TOURIST DATA ANALYSIS

Mentor:

Bhagyaraj Gopalam

Team Members

Name of the Employee	Employee ID
Aashika Kancharlapalli	46343405
Lakshmi H P	46343404
Lekha R	46343662
Meghna K J	46343660
Priya Gupta	46194505
Srijita Pal	46345384
Sushree Sibarpita Dey	46343403

Table of Contents

1. Introduction

1.1 Description of the Project

The tourist data analysis project aims to extract insights from tourism-related data, unveiling patterns, preferences, and behaviors to guide strategic decisions and enhance tourism offerings. Leveraging advanced data analytics techniques, the project focuses on automating daily data analysis stored in an Amazon S3 bucket. This automation is facilitated by integrating AWS Glue for seamless ETL (Extract, Transform, Load) operations and AWS Redshift for scalable data warehousing capabilities. By harnessing the strengths of AWS Glue, AWS Redshift, and Amazon S3, the project ensures efficient data processing, bolstering reliability and responsiveness. This integration empowers stakeholders with actionable insights, facilitating informed decisions that drive continuous improvements in tourism strategies and operational efficiencies. Additionally, the project includes the development of effective dashboards designed to facilitate comprehensive data understanding and interpretation.

1.2. Required Services

AWS Account

Amazon Web Services has been used to build an end-to-end log analytics solution that collects, ingests, processes, and loads both batch data and streaming data, and makes the processed data available to your users in the analytics system they are already using in near real-time.

AWS S3

S3 stands for Simple Storage Service. It is Object-based storage, i.e., you can store images, word files, pdf files, etc. It has unlimited storage. Files are stored in Bucket. A bucket is like a folder available in S3 that stores the files.

AWS IAM

Using AWS Identity and Access Management (IAM), we can specify who can access which AWS services and resources, and under which conditions. IAM is a feature of our AWS account and is offered at no additional charge. With IAM, you define who can access what by specifying fine-grained permissions. IAM then enforces those permissions for every request. Access is denied by default and access is granted only when permissions specify an "Allow."

AWS Glue

AWS Glue is an ETL (Extract, Transform, Load) service provided by Amazon Web Services. It automates much of the heavy lifting involved in data integration and ETL tasks,

making it easier for businesses to build scalable and efficient data pipelines. Operating on a serverless architecture, AWS Glue eliminates the need for users to provision or manage infrastructure. It automatically scales resources based on the workload demands, optimizing performance and cost efficiency. Users pay only for the resources consumed during job execution, making it a cost-effective solution. AWS Glue seamlessly integrates with other AWS services such as Amazon S3, Amazon Redshift, Amazon RDS, Amazon Aurora, and various relational databases. This allows businesses to ingest data from multiple sources, transform it, and load it into data lakes or data warehouses for further analysis and reporting.

Amazon Redshift

Amazon Redshift is a fully managed, petabyte-scale data warehouse service offered by AWS, designed for processing large-scale analytical workloads. It uses SQL to analyse structured and semi-structured data across data warehouses, operational databases, and data lakes, using AWS-designed hardware and machine learning to deliver the best price performance at any scale. Redshift stores data in a columnar format, optimizing query efficiency by reducing I/O and enhancing compression. It includes seamless integration with AWS services like S3 and Glue for data loading and transformation, support for popular BI tools such as Tableau and QuickSight, and scalability to accommodate varying workload demands.

Amazon QuickSight

Amazon QuickSight is a powerful BI tool that democratizes data analytics, allowing users across an organization to make informed decisions based on real-time data insights. Its combination of ease of use, scalability, advanced analytics capabilities, and cost-effectiveness makes it an asset for businesses aiming to harness the power of their data. Amazon QuickSight powers data-driven organizations with unified business intelligence (BI) at hyperscale. With QuickSight, all users can meet varying analytic needs from the same source of truth through modern interactive dashboards, paginated reports, natural language queries and embedded analytics. With Amazon Q in QuickSight, business analysts and business users can use natural language to build, discover, and share meaningful insights in seconds, turning insights into impacts faster.

1.3. Project Workflow

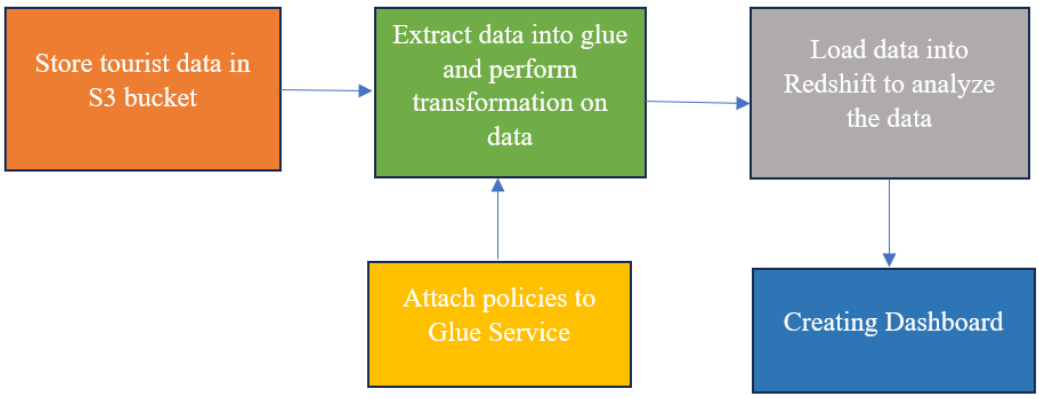

Figure 1: Project Workflow

1. **Load the tourist Data into S3 Bucket:**

The initial step involves loading tourist data to S3 bucket. This is achieved using AWS S3. Handling data with null values and inconsistent formats when loading into an S3 bucket for further processing involves several considerations and steps to ensure data quality and compatibility.

2. **Attach IAM policies:**

An IAM (Identity and Access Management) role is essential for AWS Glue to access other AWS services and resources securely. When we create an AWS Glue job , we specify an IAM role that grants permissions for Glue to perform actions on our behalf. We have selected AWS Glue as our service. Permission policies were attached here are AWS S3 full access, AWS Glue Console full access, CloudWatch full access to the Glue.

3. **Extract, Transform and load data into Glue:**

To process our data stored in S3 which includes null values or blanks, inconsistencies in data formats, and attributes represented as percentages across 11 data files spanning from 2014 to 2021. Our objective is to standardize this data for analysis by filling null values with zeros, correcting format discrepancies, and converting percentage attributes into numerical values. Additionally, we aim to extract and consolidate data for each year into individual files, renaming attributes as needed to align with our operational requirements. This systematic approach ensures that our data is consistent, formatted correctly, and organized by year for effective analysis and decision-making. After preparing and organizing our data as outlined, the next step involves loading it into Amazon Redshift. To accomplish this, we use the Redshift URL, password, and database details provided by AWS.

4. **Redshift:**

First, we created a Redshift cluster with a configuration of two nodes using "dc2.large" instances. We named the initial database "dev." Once the data has been successfully loaded into Amazon Redshift, corresponding tables are created within the specified database. These tables mirror the structure and schema of the data that was loaded, ensuring consistency and alignment with the dataset from S3.

Here's an overview of what happens after the data loading process is completed:

- Amazon Redshift automatically creates tables based on the schema defined during the data loading process. The schema includes attributes such as column names, data types, and any constraints specified.

- Each table corresponds to a dataset or a subset of data loaded from S3, organized according to the defined structure (e.g., year-wise data partitions or consolidated files). Data Storage

- The loaded data is stored in Redshift's columnar storage format, optimized for query performance and efficient data retrieval. Redshift organizes data across nodes in a cluster, using a distributed architecture to handle large volumes of data and complex queries effectively.

Availability for Queries: Once the data is loaded and processed, it becomes available for querying. We can run SQL queries using Redshift's query editor.

5. **Quick Sight :**

Ensure our data is properly loaded and structured in Amazon Redshift. This involves defining schemas, tables, and optimizing for performance using distribution keys, sort keys, and compression as needed.

Connecting QuickSight to Redshift:

- Use Amazon QuickSight's interface to connect to our Amazon Redshift cluster. QuickSight allows us to import data directly from Redshift .This step involves selecting the tables and columns we want to use in our dashboard. Once data is imported, we can perform additional data preparation steps within Quick Sight if needed, such as filtering, aggregating, or joining datasets from multiple tables.Use QuickSight's drag-and-drop interface to create various types of visualizations (e.g., Vertical bar chart Horizontal stacked bar chart, Line chart, Vertical stacked 100% bar chart, Donut chart and points on map) based on our imported data. Customize these visualizations with different styles and formats to convey insights effectively.

