DIWATA GAS CORPORATION

EMPOWERING ORDER MANAGEMENT, LOGISTICS, AND INVENTORY THROUGH A STREAMLINED DATA ARCHITECTURE

1 INTRODUCTION

1.1 Company Background

Diwata Gas Corporation manufactures and distributes industrial and medical gases. Their customer base includes hospitals, construction companies, steel fabrication shops, and repair shops, among others. Most of the company's branches are in Luzon, with one in the Visayas area. They offer 17 gas products such as oxygen, acetylene, and carbon dioxide, available in cylinders of five different sizes.

To give an overview, the business model operates on the basis that the cylinders or gas tanks are either owned by the customer or Diwata Gas Corporation. Diwata Gas Corporation is exclusively responsible for refilling these tanks. The primary product sold is the gas within the tanks. After delivery and consumption, the company's drivers retrieve the empty cylinders to be refilled, continuing this cycle of delivery and collection.

1.2 Problem Statement

The company has long faced a problem that impacts the marketing, logistics, and inventory departments. Due to the manual and independent processes of these departments, there is inefficiency in routing customer orders. The departments are somewhat disconnected as they transfer information ineffectively. The marketing department is responsible for collecting and managing all incoming customer orders, while the inventory management team ensures the release of gas cylinders to fulfill these orders. These orders are then assigned to delivery drivers, which are managed by the logistics team. Although the process appears to flow smoothly at a high level, the departments are considered disconnected because they handle handovers by passing around Excel files and physical documents without clear instructions or context on how to proceed.

Given this, the question now is: **How can the team design an effective and efficient data architecture which links the processes of marketing,**

logistics, and inventory departments and semi-automatically handles transactional order data and generates reports?

2 ARCHITECTURE

The architecture designed effectively meets the operational requirements of Diwata Gas Corporation by integrating four key business processes: sales tracking, inventory management, fleet assignment, and cylinder complaints management. This is accomplished by routing, storing, and processing data from field inputs captured through a web application to producing analytical reports and historical data summaries using OLAP (Online Analytical Processing) and NoSQL databases.

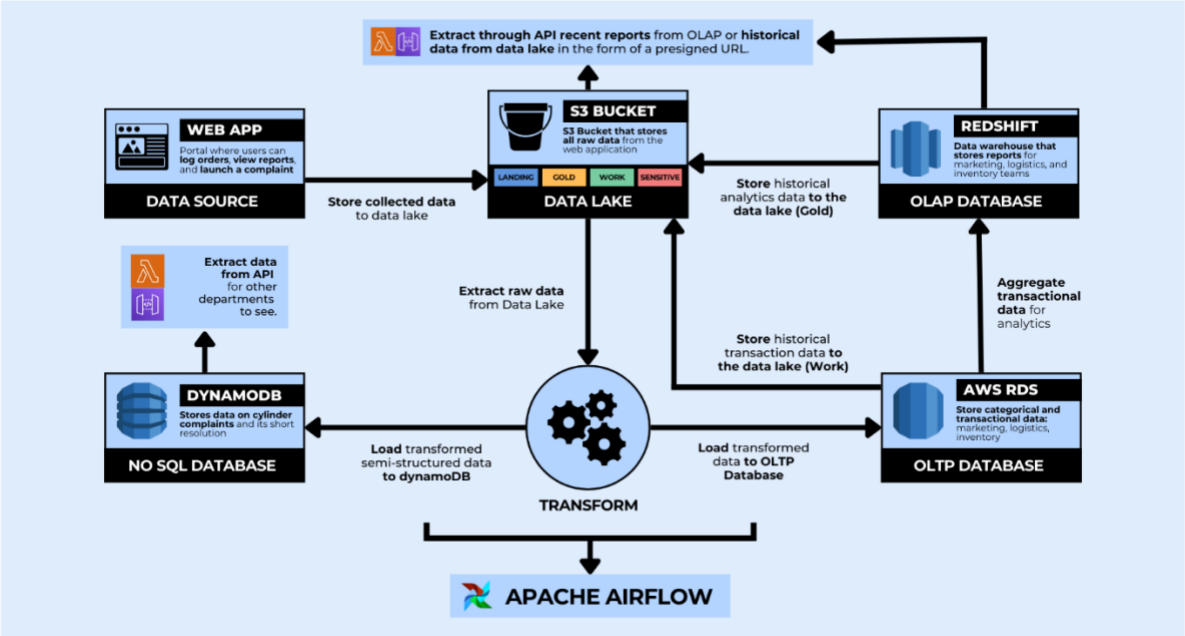


Figure 1. Diwata Gas Corporation Data Architecture

To provide a comprehensive overview of each component, the system architecture starts with the web application, which acts as the main entry point for data. Initially, to ensure data accuracy and consistency, staff members enter data on behalf of customers. As input fields are improved and directed acyclic graphs (DAGs) capable of handling complex rules are introduced, the system will evolve to allow customers to directly input on their own in the future.

The architecture incorporates three specific forms that require information coming from customers: the customer registration form, which adds them to the customer database; the customer order form, which

facilitates the purchase of various filled gas cylinders; and the customer complaint form, which enables them to report any product issues. Additionally, there are two internal forms: the logistics form, which assigns orders to the company's fleet, and the complaint resolution form, which addresses and resolves customer complaints.

From the web application, the raw data is captured in JSON format and transferred to a data lake, stored in an S3 bucket. This bucket includes a designated landing zone for long-term storage of such data and acts as a staging area. Here, the data is organized and cleansed of any sensitive information before further processing. Subsequently, the data undergoes ETL (extract, transform, load) processing with Apache Airflow, which facilitates its daily transfer or *upserting* or both to both Postgres-based OLTP and DynamoDB-based NoSQL databases.

