AWS Project 2

Febin P George

Objective:

To convert a csv file into an iceberg table and test if it works

Automate the entire process using lambda and airflow

Using kinesis datastream and EMR, get the live stream data and upload it into Snowflake and S3

Create tables in Athena and query the data.

1. Create the files using the python script
2. Upload the files into s3
3. Run a crawler on a file
4. Run a glue job on the crawled table to convert it into an iceberg table

import sys

from awsglue.context import GlueContext

from awsglue.utils import getResolvedOptions

from pyspark.context import SparkContext

from pyspark.sql import SparkSession

from awsglue.job import Job

# Initialize SparkSession with Iceberg support

spark = SparkSession.builder \

.appName("Glue Iceberg Table CDC") \ .config("spark.sql.catalog.glue\_catalog", "org.apache.iceberg.spark.SparkCatalog") \ .config("spark.sql.catalog.glue\_catalog.catalog-impl", "org.apache.iceberg.aws.glue.GlueCatalog") \ .config("spark.sql.catalog.glue\_catalog.warehouse", "s3://feb-project2/ad\_revenue\_iceberg/") \ .config("spark.sql.catalog.glue\_catalog.io-impl", "org.apache.iceberg.aws.s3.S3FileIO") \

.getOrCreate()

# Get job arguments

args = getResolvedOptions(sys.argv, ['JOB\_NAME'])

# Initialize Glue Context

sc = SparkContext.getOrCreate()

glueContext = GlueContext(sc)

spark = glueContext.spark\_session

job = Job(glueContext)

job.init(args['JOB\_NAME'], args)

# Optional: Create database in Glue Catalog if it doesn’t exist spark.sql("CREATE DATABASE IF NOT EXISTS glue\_catalog.ad\_revenue\_iceberg")

# Read data from Glue Data Catalog (output of crawler)

demographics\_df = glueContext.create\_dynamic\_frame.from\_catalog( database="feb-project2db",

table\_name="ad\_revenue"

).toDF()

# Optional: Create Iceberg table (only if it doesn't exist)

spark.sql(""" CREATE TABLE IF NOT EXISTS glue\_catalog.ad\_revenue\_iceberg.ad\_revenue (

channel\_id STRING,

channel\_name STRING,

date STRING,

ad\_revenue DOUBLE

)

USING ICEBERG

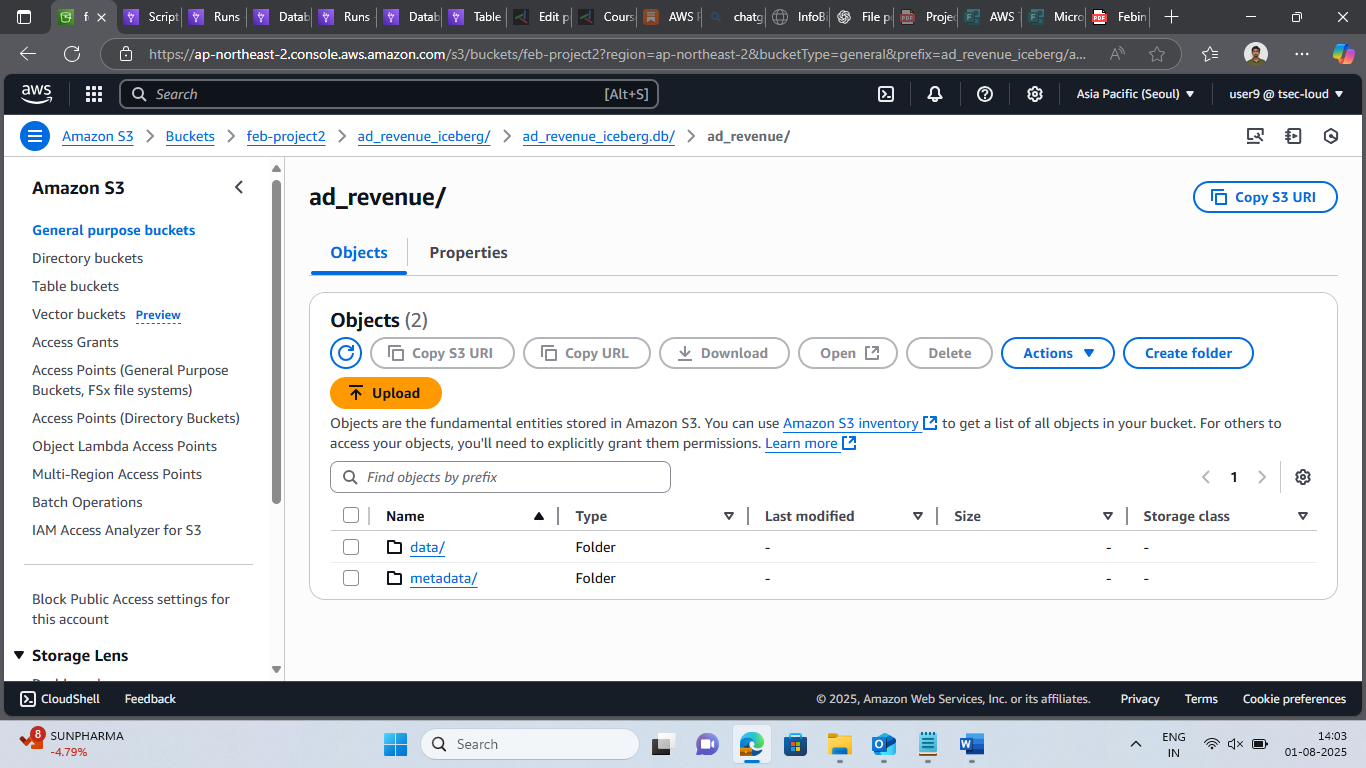
""")