6. Star Schema

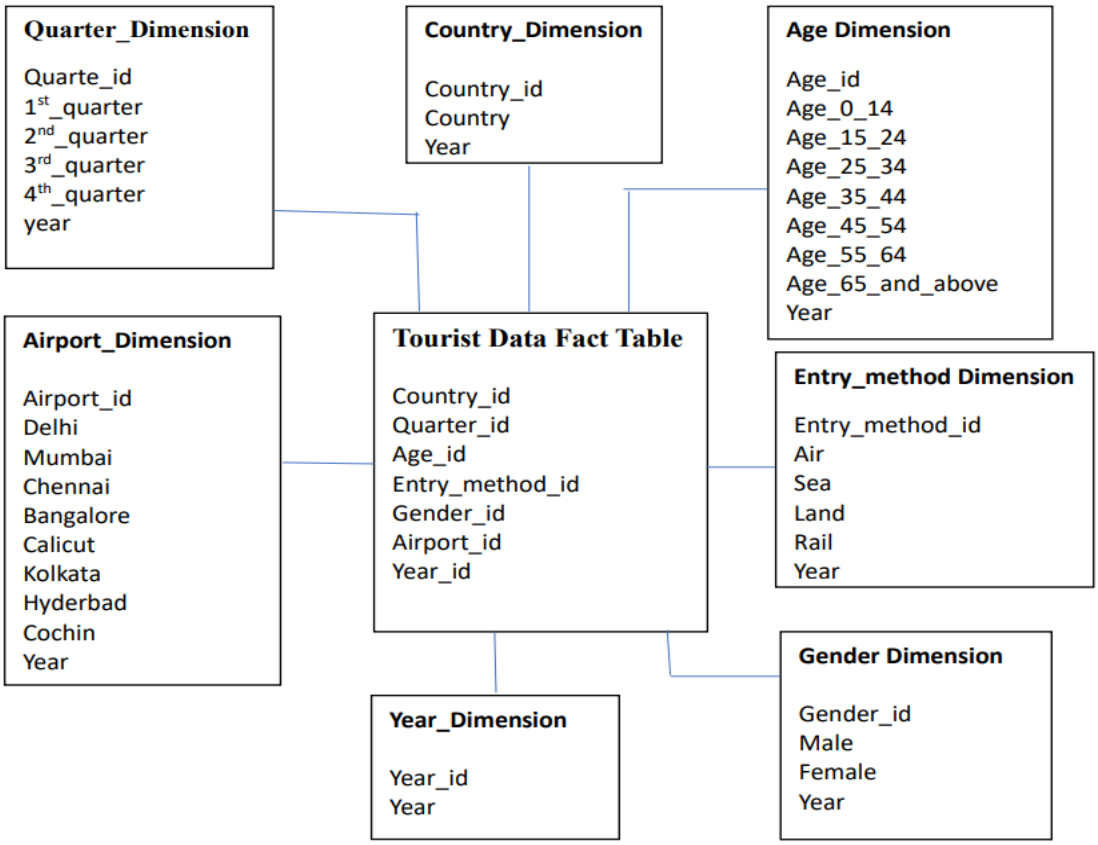

Figure 2: Star Schema

A star schema is a type of data modeling technique used in data warehousing to represent data in a structured and intuitive way. In a star schema, data is organized into a central fact table that contains the measures of interest, surrounded by dimension tables that describe the attributes of the measures.

The fact table in a star schema contains the measures or metrics that are of interest to the user or organization. For example, in a Tourist data warehouse, the fact table might contain country_id, quarter_id, and age_id so on. Each record in the fact table represents a specific event.

The dimension tables in a star schema contain the descriptive attributes of the measures in the fact table. These attributes are used to slice and dice the data in the fact table, allowing users to analyze the data from different perspectives. For example, in a Tourist data warehouse, the dimension tables might include Age, Entry_method, Airport, Year, Country, Quarter and Gender. In a star schema, each dimension table is joined to the fact table through a foreign key relationship. This allows users to query the data in the fact table using attributes from the dimension tables.

2. Architecture

2.1 Architecture Diagram

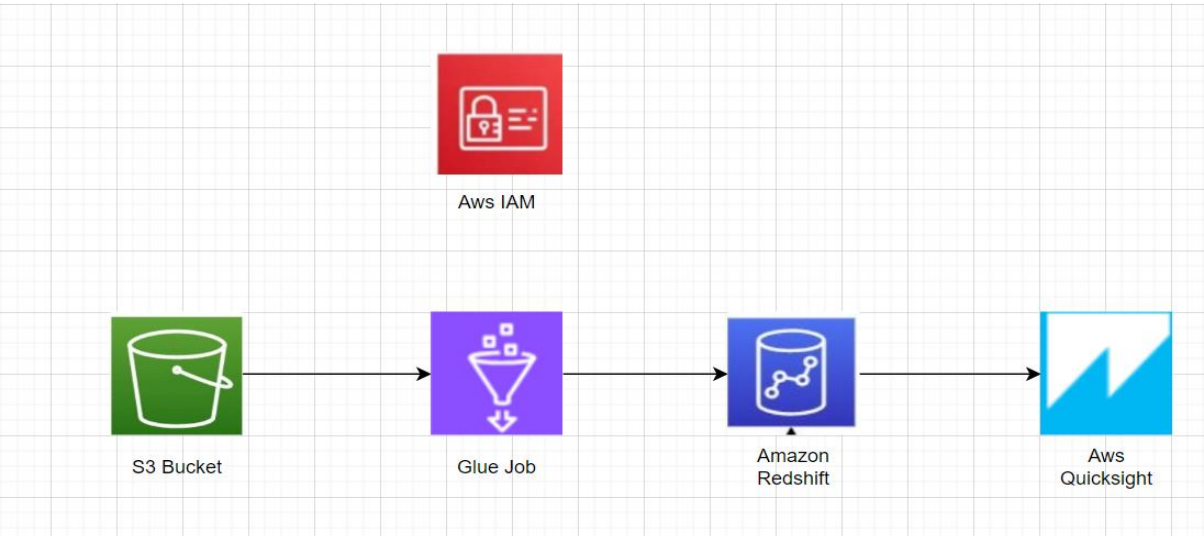

Figure 2.1 : Architecture Diagram

3. Project Steps

3.1 Creating S3 Bucket on AWS

An S3 bucket named “gen-garage-poc” has been created. The folder named “tourism2” was created to store files.

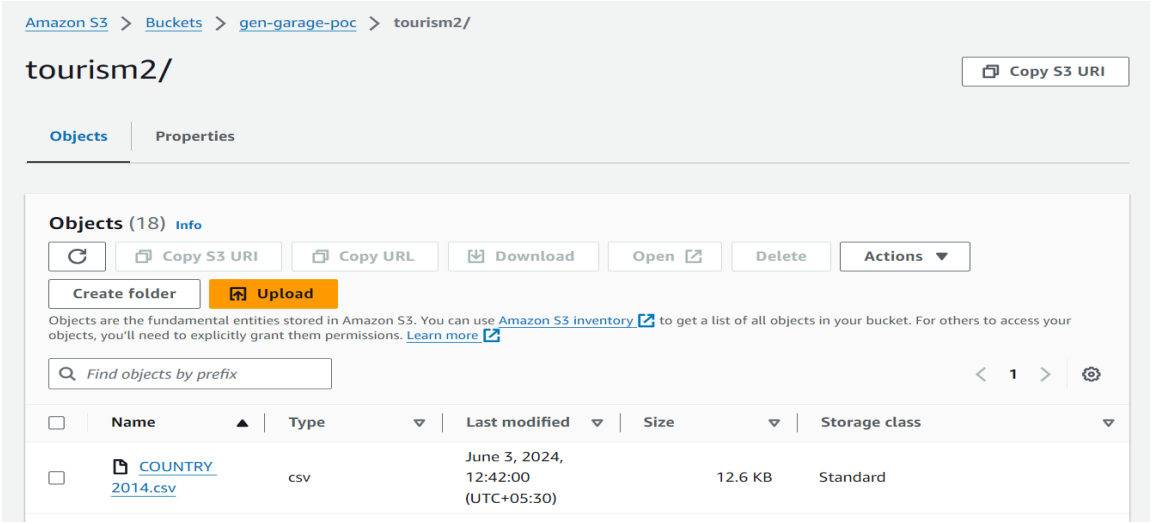

Figure 3.1: S3 Bucket and folders in the bucket

3.2 Attaching Policies to IAM user

Attached permissions and policies to provide access to run the Glue.