The DynamoDB database, a key-value NoSQL database, is employed to handle customer complaints. Its scalability is particularly advantageous, allowing it to manage a high volume of complaints while maintaining high availability to respond to customer issues promptly. Moreover, its key-value feature allows users to easily query complaints based on the serial number of the tank as well as the specific complaint ID. Additionally, the OLTP database is used to store the company's transaction data, including details related to products, customers, and delivery information. Using Apache Airflow, OLTP data is aggregated and transferred or *upserted* daily to the Redshift-based OLAP database which organizes data into reports in the form of three distinct fact tables using Redshift. These are the daily sales tracker, daily inventory tracker, and driver incentives tracker.

Using Apache Airflow, monthly historical reports and data from the OLAP and OLTP databases are stored every month in the Gold and Work zones in the data lake, respectively. These historical reports and data, along with the customer complaints from DynamoDB can be accessed or requested by users through API requests or calls, which would return JSON-formatted responses.

For additional details on each component of the architecture, the following two sections will delve deeper into the design considerations and technical specifics.

3 DESIGN CONSIDERATIONS

3.1 Web Application

The design of the static webpages hosted on Amazon Simple Storage Service (S3) emphasizes scalability, high availability, user-friendliness, and security (Muthiah, 2023). By leveraging S3's robust infrastructure, the system can

efficiently accommodate large number of employees and customers without server limitations. Each webpage serves a specific function, such as order management, customer registration, or logistics tracking. This clear separation of functions not only enhances performance but also simplifies future updates and maintenance.

The S3 API is optimized for handling light data inputs and image uploads, integrating seamlessly with S3 for scalable storage while ensuring secure interactions with the database. Despite high expected user traffic, the data being processed is minimal, allowing for efficient handling. Uploaded files are validated to accept only appropriate file types, safeguarding the system from potential risks. S3's scalability ensures the system can handle frequent uploads and large data volumes, with multipart uploads further optimizing performance. The database stores only essential metadata like file URLs and sizes, reducing its load and ensuring fast access to stored files.

To maintain data consistency, the S3 API automatically creates corresponding database records for each file uploaded, linking stored files with their associated metadata. Comprehensive error handling is built into the system, providing clear messages for upload issues and integrated logging to aid in debugging. This approach, which hinges on data consistency and quality, enables the system to scale easily, adapt to future storage needs, and be ready for more complex processing requirements as they arise.

Overall, the design of this platform prioritizes robust functionality and scalability, creating a user-friendly experience that can grow alongside the company while ensuring operational efficiency.

3.2 Data Lake

The data lake is designed to address the company's need for a centralized data repository, serving as a single source of truth for a wide range of data, including orders, sales, inventory, logistics, and customer data. By managing the entire data lifecycle, from raw data ingestion to structured data storage and historical retention from OLAP and OLTP systems, it ensures seamless data access via API endpoints, improving operational efficiency.

To support this growing data ecosystem, AWS S3 was selected as the storage platform due to its unmatched scalability and flexibility. As the company's data needs evolve, S3 provides secure and adaptable storage for a variety of data formats, such as structured data from traditional systems, semi-structured information from NoSQL databases, and unstructured content such as images. This centralized approach allows the data lake to serve as a cohesive hub, streamlining data management and usage.

3.3 NoSQL

DynamoDB is an ideal choice for Diwata Gas's Complaint Management System due to its ability to efficiently handle semi-structured data. To further highlight why it is used instead of a relational database, some attributes of the complaints may differ in length and may not be present across all complaints, which necessitates having a database with a flexible schema instead of a fixed schema (Google Cloud, n.d.).

The system needs to facilitate tasks such as creating and updating complaints, escalating complaints, creating and reading comments, retrieving all complaints by a customer, and gathering all comments by an agent, along with viewing all escalations. Since these operations can predominantly rely on key-value lookups (i.e., looking up the complaints by cylinder serial number or complaint ID or both), DynamoDB's design aligns perfectly with these transactional needs (Amazon Web Services, 2024).

Beyond basic functionality, DynamoDB is highly scalable (Google Cloud, 2024), ensuring it can grow with Diwata Gas as the number of customers and complaints increase. By leveraging DynamoDB, Diwata Gas can efficiently manage its growing customer base and complaint volume while ensuring scalable, reliable, and compliant data storage and retrieval.

3.4 Online Transactional Processing (OLTP) Database

The OLTP database was designed to primarily store individual order transactions between the customer and the company in an organized manner. It was designed to connect the processes of relevant departments such as sales and marketing, who collates customer orders; inventory, who releases cylinders to fulfill orders; and logistics, who assign drivers to deliver goods.

The customer order table is closely linked to the sales and marketing team, capturing all necessary transaction details. It provides comprehensive information to support the sales and marketing efforts, including specifics about the branch, customer, responsible salesperson, and sales area, all interconnected through dedicated tables linked to the main order table. Each order also details multiple cylinder products, specifying their contents and dimensions via linked tables like cylinder, product, and cylinder type—information vital for inventory management. Additionally, order transactions are tied to the logistics table, which assigns drivers to each order. This logistics table, in turn, connects to a driver table that holds detailed information about the fleet members.

3.5 Online Analytical Processing (OLAP) Database

The team developed the OLAP database to produce actual reports that the departments regularly review on a periodic basis. To facilitate this, three fact

tables were established, which are aggregations derived from data in the OLTP database.