# Write data to Iceberg table demographics\_df.writeTo("glue\_catalog.ad\_revenue\_iceberg.ad\_revenue").overwritePartitions()

# Optional: Query the data to verify

spark.sql("SELECT \* FROM glue\_catalog.ad\_revenue\_iceberg.ad\_revenue").show()

job.commit()



1. Test the data in athena
2. Create an airflow which will automate the process

from airflow import DAG  
from airflow.providers.amazon.aws.operators.glue import GlueJobOperator  
from datetime import datetime

default\_args = {  
    'owner': 'airflow',  
    'start\_date': datetime(2023, 1, 1),  
    'retries': 0  
}

with DAG(  
    dag\_id='trigger\_glue\_dag',  
    default\_args=default\_args,  
    schedule\_interval=None,  
    catchup=False,  
    tags=["glue", "iceberg"]  
) as dag:

    run\_glue\_job = GlueJobOperator(  
        task\_id='run\_iceberg\_glue\_job',  
        job\_name='feb-project2-adrevenuejob',        
        region\_name='ap-northeast-2'             
    )

    run\_glue\_job

1. Create a lambda function which will trigger the airflow when triggered by a put function in the bucket.

import json  
import boto3  
import urllib3  
from datetime import datetime  
  
MWAA\_ENV\_NAME = "feb-airflow"  
DAG\_NAME = "trigger\_glue\_dag"  
  
def lambda\_handler(event, context):  
    print("Received S3 Event:")  
    print(json.dumps(event, indent=2))  
  
    mwaa = boto3.client('mwaa')  
  
    try:  
        # 1. Create MWAA CLI token  
        resp = mwaa.create\_cli\_token(Name=MWAA\_ENV\_NAME)  
        web\_token = resp['CliToken']  
        web\_server\_hostname = resp['WebServerHostname']  
        print("CLI token and hostname retrieved successfully.")  
  
        # 2. Generate Airflow CLI trigger command  
        execution\_date = datetime.utcnow().isoformat()  
        cli\_command = f"dags trigger -e {execution\_date} {DAG\_NAME}"  
        trigger\_url = f"https://{web\_server\_hostname}/aws\_mwaa/cli"  
  
        print(f"Triggering DAG: {DAG\_NAME} at {execution\_date}")  
  
        # 3. Send request using urllib3  
        http = urllib3.PoolManager()  
        headers = {  
            "Authorization": f"Bearer {web\_token}",  
            "Content-Type": "text/plain"  
        }  
  
        response = http.request(  
            "POST",  
            trigger\_url,  
            headers=headers,  
            body=cli\_command.encode("utf-8"),  
            timeout=10.0  
        )  
  
        # 4. Handle response  
        status = response.status  
        response\_body = response.data.decode("utf-8")  
  
        print(f"HTTP Status: {status}")  
        print(f"Response Body:\n{response\_body}")  
  
        return {  
            'statusCode': status,  
            'body': json.dumps({  
                'message': f"DAG trigger attempted with status {status}",  
                'response': response\_body  
            })  
        }  
  
    except Exception as e:  
        print(f"Exception occurred: {str(e)}")  
        return {  
            'statusCode': 500,  
            'body': json.dumps({'error': str(e)})  
        }

1. Update the glue job so that it gets the file from the bucket

import sys

from awsglue.context import GlueContext

from awsglue.utils import getResolvedOptions

from pyspark.context import SparkContext

from pyspark.sql import SparkSession

from awsglue.job import Job

# Initialize SparkSession with Iceberg support

spark = SparkSession.builder \

.appName("Glue Iceberg Table Upsert") \

.config("spark.sql.catalog.glue\_catalog", "org.apache.iceberg.spark.SparkCatalog") \

.config("spark.sql.catalog.glue\_catalog.catalog-impl", "org.apache.iceberg.aws.glue.GlueCatalog") \

.config("spark.sql.catalog.glue\_catalog.warehouse", "s3://feb-project2/ad\_revenue\_iceberg/") \

.config("spark.sql.catalog.glue\_catalog.io-impl", "org.apache.iceberg.aws.s3.S3FileIO") \

.getOrCreate()

# Get job arguments

args = getResolvedOptions(sys.argv, ['JOB\_NAME'])

# Initialize Glue Context

sc = SparkContext.getOrCreate()

glueContext = GlueContext(sc)

spark = glueContext.spark\_session

job = Job(glueContext)

job.init(args['JOB\_NAME'], args)

# Create Iceberg DB if it doesn't exist

spark.sql("CREATE DATABASE IF NOT EXISTS glue\_catalog.ad\_revenue\_iceberg")

