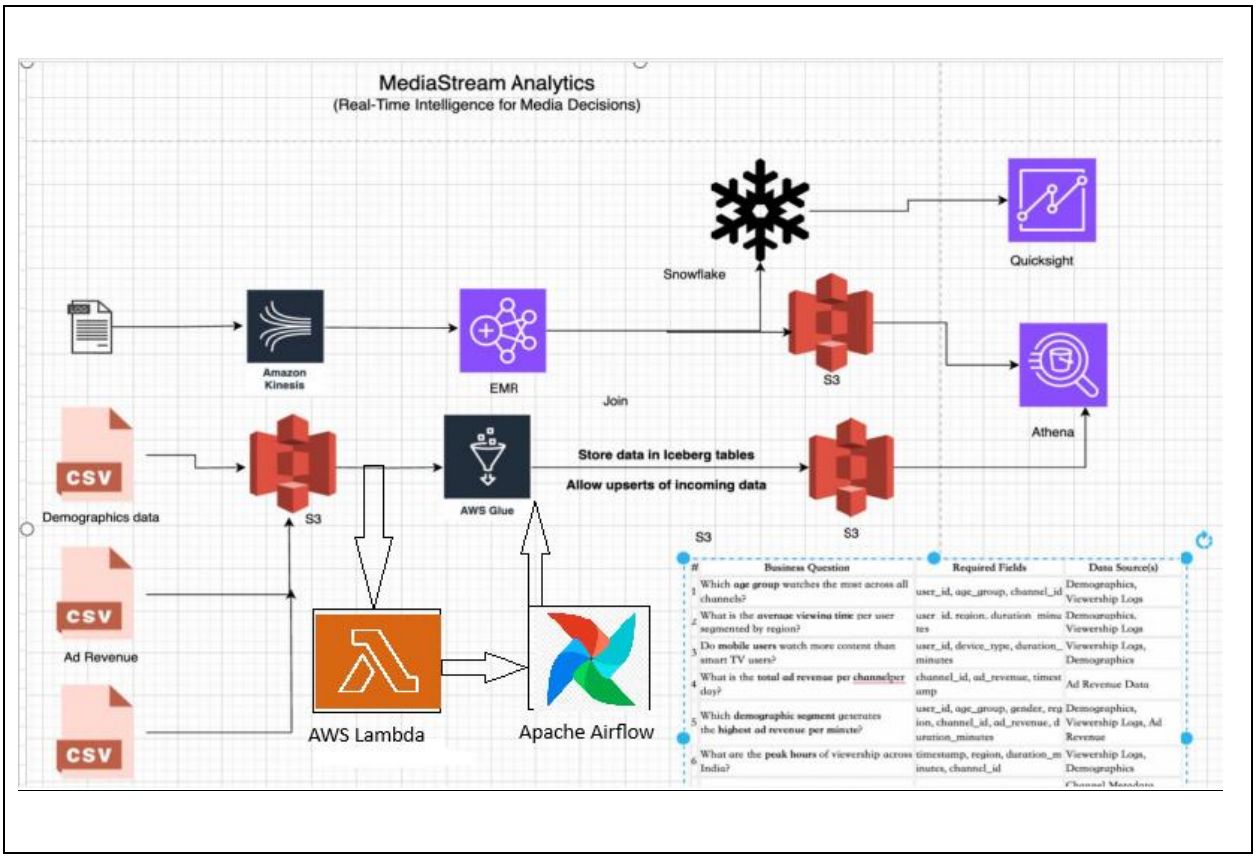
Project Report

Media Stream Analytics

Priti Ranjan Samal

Contents :

Media Stream Analytics - Project overview



1. Business Objective

MediaStream Analytics is a cloud-native big data analytics solution developed to generate actionable insights from large-scale media consumption data. The platform is designed to support both real-time and batch data processing to enhance decision-making related to programming strategies, monetization, and audience engagement.

The system ingests data such as viewership logs, ad revenue reports, channel metadata, and user demographics, cleans and processes them using a modern data pipeline, and makes them available for analysis using tools like Amazon Athena, Snowflake, and Amazon QuickSight

2. Key Data Sources

- Viewership Logs – Ingested in real-time via Amazon Kinesis
- Ad Revenue Reports – Stored in Amazon S3
- Channel Metadata – Stored in Amazon S3
- Demographic Information – Stored in Amazon S3

- Cooked PII Data – Processed and hashed through AWS Lambda

3. Data Pipeline Architecture

- Ingestion:
    - Real-time: Viewership logs ingested using Kinesis
    - Batch: Ad revenue, metadata, and demographic files ingested from S3
- Processing:
    - Cleaning and transformation using Spark on EMR
    - Cooked PII handled via Lambda functions
- Storage:
    - Processed datasets are written to Snowflake and Amazon S3
- Analytics & Reporting:
    - Athena and QuickSight used for querying and dashboarding
    - Both ad-hoc and scheduled reporting supported
- Architecture Pattern: Implements Lambda architecture supporting both batch and real-time layers

4. Core Business Questions Answered

- Which channels generate the highest ad revenue per day?
- What demographic segments generate the most ad revenue per minute?
- What are the peak viewing hours across India?
- How does genre popularity vary by gender?
- Which channels are most popular among premium subscribers?
- Which shows drive higher engagement in female metro audiences?

5. Project Goals

Technical Goals

- Implement real-time and batch data pipelines using AWS Kinesis, Spark on EMR, and Snowflake
- Learn and apply ETL and data warehousing best practices

- Enable data lake architecture with raw, cleaned, and curated zones in Amazon S3
- Develop and visualize key KPI dashboards using Amazon QuickSight
- Ensure data governance, including PII protection and hashing with Lambda

Non-Technical Goals

- Gain domain knowledge of the media and entertainment industry
- Understand how user behavior, demographics, and ad performance influence business strategy
- Learn to map business objectives to technical data processing workflows

Creating Iceberg Table for Data Stored in S3 and Loading to Athena Table

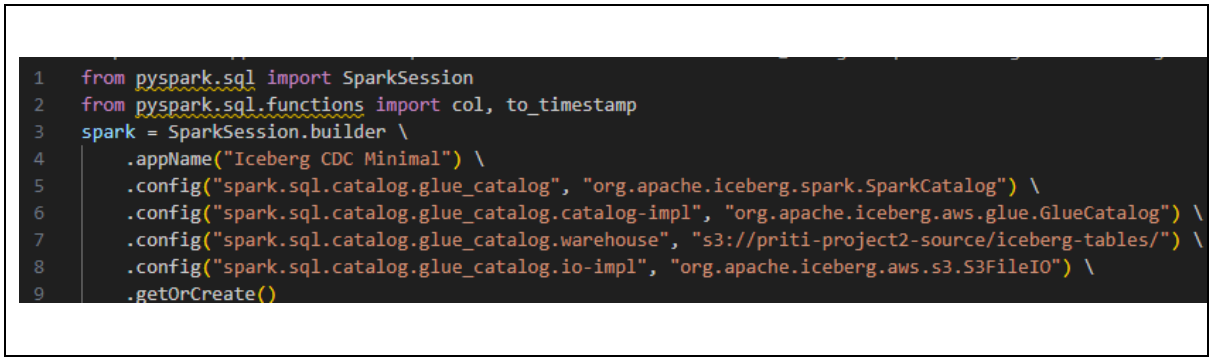
Loading dataset to S3 Bucket

- For loading data into S3 from local dataset stored in CSV format we would directly upload the CSV from our local system to the dedicated S3 bucket .