The first fact table is designed for the sales and marketing team, enabling them to track daily sales and compile aggregated monthly reports. This daily sales fact table provides insights into business performance based on quantity sold and total sales revenue. The second fact table serves the inventory team by allowing them to monitor daily cylinder stock levels. This enables them to identify available cylinders for release and those that need refilling or repairs. The final fact table supports the logistics team by tracking the number of pickups and deliveries made by drivers daily, which is crucial since drivers receive incentives for each cylinder they pick up and deliver.

It is important to note that the dimensions and facts in the OLAP databases align with the fields used in the actual departmental reports. For the daily inventory report, all fields mirror those used in real-world operations. However, for sales and logistics monitoring reports, the team has introduced additional fields to enhance analytical capabilities. In sales monitoring, the usual fields include date, customer, cylinder type, and product, with the team adding dimensions like sales area, branch, and payment method to facilitate a more detailed analysis of sales data. Additionally, an area dimension has been added to the logistics monitoring fact table to offer a geographical view of driver deliveries.

3.6 Workflow Manager

A workflow manager, in this case Apache Airflow, is integrated into the architecture to automatically manage the scheduling of the ETL processes of the OLTP, OLAP, and NoSQL database. Moreover, it is also used to manage the scheduling of the RETL processes of both the OLTP and OLAP databases. In addition to these functionalities, the manager also stores the connection credentials of the company to these databases and connects these credentials to these databases, allowing operations to be done on these databases.

3.7 API Endpoints

API endpoints were created in order (1) to provide an endpoint for users to request or download information from the NoSQL database, OLAP database, and data lake, (2) to provide a micro or modular feature which may be integrated into the web application in the future, and (3) to provide read-only access (and not write access) of the database to possible end users such as complaints IT officers, data analysts, and data scientists.

4 COMPONENTS

4.1 Web Application

The web applications developed for Diwata Gas are designed to optimize operational workflows and enhance customer service through structured data collection across key business functions.

Order Management Form

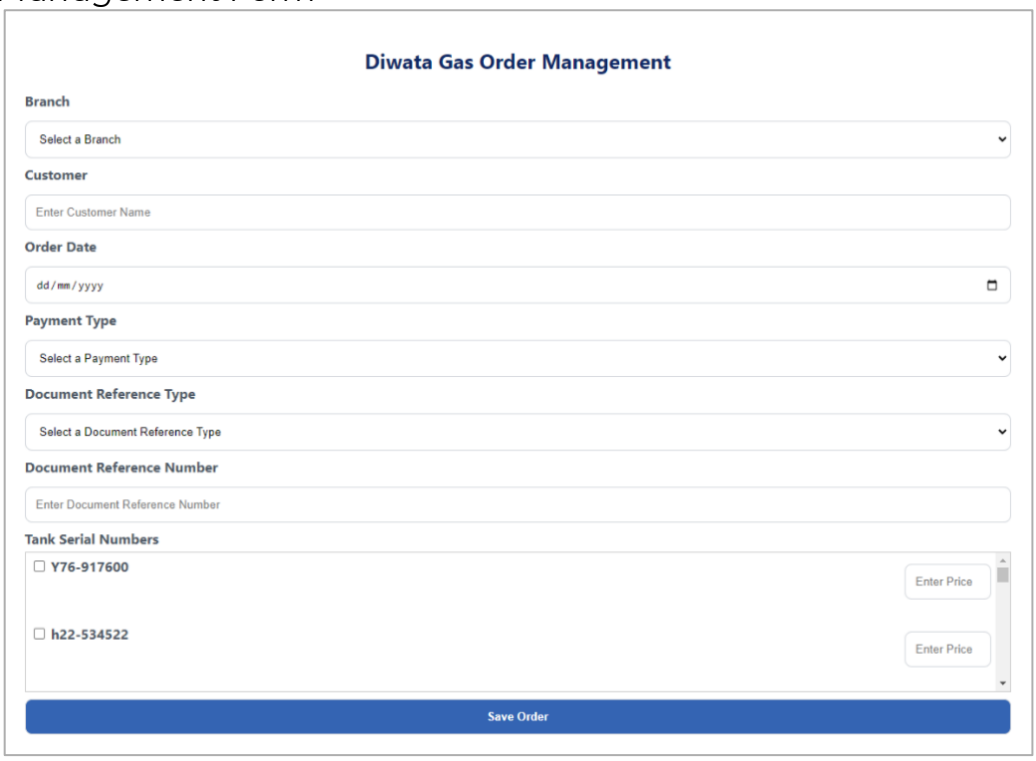


Figure 2. Diwata Gas Order Management Form Webpage

This form (see **Figure 2**) collects the branch name, customer information, order date, payment method, and document references. By centralizing this data, the form enables near real-time order tracking, allowing for efficient multi-branch inventory management and optimized delivery schedules. This seamless order tracking forms the backbone of the company's sales operations, ensuring that customer demands are met swiftly and accurately.