# Read new CSV records from S3

csv\_df = spark.read \

.format("csv") \

.option("header", "true") \

.option("inferSchema", "true") \

.load("s3://feb-project1/your-folder-name/") # This is my actual path

# Create a temporary view for SQL merge

csv\_df.createOrReplaceTempView("incoming\_updates")

# Perform MERGE INTO for upsert (match on channel\_id, channel\_name, and date)

spark.sql("""

MERGE INTO glue\_catalog.ad\_revenue\_iceberg.ad\_revenue AS target

USING incoming\_updates AS source

ON target.channel\_id = source.channel\_id

AND target.channel\_name = source.channel\_name

AND target.date = source.date

WHEN MATCHED THEN UPDATE SET

target.ad\_revenue = source.ad\_revenue

WHEN NOT MATCHED THEN INSERT \*

""")

job.commit()

1. Test it by uploading test data to see if it is upserting

Test datas:

1. channel\_id,channel\_name,date,ad\_revenue

febin\_music,F Music,01-08-2025,12.5

febin\_games,F Games,02-08-2025,18.9

1. channel\_id,channel\_name,date,ad\_revenue

febin\_music,F Music,01-08-2025,12.5

febin\_games,F Games,02-08-2025,18.9

febin\_music,F Music,01-08-2025,12345.1

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

1. Set up kinesis
2. Create an ec2 instance to simulate the flow of the real time data

import boto3

import csv

import json

import time

# AWS Kinesis Configuration

STREAM\_NAME = "feb-stream1"

REGION = "ap-northeast-2"

# Initialize boto3 client

kinesis = boto3.client("kinesis", region\_name=REGION)

# Open the CSV file

with open("viewership\_logs.csv", mode="r") as file:

reader = csv.DictReader(file)

for row in reader:

# Optional: convert data types (e.g., float, bool, int)

row["duration\_minutes"] = int(row["duration\_minutes"])

row["ads\_watched"] = int(row["ads\_watched"])

row["ad\_revenue"] = float(row["ad\_revenue"])

row["engagement\_score"] = float(row["engagement\_score"])

row["buffer\_count"] = int(row["buffer\_count"])

row["completion\_percentage"] = float(row["completion\_percentage"])

row["is\_live"] = True if row["is\_live"].lower() == "true" else False

# Send to Kinesis as JSON

kinesis.put\_record(

StreamName=STREAM\_NAME,

Data=json.dumps(row),

PartitionKey=row["region"] # Use region as partition key

)

print(f"Sent to Kinesis: {row['session\_id']}")

time.sleep(1)

1. Create an EMR cluster to get the data from kinesis and send it to snowflake and s3

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, expr, split

from pyspark.sql.types import \*

# 1. Define the schema of the CSV-style records

schema = StructType([

StructField("session\_id", StringType()),

StructField("user\_id", StringType()),

StructField("channel\_id", StringType()),

StructField("channel\_name", StringType()),

StructField("show\_name", StringType()),

StructField("genre", StringType()),

StructField("timestamp", StringType()),

StructField("duration\_minutes", IntegerType()),

StructField("region", StringType()),

StructField("subscription\_type", StringType()),

StructField("device", StringType()),

StructField("platform", StringType()),

StructField("is\_live", StringType()),

StructField("ads\_watched", IntegerType()),

StructField("ad\_revenue", FloatType()),

StructField("engagement\_score", FloatType()),

StructField("buffer\_count", IntegerType()),

StructField("completion\_percentage", FloatType())

])

# 2. Initialize Spark Session

print("🚀 Initializing Spark session...")

spark = SparkSession.builder \

.appName("KinesisToS3AndSnowflake") \

.getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

print("✅ Spark session initialized.")

# 3. Read from Kinesis

print("🔄 Connecting to Kinesis...")

df\_raw = spark.readStream \

.format("kinesis") \

.option("streamName", "feb-stream1") \

.option("endpointUrl", "https://kinesis.ap-northeast-2.amazonaws.com") \

.option("region", "ap-northeast-2") \

.option("startingPosition", "LATEST") \

.load()

print("✅ Connected to Kinesis.")

# 4. Convert and split raw records

df\_string = df\_raw.withColumn("data\_string", expr("CAST(data AS STRING)"))

df\_split = df\_string.withColumn("fields", split(col("data\_string"), ","))

# 5. Assign to structured schema

df\_parsed = df\_split.select(

col("fields").getItem(0).alias("session\_id"),

col("fields").getItem(1).alias("user\_id"),

col("fields").getItem(2).alias("channel\_id"),

col("fields").getItem(3).alias("channel\_name"),

col("fields").getItem(4).alias("show\_name"),

col("fields").getItem(5).alias("genre"),

col("fields").getItem(6).alias("timestamp"),

col("fields").getItem(7).cast("int").alias("duration\_minutes"),

col("fields").getItem(8).alias("region"),

col("fields").getItem(9).alias("subscription\_type"),

col("fields").getItem(10).alias("device"),

col("fields").getItem(11).alias("platform"),

col("fields").getItem(12).alias("is\_live"),

col("fields").getItem(13).cast("int").alias("ads\_watched"),

col("fields").getItem(14).cast("float").alias("ad\_revenue"),

col("fields").getItem(15).cast("float").alias("engagement\_score"),

col("fields").getItem(16).cast("int").alias("buffer\_count"),

col("fields").getItem(17).cast("float").alias("completion\_percentage")