Creating Iceberg table from the S3 Data Source

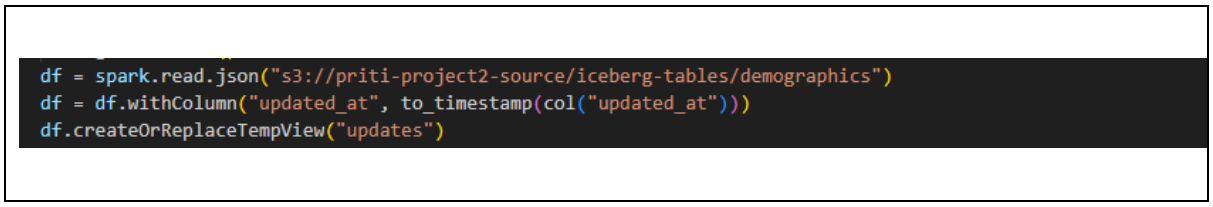
- We would be using Iceberg Table in Athena.
- We do so because Iceberg provide us with various functionalities such as :
  - ACID compliance over S3 Bucket .
  - Time-travel provide us with feature of data Retention .
  - Optimized for incremental updates with MERGE and UPSERT .
- We Would be working with three datasets :
  - Ad Revenue
  - Channel Metadata
  - Demographics
- In order to create Iceberg table from these datasets we have to first update the datasets in the S3 bucket .
- Once updated we would create an ETL Spark job in AWS Glue and execute it in order to create iceberg table .
- Iceberg tables help in handling incremental data efficiently. When a new job runs, the system checks whether the data already exists in the table if it does, the record is updated if not, a new record is inserted .

- Now we would be creating an AWS Glue ETL spark job which would create a iceberg table and also include feature of insert and update .
- This part configures the Spark job by creating a Spark session named **"Iceberg CDC Minimal"**, and sets it up to use Apache Iceberg with AWS Glue as the catalog and Amazon S3 as the data lake storage.



```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_timestamp
spark = SparkSession.builder \
    .appName("Iceberg CDC Minimal") \
    .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://priti-project2-source/iceberg-tables/") \
    .config("spark.sql.catalog.glue_catalog.io-impl", "org.apache.iceberg.aws.s3.S3FileIO") \
    .getOrCreate()
```

- Here we read the data from the mentioned S3 bucket where the datasets are uploaded in this case my bucket is “s3://priti-project2-source/” and the data source object Is “ iceberg-tables/demographics ” .
- Along with reading the data from the source we are also applying transformations such as converting the “ update_at ” attribute to timestamp format and creating a temporary table in order to use it to create Iceberg table .



```
df = spark.read.json("s3://priti-project2-source/iceberg-tables/demographics")
df = df.withColumn("updated_at", to_timestamp(col("updated_at")))
df.createOrReplaceTempView("updates")
```

- This code creates a Glue database “ icebergdb “ and an Iceberg table customers with demographics related columns, partitioned by day using the “ updated_at “ field for efficient S3-based storage and querying .

```
spark.sql("CREATE DATABASE IF NOT EXISTS glue_catalog.icebergdb")
spark.sql("""
CREATE TABLE IF NOT EXISTS glue_catalog.icebergdb.customers (
    user_id STRING,
    gender STRING,
    age_group STRING,
    range TIMESTAMP,
    subscription_type STRING
)
USING ICEBERG
PARTITIONED BY (days(updated_at))
""")
```

- Here the data is checked if the record in the updated data source is already existing in the iceberg table then it just updates the existing record . The record is checked on basis of the attribute id here target is the table already existing called customers and the source is the object from the earlier mentioned object .

```
spark.sql("""
MERGE INTO glue_catalog.icebergdb.customers AS target
USING updates AS source
ON target.id = source.id
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

- Once the spark job is done save it with a suitable recognizable job name with role having S3 access because this job name will be used later in MWAA dag trigger in order to execute the job .
- Once the job is saved then execute the job and check if it is creating the table in athena . If yes it is creating the table the upload a updated CSV file with some additional records and record with existing id with few updated attributes run the tests and check the table to find out if the new records are entered and the existing record is updated or not . Do the same for other two tables Ad Revenue , Channel Metadata .

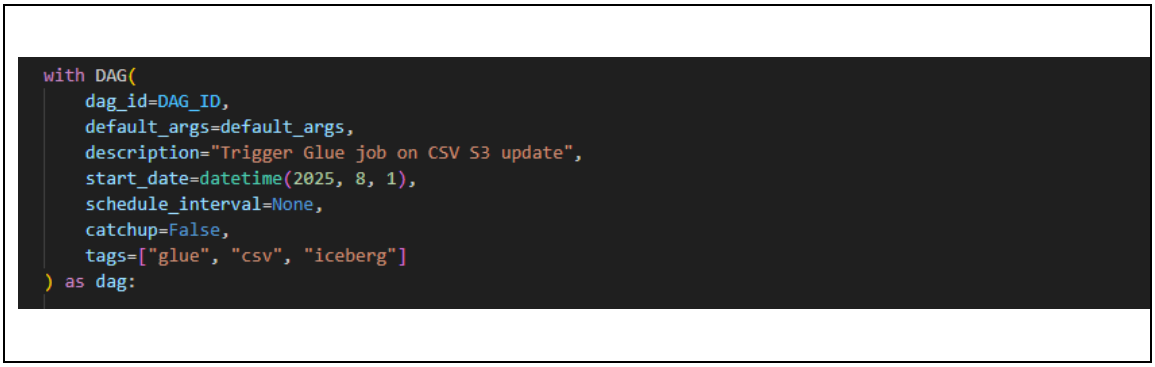
Creating an MWAA Environment to Trigger the ETL Job

- In this we will be setting up an MWAA Environment in AWS which is a managed service for running Apache Airflow .
- Here we will design a workflow (DAG) that automatically triggers an ETL job already created earlier that extracts and transforms the data .
- In order to create the MWAA environment we would first create a MWAA VPC .
- Once the MWAA VPC is started then start creating the environment with the credentials required .
- While creating the environment we need to provide with the dag directory containing the dag script .
- In order to create the dag script we would create a python file with the required details in order to trigger the ETL job .
- This script contains a DAG named "trigger ETL job on file update on s3" that is configured to run a Glue job called "demographicsetlpipeline" in the ap-northeast-1 AWS region. The default_args dictionary sets basic DAG parameters, such as the owner being airflow. The GlueJobOperator will be used later in the DAG to trigger the specified AWS Glue ETL job.

```
from airflow import DAG
from airflow.providers.amazon.aws.operators.glue import GlueJobOperator
from datetime import datetime
GLUE_JOB_NAME = "demographicsetlpipeline"
AWS_REGION = "ap-northeast-1"
DAG_ID = "trigger ETL job on file update on s3 "

default_args = {
    'owner': 'airflow',
}
```

- This block defines the DAG using the Airflow DAG context manager. It sets the dag name using DAG_ID which gives us the name by which it would be displayed in the Airflow UI .
- Default args. and description: this gives the description of the dag by which we get the information of what does the dag do here the description is "Trigger Glue job on CSV S3 update".
- The DAG is scheduled to start on August 1, 2025, but with schedule_interval=None, it will run only when triggered manually or programmatically. It won't execute automatically .
- The catchup=False ensures that missed runs between the start date and now won't be executed.
- The tags help categorize the DAG with labels like "glue", "csv", and "iceberg" for easy filtering in the Airflow UI.