Customer Registration Form

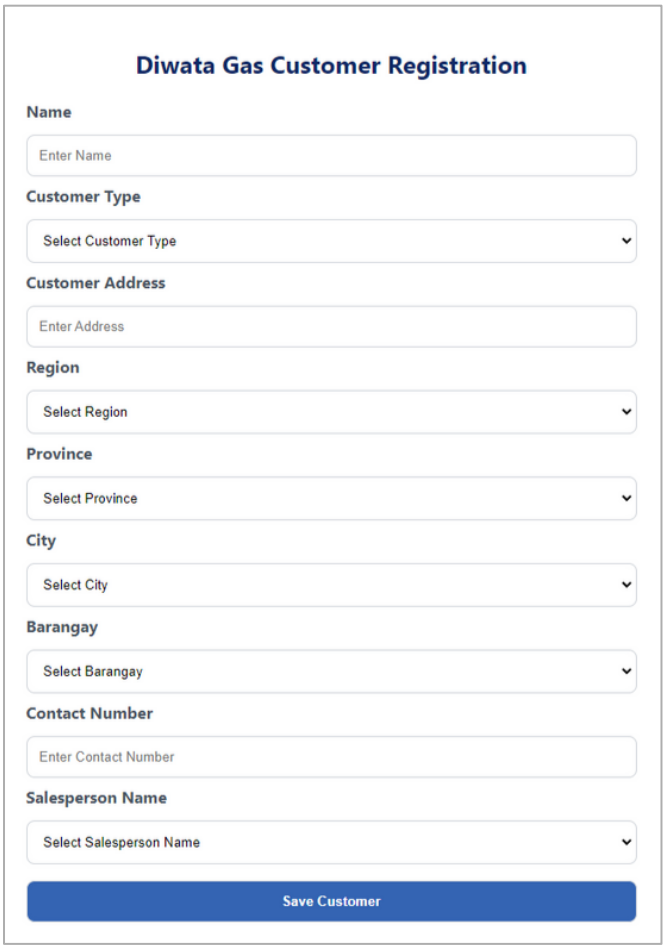


Figure 3. Diwata Gas Customer Registration Form Webpage

On the other hand, the Customer Registration Form (see **Figure 3**) gathers detailed information about each customer, including their name, type, address, and contact details. By incorporating dropdown menus for geographic data such as region, province, city, and barangay, this form provides details in logistics planning and service delivery. The ability to assign a specific salesperson further improves customer engagement, creating a direct line of communication that supports both sales and aftercare.

Logistics Management Form

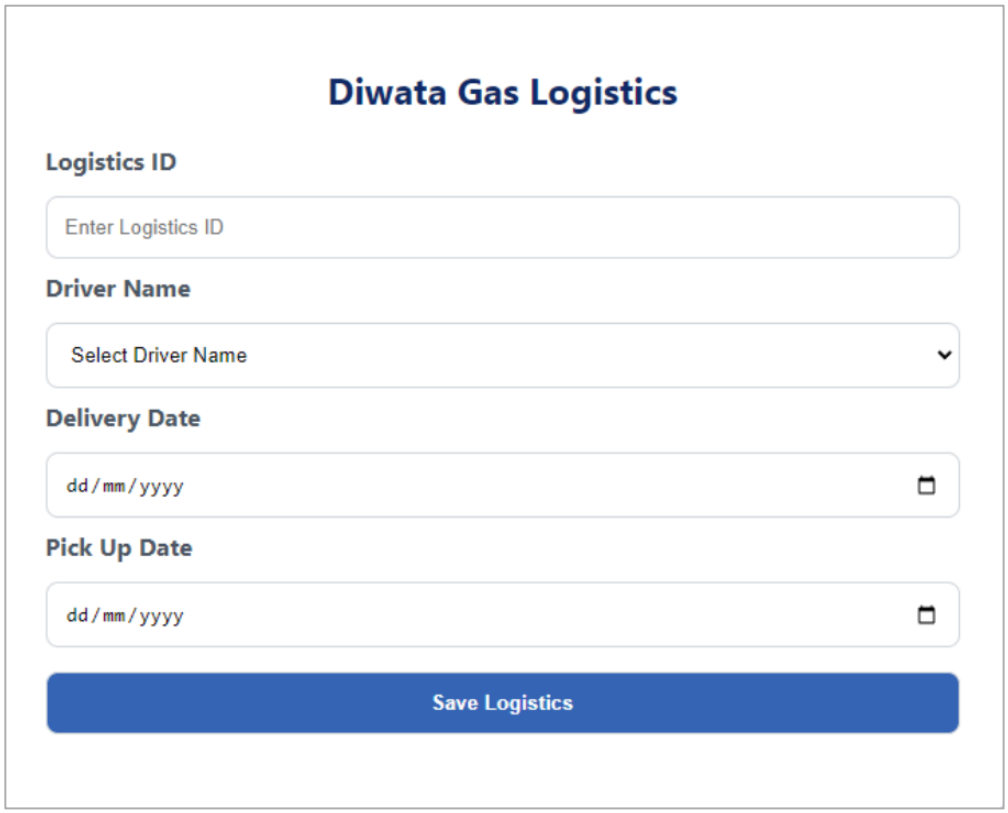


Figure 4. Diwata Gas Logistics Form Webpage

Once orders are placed and customers registered, the Logistics Management Form (see **Figure 4**) comes into play, capturing critical logistics metadata like logistics ID, driver details, delivery dates, and pickup schedules. This form ensures complete traceability throughout the supply chain, linking seamlessly with the order management system to monitor deliveries in real time. As a result, Diwata Gas can optimize dispatch operations and enhance the overall efficiency of its logistics network.