)

# 6. Function to write batch to S3 and Snowflake

def write\_batch(batch\_df, epoch\_id):

print(f"🔁 Processing batch {epoch\_id}...")

record\_count = batch\_df.count()

print(f"📦 Records in batch: {record\_count}")

if record\_count == 0:

print("⚠️ Empty batch. Skipping writes.")

return

try:

# Snowflake options

snowflake\_options = {

"sfURL": "WJJCTOT-TYB13621.snowflakecomputing.com",

"sfUser": "FEBINPGEORGE",

"sfPassword": "rDpYpjS!E9iqG-E",

"sfDatabase": "FEB\_AMAZON",

"sfSchema": "PUBLIC"

"sfWarehouse" : "COMPUTE\_WH",

"sfRole": "ACCOUNTADMIN"

}

print("⛅ Writing to Snowflake...")

batch\_df.write \

.format("net.snowflake.spark.snowflake") \

.options(\*\*snowflake\_options) \

.option("dbtable", "channel\_logs") \

.mode("append") \

.save()

print("✅ Write to Snowflake succeeded.")

except Exception as e:

print("❌ Error writing to Snowflake:", e)

try:

print("📤 Writing to S3...")

batch\_df.write \

.format("csv") \

.mode("append") \

.save("s3://feb-project2/stream/")

print("✅ Write to S3 succeeded.")

except Exception as e:

print("❌ Error writing to S3:", e)

# 7. Start the streaming query

print("📡 Starting streaming query...")

query = df\_parsed.writeStream \

.foreachBatch(write\_batch) \

.outputMode("append") \

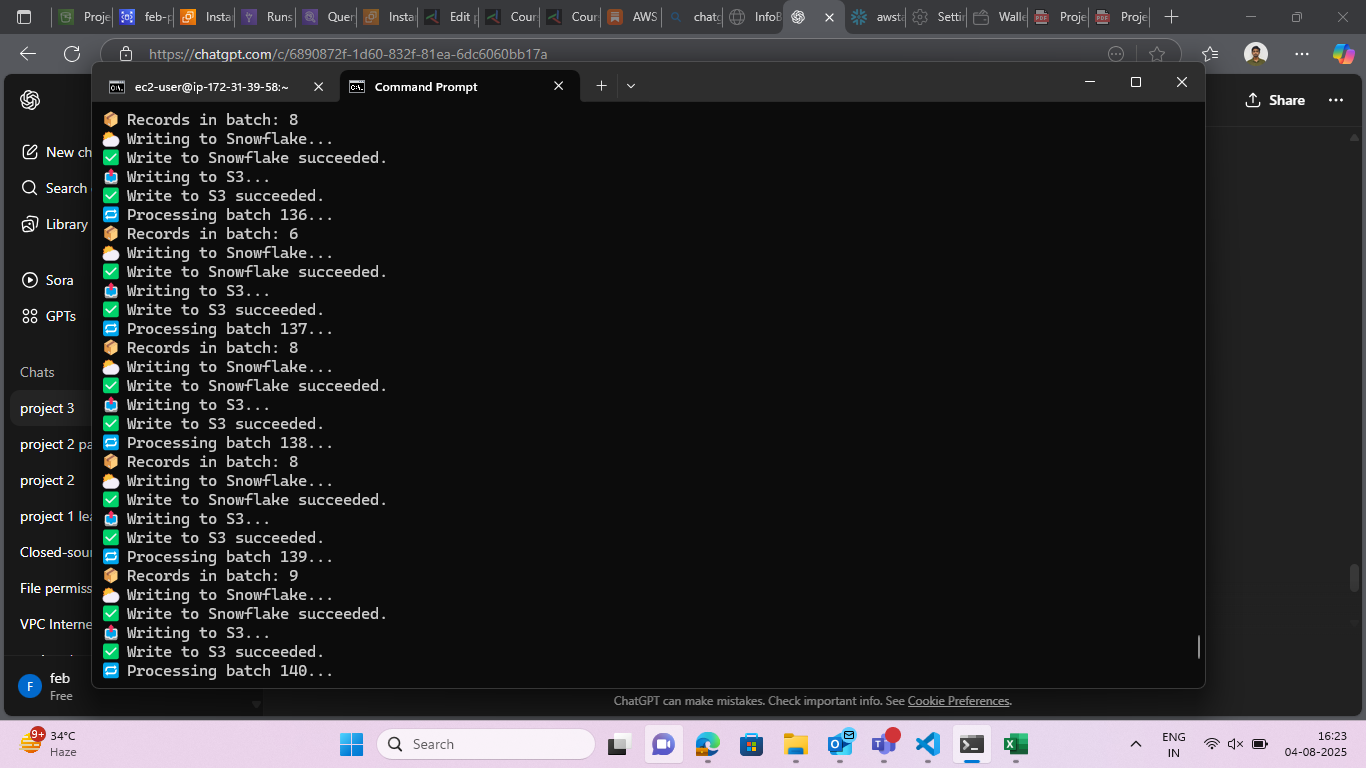
.option("checkpointLocation", "s3://feb-project2/temp/") \

