



Northeastern University

College of Engineering

IE 6750: DATA WAREHOUSING & INTEGRATION

Milestone #6

Group No. 7

Student 1: Rahul Daruka
[\(daruka.r@northeastern.edu\)](mailto:daruka.r@northeastern.edu)

Student 2: Sharvari Deshpande
[\(deshpande.sha@northeastern.edu\)](mailto:deshpande.sha@northeastern.edu)

Percentage of effort contributed by student 1: 50%

Percentage of effort contributed by student 2: 50%

Signature of student 1: Rahul Daruka

Signature of student 2: Sharvari Deshpande

Submission Date: 24th November 2024

Table of Contents

Sr No.	Topic
1.	<u>Introduction</u>
2.	<u>Problem Statement</u>
3.	<u>Objectives of the Data Pipeline</u>
4.	<u>Dataset for Analysis</u>
5.	<u>Volume and Variety in Data Sources</u>
6.	<u>Fact Tables</u>
7.	<u>Dimension Tables</u>
8.	<u>Strategic and Operational Insights for the Data Pipeline</u>
9.	<u>Pipeline Tools</u>
10.	<u>Programming Choice</u>
11.	<u>Architectural Diagram of the Data Pipeline</u>
12.	<u>Pipeline Execution</u>
13.	<u>Dashboard Creation</u>

1. Introduction:

In today's rapidly urbanizing world, the efficient management of parking resources has become a critical component of urban infrastructure. With vehicle ownership rates climbing and urban populations expanding, the strain on existing parking facilities has reached unprecedented levels. Traditional parking management systems are increasingly proving inadequate, primarily due to their reliance on outdated static data warehousing methods. These systems are unable to process and analyze data in real time, resulting in several significant issues:

- Congestion in Parking Facilities: Due to the lack of real-time data processing, traditional systems cannot effectively manage the inflow and outflow of vehicles, leading to frequent congestion. This congestion is not just a minor inconvenience; it significantly affects the daily commute of thousands, contributing to overall urban traffic congestion.
- Increased Driver Frustration: Inefficiencies in parking management directly impact user experience, manifesting as increased time spent searching for parking spots. This not only frustrates drivers but also leads to increased fuel consumption and emissions.
- Missed Revenue Opportunities: Inadequate utilization of parking spaces and inefficient pricing strategies, due to the absence of dynamic data, result in lost revenue opportunities for parking operators. This is compounded by the potential loss of customers who may seek alternative parking options due to dissatisfaction with service levels.

2. Problem Statement:

The central challenge that plagues current parking management systems is the significant latency inherent in their data handling and processing capabilities. This latency severely impedes the system's ability to make informed and timely decisions that are critical in managing the dynamic demands of urban parking. Traditional systems, typically reliant on static data warehouses, are not designed to handle the rapid fluctuations and complex variables that characterize modern urban parking scenarios. This inefficiency manifests in several critical areas impacting overall service delivery and system performance:

1. Inability to Adapt to Real-Time Demands:

Traditional parking systems fail to adapt quickly to real-time conditions due to slow data processing speeds. In a typical scenario, data collected from parking sensors, ticketing systems, and user inputs must undergo a lengthy process of extraction, loading, and analysis before actionable insights can be derived. During this delay, the actual parking situation may have altered significantly, rendering the insights outdated and ineffective by the time they are available for use.

2. Suboptimal Space Utilization:

The latency in data processing directly affects the ability of these systems to efficiently manage parking spaces. Without access to real-time occupancy data, parking operators cannot implement dynamic allocation strategies or adjust to sudden increases or decreases in demand. This often leads to scenarios where parking lots are either congested or underutilized, neither of which is optimal for maximizing usage or revenue.

3. Poor Customer Experience:

For customers, the most tangible impact of these outdated systems is the difficulty in finding available parking spaces. The lack of real-time data means that customers are often provided with inaccurate information regarding space availability. This not only leads to increased time spent searching for parking but also contributes to traffic congestion and frustration, degrading the overall user experience.

4. Missed Revenue Opportunities:

Effective revenue management in parking systems depends heavily on the ability to implement flexible pricing models that can adapt to changing demand patterns. However, without real-time data analytics, parking operators miss out on implementing dynamic pricing strategies that could capitalize on peak demand periods. Additionally, the inability to offer targeted promotions or discounts in real-time further diminishes potential revenue.

5. Integration Challenges:

Current systems often operate in silos with minimal interoperability between different data sources such as payment gateways, vehicle detection systems, and customer feedback platforms. This lack of integration complicates the data landscape, making it difficult to synthesize data from multiple sources into a coherent analytical framework. As a result, the full potential of the data remains untapped, limiting the scope and accuracy of predictive analytics.

6. Inadequate Incident and Crisis Management:

In situations where rapid response is crucial, such as managing parking during large events or resolving access disputes, the delayed data processing further exacerbates the problem. Operators lack the necessary situational awareness to manage incidents effectively, leading to operational inefficiencies and potential security risks.

The proposed solution, the ParkEasy data pipeline, aims to revolutionize how parking data is managed by facilitating the seamless integration and real-time processing of diverse data streams. By transforming data handling capabilities, ParkEasy seeks to enhance operational efficiency, improve customer service, maximize revenue potential, and streamline incident management in urban parking environments.

3. Objectives of the Data Pipeline:

The overarching objective of the SmartPark data pipeline is to transform parking management into a highly responsive, customer-centric service. Specific goals include:

- **Real-time Data Flow:** Facilitating the continuous flow of data from various sources to ensure that information regarding parking space availability and usage is always current.
- **Operational Efficiency:** Automating key management functions such as dynamic pricing adjustments, parking slot reservations, and quick incident response.
- **Enhanced Customer Experience:** Providing real-time updates to customers about parking availability through user-friendly mobile interfaces, thereby improving user interaction and satisfaction.

- **Predictive and Strategic Insights:** Employing historical data analysis in conjunction with real-time data to forecast parking demand, identify usage patterns, and guide strategic management decisions.
-

4. Dataset for Analysis:

The data population methodology for ParkEasy entails a systematic approach to creating and maintaining a robust database that can support both initial system functionalities and scalable future needs. This process is divided into two main phases: the initial manual generation of data and subsequent plans for automated data generation. This strategy ensures that while the system starts with a solid foundation of manually populated data, it is also designed to evolve through more sophisticated, automated methods that can handle increased data volume and complexity.

Phase 1: Manual Data Population

Initially, the data required for the ParkEasy system will be generated manually to set up the core structure and to ensure that all system components function correctly with realistic data sets.

1. Database Creation:

- **Tool Used:** PostgreSQL is utilized for database creation due to its robustness and ability to handle complex query operations.
- **Schema Setup:** The database schema is designed based on the relational model of the ParkEasy system which includes various entities such as parking lots, users, and transactions.
- **Table Definition:** SQL commands are crafted to define tables with appropriate constraints, data types, and indices to optimize performance and data integrity.

2. Table Structure Implementation:

- **Core Tables:** Tables such as Admin, Member, ParkingLot, ParkingSlot, Booking, Payment, and Incident are created to support essential functionalities.
- **Data Entry:** Each table is populated using SQL INSERT statements that introduce realistic but fabricated data reflecting typical operational scenarios.

3. Data Insertion:

- **Execution:** Manual insertion of data through SQL scripts ensures that all relationships and constraints among tables are respected.
- **Validation:** Regular checks are performed to validate data consistency and integrity across the system.

Phase 2: Automated Data Population (Future Strategy)

As the ParkEasy system matures and expands, the data population strategy will shift towards automation to efficiently manage larger datasets and more complex scenarios.

1. Automated Data Generation:

- **Tool:** Use of Mockaroo, a sophisticated online data generator, to create structured data in CSV format that mirrors the database schema.
- **Scope:** Generation of data for dynamic entities such as Booking, Payment, and Vehicle, which require frequent updates and high volume.

2. Scripted Data Insertion:

- **Implementation:** Development of Python and SQL scripts to automate the data insertion process.

- **Bulk Operations:** Utilization of PostgreSQL's COPY command for bulk data import from CSV files, enhancing the speed and efficiency of the data population process.
- 3. Scalability for Transactional Data:**
- **High Volume Data Handling:** Focus on generating and managing large volumes of transactional data to support extensive analytics and system testing.
 - **Real-Time Simulation:** Integration of mechanisms to simulate real-time data updates, critical for entities like booking statuses and incident management.

5. Volume and Variety in Data Sources:

The ParkEasy data pipeline is designed to manage and analyze data from multiple sources, accommodating both the volume and variety inherent in the operation of 5-10 parking lots, each containing over 50+ individual parking slots. With more than 100+ active members, the system handles a significant volume of bookings and financial transactions, necessitating a robust infrastructure that can process and analyze data efficiently. Here's a detailed look at how volume and variety are considered in the SmartPark data sources:

Volume Considerations-

The system is engineered to handle large-scale data inputs and interactions generated from day-to-day operations:

1. Granular Time Stamps:

- **Detail:** Every transaction and booking record includes precise time stamps down to the second.
- **Volume:** This results in a massive increase in data granularity, capturing every moment of operation which accumulates vast amounts of data over time.
- **Impact:** The detailed time stamps require the database to handle a higher transaction load and provide the basis for more detailed time-series analysis.

2. Extensive Logging of User Interactions:

- **Detail:** Every interaction by the 100+ members with the parking management system, including payment transactions, is logged.
- **Volume:** This leads to a large dataset comprising user actions, preferences, and feedback.
- **Impact:** Requires robust data management strategies to ensure that the system can quickly access and analyze these interactions for customer service and operational improvements.

3. High-Density Booking Data:

- **Detail:** Each parking lot supports over 50 slots, and with multiple bookings per slot per day, the dataset grows exponentially.
- **Volume:** Accumulates detailed records of bookings, cancellations, and modifications.
- **Impact:** Necessitates efficient indexing and partitioning in the database to speed up data retrieval and enhance performance during peak load times.

Variety Considerations-

1. Diverse Transaction Types:

- **Detail:** Transactions not only include payments but also fines, refunds, and adjustments.

- **Management:** The database schema includes various transaction types, each requiring specific handling, validation, and processing within the SQL environment.

2. Multiple User Roles and Access Levels:

- **Detail:** Different data views and permissions for admins, regular members, VIP members, and guest users.
- **Management:** Implementation of role-based access control within the SQL database ensures that data integrity and security are maintained while providing customized data views based on user roles.

The primary data source for the ETL process was an operational OLTP (Online Transaction Processing) database. This OLTP system contains transactional data generated manually using unput statements, including entities such as Admin, ParkingLot, ParkingSlot, Booking, Payment, Incident, and Member. This source provides transactional records that form the basis for the fact tables in the data warehouse.

Prepopulated Time Dimension:

- The time dimension in the data warehouse was prepopulated as a static reference table, providing detailed temporal attributes such as year, quarter, month, day, and time of day. This prepopulated time dimension serves as a reference that is essential for time-based analysis across the data warehouse.
- Using a prepopulated time dimension provides flexibility and efficiency, as it allows for consistent and easy querying of time-related data without needing frequent updates. The time dimension is fundamental for creating time-based reports, tracking trends over periods, and supporting calculations like monthly revenue, booking durations, and incident resolutions over time.

6. Fact Tables:

Fact tables contain measurable, quantitative data and foreign keys that reference dimension tables. They form the core of the data warehouse as they store the events or transactional data.

1. Booking Fact Table

- **Purpose:** Stores information related to parking bookings.
- **Measures:**
 - Booking ID (PK): Unique identifier for each booking.
 - Transaction ID (FK): Links to the payment fact table.
 - Status: The status of the booking (e.g., active, completed, canceled).
 - Total Booking Cost: The total booking cost is calculated as the difference between End_DateTime and Start_DateTime multiplied by the price
 - Total Booking Duration: The total booking duration is calculated as the sum of the difference between End_DateTime and Start_DateTime for all bookings in the FACT_Booking table
- **Cardinality:**
 - 1, N bookings can occur for each Parking Slot and Member.
- **Relationships:**
 - Linked with dimensions: Parking Slot, Vehicle, Time, Member.

- **Example Query:** Total number of bookings per parking lot by month.

2. Payment Fact Table

- **Purpose:** Stores payment details for parking bookings.
- **Measures:**
 - Transaction ID (PK): Unique identifier for each payment.
 - Payment Status: Status of the payment (e.g., completed, pending).
 - Amount: The total amount paid.
- **Cardinality:**
 - 1, N payments are linked to each Member and Booking.
- **Relationships:**
 - Linked with dimensions: Time, Member.
- **Example Query:** Total revenue collected by year from all members.

3. Incident Fact Table

- **Purpose:** Logs incidents related to parking activities.
- **Measures:**
 - Incident ID (PK): Unique identifier for each incident.
 - Resolution Status: Indicates if the incident is resolved.
 - Incident Count: Tracks the number of incidents per period.
- **Cardinality:**
 - 1, N incidents can occur for each Admin and Parking Slot.
- **Relationships:**
 - Linked with dimensions: Admin, Parking Slot, Incident Details, Time.
- **Example Query:** Monthly count of incidents by parking lot.