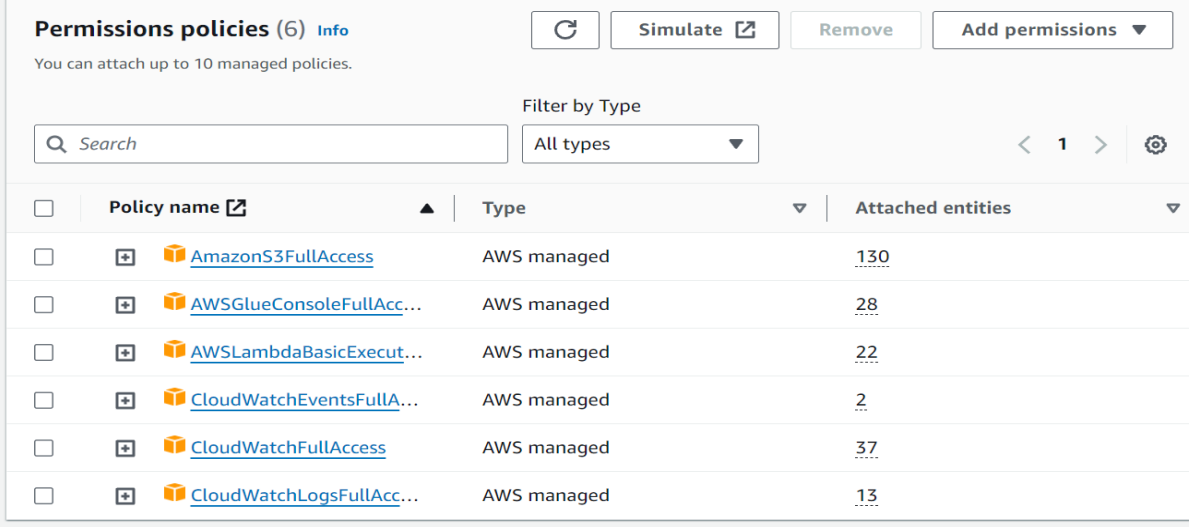

Figure 3.2: Permissions Policies attached to IAM user.

3.3 Creating workflow

A Glue job named “tur_mum” and “Top1_country_mum” has been configured. We Added 11 attributes to the workflow path in AWS Glue.

Workflow Name: Tourist_final_trigger1
Purspose : Workflow ensures that tasks are executed in the correct sequence and dependencies between tasks are managed properly. By defining workflows, we can automate the execution of our ETL processes. This reduces manual intervention and the likelihood of errors that can occur with manual execution. We can schedule workflows to run at specific times or intervals, ensuring that data processing tasks are executed according to business needs or data availability.

Attributes Added:

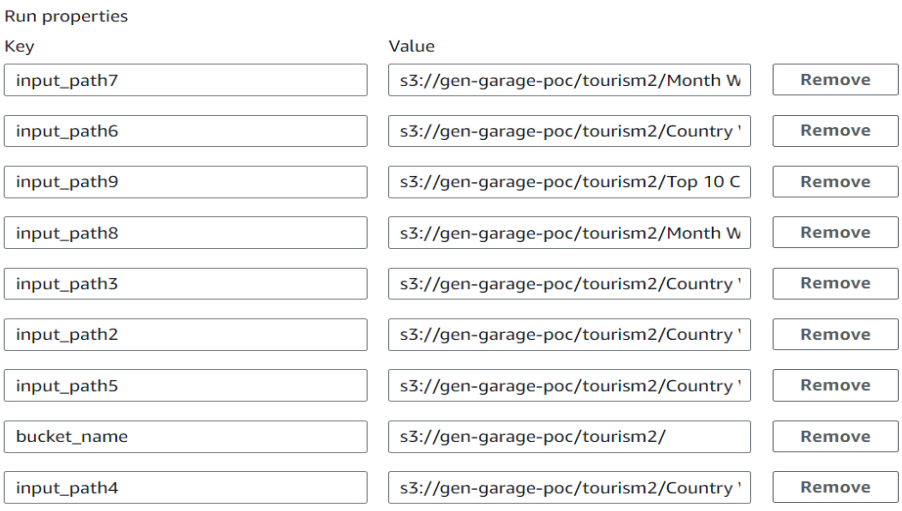

Figure 3.3.1:Adding attributes in workflow

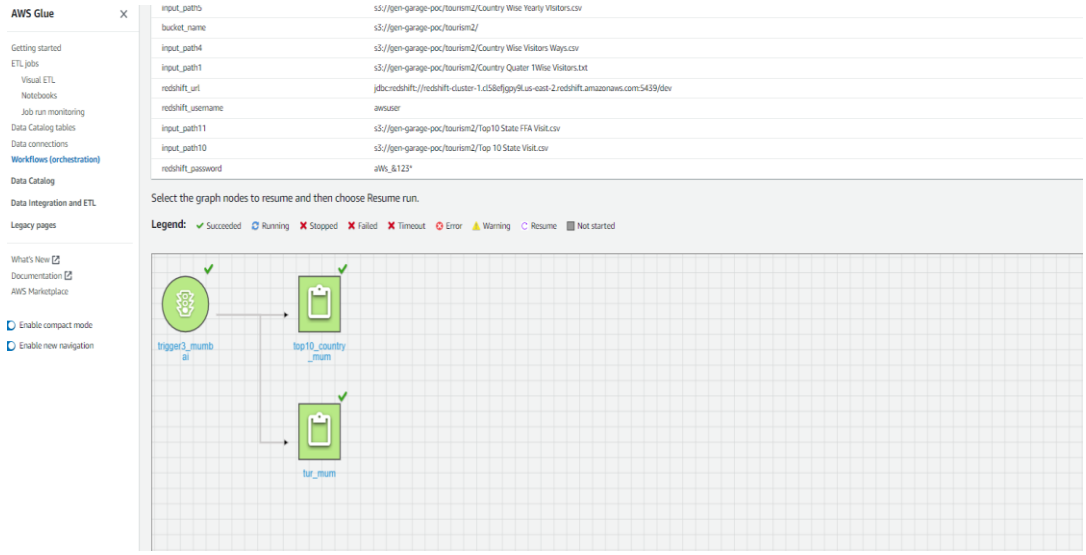

Figure 3.3.2: Graph nodes in workflow

```
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
import boto3
sc = SparkContext.getOrCreate()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
glue_client = boto3.client('glue')
glue_args = getResolvedOptions(sys.argv, ['WORKFLOW_NAME', 'WORKFLOW
workflow_args = glue_client.get_workflow_run_properties(Name=glue_ar

input_path1 = workflow_args['input_path1']
input_path2 = workflow_args['input_path2']
input_path3 = workflow_args['input_path3']
input_path4 = workflow_args['input_path4']
input_path5 = workflow_args['input_path5']
input_path6 = workflow_args['input_path6']
bucket_name = workflow_args['bucket_name']
redshift_url = workflow_args['redshift_url']
redshift_username =  workflow_args['redshift_username']
```

Figure 3.3.3: Added workflow path in Glue

```
# Reading data from S3
dyf1 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path1]},
    format="csv",
    format_options={"withHeader": True}
)
df1 = dyf1.toDF()
df1.show()
dyf2 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path2]},
    format="csv",
    format_options={"withHeader": True}
)
df2 = dyf2.toDF()
df2.show()
dyf3 = glueContext.create_dynamic_frame_from_options(
```

Figure 3.3.4: Reading data from s3

```
# Adding Country_ID column to each DataFrame
df1 = df1.withColumn("Country_ID", monotonically_increasing_id() + 100)
df2 = df2.withColumn("Country_ID", monotonically_increasing_id() + 100)
df3 = df3.withColumn("Country_ID", monotonically_increasing_id() + 100)
df4 = df4.withColumn("Country_ID", monotonically_increasing_id() + 100)
df5 = df5.withColumn("Country_ID", monotonically_increasing_id() + 100)
df6 = df6.withColumn("Country_ID", monotonically_increasing_id() + 100)
# Joining DataFrames on Country_ID
joined_df = df1.join(df2, on="Country_ID", how="inner") \
               .join(df3, on="Country_ID", how="inner") \
               .join(df4, on="Country_ID", how="inner") \
               .join(df5, on="Country_ID", how="inner") \
               .join(df6, on="Country_ID", how="inner")
# Initialize empty DataFrames for each dimension table
df_con_id = None
df_quarter = None
df_age = None
df_entry_method = None
df_gender = None
df_airport = None
df_year = None
```