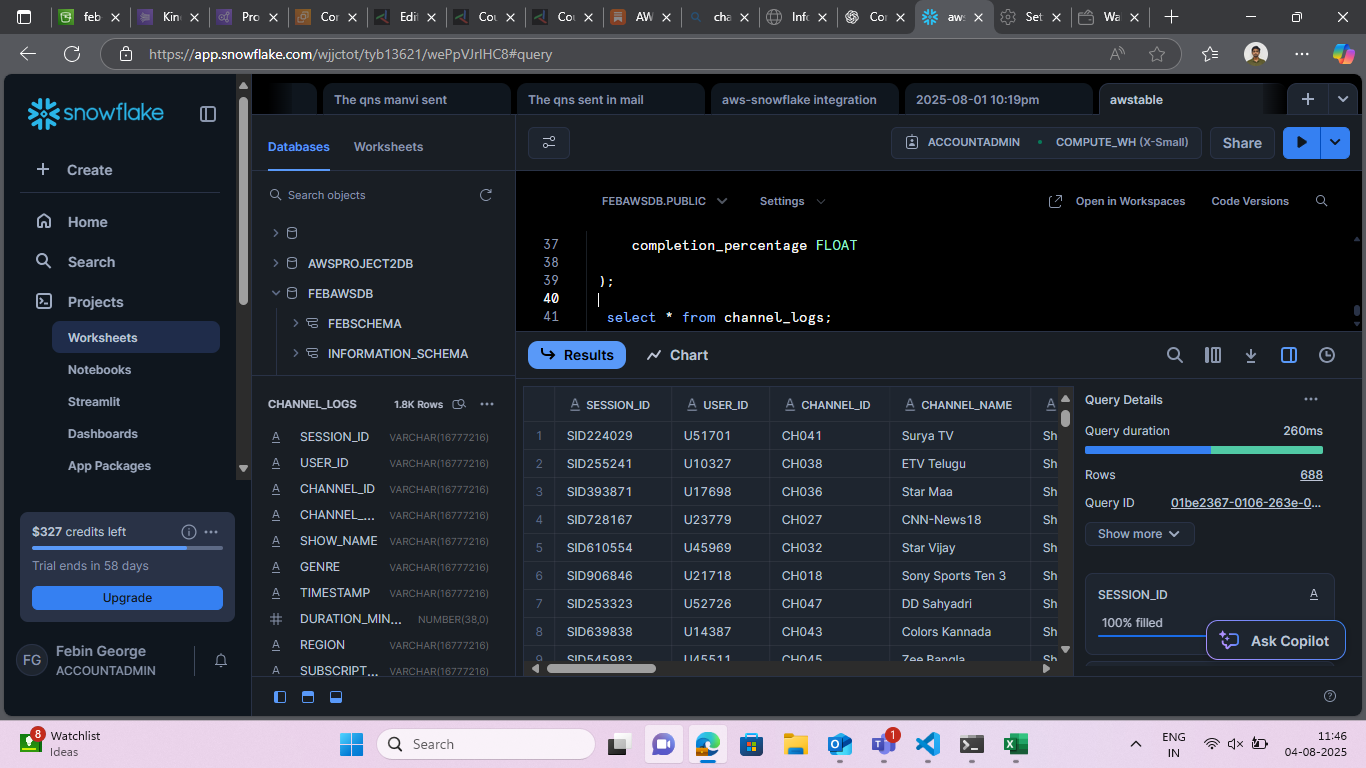
.trigger(processingTime="10 seconds") \

.start()

print("✅ Streaming started.")

query.awaitTermination()





1. Run a crawler and get the realtime data from S3 to Athena
2. Query the data

Total viewership duration per channel

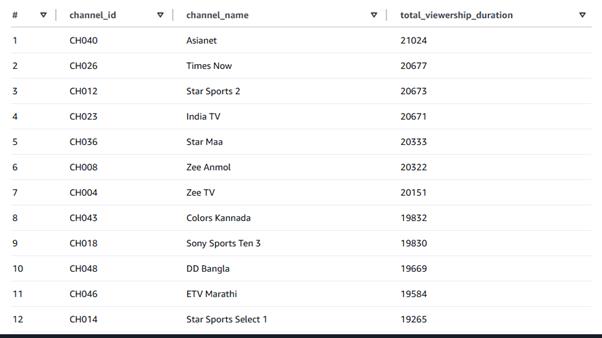
SELECT

channel\_id, channel\_name, SUM(duration\_minutes) AS total\_viewership\_duration

FROM viewership\_logs

GROUP BY channel\_id, channel\_name

ORDER BY total\_viewership\_duration DESC;



Average engagement by device

SELECT

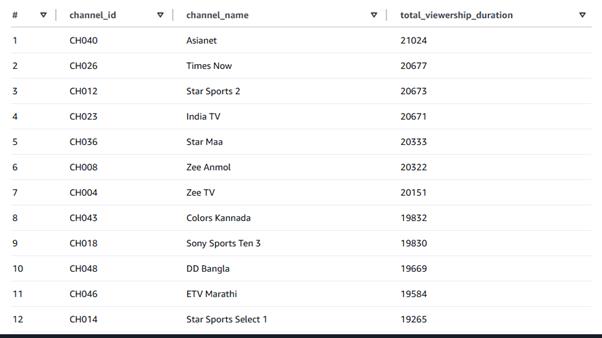
device,

AVG(engagement\_score) AS avg\_engagement\_score

FROM viewership\_logs

GROUP BY device

ORDER BY avg\_engagement\_score DESC;