7. Dimension Tables: Dimension tables store descriptive information about the business entities involved in the fact events. They provide context to the facts and allow the users to slice, dice, and roll-up data for analysis.

1. Time Dimension

- **Attributes:**
 - Date_Time, Day_of_Week, Time
- **Cardinality:**
 - 1, N bookings and payments can be associated with each time period.
- **Hierarchy:**
 - Time → Year → Quarter → Month → Day.
- **Example Query:** Number of bookings per quarter for each parking lot.

2. Member Dimension

- **Attributes:**
 - Member ID (PK): Unique identifier for each member.
 - Member_Name, Phone No, Email, Address.
 - Membership ID (FK): Links to the Membership dimension.

- Cardinality:
 - 1, N bookings and payments can be made by each member.
- Example Query: Number of bookings made by premium members in 2024.

3. Parking Slot Dimension

- Attributes:
 - Slot ID (PK): Unique identifier for each parking slot.
 - Type: Type of parking slot
 - Status: Availability of the slot (free, occupied).
 - Price: Pricing for the parking slot.
- Cardinality:
 - 1, N bookings can be associated with each parking slot.
- Example Query: Parking slot usage analysis by status and price.

4. Parking Lot Dimension

- Attributes:
 - Lot ID (PK): Unique identifier for each parking lot.
 - Location ID (FK): Links to the Location dimension.
 - Name: Name of the lot
 - Capacity: Number of parking slots available.
- Cardinality:
 - 1, N parking slots are contained in each parking lot.
- Example Query: Total revenue per parking lot per year.

5. Vehicle Dimension

- Attributes:
 - Vehicle Number (PK): Unique identifier for each vehicle.
 - Make ID (FK): Links to the vehicle Make dimension.
 - Color
- Cardinality:
 - 1, N bookings can be associated with each vehicle.
- Example Query: Popular vehicle types used in the downtown parking lot.

6. Admin Dimension

- Attributes:
 - Admin ID (PK): Unique identifier for each administrator.
 - Admin_Name, Email, Phone No., Address
- Cardinality:
 - 1, N incidents can be managed by each admin.
- Example Query: Number of incidents resolved by each administrator.

7. Incident Details Dimension

- Attributes:
 - Type, Description, Severity.
- Cardinality:
 - 1, N incidents can be associated with each incident type.

- Example Query: Severity analysis of incidents reported in 2023.

8. Membership Dimension

- Attributes:
 - Membership ID (PK): Unique identifier for each membership.
 - Membership_Name, Status, Membership Level.
- Cardinality:
 - 1, N members can hold a membership.
- Example Query: Booking analysis based on membership level (e.g., premium vs. basic).

9. Discount Dimension

- Attributes:
 - Discount ID (PK): Unique identifier for each discount.
 - Percentage, Description, Status
- Cardinality:
 - 1, N memberships may have a discount applied.
- Example Query: Total discount provided to premium members in 2024.

10. Location and Region Dimension

- Attributes:
 - Location ID (PK): Unique identifier for each location.
 - Location Name, Region ID (FK): Links to the Region dimension.
- Cardinality:
 - 1, N parking lots belong to a location.
 - 1, N locations belong to a region.
- Hierarchy:
 - Region → Location → Parking Lot → Parking Slot.
- Example Query: Total bookings by region for each year.

11. Vehicle Make and Type Dimension

- Attributes:
 - Make ID (PK): Unique identifier for each vehicle make.
 - Make_Name, Type ID (FK): Links to the Type dimension.
- Cardinality:
 - 1, N vehicle makes are of a specific type.
- Example Query: Analyze vehicle types used for parking in different locations.

Hierarchies:

- **Time Hierarchy:** Year → Quarter → Month → Day.
- **Location Hierarchy:** Region → Location → Parking Lot → Parking Slot.

Vehicle Hierarchy: Type → Make → Vehicle.

8. Strategic and Operational Insights for the Data Pipeline:

The ParkEasy data pipeline is designed to deliver a broad spectrum of strategic and operational insights that will enhance the efficiency and effectiveness of parking management. Leveraging

the data collected and processed through this advanced system, stakeholders can make informed decisions that not only improve daily operations but also shape long-term strategic initiatives. Below are the detailed insights that ParkEasy aims to achieve:

Operational Analytics

Operational insights focus on leveraging real-time and transactional data to streamline daily parking operations, ensuring optimal resource utilization and customer satisfaction:

1. Real-Time Availability Analysis:
 - Objective: Employ real-time data analytics to provide immediate updates on parking space availability.
2. Dynamic Pricing Optimization:
 - Objective: Implement data-driven pricing models that respond to changes in demand in real-time.
3. Incident Response Analytics:
 - Objective: Enhance the speed and effectiveness of incident management using automated systems.
4. Operational Efficiency Metrics:
 - Objective: Continuously monitor and improve the layout and operations of parking lots.

Strategic Analytics

Strategic insights utilize deep analytics to support long-term business goals, infrastructure planning, and policy development:

1. Customer Behavioral Analytics:
 - Objective: Deepen understanding of customer preferences and behaviors to enhance service personalization.
2. Capacity and Demand Forecasting:
 - Objective: Predict future parking capacity needs and plan infrastructure developments accordingly.
3. Sustainability Impact Studies:
 - Objective: Assess and improve the environmental sustainability of parking operations.
4. Market and Competitive Analytics:
 - Objective: Maintain a competitive edge by analyzing market trends and competitor strategies.
5. Regulatory Compliance Monitoring:
 - Objective: Ensure adherence to evolving regulatory requirements related to urban mobility and parking management.

9. Pipeline Tools:

The Parkeasy Data Pipeline employs a comprehensive set of tools to extract, transform, and load (ETL) data efficiently while maintaining scalability and robustness. These tools are integral in ensuring the seamless integration of data sources, accurate data transformation, and effective data loading into the target warehouse for analysis. Below is a detailed breakdown of each tool used in the pipeline, its functionality, and its specific role within the architecture.

1. AWS Glue:

AWS Glue is the central ETL tool orchestrating data extraction, transformation, and loading processes. It automates schema discovery, manages metadata, and performs custom Python-based data transformations.

Components used:

- **Data Crawlers:** Glue crawlers are set up to automatically discover schemas from the raw data in the source systems. These schemas are stored in the Glue Data Catalog, a centralized metadata repository. Crawlers are particularly useful for handling:
 - ➔ Data from the PostgreSQL database (hosted on Amazon RDS).
 - ➔ CSV files stored in an Amazon S3 bucket.
- **ETL Jobs:** Glue ETL jobs are configured with Python scripts to perform transformations such as:
 - ➔ Renaming columns to standardize naming conventions.
 - ➔ Adding surrogate keys to ensure data consistency.
 - ➔ Typecasting columns (e.g., casting percentages to float).
 - ➔ Complex operations like parsing dates or handling hierarchical data.

Integration: Glue integrates seamlessly with Amazon S3, Redshift, and the Glue Data Catalog, acting as a bridge between raw data sources and the processed data warehouse.

Role in the Pipeline:

1. Extracts data from Amazon RDS and S3.
2. Automates data transformation and schema discovery.
3. Writes processed data into Amazon Redshift for further analysis.

2. Amazon S3:

Amazon S3 acts as a storage layer for the pipeline, providing scalable and durable storage for semi-structured data. It is particularly used for storing flat files, such as CSV files.

Role in the Pipeline:

1. Hosts raw data files, including pre-populated dimension tables like the time dimension.
2. AWS Glue crawlers extract data from S3 to populate the Glue Data Catalog.
3. Acts as a temporary storage location for intermediate files during the ETL process.

Use Case:

Storing the time dimension table as a CSV file, which is pre-processed and integrated with other data sources during the ETL process.

3. Amazon RDS (PostgreSQL):

Amazon RDS is a managed database service hosting the PostgreSQL database, which serves as the transactional data source for the ETL pipeline.

Role in the Pipeline:

1. Acts as the primary source of transactional data, including fact and dimension tables.

2. AWS Glue jobs use JDBC connections to extract data from the RDS-hosted PostgreSQL database for processing.
3. Ensures high availability and reliability for data extraction operations.

Use case:

Source tables such as public_admin, public_membership, and public_vehicle are extracted from PostgreSQL for transformation and loading into Redshift.

4. AWS Glue Data Catalog:

The AWS Glue Data Catalog serves as the centralized metadata repository for the entire pipeline. It maintains information about the structure, schema, and locations of data sources.

Role in the Pipeline:

1. Stores metadata for the transactional PostgreSQL tables and CSV files.
2. Enables querying of transformed data using Amazon Athena.
3. Acts as the backbone for schema discovery and validation during ETL.

5. Amazon Redshift:

Amazon Redshift is the final destination for the transformed data. It is a fully managed data warehouse optimized for analytical workloads.

Role in the Pipeline:

1. Acts as the target system for the processed dimension and fact tables.
2. Data from AWS Glue ETL jobs is loaded into Redshift tables like dim_admin, dim_vehicle, and dim_membership.
3. Supports complex SQL queries for reporting and business intelligence.

Use case:

Serves as the core data warehouse for running analytics on Parkeasy data.

6. Amazon Athena:

Amazon Athena is a serverless query service that enables SQL-based querying of the data cataloged in AWS Glue.

Role in the pipeline:

1. Provides an interface to query transformed data stored in S3 via the Glue Data Catalog.
2. Facilitates ad hoc querying without requiring a dedicated data warehouse or infrastructure.

Use Case:

Enables business users to query processed data directly from S3 for quick insights.

7. AWS Step Functions:

AWS Step Functions orchestrate the workflow of the ETL pipeline, ensuring smooth execution of dependent tasks.

Role in the pipeline:

1. Manages the sequence of operations, including:

- Data extraction from RDS and S3.
 - Running Glue ETL jobs.
 - Loading data into Redshift.
- Provides fault tolerance by retrying failed steps or alerting users in case of issues.

A lambda function was created for the same which was used to trigger the crawlers to automated pulling the data

8. Amazon Cloudwatch:

Amazon CloudWatch is the monitoring and logging tool for the pipeline.

Role in the pipeline:

- Logs and monitors the execution of Glue jobs and Step Functions.
- Provides visibility into pipeline performance and potential bottlenecks.

10. Programming Choice

The programming language used for the Parkeeasy ETL Pipeline is Python. Python scripts play a pivotal role in the pipeline, especially in the ETL jobs executed by AWS Glue. These scripts are tailored to meet the pipeline's transformation requirements and handle diverse data sources, such as PostgreSQL (via Amazon RDS) and flat files (CSV format). Below is an overview of how Python has been used:

1. Data Extraction:

Python scripts leverage AWS Glue's PySpark API to extract data from:

- PostgreSQL tables via a JDBC connection.
- CSV files stored in Amazon S3.

Example –

```

18 ## Read data from Glue Data Catalog (public_admin table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20     database="parkeeasy_data_catalog", # Replace with your Glue Data Catalog database
21     table_name="public_admin", # Replace with your table name
22     transformation_ctx="datasource0"
23 )

```

2. Data Transformation:

Python handles complex transformations, including:

- Column Renaming:** Standardizes naming conventions for easier analysis.
- Data Type Casting:** Converts data to appropriate formats (e.g., strings to floats).
- Key Generation:** Adds surrogate keys using Spark functions like `monotonically_increasing_id`.
- Custom Logic:** Python scripts include custom transformations such as date parsing or conditional logic.

Example –

```
28 ## Transformations
29 df_transformed = df.withColumnRenamed("adminid", "AdminID") \
30                 .withColumnRenamed("name", "Name") \
31                 .withColumnRenamed("address", "Address") \
32                 .withColumnRenamed("email", "Email") \
33                 .withColumnRenamed("phoneno", "PhoneNo") \
34                 .withColumnRenamed("username", "Username") \
35                 .withColumn("AdminKey", monotonically_increasing_id())
36
```

3. Data Loading:

Python scripts write transformed data into Amazon Redshift using Glue's JDBC connection APIs.

Example –

```
40 ## Write to Redshift (dim_admin table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42     frame=dynamic_frame_out,
43     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
44     connection_options={
45         "dbtable": "dim_admin", # Target table in Redshift
46         "database": "dev"       # Target database in Redshift
47     },
48     redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/" # Temporary directory in S3
49 )
```

4. Error handling and logging:

Python enables robust error handling to manage job failures, ensuring retries and logging details to CloudWatch Logs for debugging.