Figure 3.3.5: Generating new Country_ID's for all the data files and joining all the files based on country_id

```
# Processing each year
for y in range(2014, 2021):
    df_year_temp = joined_df.withColumn("year", lit(y))
    df_year_temp.createOrReplaceTempView("rename_df_year")
    query = f"""
        SELECT
            `Country_ID`,
            `Country`,
            `year`,
            round((`{y} AIR` / 100) * `{y}`, 0) AS AIR,
            round((`{y} SEA` / 100) * `{y}`, 0) AS SEA,
            round((`{y} RAIL` / 100) * `{y}`, 0) AS RAIL,
            round((`{y} LAND` / 100) * `{y}`, 0) AS LAND,
            round((`{y} Delhi (Airport)` / 100) * `{y}`, 0) AS DELHI,
            round((`{y} Mumbai (Airport)` / 100) * `{y}`, 0) AS MUMBAI,
            round((`{y} Chennai (Airport)` / 100) * `{y}`, 0) AS CHENNAI,
            round((`{y} Calicut (Airport)` / 100) * `{y}`, 0) AS CALICUT,
            round((`{y} Benguluru (Airport)` / 100) * `{y}`, 0) AS BENGULURU,
            round((`{y} Kolkata (Airport)` / 100) * `{y}`, 0) AS KOLKATA,
            round((`{y} Hyderabad (Airport)` / 100) * `{y}`, 0) AS HYDERABAD,
```

Figure 3.3.6: Converting percentage into numbers for all years and renaming attributes

```
result = spark.sql(query)
columns = result.columns
null_df = result.select([when(col(c).isNull(), 0).otherwise(col(c)).alias(c) for c in columns])
final_df = null_df.orderBy("Country_ID")
```

Figure 3.3.7: replacing all null values with zero.

```
# Create dimension tables for each year
temp_df_con_id = final_df.select("Country_ID", "Country").orderBy("Country_ID")
temp_df_quarter = final_df.select(
    (F.monotonically_increasing_id() + 100).alias("quarter_id"), "year",
    "1st_quarter", "2nd_quarter", "3rd_quarter", "4th_quarter"
).orderBy("quarter_id")
temp_df_age = final_df.select(
    (F.monotonically_increasing_id() + 100).alias("age_id"), "year",
    "age_0_14", "age_15_24", "age_25_34", "age_35_44", "age_45_54", "age_55_64", "age_65_and_above"
).orderBy("age_id")
temp_df_entry_method = final_df.select(
    (F.monotonically_increasing_id() + 100).alias("entry_method_id"), "year",
    "AIR", "SEA", "RAIL", "LAND"
).orderBy("entry_method_id")
temp_df_gender = final_df.select(
    (F.monotonically_increasing_id() + 100).alias("gender_id"), "year",
    "MALE", "FEMALE"
).orderBy("gender_id")
temp_df_airport = final_df.select(
    (F.monotonically_increasing_id() + 100).alias("airport_id"), "year",
    "DELHI", "MUMBAI", "CHENNAI", "CALICUT",
    "BENGULURU", "KOLKATA", "HYDERABAD", "COCHIN"
).orderBy("airport_id")
temp_df_year = final_df.select(
    (F.monotonically_increasing_id() + 100).alias("year_id"), F.lit(y).alias("year")
).orderBy("year_id")
```

Figure 3.3.8:Creating fact and Dimension Table

```
    # Combine dimension tables for all years
    if df_con_id is None:
        df_con_id = temp_df_con_id
        df_quarter = temp_df_quarter
        df_age = temp_df_age
        df_entry_method = temp_df_entry_method
        df_gender = temp_df_gender
        df_airport = temp_df_airport
        df_year = temp_df_year
    else:
        df_con_id = df_con_id.union(temp_df_con_id)
        df_quarter = df_quarter.union(temp_df_quarter)
        df_age = df_age.union(temp_df_age)
        df_entry_method = df_entry_method.union(temp_df_entry_method)
        df_gender = df_gender.union(temp_df_gender)
        df_airport = df_airport.union(temp_df_airport)
        df_year = df_year.union(temp_df_year)
# Function to add a unique ID starting from 100
def add_unique_id(df, id_name):
    window_spec = Window.orderBy(monotonically_increasing_id())
    return df.withColumn(id_name, (F.row_number().over(window_spec) + 99).cast("integer"))
# Add unique IDs to dimension tables
df_con_id = add_unique_id(df_con_id, "Country_ID")
df_quarter = add_unique_id(df_quarter, "quarter_id")
df_age = add_unique_id(df_age, "age_id")
df_entry_method = add_unique_id(df_entry_method, "entry_method_id")
df_gender = add_unique_id(df_gender, "gender_id")
df_airport = add_unique_id(df_airport, "airport_id")
df_year = add_unique_id(df_year, "year_id")
```

Figure 3.3.9 : Combine all dimensional table

```
# List of combined dimension tables and their corresponding DataFrames
combined_dim_tables = {
    'country_dim_2': df_con_id,
    'quarter_dim_2': df_quarter,
    'age_dim_2': df_age,
    'entrymethod_dim_2': df_entry_method,
    'gender_dim_2': df_gender,
    'airport_dim_2': df_airport,
    'year_dim_2': df_year
}
# Redshift connection details
redshift_url = "jdbc:redshift://redshift-cluster-1.cl58efjgpy9l.us-east-2.redshift.amazonaws.com:5439/dev"
redshift_username = "awsuser"
redshift_password = "aWs_&123*"
# Write each combined dimension table to Redshift
for table_name, table_df in combined_dim_tables.items():
    table_df.write \
        .format("jdbc") \
        .option("url", redshift_url) \
        .option("dbtable", table_name) \
        .option("user", redshift_username) \
        .option("password", redshift_password) \
        .mode("overwrite") \
        .save()
    # table_df.show()
```

Figure 3.3.10 : Loading dimensional table into redshift

```
Country_FFA = spark.read.format("csv").option("header","true").load(input_path9)
# Country_FFA.show()
top_country_cols = [f"top{i}_country" for i in range(1, 11)]
top_ffa_cols = [f"top{i}_ftas" for i in range(1, 11)]

# Create an empty DataFrame to hold the transformed data
transformed_data = None

# Loop through each top country and FFA column to create the new rows
for i in range(10):
    temp_df = Country_FFA.select(
        col("year").cast("int").alias("year"),
        col(top_country_cols[i]).alias("country"),
        col(top_ffa_cols[i]).cast("int").alias("ffa")
    )

    if transformed_data is None:
        transformed_data = temp_df
    else:
        transformed_data = transformed_data.union(temp_df)

# Define a window specification to partition by year and order by FFA
window_spec = Window.partitionBy("year").orderBy("ffa")

# Add an ID column that resets for each year
transformed_data = transformed_data.withColumn("id", row_number().over(window_spec))
```

Figure 3.3.11：Unpivoting Data

```
redshift_url = "jdbc:redshift://redshift-cluster-1.cl58efjgpy9l.us-east-2.redshift.amazonaws.com:5439/dev"
redshift_username = "awsuser"
redshift_password = "aWs_&123*"
redshift_table = "Top10_State_FFA_Visit_2"
transformed_data3.write \
    .format("jdbc") \
    .option("url", redshift_url) \
    .option("dbtable", redshift_table) \
    .option("user", redshift_username) \
    .option("password", redshift_password) \
    .mode("append") \
    .save()
```

Figure 3.3.12 : Loading data into redshift

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Country o	2014 1st c	2014 2nd	2014 3rd c	2014 4th c	2015 1st c	2015 2nd	2015 3rd c	2015 4th c	2016 1st c	2016 2nd	2016 3rd c	2016 4th c	2017 1st c	2017
2	Canada	33.1	14.5	15.7	36.7	34.2	14.5	15.9	35.4	34.5	14.1	16.5	34.9	33.9	1
3	United Sta	25.7	22	20.6	31.7	26.4	22.1	19.6	31.9	25.7	21.2	20.8	32.3	25.3	2
4	Argentina	46.8	15.6	13.9	23.7	40.6	17.5	14.6	27.3	41.7	14.1	16.2	28	44.3	1
5	Brazil	31	18.8	18.7	31.5	34.8	19	19.6	26.6	29.7	16.7	19.7	33.9	31.9	1
6	Mexico	23.6	20.3	26.5	29.6	28.1	21.3	23.4	27.2	25.4	17.9	23.4	33.3	24.3	2
7	Austria	33.1	17.1	20.5	29.3	33.6	16.9	21.6	27.9	34.4	15.6	20.5	29.5	33.2	1
8	Belgium	28.8	18.5	22.4	30.3	28.5	18.1	22.6	30.8	30.2	16	22.7	31.2	27.2	1
9	Denmark	40	16	16.2	27.8	38.1	15.7	17.3	28.9	38	15	17.4	29.7	36.3	1
10	Finland	38.6	17.5	13.5	30.4	39.1	16.1	12.7	32.1	40.2	15.5	12.7	31.6		
11	France	32.3	17.8	22.4	27.5	33.2	17.6	22.1	27.1	32.6	17.2	22	28.2	33.3	1
12	Germany	32.1	17.6	19.9	30.4	33.6	16.6	19.4	30.4	32.9	16.2	19.1	31.8		
13	Greece	28.9	18.4	22.7	30	28.9	20.5	19.6	31	27.3	18.2	21.2	33.3	27.3	1
14	Ireland	26.1	23.7	21.6	28.6	26.3	22.1	22.4	29.2	27.5	21.1	22.4	29	25.2	2
15	Italy	30.2	16.4	23.6	29.8	29.3	16.6	23.1	31	29.1	15.9	23.1	31.9	28.6	1