```
with DAG(
    dag_id=DAG_ID,
    default_args=default_args,
    description="Trigger Glue job on CSV S3 update",
    start_date=datetime(2025, 8, 1),
    schedule_interval=None,
    catchup=False,
    tags=["glue", "csv", "iceberg"]
) as dag:
```

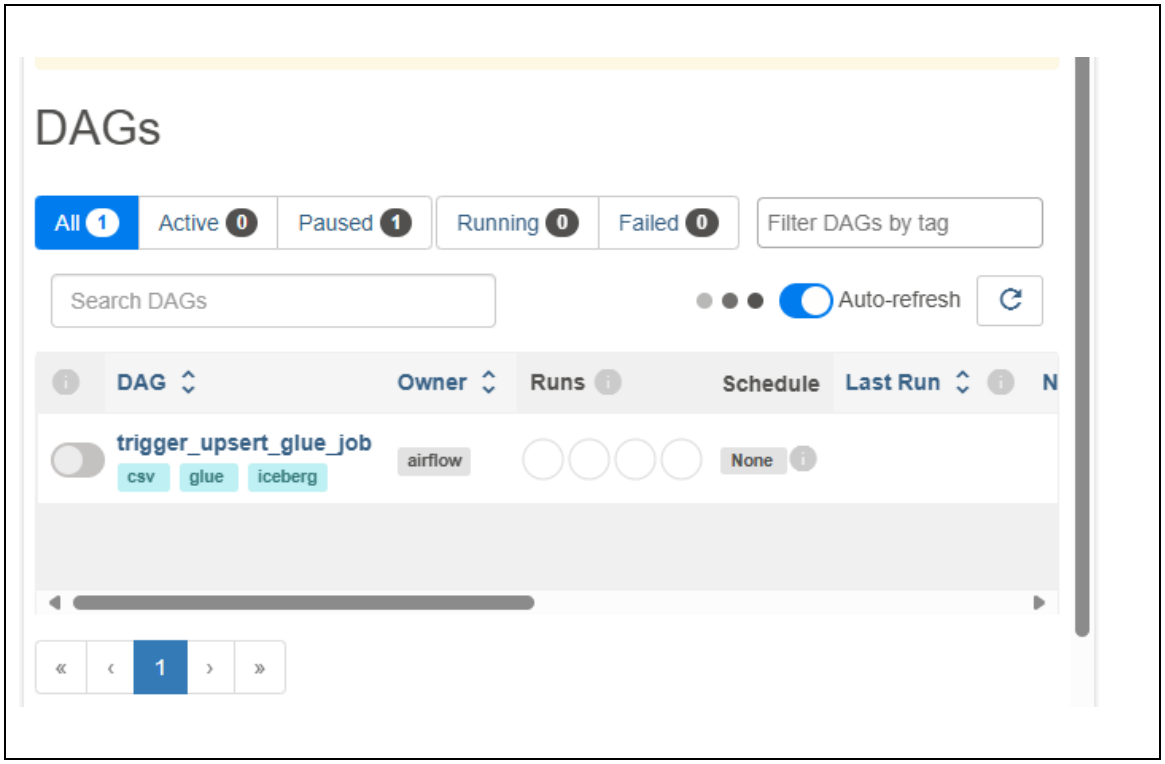
- This code defines a task named run_glue using GlueJobOperator, which triggers the AWS Glue job specified by GLUE_JOB_NAME and in the region AWS_REGION specified earlier .
- The task ID run_csv_to_iceberg_glue_job identifies this step within the DAG.

- Finally, run_glue is called to include the task in the DAG workflow for execution when the DAG runs.

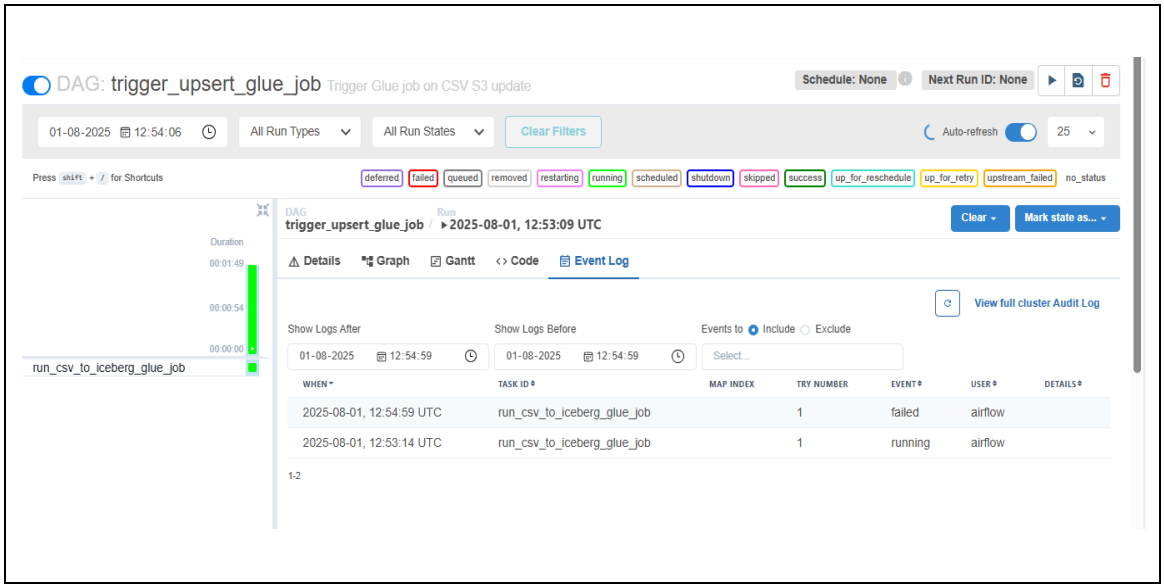
```
run_glue = GlueJobOperator(
    task_id="run_csv_to_iceberg_glue_job",
    job_name=GLUE_JOB_NAME,
    region_name=AWS_REGION
)

run_glue
```

- Once the dag Script is done add it to a folder called as dags/ in the S3 bucket and pass the S3 uri for the same folder while creating the environment .
- Once the environment is set up upload a updated CSV dataset to the source S3 bucket and run the ETL job using the Airflow and check if it is working with the records .



- Once done verify for other table and check the status of the ETL job in the event logs section of the airflow UI .


DAG: trigger_upsert_glue_job
Trigger Glue job on CSV S3 update
Schedule: None
Next Run ID: None
01-08-2025 12:54:06
All Run Types
All Run States
Clear Filters
Auto-refresh
25
Press shift + / for Shortcuts
deferred failed queued removed restarting running scheduled shutdown skipped success up_for_reschedule up_for_retry upstream_failed no_status
DAG
trigger_upsert_glue_job
Run
2025-08-01, 12:53:09 UTC
Clear
Mark state as...
Duration
00:01:49
00:00:54
00:00:00
run_csv_to_iceberg_glue_job
Details
Graph
Gantt
Code
Event Log
View full cluster Audit Log
Show Logs After
01-08-2025 12:54:59
Show Logs Before
01-08-2025 12:54:59
Events to Include Exclude
Select...
WHEN
TASK ID
MAP INDEX
TRY NUMBER
EVENT
USER
DETAILS
2025-08-01, 12:54:59 UTC
run_csv_to_iceberg_glue_job
1
failed
airflow
2025-08-01, 12:53:14 UTC
run_csv_to_iceberg_glue_job
1
running
airflow
1-2


Press shift + / for Shortcuts
deferred failed queued removed restarting running scheduled shutdown skipped success up_for_reschedule up_for_retry upstream_failed no_status
DAG
trigger_upsert_glue_job
Run
2025-08-01, 13:00:37 UTC
Clear
Mark state as...
Duration
00:01:50
00:00:55
00:00:00
run_csv_to_iceberg_glue_job
Details
Graph
Gantt
Code
Event Log
Layout:
Left -> Right
run_csv_to_iceberg_glue_job
success
GlueJobOperator

Automating the MWAA Environment Using a Lambda Function

- In order to automate the MWAA Environment created in the earlier step we would use AWS Lamda Function .
- Lambda Function can be used to execute an action whenever the function is triggered .
- In order to trigger the function we would set the trigger on the S3 object with PUT event which would result on triggering the lambda function whenever a dataset is added to that particular S3 object .
- On the function getting triggered it will execute the Airflow event resulting in executing the particular ETL job mentioned in the dag .
- So we would be creating the lambda function using python .
- This script imports the required libraries along with specifying the MWAA Env Name and the DAG name it is required to specify which MWAA env and DAG is needed to be executed on function getting triggered .