Python scripts for all the jobs –

1. Admin_job

To exit full screen, press and hold **esc**

Last modified on 24/11/2024, 13:52:22

Actions | **Save** | **Run**

```

Script | Job details | Runs | Data quality | Schedules | Version Control | Upgrade analysis - preview

Script | Info
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 ## Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 ## Read data from Glue Data Catalog (public_admin table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20     database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21     table_name="public_admin", # Replace with your table name
22     transformation_ctx="datasource0"
23 )
24
25 df = datasource0.toDF()
26
27 df_transformed = df.withColumnRenamed("adminid", "AdminID") \
28     .withColumnRenamed("name", "Name") \
29     .withColumnRenamed("address", "Address") \
30     .withColumnRenamed("email", "Email") \
31     .withColumnRenamed("phoneno", "PhoneNo") \
32     .withColumnRenamed("username", "Username") \
33     .withColumn("AdminKey", monotonically_increasing_id())
34
35 dynamic_cframe_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_cframe_out")
36
37 glueContext.write_dynamic_frame.from_jdbc_conf(
38     frame=dynamic_cframe_out,
39     connection_type="redshift",
40     connection_options={
41         "TempDir": "s3://parkeasy-temp-dir",
42         "JDBCConnectionURL": "jdbc:redshift://redshift-cluster-1.cjwv4yqz7s3r.us-east-1.redshift.amazonaws.com:5439/public",
43         "DBUser": "admin",
44         "DBPassword": "Parkeeasy@123",
45         "DBName": "public"
46     },
47     redshift_tmp_dir="s3://parkeasy-temp-dir"
48 )
49
50 
```

Python | Ln 1, Col 1 | Errors: 0 | Warnings: 0

Last modified on 24/11/2024, 13:52:22

Actions | **Save** | **Run**

```

Script | Job details | Runs | Data quality | Schedules | Version Control | Upgrade analysis - preview

Script | Info
22     transformation_ctx="datasource0"
23 )
24
25 ## Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 ## Transformations
29 df_transformed = df.withColumnRenamed("adminid", "AdminID") \
30     .withColumnRenamed("name", "Name") \
31     .withColumnRenamed("address", "Address") \
32     .withColumnRenamed("email", "Email") \
33     .withColumnRenamed("phoneno", "PhoneNo") \
34     .withColumnRenamed("username", "Username") \
35     .withColumn("AdminKey", monotonically_increasing_id())
36
37 ## Convert back to DynamicFrame
38 dynamic_cframe_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_cframe_out")
39
40 ## Write to Redshift (dim_admin table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42     frame=dynamic_cframe_out,
43     connection_type="redshift",
44     connection_options={
45         "TempDir": "s3://parkeasy-temp-dir",
46         "JDBCConnectionURL": "jdbc:redshift://redshift-cluster-1.cjwv4yqz7s3r.us-east-1.redshift.amazonaws.com:5439/public",
47         "DBUser": "admin",
48         "DBPassword": "Parkeeasy@123",
49         "DBName": "public"
50     },
51     redshift_tmp_dir="s3://parkeasy-temp-dir"
52 )
53
54 
```

Python | Ln 1, Col 1 | Errors: 0 | Warnings: 0

```

43     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
44     connection_options={
45         "dbtable": "dim_admin", # Target table in Redshift
46         "database": "dev"       # Target database in Redshift
47     },
48     redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/" # Temporary directory in S3
49 )
50
51 job.commit()

```

2. Discount_job

The screenshot shows the AWS Glue Studio interface. On the left, there's a sidebar with navigation links for AWS Glue, Data Catalog, Data Integration and ETL, and ETL jobs. The main area is titled 'Discount_test_job' and shows a Python script. The script imports necessary modules like sys, awsglue.transforms, awsglue.utils, pyspark.context, awsglue.context, and awsglue.job. It initializes a SparkContext and a GlueContext, reads data from a public_discount table in the parkeasy_data_catalog database, and performs some operations. The script is currently at line 1, column 1, with 0 errors and 0 warnings.

```

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
from pyspark.sql.functions import col, monotonically_increasing_id
from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion

## Initialize Glue Context and Job
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

## Read data from Glue Data Catalog (public_discount table)
datasource0 = glueContext.create_dynamic_frame.from_catalog(
    database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
    table_name="public_discount", # Replace with your table name
)

```

Screenshot of the AWS Glue Studio home page showing the script editor for a job named "Discount_test_job".

The sidebar on the left shows navigation links for AWS Glue, Data Catalog, and Data Integration and ETL.

The main area displays the script code:

```

22     transformation_ctx="datasource0"
23   )
24
25     ## Convert DynamicFrame to Spark DataFrame for transformations
26   df = datasource0.toDF()
27
28     ## Transformations
29   df_transformed = df.withColumnRenamed("discountid", "DiscountID") \
30     .withColumnRenamed("status", "Status") \
31     .withColumnRenamed("description", "Description") \
32     .withColumnRenamed("percentage", "Percentage") \
33     .withColumn("DiscountKey", monotonically_increasing_id()) \
34     .withColumn("Percentage", col("Percentage").cast("float")) # Cast percentage to float for Redshift
35
36     ## Convert back to DynamicFrame
37   dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
38
39     ## Write to Redshift (dim_discount table)
40   glueContext.write_dynamic_frame.from_jdbc_conf(
41     frame=dynamic_frame_out,
42     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
43     connection_options={# Target table in Redshift
44       "dbtable": "dim_discount", # Target table in Redshift
45       "database": "dev" # Target database in Redshift
46     },
47     redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/" # Temporary directory in S3
48   )
49
50   job.commit()
51

```

The script was last modified on 24/11/2024, 13:56:24. The status bar indicates 0 errors and 0 warnings.

3.ParkingLot_job

AWS Glue Job Editor - Lot_job

```

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
from pyspark.sql.functions import col, monotonically_increasing_id
from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion

## Initialize Glue Context and Job
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

## Read data from Glue Data Catalog (public_parkinglot table)
datasource0 = glueContext.create_dynamic_frame.from_catalog(
    database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
    table_name="public_parkinglot", # Replace with your table name
    transformation_ctx="datasource0"
)

```

Python Ln 1, Col 1 Errors: 0 | Warnings: 0

AWS Glue Job Editor - Lot_job

```

transformation_ctx="datasource0"
)
## Convert DynamicFrame to Spark DataFrame for transformations
df = datasource0.toDF()
## Transformations
df_transformed = df.withColumnRenamed("lotid", "LotID") \
    .withColumnRenamed("name", "Name") \
    .withColumnRenamed("location", "Location") \
    .withColumnRenamed("region", "Region") \
    .withColumnRenamed("capacity", "Capacity") \
    .withColumn("LotKey", monotonically_increasing_id()) # Add surrogate key
## Convert back to DynamicFrame
dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
## Write to Redshift (dim_parkinglot table)
glueContext.write_dynamic_frame.from_jdbc_conf(
    frame=dynamic_frame_out,
    catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
    connection_options={}
)

```

Python Ln 1, Col 1 Errors: 0 | Warnings: 0

```

43     connection_options={
44         "dbtable": "dim_parkinglot", # Target table in Redshift
45         "database": "dev"           # Target database in Redshift
46     },
47     redshift_tmp_dir="s3://project-parkeasy-data-bucket/Tmp/" # Temporary directory in S3
48 )
49
50 job.commit()
51

```

4.Time_job

The screenshot shows the AWS Glue Studio interface with the 'time_job' script open. The left sidebar contains navigation links for AWS Glue, Data Catalog, and Data Integration and ETL. The main area displays the script code:

```

1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame
9
10 ## Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 ## Read CSV file directly from S3
19 csv_file_path = "s3://project-parkeasy-data-bucket/csv folder/DIM_Time (3).csv" # Replace with the actual path
20 df = spark.read.format("csv") \
21     .option("header", "true") \
22     .option("sep", ",") \

```

The code imports necessary libraries and initializes a SparkContext, GlueContext, and a Job. It then reads a CSV file from S3 using the specified path.

The screenshot shows the AWS Glue Studio interface. On the left, a sidebar navigation bar includes links for Getting started, ETL jobs (Visual ETL, Notebooks, Job run monitoring), Data Catalog tables, Data connections, Workflows (orchestration), Data Catalog (Databases, Tables, Stream schema registries, Schemas, Connections, Crawlers, Classifiers, Catalog settings), Data Integration and ETL (ETL jobs, Visual ETL, Notebooks, Job run monitoring, Interactive Sessions, Data classification tools, Sensitive data detection), CloudShell, and Feedback.

The main workspace is titled "time_job". It shows the "Script" tab selected. The script content is as follows:

```

22     .option("inferSchema", "true") \
23     .load(csv_file_path)
24
25     ## Transformations
26     df_transformed = df.withColumnRenamed("timeid", "TimeID") \
27         .withColumnRenamed("year", "Year") \
28         .withColumnRenamed("quarter", "Quarter") \
29         .withColumnRenamed("month", "Month") \
30         .withColumnRenamed("day", "Day") \
31         .withColumnRenamed("date", "Date") \
32         .withColumn("TimeKey", monotonically_increasing_id()) # Add surrogate key
33
34     ## Convert to DynamicFrame
35     dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
36
37     ## Write to Redshift (dim_time table)
38     glueContext.write_dynamic_frame.from_jdbc_conf(
39         frame=dynamic_frame_out,
40         catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
41         connection_options={
42             "dbtable": "dim_time", # Target table in Redshift
43             "database": "dev"      # Target database in Redshift
44         },
45         redshift_tmp_dir="s3://project-parkeasy-data-bucket(Temp)" # Replace with your S3 temporary directory
46     )
47
48 job.commit()

```

At the bottom of the script editor, it says "Python Ln 1, Col 1" and "Errors: 0 | Warnings: 0". The status bar at the bottom right indicates "© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences".

5. Membership_job

AWS Glue - membership_job

```

1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id, to_date
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 ## Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 ## Read data from Glue Data Catalog (public_membership table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20     database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21     table_name="public_membership", # Replace with your table name
22     transformation_ctx="datasource0"
23 )
24
25 ## Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 ## Transformations
29 df_transformed = df.withColumnRenamed("membershipid", "MembershipID") \
30     .withColumnRenamed("name", "Name") \
31     .withColumnRenamed("status", "Status") \
32     .withColumnRenamed("discountid", "DiscountID") \
33     .withColumn("MembershipKey", monotonically_increasing_id()) \
34     .withColumn("StartDateKey", to_date("startdate", "yyyy-MM-dd")) \
35     .withColumn("EndDateKey", to_date("enddate", "yyyy-MM-dd")) # Convert to Date format
36
37 ## Convert back to DynamicFrame
38 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
39
40 ## Write to Redshift (dim_membership table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42     frame=dynamic_frame_out,
43     ...
44

```

Last modified on 24/11/2024, 13:57:11

[Actions](#) | [Save](#) | [Run](#)

AWS Glue - membership_job

```

22     transformation_ctx="datasource0"
23 )
24
25 ## Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 ## Transformations
29 df_transformed = df.withColumnRenamed("membershipid", "MembershipID") \
30     .withColumnRenamed("name", "Name") \
31     .withColumnRenamed("status", "Status") \
32     .withColumnRenamed("discountid", "DiscountID") \
33     .withColumn("MembershipKey", monotonically_increasing_id()) \
34     .withColumn("StartDateKey", to_date("startdate", "yyyy-MM-dd")) \
35     .withColumn("EndDateKey", to_date("enddate", "yyyy-MM-dd")) # Convert to Date format
36
37 ## Convert back to DynamicFrame
38 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
39
40 ## Write to Redshift (dim_membership table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42     frame=dynamic_frame_out,
43     ...
44

```

Last modified on 24/11/2024, 13:57:11

[Actions](#) | [Save](#) | [Run](#)

```

43     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
44     connection_options={
45         "dbtable": "dim_membership", # Target table in Redshift
46         "database": "dev" # Target database in Redshift
47     },
48     redshift_tmp_dir="s3://project-parkeasy-data-bucket(Temp/" # Temporary directory in S3
49 )
50
51 job.commit()
52

```

6. ParkingSlot_job

The screenshot shows the AWS Glue Studio interface. On the left, there's a sidebar with navigation links for AWS Glue, ETL jobs, Data Catalog, Data Integration and ETL, and CloudShell. The main area is titled 'Slot_job' and shows the script content. The script imports necessary libraries and initializes a Glue Context and SparkSession. It then reads data from a Glue Data Catalog table named 'public_parkingslot'. The code is written in Python.

```

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
from pyspark.sql.functions import col, monotonically_increasing_id
from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion

## Initialize Glue Context and Job
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

## Read data from Glue Data Catalog (public_parkingslot table)
datasource0 = glueContext.create_dynamic_frame.from_catalog(
    database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
    table_name="public_parkingslot", # Replace with your table name
    transformation_ctx="datasource0"
)

```

The screenshot shows the AWS Glue Studio interface. On the left, there's a sidebar with navigation links for AWS Glue, ETL jobs, Data Catalog, Data Integration and ETL, and CloudShell. The main area is titled "Slot_job" and shows a Python script for a transformation job. The script reads data from a "datasource0", converts it to a DataFrame, performs several column renamings, and then writes it back to a Redshift table named "dim_parkingslot". The code is as follows:

```

22     transformation_ctx="datasource0"
23 )
24
25 ## Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 ## Transformations
29 df_transformed = df.withColumnRenamed("slotid", "SlotID") \
30     .withColumnRenamed("status", "Status") \
31     .withColumnRenamed("price", "Price") \
32     .withColumnRenamed("type", "Type") \
33     .withColumnRenamed("lotid", "LotID") \
34     .withColumn("SlotKey", monotonically_increasing_id()) # Surrogate key
35
36 ## Convert back to DynamicFrame
37 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
38
39 ## Write to Redshift (dim_parkingslot table)
40 glueContext.write_dynamic_frame.from_jdbc_conf(
41     frame=dynamic_frame_out,
42     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
43     connection_options={
44         "dbtable": "dim_parkingslot", # Target table in Redshift
45         "database": "dev"           # Target database in Redshift
46     },
47     redshift_tmp_dir="s3://project-parkeasy-data-bucket(Temp)" # Temporary directory in S3
48 )
49
50 job.commit()
51

```

The CloudShell tab is selected at the bottom of the editor.