Daily ad revenue per channel

SELECT

channel\_id,

channel\_name,

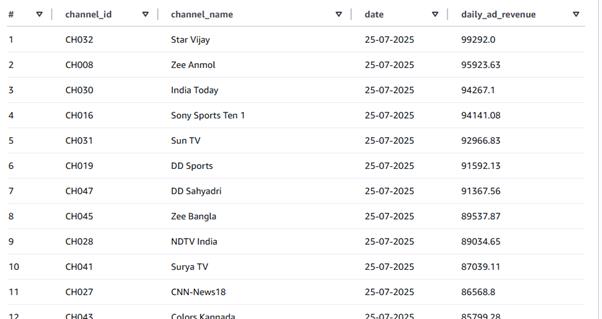
date,

SUM(ad\_revenue) AS daily\_ad\_revenue

FROM ad\_revenue

GROUP BY channel\_id, channel\_name, date

ORDER BY date, daily\_ad\_revenue DESC;



Gender-wise average completion percentage

SELECT

d.gender,

AVG(v.completion\_percentage) AS avg\_completion\_percentage

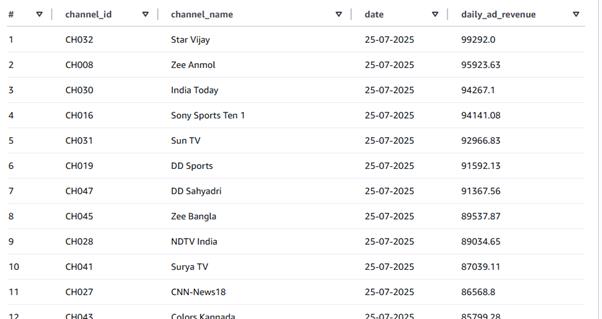
FROM viewership\_logs v

JOIN demographics d

ON v.user\_id = d.user\_id

GROUP BY d.gender

ORDER BY avg\_completion\_percentage DESC;



Most watched genres in each region

WITH genre\_duration AS (

SELECT

 region,

 genre,

 SUM(duration\_minutes) AS total\_watch\_time

FROM viewership\_logs

GROUP BY region, genre

),

ranked\_genres AS (

SELECT

 region,

 genre,

 total\_watch\_time,

 ROW\_NUMBER() OVER (PARTITION BY region ORDER BY total\_watch\_time DESC) AS rank

FROM genre\_duration

)

SELECT

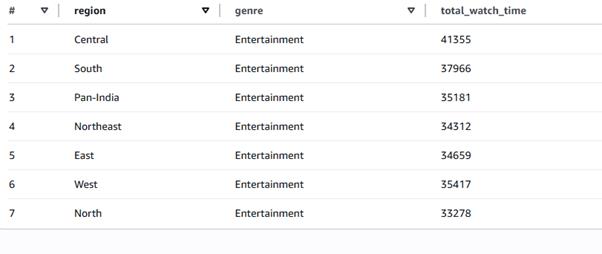
region,

genre,

total\_watch\_time

FROM ranked\_genres

WHERE rank = 1;



Top 5 channels with highest ad revenue in the past 7 days

SELECT

channel\_id,

channel\_name,

SUM(ad\_revenue) AS total\_revenue

FROM ad\_revenue

WHERE date >= date\_format(current\_date - interval '7' day, '%Y-%m-%d')

GROUP BY channel\_id, channel\_name

ORDER BY total\_revenue DESC

LIMIT 5;



Peak viewership hours by region

SELECT

region,

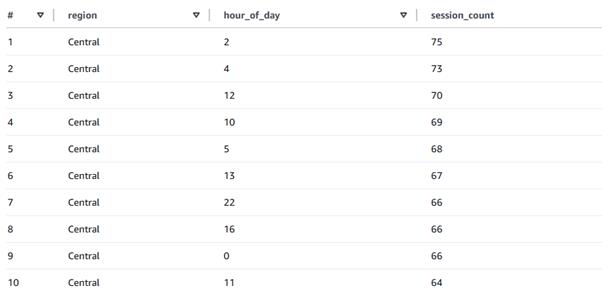
hour(parse\_datetime(timestamp, 'yyyy-MM-dd HH:mm:ss')) AS hour\_of\_day,

COUNT(\*) AS session\_count

FROM viewership\_logs

GROUP BY region, hour(parse\_datetime(timestamp, 'yyyy-MM-dd HH:mm:ss'))

ORDER BY region, session\_count DESC;



Which age group watches the most live content?