```
import json
import boto3
import urllib3
from datetime import datetime

MWAA_ENV_NAME = "mwaa-priti-environment"
DAG_NAME = "trigger_ETL_job_on_file_update_on_s3"
```

- This code defines the lambda_handler function, which is triggered when a file is uploaded to the mentioned S3 bucket. It prints the event details for logging and then creates a **boto3 MWAA client** to interact with the Airflow environment .

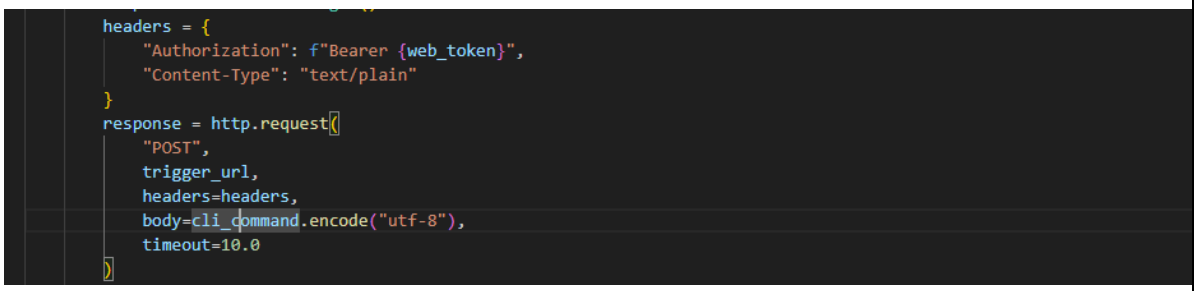
```
def lambda_handler(event, context):
    print("Received S3 Event:")
    print(json.dumps(event, indent=2))

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

- This block triggers the Airflow DAG by first generating a temporary CLI token and web server URL using mwaa.create_cli_token().
- It then builds a DAG trigger command with the current timestamp and prepares the HTTP request using urllib3 to send this command to the Airflow CLI endpoint.

```
    try:
        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.")
        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}")
        http = urllib3.PoolManager()
```

- This block sets the request headers, including the Bearer token for authentication, and sends a POST request to the Airflow CLI endpoint using urllib3.
- The request body contains the DAG trigger command, and a timeout of 10 seconds is set for the request.



```
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
)
```

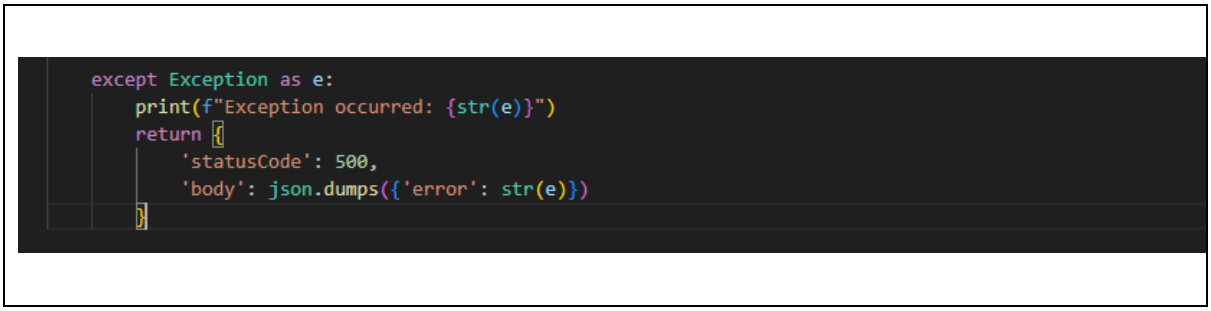
- This block extracts the HTTP response status and body from the Airflow CLI trigger request.
- It logs both to the console, then returns them in a structured JSON format, indicating whether the DAG trigger was successful and providing the response details.

```
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
    })
}
```

- This block handles any errors that occur during the DAG trigger process.
- If an exception is raised, it logs the error message and returns a JSON response with status code 500 and the error details, indicating the failure.



```
except Exception as e:
    print(f"Exception occurred: {str(e)}")
    return {
        'statusCode': 500,
        'body': json.dumps({'error': str(e)})
    }
```

- Once the script is done and create a test case in order to test the code . In test section add the test event JSON .
- Once added save it and go back to code section deploy the code and run tests in it .
- In order to let the Lambda access the S3 we have to assign it a role with S3 access policy attach to the role and attach the role to the lambda function .
- Once the tests are passed then upload a updated CSV file to S3 folder and check if the event is getting triggered .
- Once the file is uploaded go to the monitoring section in lambda function dashboard and go to cloudwatch logs over there it would be printing the logs of the lambda function which would give the idea if it is working fine or not .
- The Iceberg tables data is stored in the S3 bucket mentioned during creation of the ETL script created earlier .
- It stores the data in the mentioned folder of the bucket in two formats one contains data and the other contains metadata .
- The data is stored in parquet format . Because the iceberg format is build over the parquet format .
- once the data is stored for the first table we would do the same for the remaining other two tables Ad Revenue and channel metadata .

Creating table from Real-time data for the current date for viewership logs

- We will be generating live data from the existing and provided CSV dataset for viewership logs .
- In order to generate the data from it we would be using a python script in order to read the data from CSV and simultaneously send the data into kinesis data stream .
- Once the data is sent to kinesis data stream it would be captured in the other end and will be sent further using spark streaming to snowflake and S3 simultaneously .

Setting up EC2 instance and sending data via Kinesis Data Stream

- Create and EC2 instance and launch it . Alongside move the viewership_logs.csv to the instance .
- In order to send the logs over the EC2 to the EMR cluster we would be creating and using Kinesis Data Stream .
- Go to AWS Kinesis select the create data stream option and create a data stream with a suitable name as it would be used later on .
- Once the EC2 instance is launched log in to it using SSH and local .pem file stored in the system .
- After getting into the system we would create a python script in order to read the data from the csv file and send it over the kinesis stream .

- We are importing the required libraries such as boto3 , pandas and json in order to use the aws service and perform operations on the data extracted from the csv file .
- It define the Kinesis data stream that will be used in order to send the data over from EC2 to EMR.
- We do have to specify the region where the stream is located in my case it is to be in ‘ ap-northeast-1 ’.
- Followed by providing the path to the data source here we are using ‘ viewership_logs.csv ’ as the data source it is the same csv file that we moved earlier to the EC2 instance from local system .
- We create a boto3 kinesis client in order to access the kinesis operations . Providing the region name specified earlier .

```
import boto3
import pandas as pd
import json

stream_name = "viewershipdata-stream-priti"
region_name = "ap-northeast-1"
csv_file_path = "/home/ec2-user/data/viewership_logs.csv"
kinesis_client = boto3.client('kinesis', region_name=region_name)
```

- This code reads a CSV file which is already uploaded to the EC2 instance using pandas.
- It loops through each row, converts it to JSON, and sends it as a record to an AWS Kinesis Data Stream using the put_record method.
- Each record is uniquely identified using the row index as the partition key, and a success message is printed after each send.

```
try:
    df = pd.read_csv(csv_file_path)
    print(f"Loaded CSV file with {len(df)} records.")
    for index, row in df.iterrows():
        payload = row.to_dict()
        json_payload = json.dumps(payload)

        response = kinesis_client.put_record(
            StreamName=stream_name,
            Data=json_payload.encode('utf-8'),
            PartitionKey=str(index)
        )

        print(f"Sent record {index} to Kinesis. SequenceNumber: {response['SequenceNumber']}")
```

- This block catches any errors that occur while reading the CSV or sending records to Kinesis.
- If an exception is raised, it prints a clear error message with the exception details, helping debug what went wrong.

```
except Exception as e:
    print(f"Error reading CSV or sending to Kinesis: {str(e)}")
```

Reading the data from kinesis data stream and sending it over to Snowflake and S3 bucket simultaneously

- We will be extracting the data and sending it over to snowflake and S3 simultaneously for further transformations .
- In order to extract and send the data over to snowflake we will be creating a spark code which will be extracting the data from the data stream and sending it to both the destination .
- Create and setup an EMR cluster with Spark and Hadoop installed in it .
- Once created login to the instance using ssh and .pem keypair file .

- We will import all the libraries required which includes spark session , to perform transformations functions such as col, expr,split and all other necessary types .

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, expr, split
from pyspark.sql.types import *
```

- create a Schema for the data that would be extracted and transformations will be done from kinesis data streams .

```
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())
])
```

- create a spark session using spark session builder and initiate the session .

```
print("🚀 Initializing Spark session...")
spark = SparkSession.builder \
    .appName("priti-data-stream-proj-02") \
    .getOrCreate()

print("✅ Spark session initialized.")
                                                    1,24          Top
```

- once the session is created then connect to kinesis and start reading data from the stream .
- the stream specifications such as the format the stream name from which data is to be collected , the endpointurl of the stream , the region where the stream is created and further more .