Country Quater Wise Visitors

Figure 3.3.13 : Original Data

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Country II	Country	year	AIR	SEA	RAIL	LAND	DELHI	MUMBAI	CHENNAI	CALICUT	BENGULU	KOLKATA	HYDERAB/	COCHIN	REMAININ
2	100	Canada	2016	314384	1586	0	1269	181143	57420	18083	0	13641	5076	9200	7931	24745
3	101	U.S.A	2016	1290454	3891	0	2594	400754	294405	108943	0	133585	29830	124506	58362	146554
4	102	Argentina	2016	12123	98	0	86	7914	1452	505	0	1120	455	135	172	555
5	103	Brazil	2016	21076	128	0	85	10815	3811	1426	21	1937	617	617	426	1619
6	104	Mexico	2016	15124	502	0	63	9335	2228	941	0	988	361	314	235	1287
7	105	Austria	2016	32196	761	0	132	15618	5063	1787	33	2316	695	364	3143	4070
8	106	Belgium	2016	37429	456	0	76	14804	12375	3189	0	2505	607	759	797	2924
9	107	Denmark	2016	24396	247	0	49	8873	5561	2793	0	2645	519	395	742	3189
10	108	Finland	2016	18224	73	0	73	10747	2480	808	18	735	129	0	331	0
11	109	France	2016	237036	1194	0	477	93573	52038	47503	0	23632	2864	2387	5968	10742
12	110	Germany	2016	249706	15424	0	532	99191	54515	22072	0	31645	5053	3723	10637	39092
13	111	Greece	2016	8541	461	0	45	3538	1828	615	0	715	308	317	398	1329
14	112	Ireland	2016	36221	146	0	36	10385	6085	2514	0	3571	1494	1786	6778	3827
15	113	Italy	2016	94081	1050	0	286	44655	16793	7920	95	6107	2767	1718	2958	12404
16	114	Netherlan	2016	71050	503	0	216	35345	17098	4239	0	6753	1437	1724	1221	4023

COUNTRY 2016

Figure 3.3.14 : Data after transformation(replacing blank with zero and converting percentage into values)

3.4 Load data to Redshift

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud. Redshift is designed for high performance and scalability, allowing for fast querying and analysis of data.

Cluster:

A Redshift cluster is a group of nodes that work together to store and manage data in a Redshift data warehouse. Each cluster is composed of one or more compute nodes, which are responsible for storing and processing data, and one or more leader nodes, which handle query coordination and management.

Cluster identifier: redshift-cluster-1

Node type: dc2.large

Number of nodes: 1

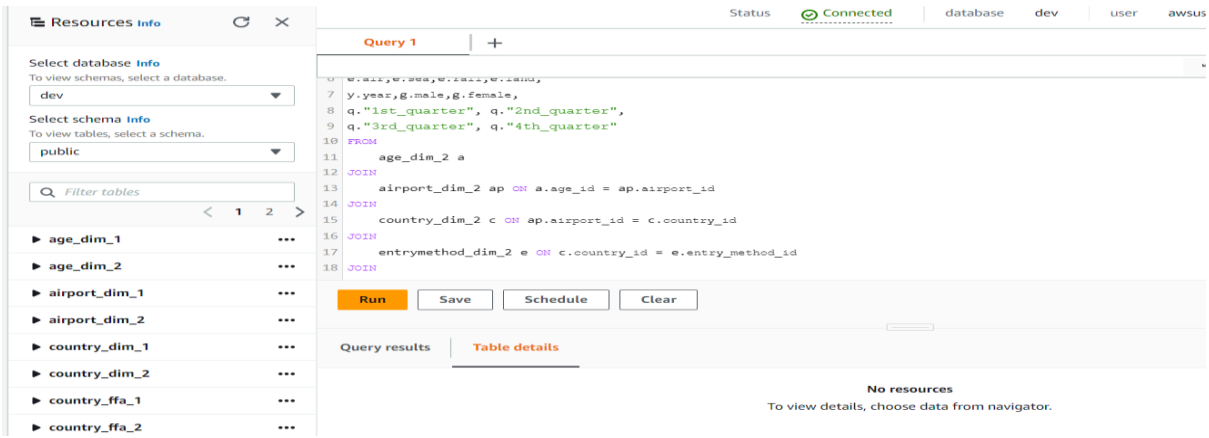

Figure 3.4: Tables are created in Redshift

4. Output

Creating an Interactive Dashboard in QuickSight with Redshift:

1. **Connecting to Redshift:**

- Manually connect Amazon QuickSight to your Amazon Redshift data source.

2. **Writing SQL Queries:**

- Begin by writing SQL queries to select and join the necessary columns from your Redshift database. This involves joining multiple tables to gather all required data.

3. **Creating Visualizations:**

- Utilize the data obtained from the queries to create various visualizations in QuickSight. These visualizations can include charts, graphs, and other visual elements that represent the data in an interactive and insightful manner.

4. **Building the Dashboard:**

- Assemble the visualizations into an interactive dashboard. Ensure the dashboard is user-friendly and provides clear insights by arranging the visual elements effectively.

5. **Adding Filters:**

- Enhance the interactivity of the dashboard by adding filters. Filters allow users to dynamically adjust the data being displayed based on specific criteria, making the dashboard more versatile and useful for different analysis needs.

By following above steps, we effectively created a comprehensive and interactive dashboard in QuickSight using data from Amazon Redshift, enabling better data visualization and analysis.