SELECT

d.age\_group,

COUNT(\*) AS live\_session\_count

FROM viewership\_logs v

JOIN demographics d ON v.user\_id = d.user\_id

WHERE v.is\_live = true

GROUP BY d.age\_group

ORDER BY live\_session\_count DESC;



Subscription type driving most revenue per channe

WITH revenue\_by\_subs AS (

SELECT

 channel\_id,

 channel\_name,

 subscription\_type,

 SUM(ad\_revenue) AS total\_revenue

FROM viewership\_logs

GROUP BY channel\_id, channel\_name, subscription\_type

),

ranked\_revenue AS (

SELECT \*,

      ROW\_NUMBER() OVER (PARTITION BY channel\_id ORDER BY total\_revenue DESC) AS rank

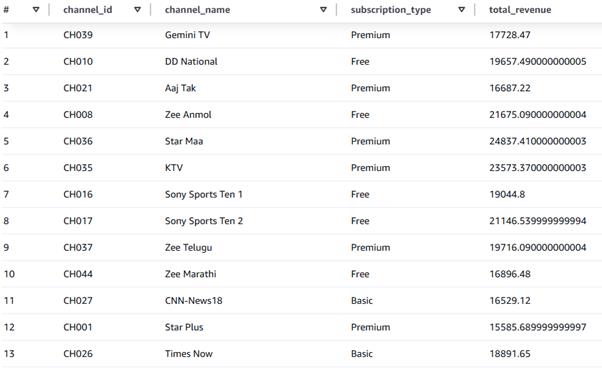
FROM revenue\_by\_subs

)

SELECT channel\_id, channel\_name, subscription\_type, total\_revenue

FROM ranked\_revenue

WHERE rank = 1;



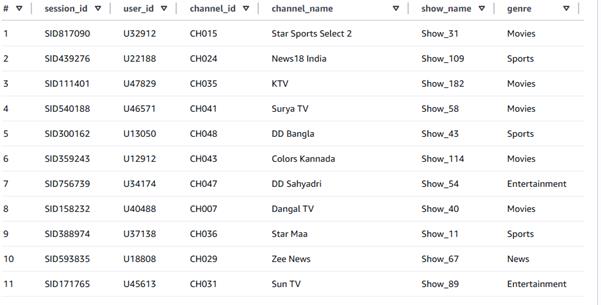
High engagement sessions (more than 90% completion and

SELECT \*

FROM viewership\_logs

WHERE completion\_percentage > 90

AND duration\_minutes > 1;



Channels with above-average ad revenue pe

WITH daily\_channel\_revenue AS (

SELECT

 channel\_id,

 channel\_name,

 date,

 SUM(ad\_revenue) AS daily\_revenue

FROM ad\_revenue

GROUP BY channel\_id, channel\_name, date

),

avg\_revenue AS (

SELECT AVG(daily\_revenue) AS avg\_daily\_revenue

FROM daily\_channel\_revenue

)

SELECT dcr.channel\_id, dcr.channel\_name, dcr.date, dcr.daily\_revenue

FROM daily\_channel\_revenue dcr, avg\_revenue ar

WHERE dcr.daily\_revenue > ar.avg\_daily\_revenue

ORDER BY dcr.daily\_revenue DESC;



Most engaging show for female users in metro regi

SELECT

show\_name,

AVG(engagement\_score) AS avg\_engagement

FROM viewership\_logs v

JOIN demographics d ON v.user\_id = d.user\_id

WHERE d.gender = 'Female'

AND v.region IN ('Central')

GROUP BY show\_name

ORDER BY avg\_engagement DESC

LIMIT 1;



Genre-wise ad revenue and completio

SELECT

genre,

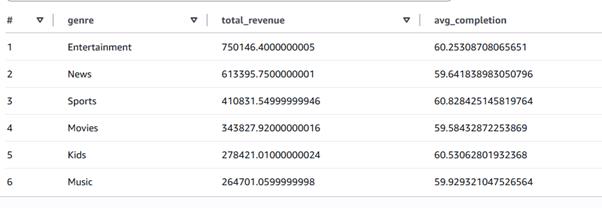
SUM(ad\_revenue) AS total\_revenue,

AVG(completion\_percentage) AS avg\_completion

FROM viewership\_logs

GROUP BY genre

ORDER BY total\_revenue DESC;



Channels with highest buffer counts but good engagem

SELECT

channel\_id,

channel\_name,

AVG(buffer\_count) AS avg\_buffer\_count,

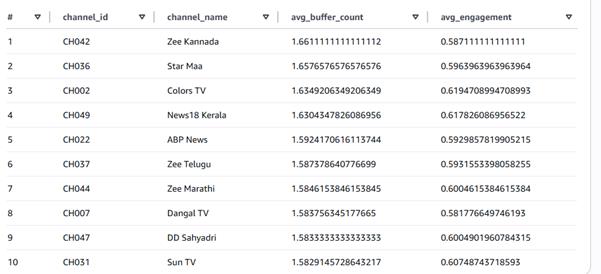
AVG(engagement\_score) AS avg\_engagement