```
print("🔄 Connecting to Kinesis...")
df_raw = spark.readStream \
    .format("kinesis") \
    .option("streamName", "priti-data-stream-proj-02") \
    .option("endpointUrl", "https://kinesis.ap-northeast-1.amazonaws.com
") \
    .option("region", "ap-northeast-1") \
    .option("startingPosition", "LATEST") \
    .load()
print("✅ Connected to Kinesis.")
```

- Here we perform some operations on the extracted data from the data stream in order to normalize the data .
- Converts the binary-encoded data column into a string and stores it in a new column called data_string and Splits each data_string into a list of values using a comma (,) as the delimiter and stores the result in a new column named fields.

```
df_string = df_raw.withColumn("data_string", expr("CAST(data AS STRING)"
))
df_split = df_string.withColumn("fields", split(col("data_string"), ",")
)
```

- This step transforms each raw comma-separated record from Kinesis into a structured DataFrame by assigning each field to a meaningful column name and converting data types where necessary (like integers or floats).
- It uses the getItem() function to extract values from the split list and alias() to name columns such as session_id, user_id, channel_name, etc., making the data ready for further processing or storage.

```
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"),
                                                    59,24         34%
```

- The function “ write_batch ” handles each micro-batch of streaming data.
- It first checks the number of records in the batch using .count(). If the batch is empty, it skips processing.
- Otherwise, it attempts to write the data to a Snowflake table (channel_logs) using Spark's Snowflake connector.
- The connection options like Snowflake URL, user credentials, database, schema, and warehouse are provided. The data is written in append mode, meaning new rows are added without overwriting existing data.

```
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": "JRZXFJC-AXB02657.snowflakecomputing.com",
            "sfUser": "Priti16",
            "sfPassword": "Pritiranjan@16",
            "sfDatabase": "LOGSTV",
            "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.")
                                                    98,24         73%
```

- This code handles errors during data writing. If writing to Snowflake fails, it catches the exception and prints the error message.
- Then, it attempts to write the same batch to Amazon S3 in CSV format using append mode, so new data is added without deleting existing files.
- If writing to S3 also fails, it catches and prints that error too. This ensures that failures in one target either it be snowflake or S3 don't stop the entire process and still try saving the data elsewhere.

```
except Exception as e:
    print("❌ Error writing to Snowflake:", e)

try:
    print("📤 Writing to S3...")
    batch_df.write \
        .format("csv") \
        .mode("append") \
        .save("s3://priti-project2-source/streamed/")
    print("✅ Write to S3 succeeded.")

except Exception as e:
    print("❌ Error writing to S3:", e)
```

- This code starts the Spark Structured Streaming job. It uses the foreachBatch method to call the write_batch function for every micro-batch of parsed data.
- The outputMode "append" ensures only new rows are processed.
- A checkpoint is saved to S3 to track progress and allow recovery in case of failure.
- The trigger processingTime="10 seconds" tells Spark to process incoming data every 10 seconds.

- Finally, query.awaitTermination() keeps the job running continuously.

```
print("🚀 Starting streaming query...")
query = df_parsed.writeStream \
    .foreachBatch(write_batch) \
    .outputMode("append") \
    .option("checkpointLocation", "s3://priti-project2-source/temp/") \
    .trigger(processingTime="10 seconds") \
    .start()

print("✅ Streaming started.")
query.awaitTermination()
                                                    128,24        Bot
```

- In order to run this script we do need some additional jar files and spark Snowflake connector we have to download it externally .

```
[ec2-user@ip-172-31-42-238 priti]$ sudo wget https://repo1.maven.org/mav
en2/net/snowflake/snowflake-jdbc/3.13.30/snowflake-jdbc-3.13.30.jar -P /
usr/lib/spark/jars/
```

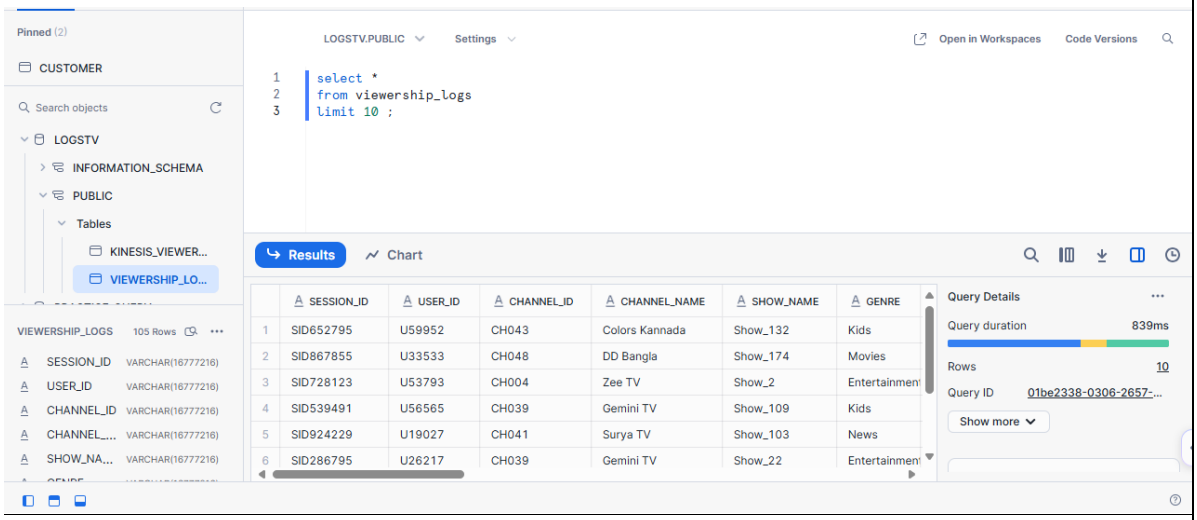
```
[ec2-user@ip-172-31-42-238 priti]$ sudo wget https://repo1.maven.org/mav
en2/net/snowflake/spark-snowflake_2.12/2.11.0-spark_3.3/spark-snowflake_
2.12-2.11.0-spark_3.3.jar -P /usr/lib/spark/jars/
```

```
[ec2-user@ip-172-31-42-238 ~]$ ls
kinesis_jars  readstream.py   transfer1.py
priti         snowflake_jars  writes3stream.py
[ec2-user@ip-172-31-42-238 ~]$
```

- once downloaded pass it with spark -submit code and execute it .

```
[ec2-user@ip-172-31-42-238 priti]$ spark-submit   --master local[*]   --
jars /usr/lib/spark/jars/snowflake-jdbc-3.13.30.jar,/usr/lib/spark/jars/
spark-snowflake_2.12-2.9.0-spark_3.1.jar   --packages com.qubole.spark:s
park-sql-kinesis_2.12:1.2.0_spark-3.0 sendwrite.py
```

- Once the data writing is completed in order to verify go to snowflake and check if the table is created and if data is present or not .



- Also check S3 if the data is sent over there or not because we will be using that on later stage in order to create table in athena and query those tables .

Query Execution in Athena

- We will be querying the tables created on various conditions and requirements .
- We have already created three table :
  - Ad revenue
  - Demographics
  - Channel Metadata
- We will have to create the table for realtime viewership logs .
- We can create the above mentioned table by simply creating and running the crawler on the data source folder of viewership_logs .
- Once the crawler is executed and table is created with data filled into it we will now start executing the queries .

Query Execution

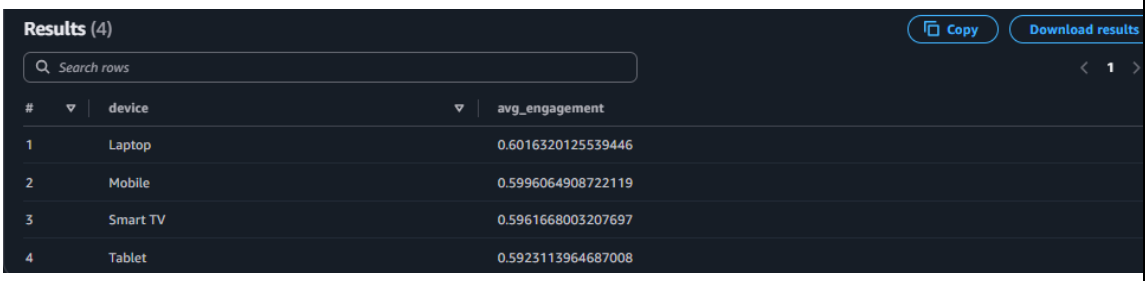
- Total viewership duration per channel .