Customer Complaint Form

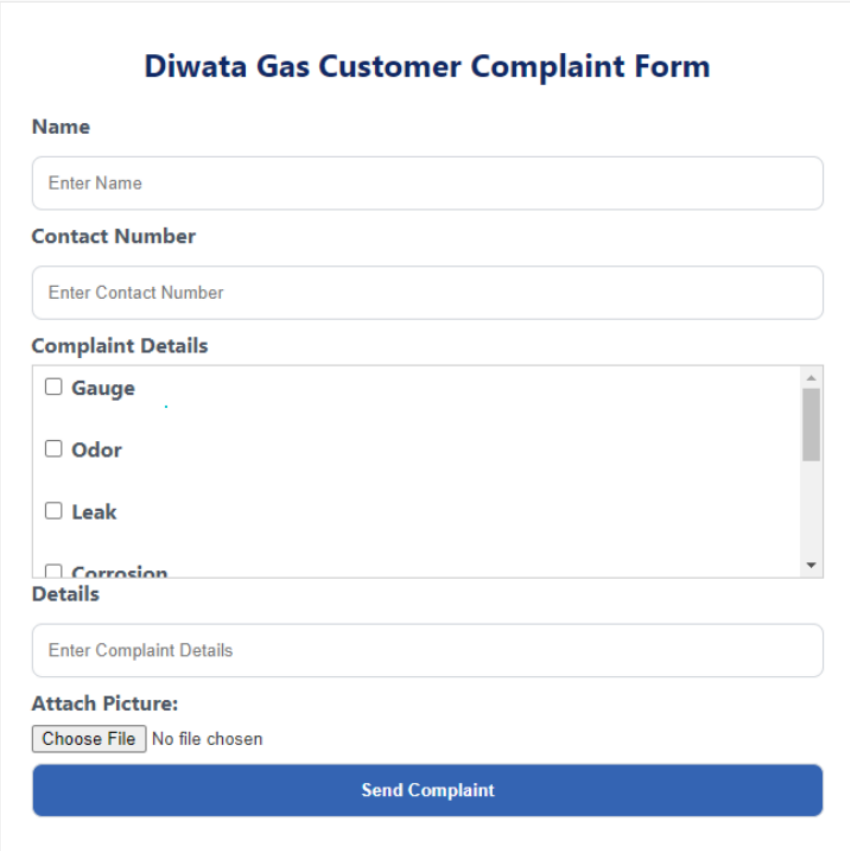


Figure 5. Diwata Gas Customer Complaint Form Webpage

To maintain high standards of product quality and service, Diwata Gas uses the Customer Complaint Form (see **Figure 5**) to standardize the reporting of issues related to gas cylinders. By collecting key identifiers, cylinder serial numbers, and categorizing complaints with predefined checkboxes, the form enables efficient handling and categorization of complaints.

Complaint Resolution Form

Figure 6. Diwata Gas Customer Complaint Resolution Form Webpage

Following the complaint process, the Complaint Resolution Form (see **Figure 6**) collects data on how the issues are resolved. It references complaint IDs and cylinder serial numbers, recording actions like product replacement or repairs. A dedicated comment section allows for detailed annotations, creating a valuable repository of data that can be analyzed to refine resolution strategies and improve operational performance over time.

These web-based forms seamlessly integrate into Diwata Gas's data lake, providing a centralized source of rich data.

4.2 Data Lake

Data Lakes needs the organization of data into distinct zones to maintain data integrity, security, and accessibility through its lifecycle. Each zone serves a specific purpose in the data management process, providing efficient workflows for data engineers, analysts, scientists, and stewards.

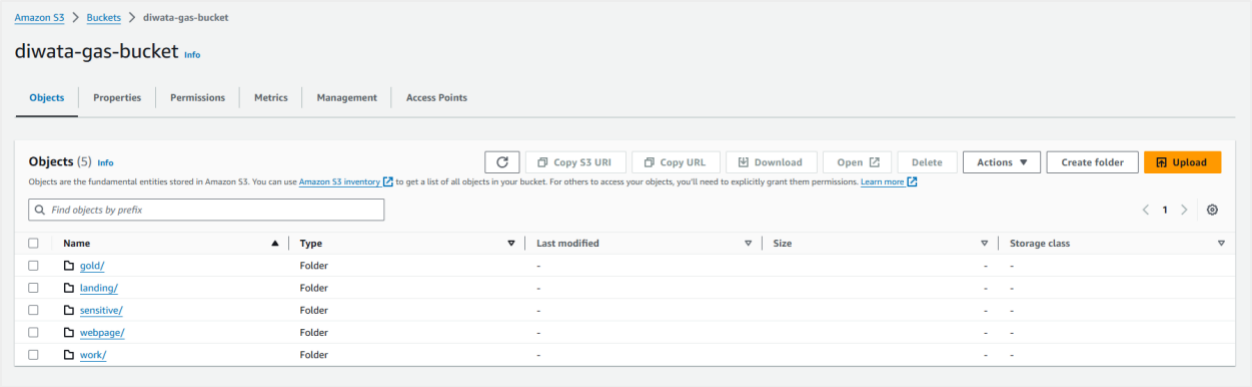


Figure 7. Diwata Gas Data Lake (S3 Bucket) and Zones

Landing Zone

Staging area where raw data is ingested into the data lake and can be stored indefinitely. This zone excludes sensitive data and is designed to quickly ingest raw information before transformation. This stores the raw inputs from the forms in web application such as Customer Order Fields, Logistics Form Fields, Customer Complaint Forms, and Customer Complaint Resolution Forms. The Landing Zone was created to provide storage for raw data to be organized and made accessible for initial processing tasks, allowing data engineers to efficiently begin their work. Access to this zone is governed minimally, with security measures in place to ensure that no sensitive data enters.

Work Zone

This zone holds cleaned and transformed historical data from OLTP, serving as a workspace for data scientists to manipulate, explore, and experiment with the data. This zone stores Monthly Customer Order and Logistics data, providing flexibility for ongoing analysis before the data becomes fully curated. The Work Zone's purpose is to act as a sandbox for further data manipulation, with minimal governance applied. Access is primarily granted to data scientists, ensuring no sensitive data is present while allowing them to work efficiently, with additional access provided to data engineers, who may need to adjust data pipelines or assist with the data transformations.