7. Vehicle_job

AWS Glue - us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Vehicle_job/script

Vehicle_job

Last modified on 24/11/2024, 13:58:07

Script | Job details | Runs | Data quality | Schedules | Version Control | Upgrade analysis - preview

Script **Info**

```

1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 # Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 # Read data from Glue Data Catalog (public_vehicle table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20     database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21     table_name="public_vehicle", # Replace with your table name
22     transformation_ctx="datasource0"
23 )
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("vehiclenumber", "VehicleNumber") \
30     .withColumnRenamed("make", "Make") \
31     .withColumnRenamed("color", "Color") \
32     .withColumnRenamed("vehicletype", "VehicleType") \
33     .withColumnRenamed("bookingid", "BookingID") \
34     .withColumnRenamed("memberid", "MemberID") \
35     .withColumn("VehicleKey", monotonically_increasing_id()) # Surrogate key
36
37 # Convert back to DynamicFrame
38 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
39
40 # Write to Redshift (dim_vehicle table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42     frame=dynamic_frame_out,
43     catalog_connection="redshift_connection_parkeasy", # Replace with your Redshift connection name
44     ...
45 )

```

Python Ln 1, Col 1 Errors: 0 | Warnings: 0

AWS Glue - us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Vehicle_job/script

Vehicle_job

Last modified on 24/11/2024, 13:58:07

Script | Job details | Runs | Data quality | Schedules | Version Control | Upgrade analysis - preview

Script **Info**

```

22     transformation_ctx="datasource0"
23 )
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("vehiclenumber", "VehicleNumber") \
30     .withColumnRenamed("make", "Make") \
31     .withColumnRenamed("color", "Color") \
32     .withColumnRenamed("vehicletype", "VehicleType") \
33     .withColumnRenamed("bookingid", "BookingID") \
34     .withColumnRenamed("memberid", "MemberID") \
35     .withColumn("VehicleKey", monotonically_increasing_id()) # Surrogate key
36
37 # Convert back to DynamicFrame
38 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
39
40 # Write to Redshift (dim_vehicle table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42     frame=dynamic_frame_out,
43     catalog_connection="redshift_connection_parkeasy", # Replace with your Redshift connection name
44     ...
45 )

```

Python Ln 1, Col 1 Errors: 0 | Warnings: 0

```

43     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
44     connection_options={
45         "dbtable": "dim_vehicle", # Target table in Redshift
46         "database": "dev"          # Target database in Redshift
47     },
48     redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/" # Temporary directory in S3
49 )
50
51 job.commit()

```

8. Member_job

The screenshot shows the AWS Glue Studio interface for a job named 'Member_job'. The left sidebar contains navigation links for AWS Glue, Data Catalog, and Data Integration and ETL. The main area displays the script code:

```

1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 # Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 # Read data from Glue Data Catalog (public_member table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20     database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21     table_name="public_member",      # Replace with your table name
22     transformation_ctx="datasource0"

```

The code imports necessary libraries and initializes a Glue Context and SparkSession. It then reads data from a table in the Glue Data Catalog.

Screenshot of the AWS Glue Studio home page showing the Member_job script.

Script Content:

```

22     transformation_ctx="datasource0"
23   )
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("memberid", "MemberID") \
30   .withColumnRenamed("phoneno", "PhoneNo") \
31   .withColumnRenamed("email", "Email") \
32   .withColumnRenamed("address", "Address") \
33   .withColumnRenamed("name", "Name") \
34   .withColumnRenamed("membershipid", "MembershipID") \
35   .withColumnRenamed("username", "Username") \
36   .withColumn("MemberKey", monotonically_increasing_id()) # Surrogate key
37
38 # Add OldAddress column with null values (if required)
39 df_transformed = df_transformed.withColumn("OldAddress", col("Address"))
40
41 # Convert back to DynamicFrame
42 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")

```

CloudShell Content:

```

42 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
43
44 # Write to Redshift (dim_member table)
45 glueContext.write_dynamic_frame.from_jdbc_conf(
46   frame=dynamic_frame_out,
47   catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
48   connection_options={
49     "dbtable": "dim_member", # Target table in Redshift
50     "database": "dev" # Target database in Redshift
51   },
52   redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/" # Temporary directory in S3
53 )
54
55 job.commit()

```

9. Fact_payment_job

AWS Glue - fact_payment_job

```

1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id, lit
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 # Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 # Read data from Glue Data Catalog (public_payment table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20     database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21     table_name="public_payment", # Replace with your table name
22     transformation_ctx="datasource0"
23 )
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("transactionid", "TransactionID") \
30     .withColumnRenamed("datetime", "DateTimeID") \
31     .withColumnRenamed("paymentstatus", "PaymentStatus") \
32     .withColumnRenamed("amount", "Amount") \
33     .withColumnRenamed("memberid", "MemberID") \
34     .withColumn("TransactionKey", monotonically_increasing_id()) # Surrogate key
35
36 # Convert Amount to Float type
37 df_transformed = df_transformed.withColumn("Amount", col("Amount").cast("float"))
38
39 # Convert back to DynamicFrame
40 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
41
42 # Write to Redshift (fact_payment table)
43 glueContext.write_dynamic_frame(frame = dynamic_frame_out, connection_type="redshift", connection_options="{'REDSHIFT_CLUSTER_ID': 'fact_payment', 'REDSHIFT_DB': 'fact_payment', 'REDSHIFT_TABLE': 'fact_payment'}")
44
45 # End of script

```

Last modified on 24/11/2024, 13:58:40

[Actions](#) | [Save](#) | [Run](#)

AWS Glue - fact_payment_job

```

22     transformation_ctx="datasource0"
23 )
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("transactionid", "TransactionID") \
30     .withColumnRenamed("datetime", "DateTimeID") \
31     .withColumnRenamed("paymentstatus", "PaymentStatus") \
32     .withColumnRenamed("amount", "Amount") \
33     .withColumnRenamed("memberid", "MemberID") \
34     .withColumn("TransactionKey", monotonically_increasing_id()) # Surrogate key
35
36 # Convert Amount to Float type
37 df_transformed = df_transformed.withColumn("Amount", col("Amount").cast("float"))
38
39 # Convert back to DynamicFrame
40 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
41
42 # Write to Redshift (fact_payment table)
43 glueContext.write_dynamic_frame(frame = dynamic_frame_out, connection_type="redshift", connection_options="{'REDSHIFT_CLUSTER_ID': 'fact_payment', 'REDSHIFT_DB': 'fact_payment', 'REDSHIFT_TABLE': 'fact_payment'}")
44
45 # End of script

```

Last modified on 24/11/2024, 13:58:40

[Actions](#) | [Save](#) | [Run](#)

```

42 # Write to Redshift (fact_payment table)
43 glueContext.write_dynamic_frame.from_jdbc_conf(
44     frame=dynamic_frame_out,
45     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
46     connection_options={
47         "dbtable": "fact_payment", # Target table in Redshift
48         "database": "dev"           # Target database in Redshift
49     },
50     redshift_tmp_dir="s3://project-parkeasy-data-bucket/Tmp/" # Temporary directory in S3
51 )
52
53 job.commit()

```

10. Fact_incident_job

The screenshot shows the AWS Glue Studio interface for editing a job named 'fact_incident_job'. The left sidebar provides navigation through various AWS services. The main workspace displays the job details, including tabs for Script, Job details, Runs, Data quality, Schedules, Version Control, and Upgrade analysis - preview. The Script tab is active, showing the following Python code:

```

1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id
8 from awsglue.dynamicframe import DynamicFrame # Import required for DynamicFrame conversion
9
10 # Initialize Glue Context and Job
11 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
12 sc = SparkContext()
13 glueContext = GlueContext(sc)
14 spark = glueContext.spark_session
15 job = Job(glueContext)
16 job.init(args['JOB_NAME'], args)
17
18 # Read data from Glue Data Catalog (public_incident table)
19 datasource0 = glueContext.create_dynamic_frame.from_catalog(
20     database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
21     table_name="public_incident", # Replace with your table name
22     transformation_ctx="datasource0"
23 )

```

The code imports necessary libraries and initializes a Glue Context and Spark Context. It then reads data from a Glue Data Catalog table named 'public_incident'.

The screenshot shows the AWS Glue Studio interface. On the left, a sidebar navigation pane includes sections for AWS Glue (Getting started, ETL jobs, Visual ETL, Notebooks, Job run monitoring, Data Catalog tables, Data connections, Workflows (orchestration)), Data Catalog (Databases, Tables, Stream schema registries, Schemas, Connections, Crawlers, Classifiers, Catalog settings), and Data Integration and ETL (ETL jobs, Visual ETL, Notebooks, Job run monitoring, Interactive Sessions, Data classification tools, Sensitive data detection). The main workspace is titled 'fact_incident_job' and was last modified on 24/11/2024, 13:58:52. It contains a tab bar with 'Script' (selected), Job details, Runs, Data quality, Schedules, Version Control, and Upgrade analysis - preview. Below the tabs is a code editor window containing the following Python script:

```

22     transformation_ctx="datasource0"
23 )
24
25 # Convert DynamicFrame to Spark DataFrame for transformations
26 df = datasource0.toDF()
27
28 # Transformations
29 df_transformed = df.withColumnRenamed("incidentid", "IncidentID") \
30     .withColumnRenamed("resolutionstatus", "ResolutionStatus") \
31     .withColumnRenamed("date_time", "date_time") \
32     .withColumnRenamed("description", "Description") \
33     .withColumnRenamed("memberid", "MemberID") \
34     .withColumnRenamed("adminid", "AdminID") \
35     .withColumn("IncidentKey", monotonically_increasing_id()) # Surrogate key
36
37 # Convert back to DynamicFrame
38 dynamic_frame_out = DynamicFrame.fromDF(df_transformed, glueContext, "dynamic_frame_out")
39
40 # Write to Redshift (fact_incident table)
41 glueContext.write_dynamic_frame.from_jdbc_conf(
42     frame=dynamic_frame_out,
43     catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
44     connection_options={
45         "dbtable": "fact_incident", # Target table in Redshift
46         "database": "dev" # Target database in Redshift
47     },
48     redshift_tmp_dir="s3://project-parkeasy-data-bucket/Tmp/" # Temporary directory in S3
49 )
50
51 job.commit()

```

The code editor shows 0 errors and 0 warnings. At the bottom right of the editor, there are buttons for Actions (dropdown), Save, and Run. Below the editor, a CloudShell tab is open, showing the continuation of the script. The footer of the page includes links for CloudShell, Feedback, and various AWS services like Gmail, YouTube, Maps, Information, NEU Class register..., Visa slot booking, SEVIS Fee Payment, IMFS, Student Hub, COVID-19 state-wi..., and All Bookmarks, along with account information for N. Virginia and SharvarID.