```
SELECT channel_name, SUM(duration_minutes) AS
total_viewership_minutes
FROM logsdata
GROUP BY channel_name
ORDER BY total_viewership_minutes DESC;
```

#	channel_name	total_viewership_minutes
1	Sony Entertainment Television	21585
2	ABP News	21053
3	Gemini TV	20704
4	Zee TV	20420
5	Colors TV	20010
6	Eurosport India	19669

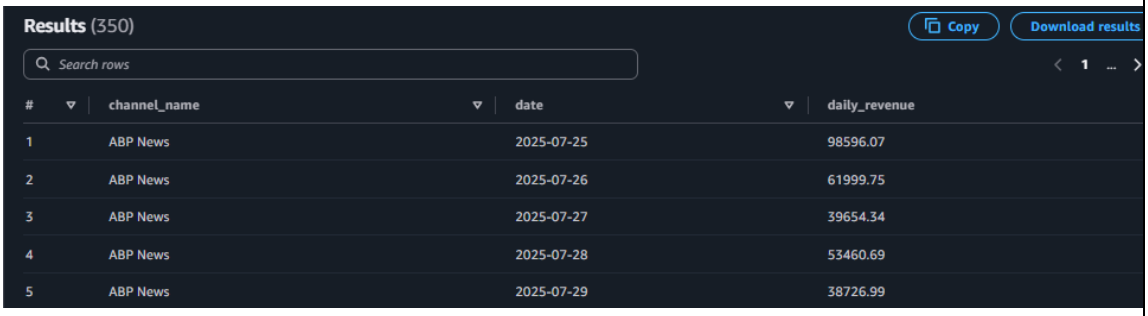
- Average engagement by device

```
SELECT device, AVG(engagement_score) AS avg_engagement
FROM logsdata
GROUP BY device
ORDER BY avg_engagement DESC;
```



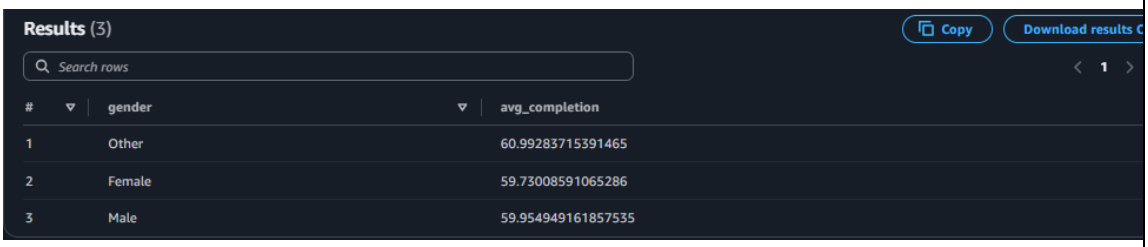
- Daily ad revenue per channel

```
SELECT channel_name, date, SUM(ad_revenue) AS
daily_revenue
FROM ad
GROUP BY channel_name, date
ORDER BY channel_name, date;
```



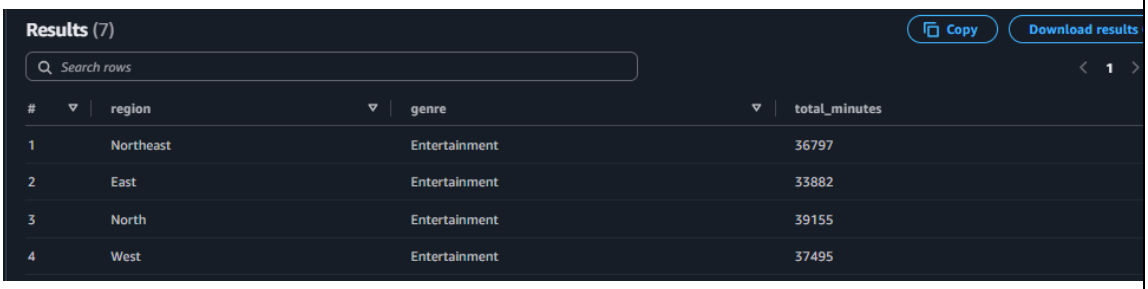
- Gender-wise average completion percentage

```
SELECT d.gender, AVG(v.completion_percentage) AS
avg_completion
FROM logsdata v
JOIN clear_demo d ON v.user_id = d.user_id
GROUP BY d.gender;
```



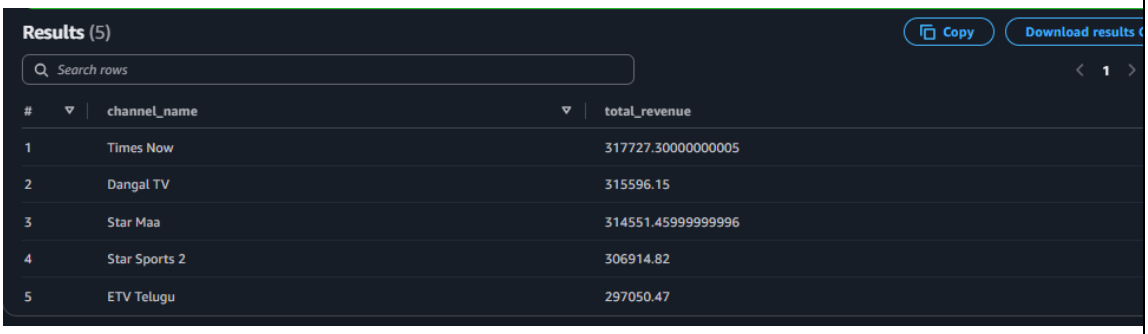
- Most watched genres in each region

```
WITH ranked_genres AS (
  SELECT
    region,
    genre,
    SUM(duration_minutes) AS total_minutes,
    RANK() OVER (PARTITION BY region ORDER BY
SUM(duration_minutes) DESC) AS rank
  FROM logsdata
  GROUP BY region, genre
)
SELECT region, genre, total_minutes
FROM ranked_genres
WHERE rank = 1;
```



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

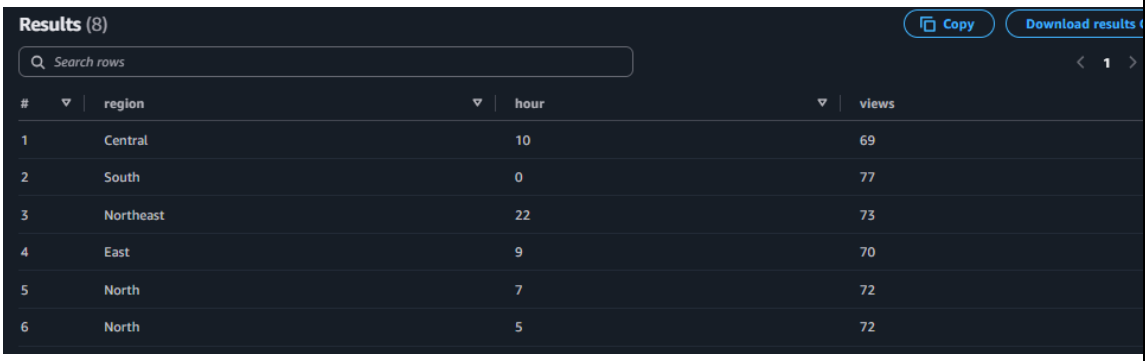
```
SELECT channel_name, SUM(ad_revenue) AS total_revenue
FROM ad
WHERE CAST(date AS DATE) >= CURRENT_DATE - INTERVAL
'7' DAY
GROUP BY channel_name
ORDER BY total_revenue DESC
LIMIT 5;
```



- Peak viewership hours by region