Gold Zone

Storage area for curated and trusted data that is ready for analysis and business reporting. This data has undergone extensive transformation and cleaning to ensure it is high quality. This stores historical data from OLAP such as Monthly Sales Fact Tables and Monthly Logistics Fact Tables. The Gold Zone was created to have reliable data for critical business decisions, offering clean and enriched data for analysis. Strict access control is enforced, with only data

scientists and business analysts granted access, as they require trusted data with clear lineage and guaranteed data quality.

Sensitive Zone

This zone is dedicated to highly confidential data such as the inputs from the webpage of the Customer Registration Form and employee details which demand the highest level of security. Its main purpose is to ensure that sensitive data remains compliant with privacy regulations and is stored securely, away from general operational or analytical data. Access to this zone is heavily restricted to data stewards and selected individuals.

Main Classes of Users and their Access in the Data Lake

At Diwata Gas, different roles within the organization have varying levels of access to the data lake, reflecting their responsibilities and the need for data security.

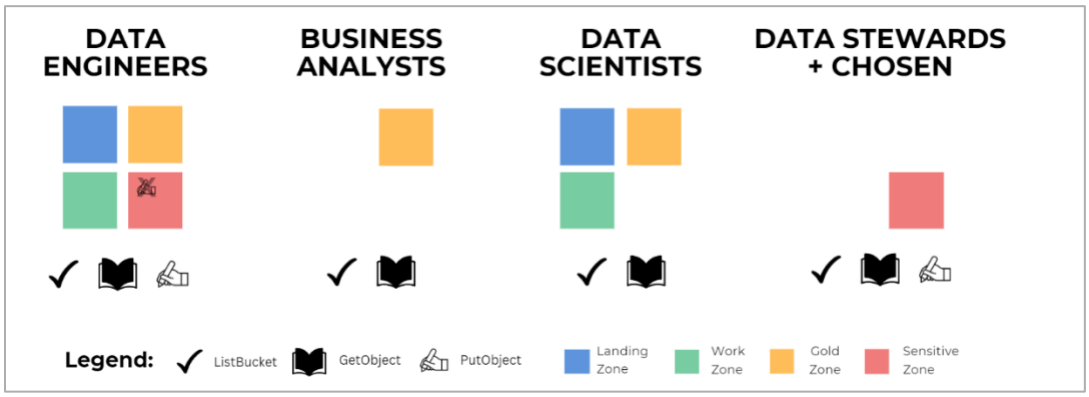


Figure 8. Diwata Gas Main Classes of Users in the Data Lake

Data Engineers

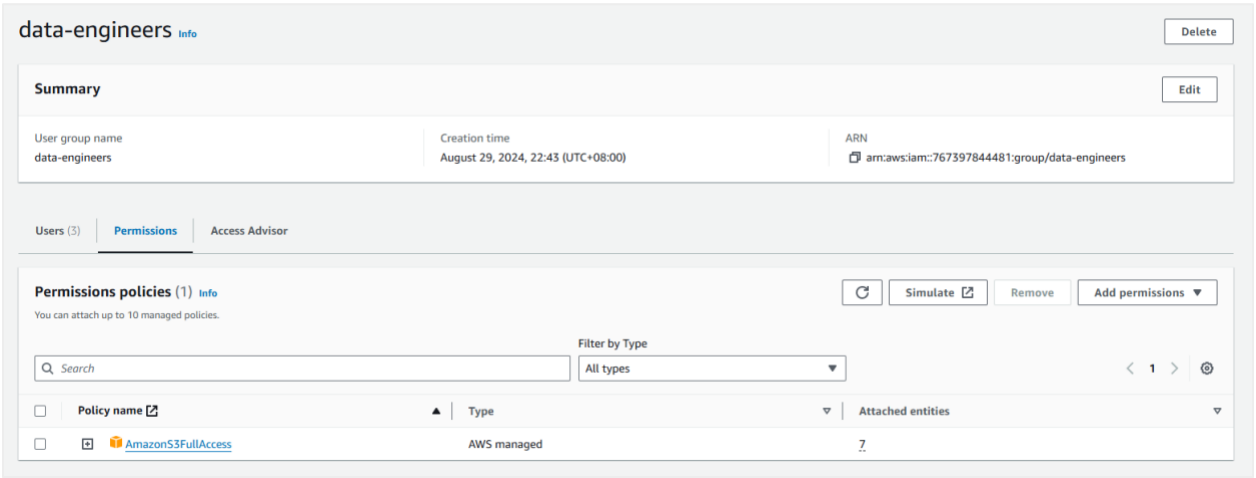