11. Fact_booking_job

AWS Glue - fact_booking_job

Last modified on 24/11/2024, 13:59:05

[Actions](#) [Save](#) [Run](#)

[Script](#) [Job details](#) [Runs](#) [Data quality](#) [Schedules](#) [Version Control](#) [Upgrade analysis - preview](#)

Script Info

```

1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql.functions import col, monotonically_increasing_id, to_date, datediff
8 from awsglue.dynamicframe import DynamicFrame
9 ## Initialize Glue Context and Job
10 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
11 sc = SparkContext()
12 glueContext = GlueContext(sc)
13 spark = glueContext.spark_session
14 job = Job(glueContext)
15 job.init(args['JOB_NAME'], args)
16 ## Read data from Glue Data Catalog (public_booking table)
17 booking_df = glueContext.create_dynamic_frame.from_catalog(
18     database="parkeasy_data_catalog", # Replace with your Glue Data Catalog database
19     table_name="public_booking", # Replace with your table name
20     transformation_ctx="booking_df"
21 ).toDF()
22 ## Read data from Redshift (DIM_ParkingSlot table) using Spark JDBC

```

Python Ln 1, Col 1 Errors: 0 Warnings: 0

AWS Glue - fact_booking_job

Last modified on 24/11/2024, 13:59:05

[Actions](#) [Save](#) [Run](#)

[Script](#) [Job details](#) [Runs](#) [Data quality](#) [Schedules](#) [Version Control](#) [Upgrade analysis - preview](#)

Script Info

```

22 ## Read data from Redshift (DIM_ParkingSlot table) using Spark JDBC
23 parking_slot_df = spark.read \
24     .format("jdbc") \
25     .option("url", "jdbc:redshift://parkeasy-redshift-cluster-dw.c9qihfbzoqpm.us-east-1.redshift.amazonaws.com:5439/dev") \
26     .option("dbtable", "dim_parkingslot") \
27     .option("user", "awsuser") \
28     .option("password", "Sharvari24") \
29     .option("driver", "com.amazon.redshift.jdbc42.Driver") \
30     .load()
31 ## Debugging step: Print columns to verify casing
32 print("ParkingSlot Columns:", parking_slot_df.columns)
33 ## Avoid column name conflicts by renaming columns in parking_slot_df
34 parking_slot_df = parking_slot_df.withColumnRenamed("slotid", "ParkingSlotID") \
35     .withColumnRenamed("price", "ParkingSlotPrice")
36 ## Join Booking data with ParkingSlot data for price
37 booking_with_price = booking_df.join(
38     parking_slot_df,
39     booking_df["slotid"] == parking_slot_df["ParkingSlotID"], # Use renamed column
40     "left"
41 )
42 ## Transformations

```

Python Ln 1, Col 1 Errors: 0 Warnings: 0

The screenshot shows the AWS Glue Studio interface. On the left, there's a sidebar with navigation links: Getting started, ETL jobs (Visual ETL, Notebooks, Job run monitoring), Data Catalog (Databases, Tables, Stream schema registries, Schemas, Connections, Crawlers, Classifiers, Catalog settings), and Data Integration and ETL (ETL jobs, Interactive Sessions, Data classification tools, Sensitive data detection). The main area is titled "fact_booking_job" and shows the "Script" tab selected. The script content is as follows:

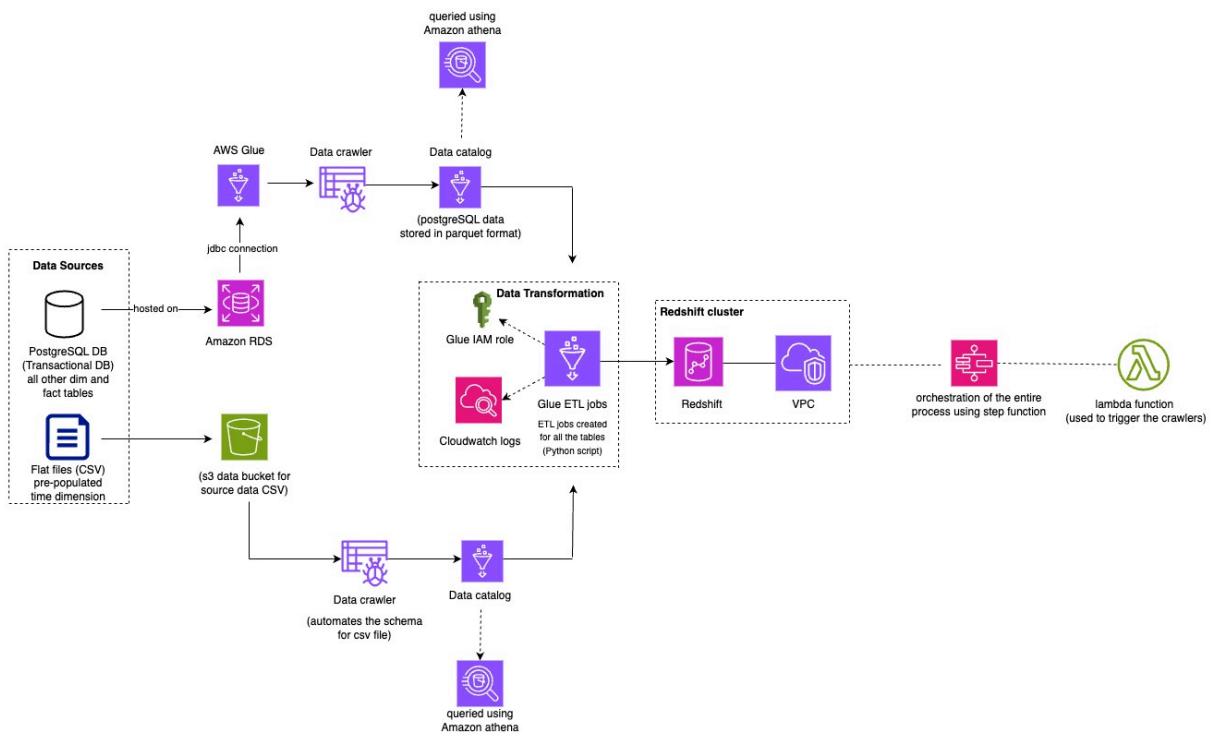
```

42  ## Transformations
43  # Convert Start_DateTime and End_DateTime to DateTime
44  booking_with_price = booking_with_price.withColumn("Start_DateTime", to_date(col("start_datetime"), "yyyy-MM-dd")) \
45  .withColumn("End_DateTime", to_date(col("end_datetime"), "yyyy-MM-dd"))
46  # Calculate BookingDuration (difference in days)
47  booking_with_price = booking_with_price.withColumn("BookingDuration",
48  datediff(col("End_DateTime"), col("Start_DateTime")))
49  # Calculate BookingCost (Duration * Price)
50  booking_with_price = booking_with_price.withColumn("BookingCost",
51  col("BookingDuration") * col("ParkingSlotPrice").cast("float"))
52  # Rename Columns to match the target schema
53  booking_transformed = booking_with_price.withColumnRenamed("bookingid", "BookingID") \
54  .withColumnRenamed("status", "Status") \
55  .withColumnRenamed("transactionid", "TransactionID") \
56  .withColumnRenamed("slotid", "SlotID") \
57  .withColumnRenamed("memberid", "MemberID") \
58  .withColumn("BookingKey", monotonically_increasing_id())
59  ## Convert back to DynamicFrame
60  dynamic_frame_out = DynamicFrame.fromDF(booking_transformed, glueContext, "dynamic_frame_out")
61  ## Write to Redshift (FACT_Booking table)
62  glueContext.write_dynamic_frame.from_jdbc_conf(
63      frame=dynamic_frame_out,
64      catalog_connection="redshift-connection-parkeasy", # Replace with your Redshift connection name
65      connection_options={
66          "dbtable": "fact_booking", # Target table in Redshift
67          "database": "dev" # Target database in Redshift
68      },
69      redshift_tmp_dir="s3://project-parkeasy-data-bucket/Temp/" # Temporary directory in S3
70  )
71  ## Commit the job
72  job.commit()

```

At the bottom of the script editor, it says "Python Ln 1, Col 1" and "Errors: 0 | Warnings: 0". Below the script editor, there's a CloudShell tab and a feedback section. The footer includes links for 2024, Privacy, Terms, and Cookie preferences.

11. Architectural Diagram of the Data Pipeline : Parkeasy Parking Management System



1. Data Sources

The pipeline begins with two primary data sources:

1. PostgreSQL Database:

- Hosted on Amazon RDS, this transactional database contains the source tables for all dimension and fact data.
- It serves as the main repository for operational data, including information about users, memberships, vehicles, parking lots, and discounts.

2. Flat Files (CSV Format):

- Stored in an Amazon S3 bucket, these files include pre-populated dimension tables such as the time dimension.
- These files complement the transactional data by providing auxiliary information for analysis.

2. Data Ingestion

Data from the above sources is ingested into the pipeline using two distinct methods:

1. JDBC Connection to Amazon RDS:

- AWS Glue connects to the PostgreSQL database on Amazon RDS using a secure JDBC connection.
- This ensures seamless access to transactional data for ETL jobs.

2. S3 Bucket Integration:

- CSV files in the S3 bucket are directly accessed by AWS Glue crawlers to automate schema discovery and load data into the pipeline.

3. AWS Glue Data Crawlers

AWS Glue Data Crawlers automate the schema discovery process for both data sources:

1. PostgreSQL Data:
 - o Crawlers identify the structure and schema of tables in the RDS-hosted PostgreSQL database.
 - o The data is cataloged in the AWS Glue Data Catalog for subsequent processing.
2. Flat Files in S3:
 - o Crawlers analyze the CSV files in the S3 bucket to infer their schema and catalog them in the Glue Data Catalog.

Key Outcome:

- The Glue Data Catalog stores metadata for all tables and files, acting as a centralized repository for schema and data definitions.

4. AWS Glue ETL Jobs

AWS Glue ETL jobs transform the raw data into a format suitable for analysis. These jobs are Python-based and use PySpark for distributed processing. Key transformations include:

1. Schema Standardization:
 - o Columns are renamed to adhere to consistent naming conventions across all tables.
2. Data Enrichment:
 - o Surrogate keys are added to enhance data integrity.
 - o Data types are cast to appropriate formats (e.g., percentages are converted to floats).
3. Business Logic Implementation:
 - o Custom transformations such as date parsing and conditional logic are applied to meet analytical requirements.

Output:

- The transformed data is prepared for loading into Amazon Redshift.

5. CloudWatch Logs

AWS Glue jobs are integrated with Amazon CloudWatch Logs to provide real-time monitoring and logging of job executions. Key functionalities include:

- Job Monitoring: Tracks the progress and status of ETL jobs.
- Error Logging: Captures errors and exceptions for debugging purposes.
- Performance Metrics: Records resource utilization and execution times to optimize job performance.

6. Amazon Redshift

The transformed data is loaded into Amazon Redshift, a fully managed data warehouse optimized for analytical queries. Key steps include:

1. Data Loading:
 - o AWS Glue writes the processed data into Redshift tables using JDBC connections.
 - o Fact and dimension tables such as dim_admin, dim_vehicle, and dim_membership are populated.
2. Data Storage:

- Redshift organizes data using columnar storage to enhance query performance.
- 3. Data Analysis:
 - Redshift supports complex SQL queries, enabling analysts to generate reports and insights.

7. Orchestration Using AWS Step Functions

AWS Step Functions orchestrate the pipeline by managing dependencies and automating the sequence of tasks. Key functionalities include:

- Workflow Management:
 - Step Functions define the order of operations, such as running data crawlers, executing Glue ETL jobs, and loading data into Redshift.
- Error Handling:
 - Built-in retry mechanisms handle job failures, ensuring the pipeline executes reliably.
- Integration:
 - Step Functions interact seamlessly with Glue, Redshift, and other AWS services to streamline the entire process.

A lambda function was created for the same which was used to trigger the crawlers to automated pulling the data

8. Data Querying with Amazon Athena

The processed data in Redshift and the Glue Data Catalog can be queried using Amazon Athena:

- 1. Querying Transformed Data:
 - Analysts use Athena to execute ad hoc SQL queries on the cataloged data stored in S3 or Redshift.
- 2. Integration with Glue Data Catalog:
 - Athena leverages the Glue Data Catalog to retrieve metadata and schema information for efficient query execution.
- 3. Insights Generation:
 - Results from Athena queries provide actionable insights for decision-making.

12. Pipeline Execution:

ETL Jobs created in Glue ETL

Screenshot of the AWS Glue Studio Jobs page (us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/jobs) showing 12 jobs listed in a table.

Create job Info

- Author in a visual interface focused on data flow. **Visual ETL**
- Author using an interactive code notebook. **Notebook**
- Author code with a script editor. **Script editor**

Example jobs Info

Your jobs (12) Info

<input type="checkbox"/> Job name	Type	Created by	Last modified	AWS Glue version
fact_booking_job	Glue ETL	Script	24/11/2024, 13:59:05	4.0
fact_incident_job	Glue ETL	Script	24/11/2024, 13:58:52	4.0
fact_payment_job	Glue ETL	Script	24/11/2024, 13:58:40	4.0
Member_job	Glue ETL	Script	24/11/2024, 13:58:26	4.0
Vehicle_job	Glue ETL	Script	24/11/2024, 13:58:07	4.0
Slot_job	Glue ETL	Script	24/11/2024, 13:57:34	4.0
membership_job	Glue ETL	Script	24/11/2024, 13:57:11	4.0
time_job	Glue ETL	Script	24/11/2024, 13:56:58	4.0
Lot_job	Glue ETL	Script	24/11/2024, 13:56:39	4.0
Discount_test_job	Glue ETL	Script	24/11/2024, 13:56:24	4.0

AWS Glue Studio Info

Create job Info

- Author in a visual interface focused on data flow. **Visual ETL**
- Author using an interactive code notebook. **Notebook**
- Author code with a script editor. **Script editor**

Example jobs Info

Your jobs (12) Info

<input type="checkbox"/> Job name	Type	Created by	Last modified	AWS Glue version
Admin_Test_job	Glue ETL	Script	24/11/2024, 13:52:22	4.0
dimension_job	Glue ETL	Script	24/11/2024, 13:36:52	4.0

Example – Job configuration details for the Admin_job

https://us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Admin_Test_job/details

AWS Glue

- Getting started
- ETL jobs**
 - Visual ETL
 - Notebooks
 - Job run monitoring
- Data Catalog tables
- Data connections
- Workflows (orchestration)

▼ Data Catalog

- Databases
- Tables
- Stream schema registries
- Schemas
- Connections
- Crawlers
- Classifiers
- Catalog settings

▼ Data Integration and ETL

ETL jobs

- Visual ETL
- Notebooks
- Job run monitoring
- Interactive Sessions
- Data classification tools
- Sensitive data detection

Admin_Test_job

Last modified on 24/11/2024, 13:52:22

Actions | **Save** | **Run**

Script | **Job details** | Runs | Data quality | Schedules | Version Control | Upgrade analysis - preview

Basic properties Info

Name
Admin_Test_job

Description - optional
Descriptions can be up to 2048 characters long.