```
SELECT region, hour, views
FROM (
   SELECT
```

```
      region,
      EXTRACT(HOUR FROM CAST("timestamp" AS
TIMESTAMP)) AS hour,
      COUNT(*) AS views,
      RANK() OVER (PARTITION BY region ORDER BY COUNT(*)
DESC) AS rnk
   FROM logsdata
   GROUP BY region, EXTRACT(HOUR FROM
CAST("timestamp" AS TIMESTAMP))
) sub
WHERE rnk = 1;
```



Results (8)

Copy Download results

Search rows

1

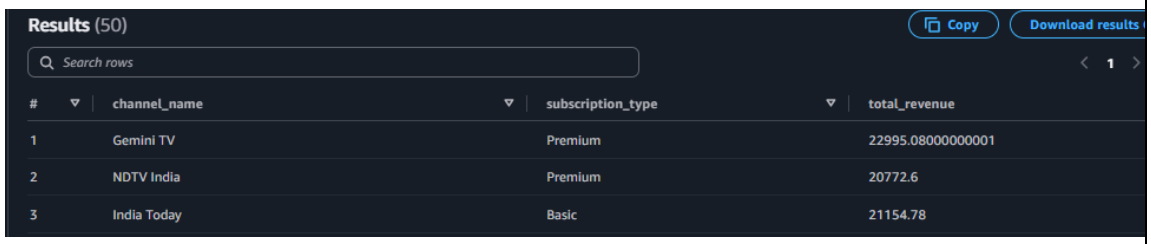
#	region	hour	views
1	Central	10	69
2	South	0	77
3	Northeast	22	73
4	East	9	70
5	North	7	72
6	North	5	72

- Which age group watches the most live content?

```
SELECT d.age_group, SUM(v.duration_minutes) AS
live_minutes
FROM logsdata v
JOIN clear_demo d ON v.user_id = d.user_id
WHERE is_live = TRUE
GROUP BY d.age_group
ORDER BY live_minutes DESC;
```

- Subscription type driving most revenue per channel

```
WITH ranked_revenue AS (
  SELECT
    channel_name,
    subscription_type,
    SUM(ad_revenue) AS total_revenue,
    RANK() OVER (PARTITION BY channel_name ORDER BY
SUM(ad_revenue) DESC) AS rnk
  FROM logsdata
  GROUP BY channel_name, subscription_type
)
SELECT channel_name, subscription_type, total_revenue
FROM ranked_revenue
WHERE rnk = 1;
```



- High engagement sessions (more than 90% completion and >1 min)

```
SELECT *
FROM logsdata
WHERE completion_percentage > 90 AND duration_minutes
> 1;
```

Results (1,246)

Copy | Download results C

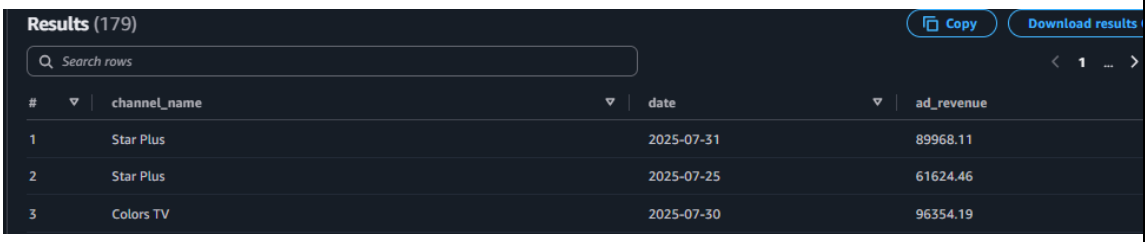
Search rows

1 ...

#	session_id	user_id	channel_id	channel_name	show_name	genre	timestamp	duration_minu
1	SID752959	U52689	CH035	KTV	Show_50	Movies	2025-07-27 03:10:42	32
2	SID910606	U46010	CH004	Zee TV	Show_190	Entertainment	2025-07-26 22:43:28	43
3	SID305655	U11120	CH016	Sony Sports Ten 1	Show_68	Sports	2025-07-26 11:16:37	41

- Channels with above-average ad revenue per day

```
WITH avg_daily AS (
  SELECT AVG(ad_revenue) AS avg_rev FROM ad
)
SELECT ar.channel_name, ar.date, ar.ad_revenue
FROM ad ar
CROSS JOIN avg_daily
WHERE ar.ad_revenue > avg_daily.avg_rev;
```



Results (179)

Copy | Download results

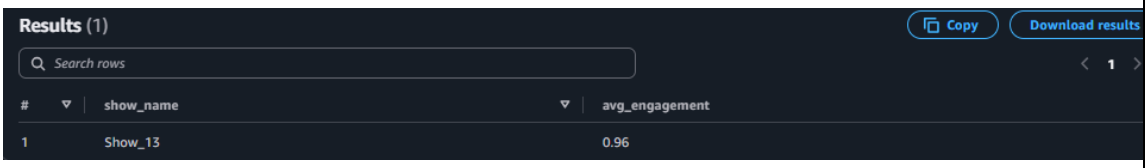
Search rows

1 ...

#	channel_name	date	ad_revenue
1	Star Plus	2025-07-31	89968.11
2	Star Plus	2025-07-25	61624.46
3	Colors TV	2025-07-30	96354.19

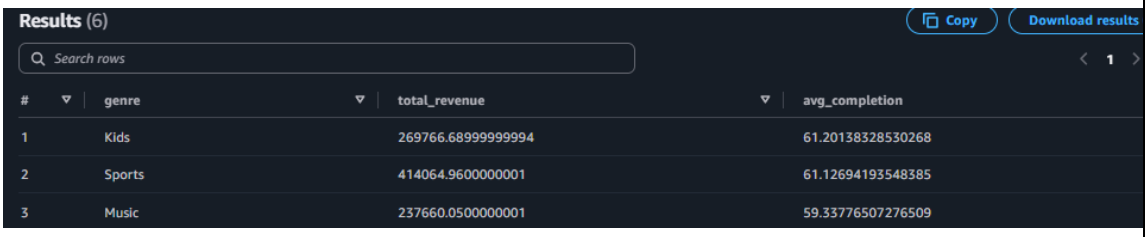
- Most engaging show for female users in metro regions

```
SELECT show_name, AVG(engagement_score) AS
avg_engagement
FROM logsdata v
JOIN clear_demo d ON v.user_id = d.user_id
WHERE d.gender = 'Female' AND d.region IN ('Metro', 'Pan-
India')
GROUP BY show_name
ORDER BY avg_engagement DESC
LIMIT 1;
```



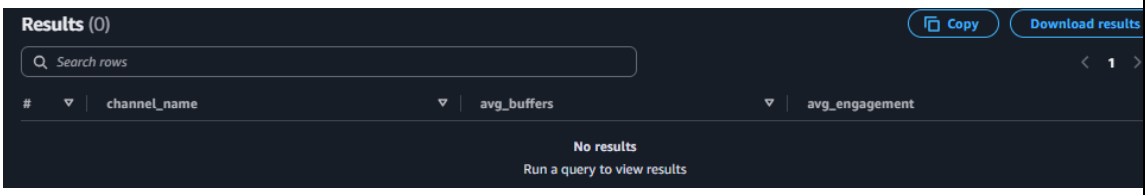
- Genre-wise ad revenue and completion comparison

```
SELECT genre,
    SUM(ad_revenue) AS total_revenue,
    AVG(completion_percentage) AS avg_completion
FROM logsdata
GROUP BY genre;
```



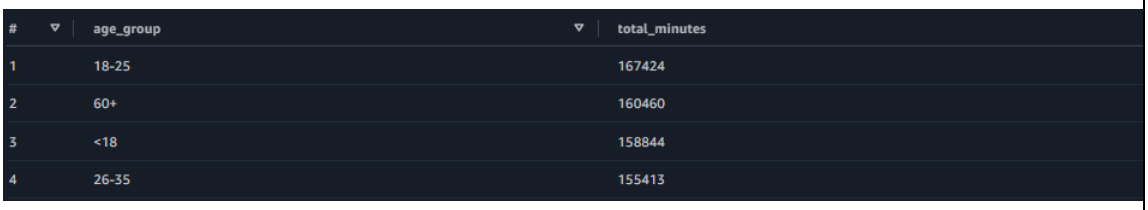
- Channels with highest buffer counts but good engagement