Figure 9. Data Engineers Permission Policy

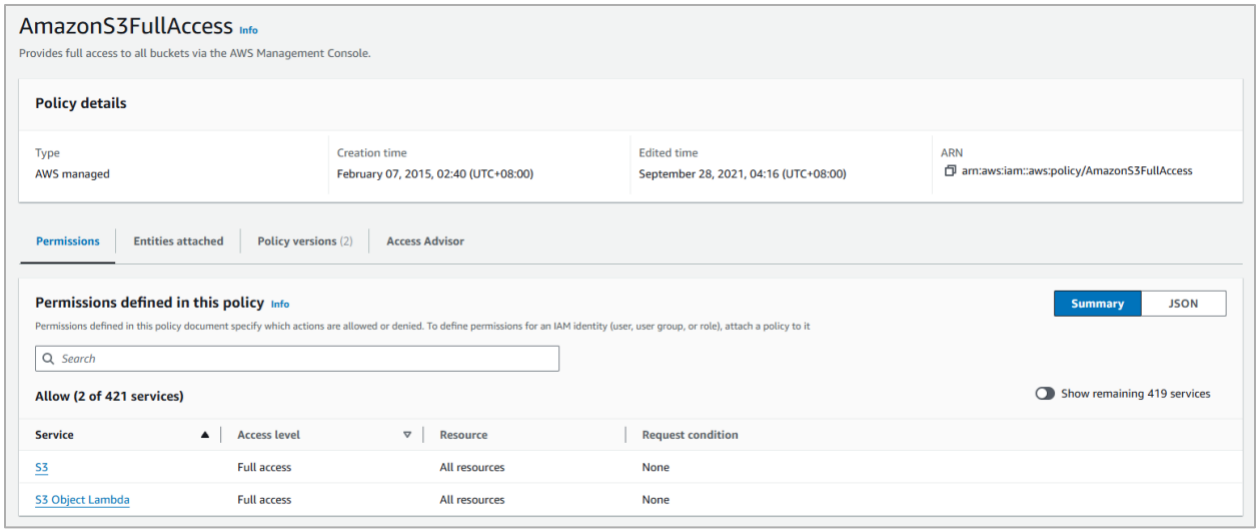


Figure 10. AmazonS3FullAccess Policy

Data Engineers play a crucial role in managing the data lake and overseeing data flow across its various zones. They are responsible for creating scripts to prevent sensitive data from being improperly loaded into the landing zone, which is essential for maintaining data security. Data Engineers have full access to all zones as shown in **Figure 9**, including the sensitive zone, and are authorized to write and modify data as needed. This comprehensive access allows them to handle tasks such as data integration, transformation, and maintenance, while they may also seek IT support for certain maintenance activities.

Data Stewards

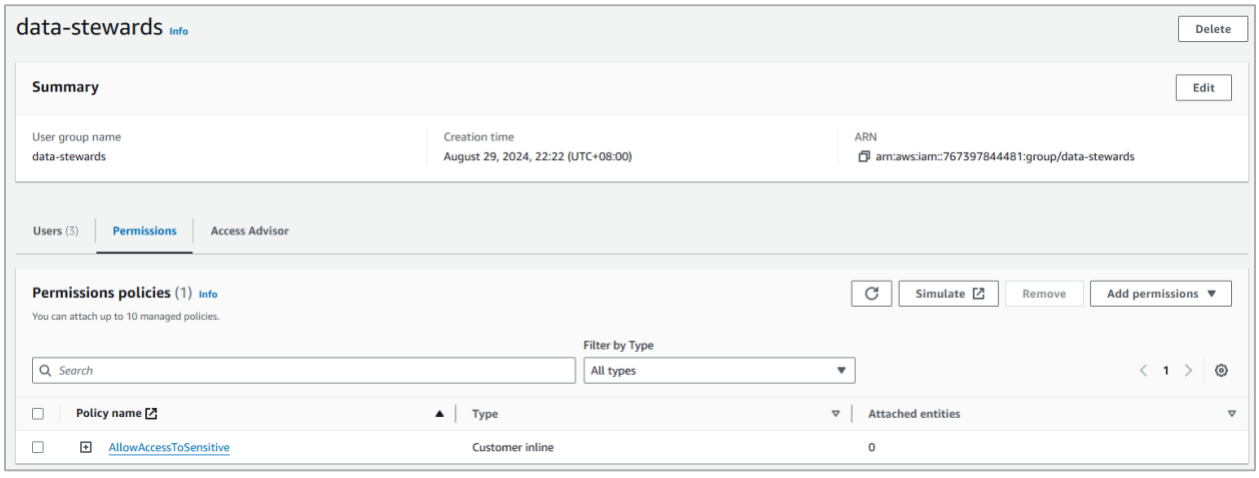


Figure 11. Data Stewards Permission Policy

Figure 12. AllowAccessToSensitive Policy

Working closely with Data Engineers are the Data Stewards, who play a vital role in data governance. While their access is not as extensive as Data Engineers, Data Stewards are granted significant permissions, including the ability to list bucket contents, read or download objects, and write data as shown in **Figure 11**. These permissions are crucial for their role in overseeing and managing data effectively. The ability to write data allows them to update, correct, or add new information, ensuring data quality and compliance with governance policies.