FROM viewership\_logs

GROUP BY channel\_id, channel\_name

ORDER BY avg\_buffer\_count DESC

LIMIT 10;



which age group watches the most across all the

SELECT

d.age\_group,

SUM(v.duration\_minutes) AS total\_watch\_minutes

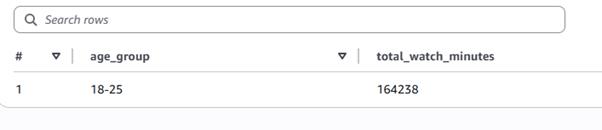
FROM viewership\_logs v

JOIN demographics d ON v.user\_id = d.user\_id

GROUP BY d.age\_group

ORDER BY total\_watch\_minutes DESC

LIMIT 1;



what is the average viewing time per user segmented by

SELECT

region,

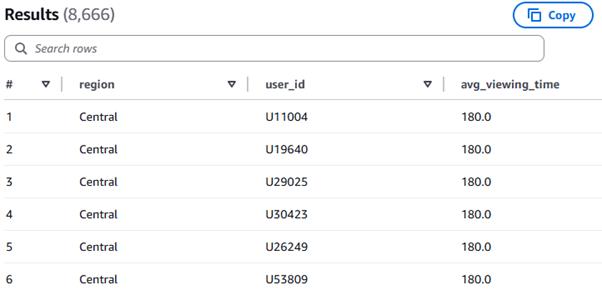
user\_id,

AVG(duration\_minutes) AS avg\_viewing\_time

FROM viewership\_logs

GROUP BY region, user\_id

ORDER BY region, avg\_viewing\_time DESC;



Do mobile users watch more content than Smart t

SELECT

device,

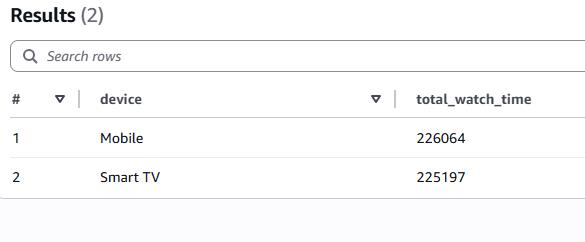
SUM(duration\_minutes) AS total\_watch\_time

FROM viewership\_logs

WHERE device IN ('Mobile', 'Smart TV')

GROUP BY device

ORDER BY total\_watch\_time DESC;



What is the total ad revenue per channel p

SELECT

channel\_id,

channel\_name,

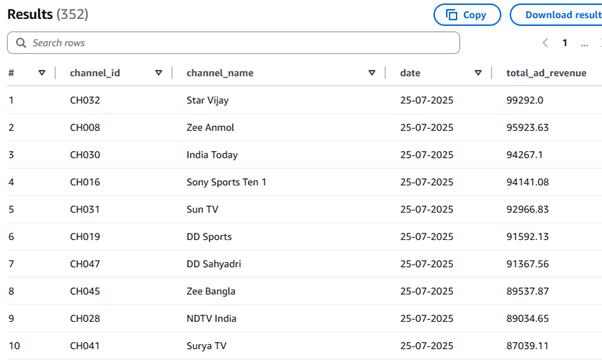
date,

SUM(ad\_revenue) AS total\_ad\_revenue

FROM ad\_revenue

GROUP BY channel\_id, channel\_name, date

ORDER BY date, total\_ad\_revenue DESC;



what are the peak hours of viewership accross

SELECT

hour(parse\_datetime(timestamp, 'yyyy-MM-dd HH:mm:ss')) AS hour\_of\_day,

COUNT(\*) AS session\_count

FROM viewership\_logs

GROUP BY hour(parse\_datetime(timestamp, 'yyyy-MM-dd HH:mm:ss'))

ORDER BY session\_count DESC

LIMIT 3; -- top 3 peak hours



how does content genre popularity vary by ge

SELECT

d.gender,

v.genre,

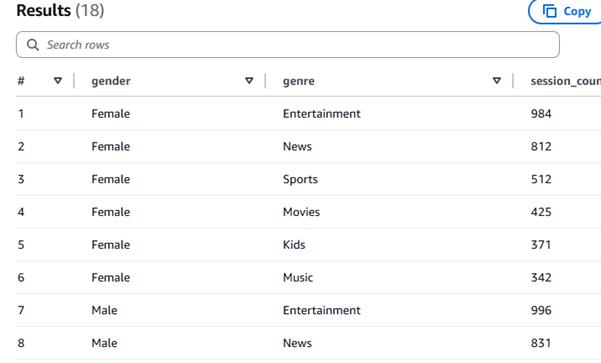
COUNT(\*) AS session\_count

FROM viewership\_logs v

JOIN demographics d ON v.user\_id = d.user\_id

GROUP BY d.gender, v.genre

ORDER BY d.gender, session\_count DESC;



which channels are more popular among premium s

SELECT

channel\_id,

channel\_name,

COUNT(\*) AS session\_count

FROM viewership\_logs

WHERE subscription\_type = 'Premium'

GROUP BY channel\_id, channel\_name

ORDER BY session\_count DESC

LIMIT 10;



are certain shows driving higher engagement among female users i

WITH female\_metro\_views AS (

SELECT

 v.show\_name,

 AVG(v.engagement\_score) AS avg\_engagement

FROM viewership\_logs v

JOIN demographics d ON v.user\_id = d.user\_id

WHERE d.gender = 'Female'

 AND v.region IN ('Central')

GROUP BY v.show\_name

)

SELECT \*

FROM female\_metro\_views

ORDER BY avg\_engagement DESC

LIMIT 10;