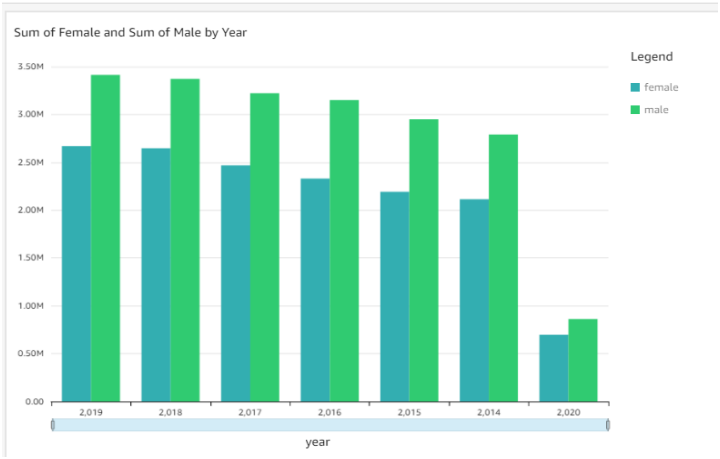

Figure 4.1: Visual for Sum of Female and Sum of Male by Year

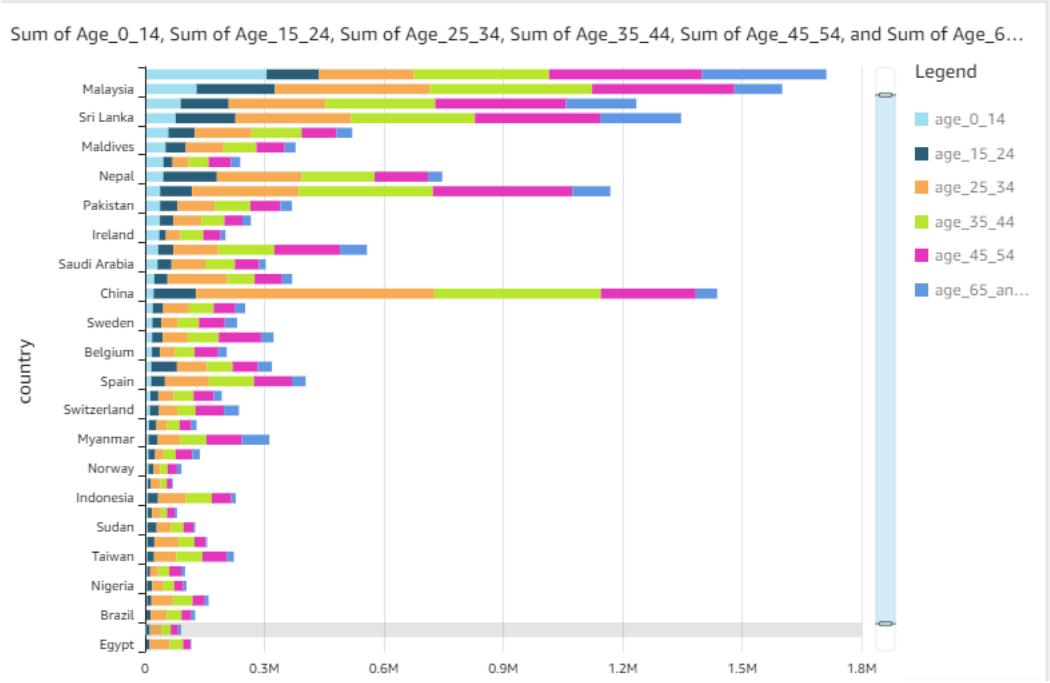

Figure 4.2: Visual for Sum of all the Ages by Country

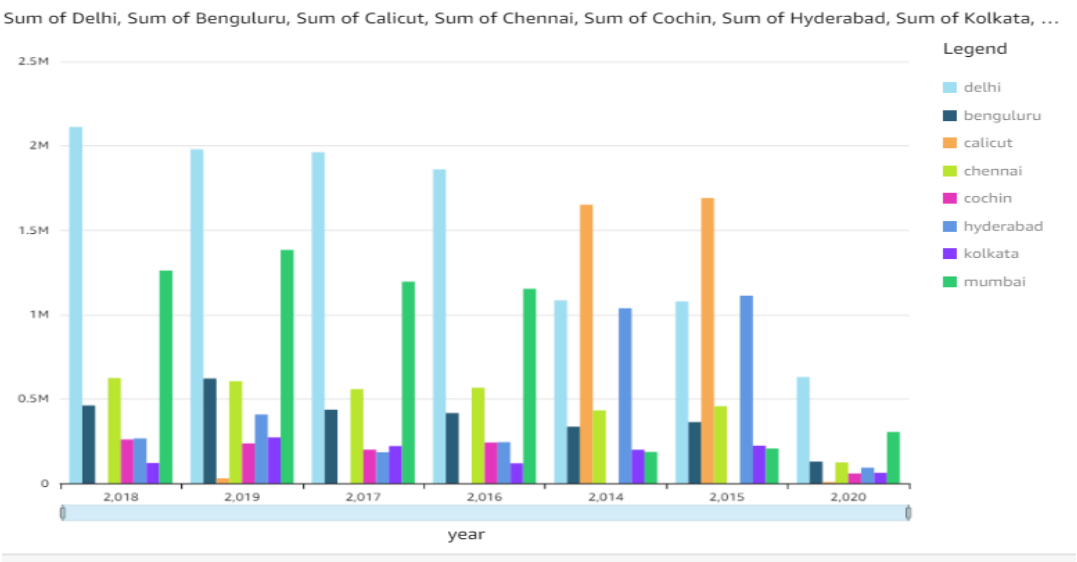

Figure 4.3: Visual for Sum of all the People entered through Airport by Year

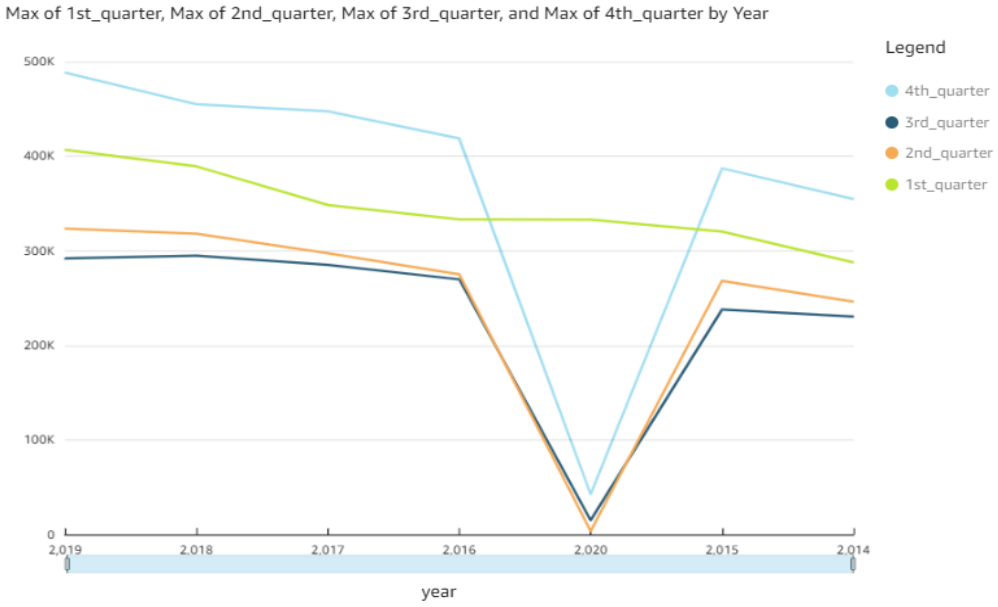

Figure 4.4: Visual for Maximum of all the quarter by Year

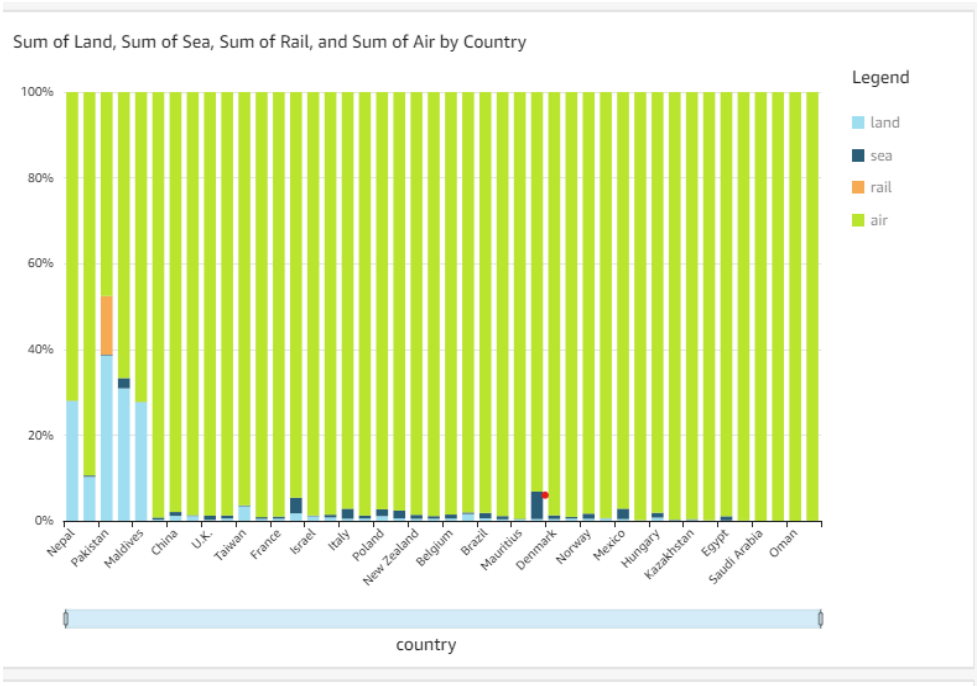

Figure 4.5: Visual for Sum of all the entry method by Country

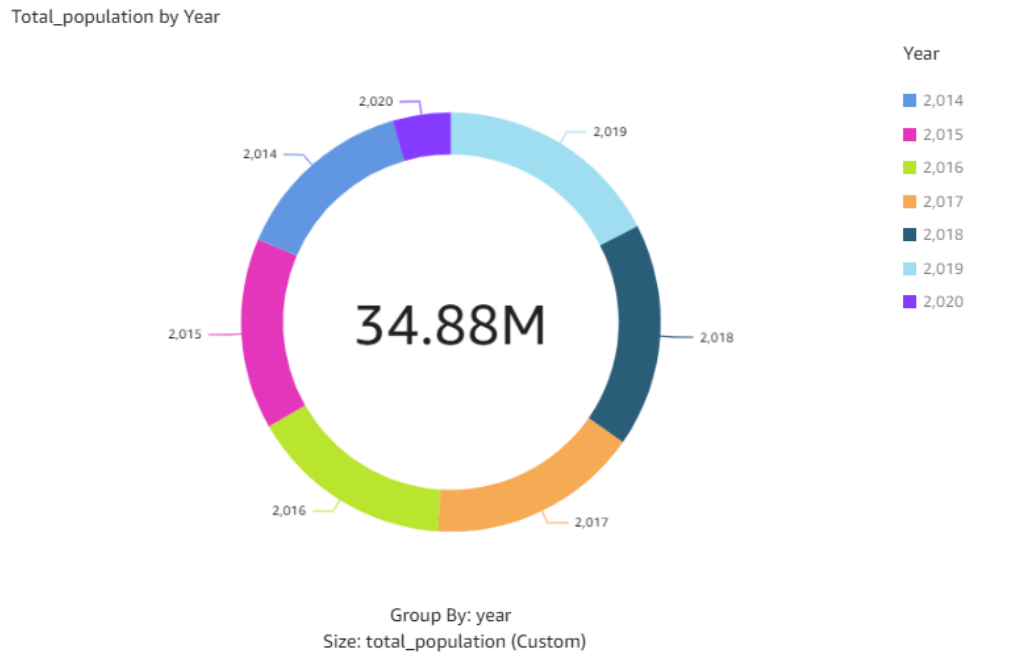

Figure 4.6: Visual for Total population by Year

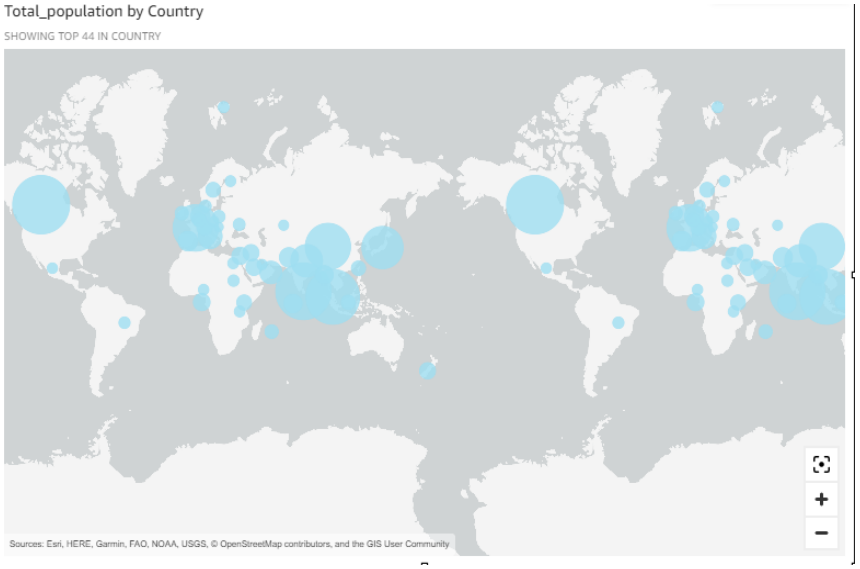

Figure 4.7: Visual for Total Population by Country

5. Code

5.1 All country files data is extracted, transformed and loaded into Redshift

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
import boto3
sc = SparkContext.getOrCreate()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
glue_client = boto3.client('glue')
glue_args = getResolvedOptions(sys.argv, ['WORKFLOW_NAME',
'WORKFLOW_RUN_ID'])
workflow_args =
glue_client.get_workflow_run_properties(Name=glue_args['WORKFLOW_NAME'],
RunId=glue_args['WORKFLOW_RUN_ID'])["RunProperties"]

input_path1 = workflow_args['input_path1']
input_path2 = workflow_args['input_path2']
input_path3 = workflow_args['input_path3']
input_path4 = workflow_args['input_path4']
input_path5 = workflow_args['input_path5']
input_path6 = workflow_args['input_path6']
```

```
bucket_name = workflow_args['bucket_name']
redshift_url = workflow_args['redshift_url']
redshift_username =  workflow_args['redshift_username']
redshift_password = workflow_args['redshift_password']

from pyspark.sql.functions import monotonically_increasing_id, col, round as round_,
when, lit
from pyspark.sql import functions as F
from pyspark.sql.window import Window



# Reading data from S3
dyf1 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path1]},
    format="csv",
    format_options={"withHeader": True}
)
df1 = dyf1.toDF()
df1.show()
dyf2 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path2]},
    format="csv",
    format_options={"withHeader": True}
)
df2 = dyf2.toDF()
df2.show()
dyf3 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path3]},
    format="csv",
    format_options={"withHeader": True}
)
df3 = dyf3.toDF()
df3.show()
dyf4 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path4]},
    format="csv",
    format_options={"withHeader": True}
)
df4 = dyf4.toDF()
df4.show()
dyf5 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path5]},
    format="csv",
```

```
    format_options={"withHeader": True}
)
df5 = dyf5.toDF()
df5.show()
dyf6 = glueContext.create_dynamic_frame_from_options(
    connection_type="s3",
    connection_options={"paths": [input_path6]},
    format="csv",
    format_options={"withHeader": True}
)
df6 = dyf6.toDF()
df6.show()
# Adding Country_ID column to each DataFrame
df1 = df1.withColumn("Country_ID", monotonically_increasing_id() + 100)
df2 = df2.withColumn("Country_ID", monotonically_increasing_id() + 100)
df3 = df3.withColumn("Country_ID", monotonically_increasing_id() + 100)
df4 = df4.withColumn("Country_ID", monotonically_increasing_id() + 100)
df5 = df5.withColumn("Country_ID", monotonically_increasing_id() + 100)
df6 = df6.withColumn("Country_ID", monotonically_increasing_id() + 100)
# Joining DataFrames on Country_ID
joined_df = df1.join(df2, on="Country_ID", how="inner") \
            .join(df3, on="Country_ID", how="inner") \
            .join(df4, on="Country_ID", how="inner") \
            .join(df5, on="Country_ID", how="inner") \