```
SELECT channel_name,
    AVG(buffer_count) AS avg_buffers,
    AVG(engagement_score) AS avg_engagement
FROM logsdata
GROUP BY channel_name
HAVING AVG(buffer_count) > 1 AND AVG(engagement_score)
> 0.7
ORDER BY avg_buffers DESC;
```



- which age group watches the most across all the channels

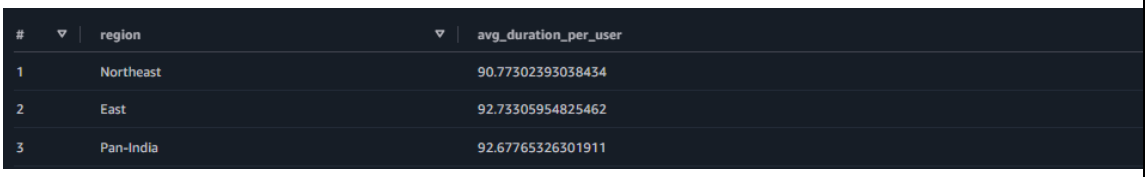
```
SELECT d.age_group, SUM(v.duration_minutes) AS
total_minutes
FROM logsdata v
JOIN clear_demo d ON v.user_id = d.user_id
GROUP BY d.age_group
ORDER BY total_minutes DESC;
```



#	age_group	total_minutes
1	18-25	167424
2	60+	160460
3	<18	158844
4	26-35	155413

- what is the average viewing time per user segmented by region

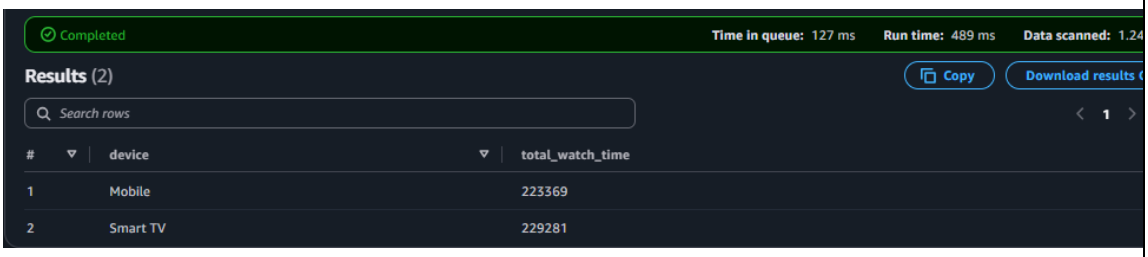
```
SELECT d.region, AVG(v.duration_minutes) AS
avg_duration_per_user
FROM logsdata v
JOIN clear_demo d ON v.user_id = d.user_id
GROUP BY d.region;
```



#	region	avg_duration_per_user
1	Northeast	90.77302393038434
2	East	92.73305954825462
3	Pan-India	92.67765326301911

- Do mobile users watch more content than Smart tv users

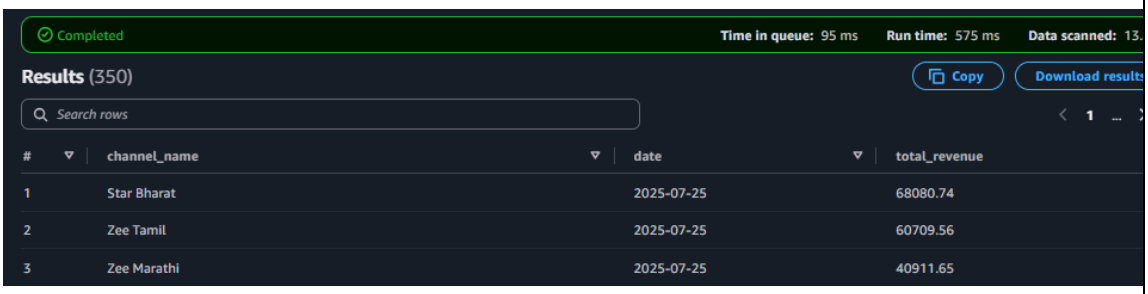
```
SELECT device, SUM(duration_minutes) AS total_watch_time
FROM logsdata
WHERE device IN ('Mobile', 'Smart TV')
GROUP BY device;
```



#	device	total_watch_time
1	Mobile	223369
2	Smart TV	229281

- What is the total ad revenue per channel per day

```
SELECT channel_name, date, SUM(ad_revenue) AS
total_revenue
FROM ad
GROUP BY channel_name, date
ORDER BY date;
```

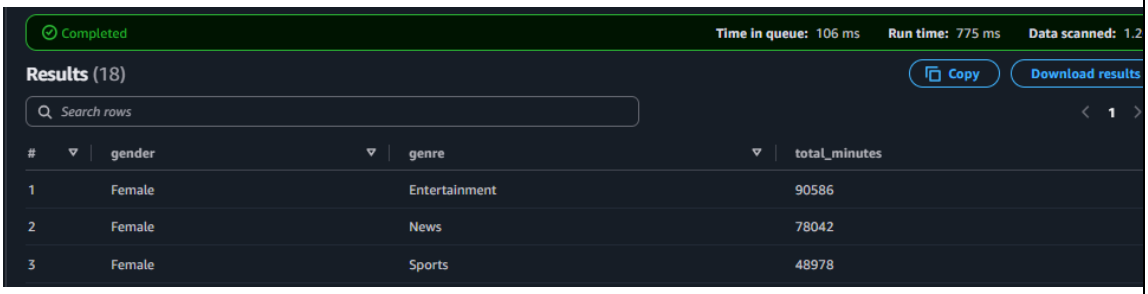


- what are the peak hours of viewership accross india

```
SELECT
   EXTRACT(HOUR FROM CAST("timestamp" AS TIMESTAMP))
AS hour,
   COUNT(*) AS view_count
FROM logsdata
GROUP BY EXTRACT(HOUR FROM CAST("timestamp" AS
TIMESTAMP))
ORDER BY view_count DESC;
```

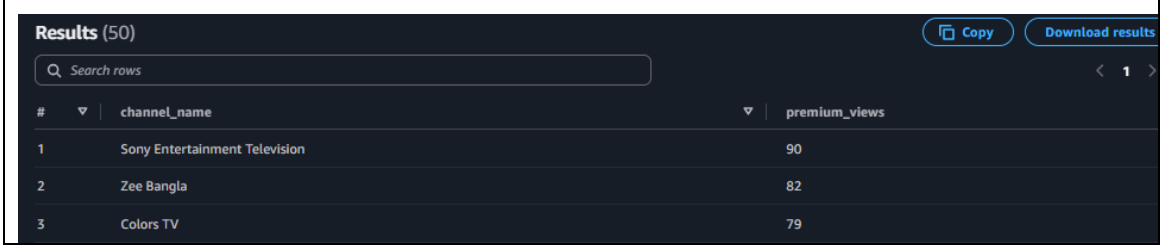
- how does content genre popularity vary by gender

```
SELECT d.gender, v.genre, SUM(v.duration_minutes) AS
total_minutes
FROM logsdata v
JOIN clear_demo d ON v.user_id = d.user_id
GROUP BY d.gender, v.genre
ORDER BY d.gender, total_minutes DESC;
```



- which channels are more popular among premium subscribers

```
SELECT channel_name, COUNT(*) AS premium_views
FROM logsdata
WHERE subscription_type = 'Premium'
GROUP BY channel_name
ORDER BY premium_views DESC;
```



- are certain shows driving higher engagement among female users in metro cities

```
SELECT show_name, AVG(engagement_score) AS
avg_engagement
FROM logsdata v
JOIN clear_demo d ON v.user_id = d.user_id
WHERE d.gender = 'Female'
GROUP BY show_name
ORDER BY avg_engagement DESC
LIMIT 10;
```