spark.hadoop.fs.s3a.secret.key", self.config.aws\_secret\_key))

return builder.getOrCreate()

def run(self) -> None:

"""

Main entry point to execute the pipeline.

"""

try:

existing\_data = self.loader.load\_existing\_data()

final\_df = self.\_process\_new\_data(existing\_data)

self.\_write\_outputs(final\_df, existing\_data)

finally:

self.\_cleanup(existing\_data)

def \_process\_new\_data(self, existing\_data: Dict) -> DataFrame:

"""

Reads tag metadata and filters relevant tables and data rows.

"""

tags\_df = self.db\_manager.query\_table(

"SELECT id, tagpath, description, unit FROM your\_tags\_table",

PipelineSchemas.TAGS

)

# Retrieve all available public table names

tables = [row.table\_name for row in

self.db\_manager.query\_table(

"SELECT table\_name FROM information\_schema.tables WHERE table\_schema = 'public'").collect()]

# Compute cutoff based on max datetime of existing files

cutoff\_date = self.\_get\_cutoff\_date(existing\_data)

filtered\_tables = self.processor.filter\_tables(tables, cutoff\_date)

# Get list of tag IDs matching the desired patterns

ids\_str = self.\_get\_matching\_ids(tags\_df)

# Read data from relevant tables

query = f"""

SELECT \* FROM {{0}}

WHERE tagid IN ({ids\_str})

AND t\_stamp >= {int(cutoff\_date.timestamp() \* 1000)}

"""

dfs = [self.db\_manager.query\_table(query.format(table), PipelineSchemas.SENSOR\_RAW) for table in filtered\_tables]

return self.processor.process\_sensor\_data(dfs[0].unionByName(\*dfs[1:]), tags\_df)

def \_write\_outputs(self, final\_df: DataFrame, existing\_data: Dict) -> None:

"""

Writes final outputs partitioned by `tagpath`.

"""

for tagpath in final\_df.select("tagpath").distinct().collect():

tagpath = tagpath[0]

col\_name = tagpath.replace("/", "\_")

# Rename sensor value column to unique tag name

sensor\_df = (final\_df

.filter(col("tagpath") == tagpath)

.drop("tagpath", "tagid")

.withColumnRenamed(

[c for c in final\_df.columns if c not in ["datetime", "tagpath", "tagid"]][0],

col\_name

))

# Append existing if applicable

if self.config.write\_mode == "append":

if existing\_df := existing\_data.get(f"{col\_name}.parquet"):

sensor\_df = existing\_df.union(sensor\_df)

output\_path = f"{self.config.s3\_directory}{col\_name}.parquet"

self.writer.write\_optimized(sensor\_df.dropDuplicates(["datetime"]).orderBy("datetime"), output\_path)

def \_get\_cutoff\_date(self, existing\_data: Dict) -> datetime.datetime:

"""

Computes the earliest datetime among all cached files,

or uses default if no files exist.

"""

if not existing\_data:

return datetime.datetime.strptime(self.config.default\_start\_date, "%Y-%m-%d")

max\_dates = [df.agg({"datetime": "max"}).collect()[0][0] for df in existing\_data.values()]

return min(max(max\_dates), datetime.datetime.now() - datetime.timedelta(days=self.config.lookback\_days))

def \_get\_matching\_ids(self, tags\_df: DataFrame) -> str:

"""

Filters tag IDs using regex patterns defined in config.

"""

condition = " OR ".join(f"tagpath ~ '{pattern}'" for pattern in self.config.sensor\_patterns)

result = self.db\_manager.query\_table(f"SELECT id FROM your\_tags\_table WHERE {condition}")

return ",".join(str(row.id) for row in result.collect())

def \_cleanup(self, data: Dict) -> None:

"""

Unpersists any cached data and clears Spark's internal cache.

"""

for df in data.values():

df.unpersist()

self.spark.catalog.clearCache()

# === Pipeline Entrypoint === #

def main(config\_source: str = "airflow", \*\*kwargs):

"""

Entrypoint for executing the pipeline with configurable source (airflow/aws).

"""

config = ConfigManager.load\_config(config\_source, \*\*kwargs)

pipeline = SensorDataPipeline(config)

pipeline.run()

if \_\_name\_\_ == "\_\_main\_\_":

main("airflow", var\_name="prod\_sensor\_config")