IAM Role
Role assumed by the job with permission to access your data stores. Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.

AWSGlueServiceRole-ParkEasy

Type
The type of ETL job. This is set automatically based on the types of data sources you have selected.

Spark

Glue version | Info
Glue 4.0 - Supports spark 3.3, Scala 2, Python 3

Language

https://us-east-1.console.aws.amazon.com/gluestudio/home?region=us-east-1#/editor/job/Admin_Test_job/details

AWS Glue

- Getting started
- ETL jobs**
 - Visual ETL
 - Notebooks
 - Job run monitoring
- Data Catalog tables
- Data connections
- Workflows (orchestration)

▼ Data Catalog

- Databases
- Tables
- Stream schema registries
- Schemas
- Connections
- Crawlers
- Classifiers
- Catalog settings

▼ Data Integration and ETL

ETL jobs

- Visual ETL
- Notebooks
- Job run monitoring
- Interactive Sessions
- Data classification tools
- Sensitive data detection

Admin_Test_job

Last modified on 24/11/2024, 13:52:22

Actions | **Save** | **Run**

Script | **Job details** | Runs | Data quality | Schedules | Version Control | Upgrade analysis - preview

Language
Python 3

Worker type
Set the type of predefined worker that is allowed when a job runs.

G 1X
(4vCPU and 16GB RAM)

Automatically scale the number of workers
 AWS Glue will optimize costs and resource usage by dynamically scaling the number of workers up and down throughout the job run. Requires Glue 3.0 or later.

Requested number of workers
The number of workers you want AWS Glue to allocate to this job.
2

Generate job insights
 AWS Glue will analyze your job runs and provide insights on how to optimize your jobs and the reasons for job failures.

Job bookmark | Info
Specifies how AWS Glue processes job bookmark when the job runs. It can remember previously processed data (Enable), update state information (Pause), or ignore state information (Disable).

Disable

Screenshot of the AWS Glue Studio Job Editor for Admin_Test_job.

Job Details Tab (Selected):

- Script filename:** Admin_Test_job.py
- Script path:** s3://project-parkeasy-data-bucket/scripts/
- Job metrics:** Enabled
- Job observability metrics:** Enabled
- Continuous logging:** Enabled
- Spark UI:** Disabled (checkbox unchecked)
- Maximum concurrency:** 1

Actions: Actions ▾, Save, Run

Screenshot of the AWS Glue Studio Job Editor for Admin_Test_job.

Job Details Tab (Selected):

- Maximum concurrency:** 1
- Temporary path:** s3://project-parkeasy-data-bucket/Temp/
- Delay notification threshold (minutes):** Info
- Security configuration:** None
- Server-side encryption:** Unchecked (checkbox)
- Use Glue data catalog as the Hive metastore:** Checked (checkbox)

Actions: Actions ▾, Save, Run

Similarly, all other jobs for the facts and dimension tables were configured in the same way

ETL jobs execution

Screenshot of the AWS Glue Studio interface showing the 'Discount_test_job' run details.

Job runs (1/15) Info

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/23/2024 19:08:03	11/23/2024 19:10:10	1 m 51 s	10 DPUs	G.1X

Run details

Job name	Start time (Local)	Glue version	Last modified on (Local)
Discount_test_job	11/23/2024 19:08:03	4.0	11/23/2024 19:10:10
Id	End time (Local)	Worker type	Log group name
jrcf34bbfb9c8bb536fb21784a152171fde07d725954c796f079fd3b4063ce34fb8c	11/23/2024 19:10:10	G.1X	/aws-glue/jobs
Validation Run Id	Run status	Start-up time	Max capacity
-	Succeeded	15 seconds	10 DPUs
Number of workers	Retry attempt number	Execution time	Execution class
10	Initial run	1 minute 51 seconds	Standard
Timeout	Trigger name	Security configuration	Cloudwatch logs

Screenshot of the AWS Glue Studio interface showing the 'Lot_job' run details.

Job runs (1/1) Info

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/23/2024 19:20:19	11/23/2024 19:22:28	1 m 53 s	10 DPUs	G.1X

Run details

Job name	Start time (Local)	Glue version	Last modified on (Local)
Lot_job	11/23/2024 19:20:19	4.0	11/23/2024 19:22:28
Id	End time (Local)	Worker type	Log group name
jrl5be051adfbba45f2f591112a098cf3ba	11/23/2024 19:22:28	G.1X	/aws-glue/jobs
Validation Run Id	Run status	Start-up time	Max capacity
-	Succeeded	16 seconds	10 DPUs
Number of workers	Retry attempt number	Execution time	Execution class

Screenshot of the AWS Glue Studio job details page for 'time_job'.

Job runs (1/14) Info
Last updated (UTC): November 24, 2024 at 23:03:38

Run details

Job name	Start time (Local)	Glue version	Last modified on (Local)
time_job	11/23/2024 20:46:35	4.0	11/23/2024 20:48:36
Id	End time (Local)	Worker type	Log group name
jr_e944fb4a29270c9293217ad2b092718	11/23/2024 20:48:36	G.1X	/aws-glue/jobs
810a9ee22279d353341f6c7a2fdf33287			
Validation Run Id	Run status	Start-up time	Max capacity
-	Succeeded	17 seconds	10 DPU
Number of workers	Retry attempt number	Execution time	Execution class
10	Initial run	1 minute 44 seconds	Standard
Timeout	Trigger name	Security configuration	Cloudwatch logs

© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

Screenshot of the AWS Glue Studio job details page for 'membership_job'.

Job runs (1/1) Info
Last updated (UTC): November 24, 2024 at 23:03:53

Run details

Job name	Start time (Local)	Glue version	Last modified on (Local)
membership_job	11/23/2024 20:59:35	4.0	11/23/2024 21:01:53
Id	End time (Local)	Worker type	Log group name
jr_050a49677cc69fd6c589293abc96dd1c	11/23/2024 21:01:53	G.1X	/aws-glue/jobs
44d6644af63de53c7bc94097085b2a26			
Validation Run Id	Run status	Start-up time	Max capacity
-	Succeeded	21 seconds	10 DPU
Number of workers	Retry attempt number	Execution time	Execution class

© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

Screenshot of the AWS Glue Studio interface showing the "Slot_job" run details.

Job runs (1/4) Info
 Last updated (UTC): November 24, 2024 at 23:04:10
 View details | Stop job run | Troubleshoot with AI | Table View | Card View

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/23/2024 21:29:33	11/23/2024 21:31:55	2 m	10 DPUs	G.1X

Run details

Job name: Slot_job	Start time (Local): 11/23/2024 21:29:33	Glue version: 4.0	Last modified on (Local): 11/23/2024 21:31:55
Id: jr_85e0cd63bb1451e5fab47dfeee7a6223	End time (Local): 11/23/2024 21:31:55	Worker type: G.1X	Log group name: /aws-glue/jobs
Validation Run Id: bcd5abdc6ebd16c3a5bf334c5dcbe73	Run status: Succeeded	Start-up time: 21 seconds	Max capacity: 10 DPUs
Number of workers: 10	Retry attempt number: Initial run	Execution time: 2 minutes	Execution class: Standard
Timeout: 10	Trigger name: Cloudwatch logs		

Screenshot of the AWS Glue Studio interface showing the "Vehicle_job" run details.

Job runs (1/1) Info
 Last updated (UTC): November 24, 2024 at 23:04:26
 View details | Stop job run | Troubleshoot with AI | Table View | Card View

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/23/2024 21:54:36	11/23/2024 21:57:04	1 m 59 s	10 DPUs	G.1X

Run details

Job name: Vehicle_job	Start time (Local): 11/23/2024 21:54:36	Glue version: 4.0	Last modified on (Local): 11/23/2024 21:57:04
Id: jr_c9f915445aa6e5ef0f5a1c79f81a00f28	End time (Local): 11/23/2024 21:57:04	Worker type: G.1X	Log group name: /aws-glue/jobs
Validation Run Id: 979e6736051ae495c453704910a4de6	Run status: Succeeded	Start-up time: 28 seconds	Max capacity: 10 DPUs
Number of workers: -	Retry attempt number: Execution time	Execution class: Standard	

Screenshot of the AWS Glue Studio home page showing the Member_job configuration.

Member_job

Last modified on 24/11/2024, 13:58:26

Actions | **Save** | **Run**

Job runs (1/1) Info

Last updated (UTC) November 24, 2024 at 23:04:40

Table View | **Card View**

Filter job runs by property

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/23/2024 21:47:05	11/23/2024 21:49:19	1 m 59 s	10 DPUs	G.1X

Run details

Job name	Start time (Local)	Glue version	Last modified on (Local)
Member_job	11/23/2024 21:47:05	4.0	11/23/2024 21:49:19
Id	End time (Local)	Worker type	Log group name
jr_fb93d0fb19f53ed809956870f76cd083 9e8274beb317945b1f312bb4d6f08890	11/23/2024 21:49:19	G.1X	/aws-glue/jobs
Validation Run Id	Run status	Start-up time	Max capacity
-	Succeeded	14 seconds	10 DPUs
Number of workers	Retry attempt number	Execution time	Execution class
-	-	-	-

© 2024, Amazon Web Services, Inc. or its affiliates. | Privacy | Terms | Cookie preferences

Screenshot of the AWS Glue Studio home page showing the fact_payment_job configuration.

fact_payment_job

Last modified on 24/11/2024, 13:58:40

Actions | **Save** | **Run**

Job runs (1/2) Info

Last updated (UTC) November 24, 2024 at 23:04:54

Table View | **Card View**

Filter job runs by property

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/23/2024 22:10:43	11/23/2024 22:12:33	1 m 36 s	10 DPUs	G.1X

Run details

Job name	Start time (Local)	Glue version	Last modified on (Local)
fact_payment_job	11/23/2024 22:10:43	4.0	11/23/2024 22:12:33
Id	End time (Local)	Worker type	Log group name
jr_87426a15324326ec77530f52863bfcbf8 19fdc23f3fa7c2236d99851fa64d934f	11/23/2024 22:12:33	G.1X	/aws-glue/jobs
Validation Run Id	Run status	Start-up time	Max capacity
-	Succeeded	14 seconds	10 DPUs
Number of workers	Retry attempt number	Execution time	Execution class
10	Initial run	1 minute 36 seconds	Standard
Timeout	Trigger name	Security configuration	Cloudwatch logs

© 2024, Amazon Web Services, Inc. or its affiliates. | Privacy | Terms | Cookie preferences

Screenshot of the AWS Glue Studio job **fact_incident_job** showing its runs tab.

Job runs (1/6) Info

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/23/2024 23:28:37	11/23/2024 23:30:54	1 m 56 s	10 DPUs	G.1X

Run details

Job name	fact_incident_job	Start time (Local)	Glue version	Last modified on (Local)
Id	jr_b7955f88d6b031baa0af5a6727b9d21	11/23/2024 23:28:37	4.0	11/23/2024 23:30:54
	76ddda1a03c074fc594c44300f6c4e137	End time (Local)	Worker type	Log group name
Validation Run Id	-	Run status	Start-up time	/aws-glue/jobs
Number of workers	10	Succeeded	21 seconds	Max capacity
Timeout		Retry attempt number	Execution time	10 DPUs
		Initial run	1 minute 56 seconds	Execution class
		Trigger name	Security configuration	Standard
				Cloudwatch logs

Screenshot of the AWS Glue Studio job **fact_booking_job** showing its runs tab.