            .join(df6, on="Country_ID", how="inner")
# Initialize empty DataFrames for each dimension table
df_con_id = None
df_quarter = None
df_age = None
df_entry_method = None
df_gender = None
df_airport = None
df_year = None
# Processing each year
for y in range(2014, 2021):
    df_year_temp = joined_df.withColumn("year", lit(y))
    df_year_temp.createOrReplaceTempView("rename_df_year")
    query = f"""
        SELECT
            `Country_ID`,
            `Country`,
            `year`,
            round((`{y} AIR` / 100) * `{y}`, 0) AS AIR,
            round((`{y} SEA` / 100) * `{y}`, 0) AS SEA,
            round((`{y} RAIL` / 100) * `{y}`, 0) AS RAIL,
            round((`{y} LAND` / 100) * `{y}`, 0) AS LAND,
            round((`{y} Delhi (Airport)` / 100) * `{y}`, 0) AS DELHI,
            round((`{y} Mumbai (Airport)` / 100) * `{y}`, 0) AS MUMBAI,
            round((`{y} Chennai (Airport)` / 100) * `{y}`, 0) AS CHENNAI,
```

```
            round((`{y} Calicut (Airport)` / 100) * `{y}`, 0) AS CALICUT,
            round((`{y} Benguluru (Airport)` / 100) * `{y}`, 0) AS BENGULURU,
            round((`{y} Kolkata (Airport)` / 100) * `{y}`, 0) AS KOLKATA,
            round((`{y} Hyderabad (Airport)` / 100) * `{y}`, 0) AS HYDERABAD,
            round((`{y} Cochin (Airport)` / 100) * `{y}`, 0) AS COCHIN,
            round(`{y}` - (
               round((`{y} Delhi (Airport)` / 100) * `{y}`, 0) +
               round((`{y} Mumbai (Airport)` / 100) * `{y}`, 0) +
               round((`{y} Chennai (Airport)` / 100) * `{y}`, 0) +
               round((`{y} Calicut (Airport)` / 100) * `{y}`, 0) +
               round((`{y} Benguluru (Airport)` / 100) * `{y}`, 0) +
               round((`{y} Kolkata (Airport)` / 100) * `{y}`, 0) +
               round((`{y} Hyderabad (Airport)` / 100) * `{y}`, 0) +
               round((`{y} Cochin (Airport)` / 100) * `{y}`, 0)
            ), 0) AS `REMAINING AIRPORTS`,
            round((`{y} 1st quarter (Jan-March)` / 100) * `{y}`, 0) AS 1st_quarter,
            round((`{y} 2nd quarter (Apr-June)` / 100) * `{y}`, 0) AS 2nd_quarter,
            round((`{y} 3rd quarter (July-Sep)` / 100) * `{y}`, 0) AS 3rd_quarter,
            round((`{y} 4th quarter (Oct-Dec)` / 100) * `{y}`, 0) AS 4th_quarter,
            round((`{y} 0-14` / 100) * `{y}`, 0) AS age_0_14,
            round((`{y} 15-24` / 100) * `{y}`, 0) AS age_15_24,
            round((`{y} 25-34` / 100) * `{y}`, 0) AS age_25_34,
            round((`{y} 35-44` / 100) * `{y}`, 0) AS age_35_44,
            round((`{y} 45-54` / 100) * `{y}`, 0) AS age_45_54,
            round((`{y} 55-64` / 100) * `{y}`, 0) AS age_55_64,
            round((`{y} 65 AND ABOVE` / 100) * `{y}`, 0) AS age_65_and_above,
            round((`{y} Male` / 100) * `{y}`, 0) AS MALE,
            round((`{y} Female` / 100) * `{y}`, 0) AS FEMALE,
            `{y}`
         FROM
            rename_df_year
         WHERE
            `{y}` IS NOT NULL
      """
      result = spark.sql(query)
      columns = result.columns
      null_df = result.select([when(col(c).isNull(), 0).otherwise(col(c)).alias(c) for c in
columns])
      final_df = null_df.orderBy("Country_ID")
      # Create dimension tables for each year
      temp_df_con_id = final_df.select("Country_ID",
"Country").orderBy("Country_ID")
      temp_df_quarter = final_df.select(
         (F.monotonically_increasing_id() + 100).alias("quarter_id"), "year",
         "1st_quarter", "2nd_quarter", "3rd_quarter", "4th_quarter"
      ).orderBy("quarter_id")
      temp_df_age = final_df.select(
         (F.monotonically_increasing_id() + 100).alias("age_id"), "year",
```