Job runs (1/16) Info

Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity ...	Worker type
Succeeded	0	11/24/2024 02:34:38	11/24/2024 02:36:52	1 m 58 s	10 DPUs	G.1X

Run details

Job name	fact_booking_job	Start time (Local)	Glue version	Last modified on (Local)
Id	jr_0631bc559aba94ea27b0f465058665c	11/24/2024 02:34:38	4.0	11/24/2024 02:36:52
	2b42ce39f8a0f337f1bdf55586032f45	End time (Local)	Worker type	Log group name
Validation Run Id	-	Run status	Start-up time	/aws-glue/jobs
Number of workers	10	Succeeded	16 seconds	Max capacity
Timeout		Retry attempt number	Execution time	10 DPUs
		Initial run	1 minute 58 seconds	Execution class
		Trigger name	Security configuration	Standard
				Cloudwatch logs

Orchestration of the entire process using step function –

State machine successfully created

GlueWorkflowStateMachine-ParkEasy

[Edit](#)[Actions ▾](#)[Start execution](#)

Details

Arn[arn:aws:states:us-east-1:593793039905:stateMachine:GlueWorkflowStateMachine-ParkEasy](#)**IAM role ARN**[arn:aws:iam::593793039905:role/GlueStepFunctionsExecutionRole](#)**Type**

Standard

Status

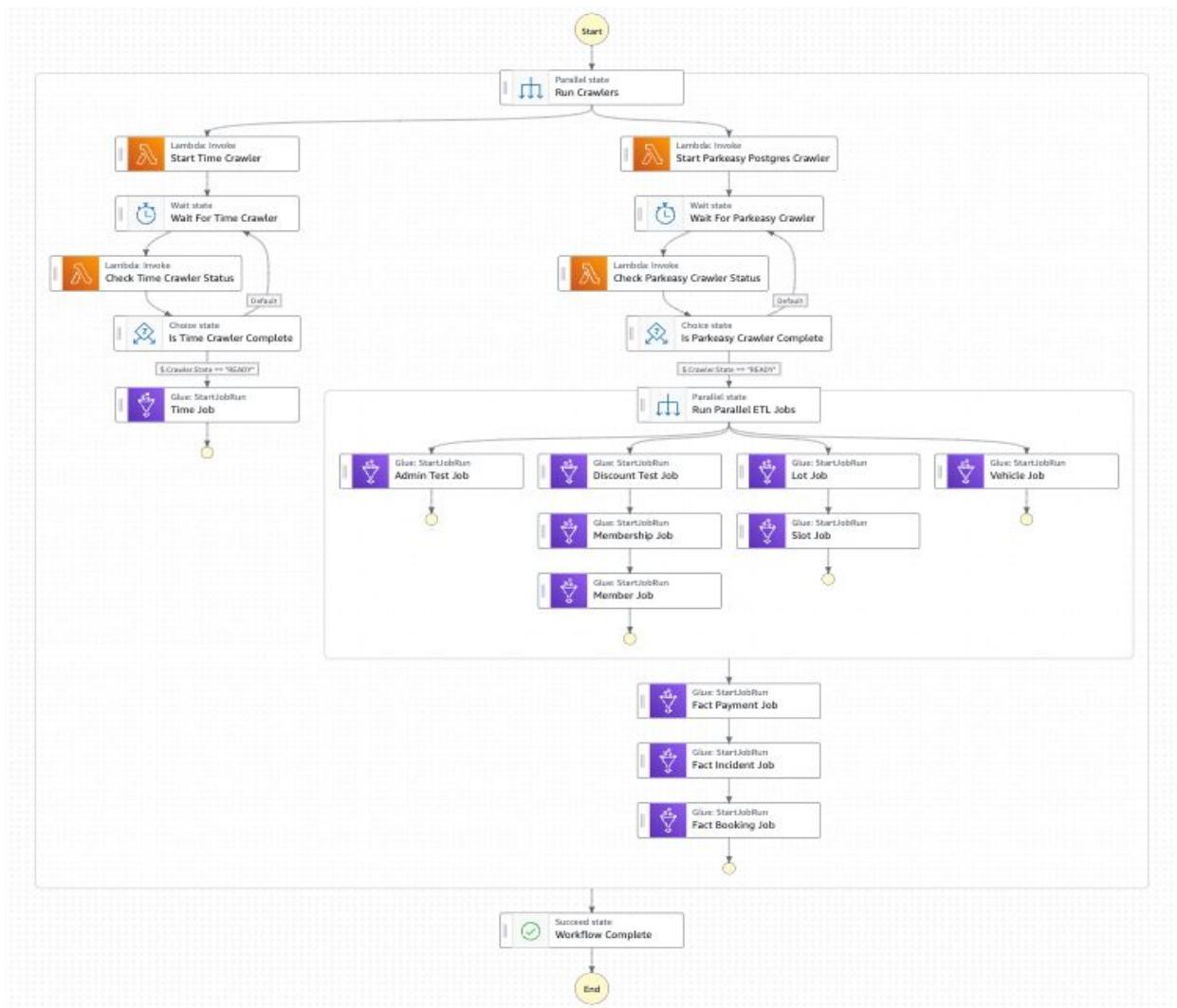
Active

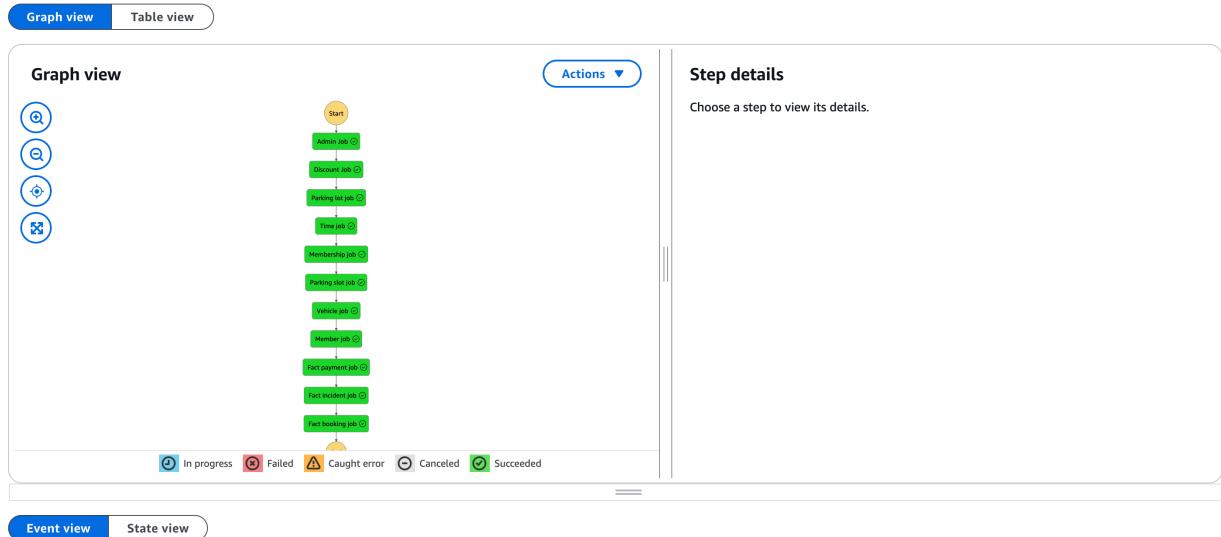
Creation date

Nov 24, 2024, 16:43:51 (UTC-05:00)

X-Ray tracing

Disabled

[Executions](#)[Monitoring](#)[Logging](#)[Definition](#)[Aliases](#)[Versions](#)[Tags](#)



To check if all the Tables were loaded completely, we made use of the Redshift query editor v2

1.Dim_admin table

```
Redshift query editor v2
Editor
Queries
Notebooks
Charts
History
Scheduled queries
CloudShell Feedback
Services Search [Option+S]
Untitled 1 Untitled 2
Run Explain Isolated session parkeasy-reds... dev
1 SELECT * FROM dim_admin;
```

Result 1 (10)					
	adminkey	adminid	name	address	email
1	1	2	Bob Smith	456 Oak Ave, Lincoln, NE	bob.smith@company.com
2	3	5	Eva Brown	202 Birch Ln, Seattle, WA	eva.brown@company.com
3	2	3	Carla Davis	789 Maple Dr, Denver, CO	carla.davis@company.com
4	4	1	Alice Johnson	123 Main St, Springfield, IL	alice.johnson@company.c...
5	6	4	David Lee	101 Pine Rd, Austin, TX	david.lee@company.com
6	5	2	Bob Smith	456 Oak Ave, Lincoln, NE	bob.smith@company.com
7	7	5	Eva Brown	202 Birch Ln, Seattle, WA	eva.brown@company.com
8	8	3	Carla Davis	789 Maple Dr, Denver, CO	carla.davis@company.com
9	10	1	Alice Johnson	123 Main St, Springfield, IL	alice.johnson@company.c...

Query ID 24117 Elapsed time: 169 ms Total rows: 10

2.Dim_discount table

The screenshot shows the AWS Redshift Query Editor v2 interface. On the left is a sidebar with icons for Editor, Queries, Notebooks, Charts, History, Scheduled queries, CloudShell, and Feedback. The main area has tabs for Untitled 1 and Untitled 2. The Untitled 1 tab contains a query editor with the following SQL:

```
1 SELECT * FROM dim_discount;
```

The results are displayed in a table titled "Result 1 (5)".

	discountkey	discountid	status	description	percentage
□	2	4	Expired	Clearance Sale	NULL
□	4	1	Expired	End of Season Sale	NULL
□	1	3	Active	Holiday Discount	NULL
□	3	5	Expired	Special Promotion	NULL
□	5	2	Active	New Customer Discount	NULL

At the bottom right of the results table, it says "Query ID 24126 Elapsed time: 17 ms Total rows: 5".

3.Dim_member table

The screenshot shows the AWS Redshift Query Editor v2 interface. On the left is a sidebar with icons for Editor, Queries, Notebooks, Charts, History, Scheduled queries, CloudShell, and Feedback. The main area has tabs for Untitled 1 and Untitled 2. The Untitled 1 tab contains a query editor with the following SQL:

```
1 SELECT * FROM dim_member;
```

The results are displayed in a table titled "Result 1 (50)".

	memberkey	memberid	phoneno	email	address
□	2	50	+1234567890938	member50@example.com	4748 Poplar Valley Ln
□	4	18	+1234567890834	member18@example.com	1516 Ashbrook Dr
□	6	8	+1234567890506	member8@example.com	505 Beech St
□	8	16	+1234567890524	member16@example.com	1314 Birchwood Blvd
□	10	36	+1234567890494	member36@example.com	3334 Birch Valley Rd
□	12	24	+1234567890749	member24@example.com	2122 Oak Ridge Rd
□	14	23	+1234567890965	member23@example.com	2021 Pine Hill Blvd
□	16	27	+1234567890710	member27@example.com	2425 Cedar Hollow Ln
□	18	29	+1234567890389	member29@example.com	2627 Walnut Grove St

At the bottom right of the results table, it says "Query ID 24139 Elapsed time: 13 ms Total rows: 50".

4.Dim_membership table

Redshift query editor v2

```
1 SELECT * FROM dim_membership;
```

Result 1 (50)

	membershipkey	membershipid	name	status	discountkey	st
□	2	38	Platinum	Inactive	NULL	20
□	4	49	Gold	Active	NULL	20
□	6	19	Bronze	Inactive	NULL	20
□	8	26	Silver	Active	NULL	20
□	10	13	Bronze	Inactive	NULL	20
□	12	46	Bronze	Active	NULL	20
□	14	9	Gold	Active	NULL	20
□	16	31	Gold	Active	NULL	20
□	18	24	Gold	Inactive	NULL	20

Query ID 24142 Elapsed time: 30 ms Total rows: 50

5.Dim_parkinglot table

Redshift query editor v2

```
1 SELECT * FROM dim_parkinglot;
```

Result 1 (10)

	lotkey	lotid	name	location	region	ca
□	2	10	South Ridge Lot	404 South Ridge Rd, Aus...	South	10
□	4	7	City Hall Lot	101 City Hall Plz, Austin, TX	East	10
□	6	5	Stadium Lot	345 Stadium Ln, Seattle, ...	South	10
□	8	2	Downtown Garage	45 Broad Ave, Lincoln, NE	South	10
□	10	3	Central Park Lot	789 Elm Dr, Denver, CO	West	10
□	1	1	Main Street Lot	123 Main St, Springfield, IL	East	10
□	3	4	Riverfront Lot	12 Riverside Rd, Austin, TX	South	10
□	5	9	North Plaza Lot	303 North Plaza, Denver, ...	North	10
□	7	8	Museum Lot	202 Museum Dr, Lincoln, ...	South	10

Query ID 24153 Elapsed time: 14 ms Total rows: 10

6. Dim_parkingslot table

The screenshot shows the AWS Redshift query editor v2 interface. On the left is a sidebar with icons for Editor, Queries, Notebooks, Charts, History, Scheduled queries, CloudShell, and Feedback. The main area has tabs for Untitled 1 and Untitled 2. Untitled 1 is active and contains the following SQL query:

```
1 SELECT * FROM dim_parkingslot;
```

The results are displayed in a table titled "Result 1 (100)". The table has columns: slotkey, slotid, status, price, type, and location. The data is as follows:

slotkey	slotid	status	price	type	location
2	441	Occupied	20.00	Compact	5
4	279	Occupied	10.00	Large	3
6	280	Occupied	15.00	Compact	3
8	357	Occupied	20.00	Large	4
10	57	Occupied	20.00	Compact	1
12	537	Occupied	15.00	Large	6
14	68	Available	20.00	Standard	1
16	581	Available	15.00	Standard	6
18	849	Available	20.00	Standard	9

At the bottom right of the results table, it says "Query ID 24158 Elapsed time: 21 ms Total rows: 100".

7. Dim_time table

The screenshot shows the AWS Redshift query editor v2 interface. The sidebar and layout are identical to the previous screenshot. Untitled 1 is active and contains the following SQL query:

```
1 SELECT * FROM dim_time;
```

The results are displayed in a table titled "Result 1 (100)". The table has columns: timekey, timeid, year, quarter, month, day, hour, minute, and second. The data is as follows:

timekey	timeid	year	quarter	month	day	hour	minute	second
1	20230101	2023	1	1	1	1	1	1
3	20230102	2023	1	1	1	1	1	2
5	20230103	2023	1	1	1	1	1	3
7	20230104	2023	1	1	1	1	1	4
9	20230105	2023	1	1	1	1	1	5
11	20230106	2023	1	1	1	1	1	6
13	20230107	2023	1	1	1	1	1	7
15	20230108	2023	1	1	1	1	1	8
17	20230109	2023	1	1	1	1	1	9

At the bottom right of the results table, it says "Query ID 24161 Elapsed time: 11 ms Total rows: 100".

8. Dim_vehicle table

The screenshot shows the AWS Redshift Query Editor v2 interface. On the left is a sidebar with icons for Editor, Queries, Notebooks, Charts, History, Scheduled queries, CloudShell, and Feedback. The main area has tabs for Untitled 1 and Untitled 2. A toolbar at the top includes Run, Explain, Isolated session, dev, Schedule, and various export options. The query in Untitled 1 is:

```
1 SELECT * FROM dim_vehicle;
```

The results table is titled "Result 1 (50)" and contains the following data:

	vehiclekey	vehiclenumber	make	vehicletype	color	rn
□	2	VN_46	Toyota	SUV	Red	29
□	4	VN_27	Ford	SUV	Red	32
□	6	VN_38	Honda	Sedan	Blue	3
□	8	VN_11	Ford	Truck	Blue	7
□	10	VN_32	Toyota	Sedan	Green	6
□	12	VN_47	Honda	Sedan	Blue	31
□	14	VN_44	Toyota	Sedan	Blue	50
□	16	VN_2	Toyota	Sedan	Red	44
□	18	VN 1	Toyota	SUV	Green	12

Query ID 24170 Elapsed time: 19 ms Total rows: 50

9. Fact_booking table

The screenshot shows the AWS Redshift Query Editor v2 interface. The sidebar and toolbar are identical to the previous screenshot. The query in Untitled 2 is:

```
1 SELECT * FROM fact_booking;
```

The results table is titled "Result 1 (100)" and contains the following data:

	bookingkey	bookingid	start_datetime	end_datetime	status	tr
□	2	948	2024-08-13	2024-08-16	Confirmed	94
□	4	856	2024-10-28	2024-10-30	Confirmed	85
□	6	477	2024-10-27	2024-10-29	Cancelled	47
□	8	943	2024-10-16	2024-10-18	Confirmed	94
□	10	129	2024-08-31	2024-09-01	Cancelled	12
□	12	353	2024-08-23	2024-08-24	Cancelled	35
□	14	732	2024-10-16	2024-10-18	Pending	73
□	16	445	2024-08-17	2024-08-20	Cancelled	44
□	18	508	2024-09-19	2024-09-21	Confirmed	50

Query ID 24177 Elapsed time: 12 ms Total rows: 100

10. Fact_incident table

Redshift query editor v2

```
1 SELECT * FROM fact_incident;
```

	incidentkey	incidentid	resolutionstatus	date_time	description	rn
□	2	20	Resolved	2024-11-27	Incident_20	19
□	4	42	Unresolved	2024-11-03	Incident_42	7
□	6	31	Unresolved	2024-11-21	Incident_31	36
□	8	35	Resolved	2024-11-17	Incident_35	24
□	10	18	Resolved	2024-11-14	Incident_18	27
□	12	17	Unresolved	2024-11-27	Incident_17	12
□	14	25	Resolved	2024-11-16	Incident_25	13
□	16	11	Resolved	2024-11-26	Incident_11	12
□	18	4	Resolved	2024-11-25	Incident 4	34

Query ID 24180 Elapsed time: 11 ms Total rows: 50

11.Fact_payment table

Redshift query editor v2

```
1 SELECT * FROM fact_payment;
```

	transactionid	datetimeid	paymentstatus	amount	memberid	pn
□	924	2024-10-11	Completed	125.42	3	1
□	978	2024-10-24	Completed	81.08	17	1
□	115	2024-10-14	Pending	41.58	42	1
□	331	2024-10-31	Completed	101.21	18	1
□	220	2024-10-23	Pending	162.25	49	1
□	405	2024-11-03	Completed	98.47	13	1
□	192	2024-10-29	Pending	66.89	20	1
□	538	2024-11-01	Completed	25.41	22	1
□	603	2024-10-30	Completed	66.44	12	1

Query ID 24191 Elapsed time: 15 ms Total rows: 100

13. Dashboard Creation:

Dashboard 2: Financial Performance & Customer Engagement Report

This dashboard aims to track the financial health of the parking management operations and gauge customer engagement through membership and booking patterns. It focuses on revenue generation, the impact of discount strategies, and customer loyalty, providing a strategic view of business growth and sustainability.

Key Performance Indicators (KPIs)

1. Total Revenue from Bookings (Over Time)

- Formula: Total Revenue = SUM(Amount) (extracted from FACT_Payment table)
- Purpose: This indicator provides a longitudinal view of revenue trends, helping to gauge the financial health and growth trajectory of the parking operations.
- Display: Shown as a cumulative monetary value, this KPI allows for immediate assessment of financial status over selectable time frames.

2. Percentage of Active Memberships

- Formula: Active Membership % = (COUNT(Active Memberships) / COUNT(Total Memberships)) × 100
- Purpose: This percentage illuminates the level of engagement and retention of members, serving as a barometer for customer loyalty and the effectiveness of membership programs.
- Display: Visualized as a percentage, this metric highlights engagement trends and can drive strategies to boost membership renewals and incentives.

3. Average Revenue Per User (ARPU):

- Formula: ARPU = Total Revenue / Total Active Members.
- Purpose: Provides a measure of the revenue generated per user, helping to assess the value contributed by each member.
- Display: As a monetary value, ideally displayed next to total revenue for a quick comparative assessment.

4. Cost of Acquisition vs Lifetime Value:

- Formula: COA = Total Marketing Spend / Number of New Members; LTV = Average Revenue Per Member × Average Membership Duration.
- Purpose: These metrics compare the cost of acquiring new members with the revenue they generate over time, indicating the efficiency and sustainability of marketing efforts.
- Display: Displayed as a dual-line chart for direct comparison.

5. Member Churn Rate:

- Formula: Churn Rate = (Number of Members at Start of Period - Number of Members at End of Period) / Number of Members at Start of Period.
- Purpose: Measures the rate at which members discontinue their services, crucial for identifying issues in customer satisfaction or competitive challenges.

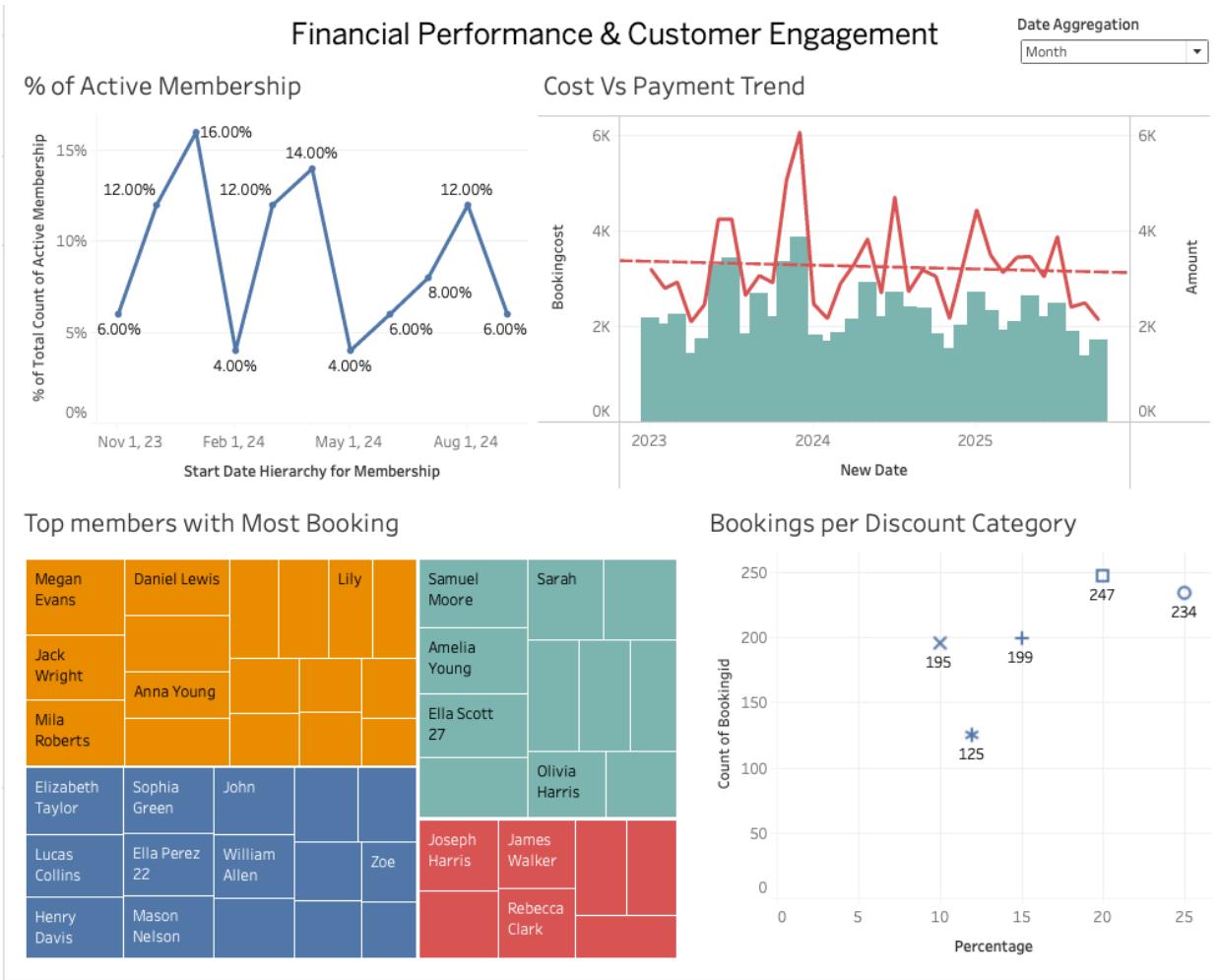
- Display: Visualized as a simple percentage, highlighting trends over selectable time frames.

6. Utilization Rate by Member Tier:

- Formula: Utilization Rate = Number of Bookings by Tier / Total Number of Slots Available.
- Purpose: Assesses how different membership tiers are using the parking facilities, which can help in tailoring services and benefits for different customer segments.
- Display: Presented as a segmented bar chart for each membership tier.

Visualizations

1. Number of Bookings Per Discount Category (Scatter Plot)
 - Purpose: Analyzes the impact of discount categories on booking behavior, guiding promotional strategies and discount offerings.
 - Key Metric: Bookings Impact from Discounts.
 - Details: The scatter plot categorizes bookings by discount levels, providing a visual correlation between discounts and booking frequency.
2. Members with Most Bookings (Tree Map)
 - Purpose: Identifies top customers by volume of bookings, essential for recognizing and rewarding loyalty and understanding customer value.
 - Key Metric: Top Customers by Bookings.
 - Details: The tree map offers a hierarchical view of members, highlighting those with the highest engagement.
3. Revenue Impact from Discounts (Bar Chart)
 - Purpose: This chart displays the portion of revenue derived from discounted bookings, vital for evaluating the financial impact of discounts on overall earnings.
 - Key Metric: Revenue from Discounted Bookings.
 - Details: By showing revenue from discounted versus full-price bookings, this visualization helps assess the profitability of discount strategies.
4. Repeat Customer Rate (Pie Chart)
 - Purpose: Shows the ratio of repeat to new customers, indicating the success of customer retention efforts and overall satisfaction.
 - Key Metric: Repeat Customer Percentage.
 - Details: The pie chart simplifies the comparison, making it easy to visualize loyalty metrics at a glance.
5. Incident Resolution Rate (Line Chart)
 - Purpose: Monitors the efficiency and timeliness of incident resolutions, critical for maintaining high standards of customer service.
 - Key Metric: Incident Resolution Rate.
 - Details: Trends in incident resolution are tracked over time, highlighting improvements or areas needing attention in customer support processes.



Analysis:

Here's an analysis of the key components in the dashboard:

- Percentage of Active Membership:** This line graph shows fluctuations in active membership percentages over time, with significant peaks and troughs. It measures the proportion of members actively using the parking facilities, which is essential for understanding customer retention and loyalty trends.
- Cost vs Payment Trend:** This combined line and bar chart tracks both the booking costs and payments over time, with costs generally surpassing payments. This visualization is crucial for assessing the financial sustainability of the operations, identifying periods of revenue shortfall or surplus.
- Top Members with Most Bookings:** The tree map highlights the members with the highest number of bookings. This helps in recognizing high-value customers and tailoring marketing strategies to enhance customer loyalty and satisfaction.
- Bookings per Discount Category:** This scatter plot shows the relationship between discount levels and the number of bookings, useful for evaluating the effectiveness of different discount strategies in driving bookings.