```
        "age_0_14", "age_15_24", "age_25_34", "age_35_44", "age_45_54",
"age_55_64", "age_65_and_above"
    ).orderBy("age_id")
    temp_df_entry_method = final_df.select(
        (F.monotonically_increasing_id() + 100).alias("entry_method_id"), "year",
        "AIR", "SEA", "RAIL", "LAND"
    ).orderBy("entry_method_id")
    temp_df_gender = final_df.select(
        (F.monotonically_increasing_id() + 100).alias("gender_id"), "year",
        "MALE", "FEMALE"
    ).orderBy("gender_id")
    temp_df_airport = final_df.select(
        (F.monotonically_increasing_id() + 100).alias("airport_id"), "year",
        "DELHI", "MUMBAI", "CHENNAI", "CALICUT",
        "BENGULURU", "KOLKATA", "HYDERABAD", "COCHIN"
    ).orderBy("airport_id")
    temp_df_year = final_df.select(
        (F.monotonically_increasing_id() + 100).alias("year_id"), F.lit(y).alias("year")
    ).orderBy("year_id")
    # Combine dimension tables for all years
    if df_con_id is None:
        df_con_id = temp_df_con_id
        df_quarter = temp_df_quarter
        df_age = temp_df_age
        df_entry_method = temp_df_entry_method
        df_gender = temp_df_gender
        df_airport = temp_df_airport
        df_year = temp_df_year
    else:
        df_con_id = df_con_id.union(temp_df_con_id)
        df_quarter = df_quarter.union(temp_df_quarter)
        df_age = df_age.union(temp_df_age)
        df_entry_method = df_entry_method.union(temp_df_entry_method)
        df_gender = df_gender.union(temp_df_gender)
        df_airport = df_airport.union(temp_df_airport)
        df_year = df_year.union(temp_df_year)
# Function to add a unique ID starting from 100
def add_unique_id(df, id_name):
    window_spec = Window.orderBy(monotonically_increasing_id())
    return df.withColumn(id_name, (F.row_number().over(window_spec) +
99).cast("integer"))
# Add unique IDs to dimension tables
df_con_id = add_unique_id(df_con_id, "Country_ID")
df_quarter = add_unique_id(df_quarter, "quarter_id")
df_age = add_unique_id(df_age, "age_id")
df_entry_method = add_unique_id(df_entry_method, "entry_method_id")
df_gender = add_unique_id(df_gender, "gender_id")
df_airport = add_unique_id(df_airport, "airport_id")
df_year = add_unique_id(df_year, "year_id")
```

```
# List of combined dimension tables and their corresponding DataFrames
combined_dim_tables = {
    'country_dim_2': df_con_id,
    'quarter_dim_2': df_quarter,
    'age_dim_2': df_age,
    'entrymethod_dim_2': df_entry_method,
    'gender_dim_2': df_gender,
    'airport_dim_2': df_airport,
    'year_dim_2': df_year
}
# Redshift connection details
redshift_url = "jdbc:redshift://redshift-cluster-1.cl58efjgpy9l.us-east-
2.redshift.amazonaws.com:5439/dev"
redshift_username = "awsuser"
redshift_password = "aWs_&123*"
# Write each combined dimension table to Redshift
for table_name, table_df in combined_dim_tables.items():
    table_df.write \
        .format("jdbc") \
        .option("url", redshift_url) \
        .option("dbtable", table_name) \
        .option("user", redshift_username) \
        .option("password", redshift_password) \
        .mode("overwrite") \
        .save()
    # table_df.show()
      job.commit()
```

5.2 All top 10 country, state and Month wise data is loaded into Redshift

```
# redshift_url = "jdbc:redshift://redshift-cluster-1.cl58efjgpy9l.us-east-
2.redshift.amazonaws.com:5439/dev"
# redshift_username = "awsuser"
# redshift_password = "aWs_&123*"
# redshift_table = "Month_Wise_FFA_2"
# melted_df_ffa.write \
#     .format("jdbc") \
#     .option("url", redshift_url) \
#     .option("dbtable", redshift_table) \
#     .option("user", redshift_username) \
#     .option("password", redshift_password) \
#     .mode("append") \
#     .save()
from pyspark.sql import functions as F
from pyspark.sql.functions import lit
from pyspark.sql.functions import monotonically_increasing_id
from pyspark.sql.functions import col, expr
from pyspark.sql import Row
```

```
Month_FFA_dollar =
spark.read.format("csv").option("header","true").load(input_path8)
# Month_FFA_dollar.show()
melted_df_dollar = Month_FFA_dollar.selectExpr("year", "stack(12, 'january',
january, 'february', february, 'march', march, 'april', april, 'may', may, 'june', june,
'july', july, 'augest', augest, 'september', september, 'october', october, 'november',
november, 'december', december) as (month, value)")
melted_df_dollar = melted_df_dollar.withColumn("year", expr("CAST(year AS
INT)"))

melted_df_dollar = melted_df_dollar.withColumn("month_id",
(monotonically_increasing_id() % 12) + 1)
# # Show the result
melted_df_dollar.show()
print(melted_df_dollar.count())
redshift_url = "jdbc:redshift://redshift-cluster-1.cl58efjgpy9l.us-east-
2.redshift.amazonaws.com:5439/dev"
redshift_username = "awsuser"
redshift_password = "aWs_&123*"
redshift_table = "Month_FFE_dollar_2"
melted_df_dollar.write \
    .format("jdbc") \
    .option("url", redshift_url) \
    .option("dbtable", redshift_table) \
    .option("user", redshift_username) \
    .option("password", redshift_password) \
    .mode("append") \
    .save()
job.commit()
```

6. Challenges Faced and Solutions

Challenges:

- Handling data with null values and inconsistence format.
- Data stored in column format that we need convert into row.
- Challenge in Creating Dimension tables

Solutions:

- Replace null values with appropriate substitutes such as zero.
- Identify and correct errors by cross-referencing with other data sources or using domain knowledge.
- A window specification window_spec3 is created to partition the data by "year".

7. Business Benefits

- **Personalized Campaigns:** By understanding visitor demographics, preferences, and behaviors, businesses can create highly targeted marketing campaigns that resonate with specific segments, increasing the effectiveness of promotional efforts.
- **Market Segmentation:** Data analysis helps in segmenting the market into distinct groups, allowing for tailored marketing strategies that cater to the unique needs and desires of each segment.
- **Customized Offerings:** Insights into tourist preferences enable businesses to tailor their products and services to meet the specific needs and expectations of different tourist groups, enhancing customer satisfaction and loyalty.
- **Service Enhancements:** Analyzing feedback and satisfaction levels helps identify areas for improvement, allowing businesses to refine their offerings and deliver superior service.
- **Informed Investments:** Data-driven insights guide businesses in making informed decisions about where to invest resources, whether in marketing, infrastructure, or new product development.
- **Risk Management:** By understanding potential risks and market fluctuations, businesses can develop strategies to mitigate risks and capitalize on opportunities.

8. Conclusion

In conclusion, this project represents a strategic embrace of cloud-based solutions, specifically AWS services, to enhance the efficiency of Tourism data and error response for organizations with diverse service landscapes. The tourist data analysis provides a wealth of information that can be leveraged to enhance the overall tourism strategy. By understanding visitor demographics, seasonal trends, spending behaviour, and satisfaction levels, stakeholders can make informed decisions that improve tourist experiences, boost economic benefits, and promote sustainable practices. These insights are invaluable for developing targeted marketing strategies, optimizing resource allocation, and ensuring the long-term success of the tourism industry. This project serves as a testament to the power of cloud technologies in optimizing operational workflows and fortifying the resilience of digital infrastructures.

9. References

[1] AWS Redshift: https://aws.amazon.com/redshift/

[2] Amazon Glue: https://aws.amazon.com/glue /

[3] Amazon IAM: https://aws.amazon.com/iam/

[4] QuickSight : https://aws.amazon.com/quicksight/