Data Scientists

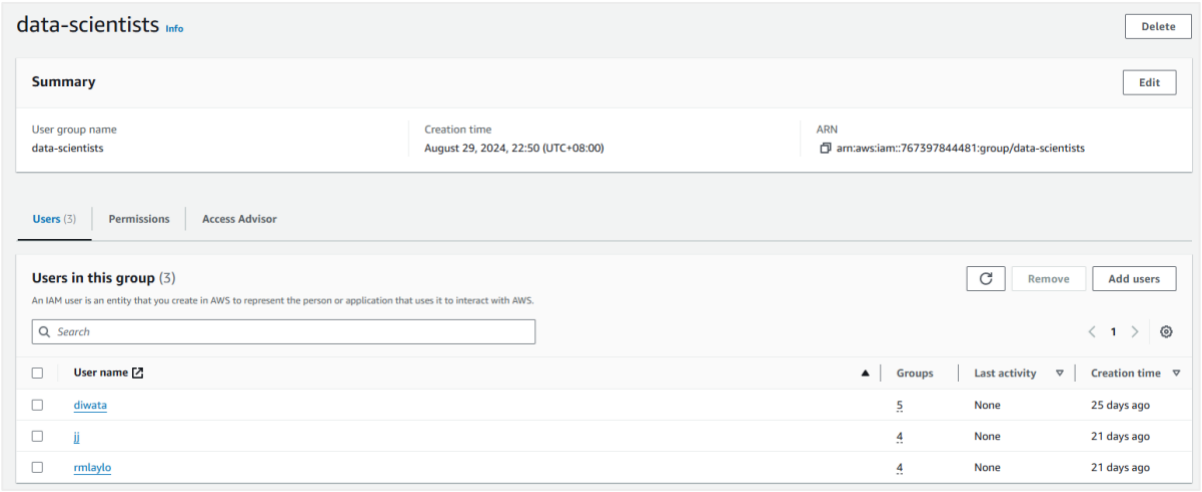


Figure 13. Data Scientists Permission Policy

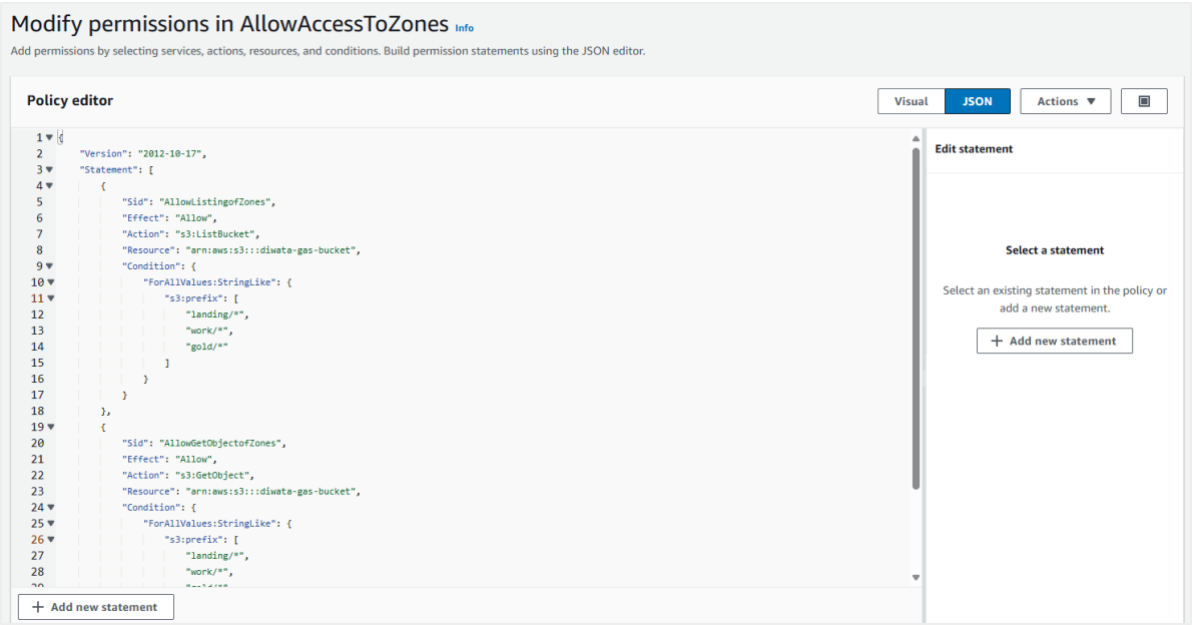


Figure 14. AllowAccessToZones Policy

Their permissions are more restricted compared to Data Engineers and Data Stewards. Data Scientists can list bucket contents and read or download objects from the landing, work, and gold zones, giving them access to raw, processed, and refined datasets. However, they lack written access to maintain data integrity and are restricted from the sensitive zone to protect confidential information. This access structure allows Data Scientists to work with various stages of data for analysis and model building while maintaining overall data security.

Business Analysts

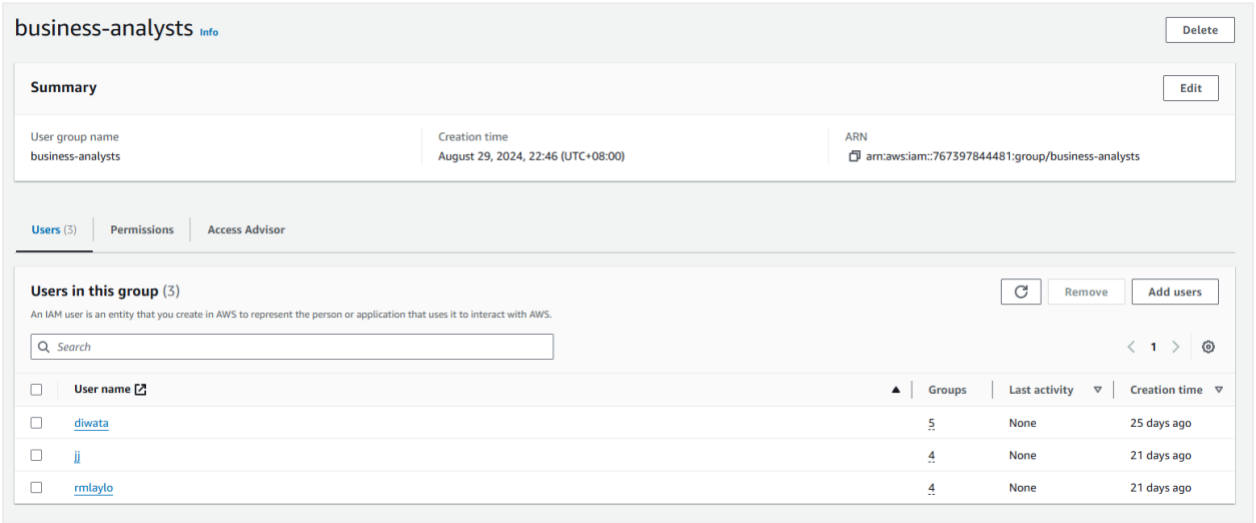


Figure 15. Business Analysts Permission Policy

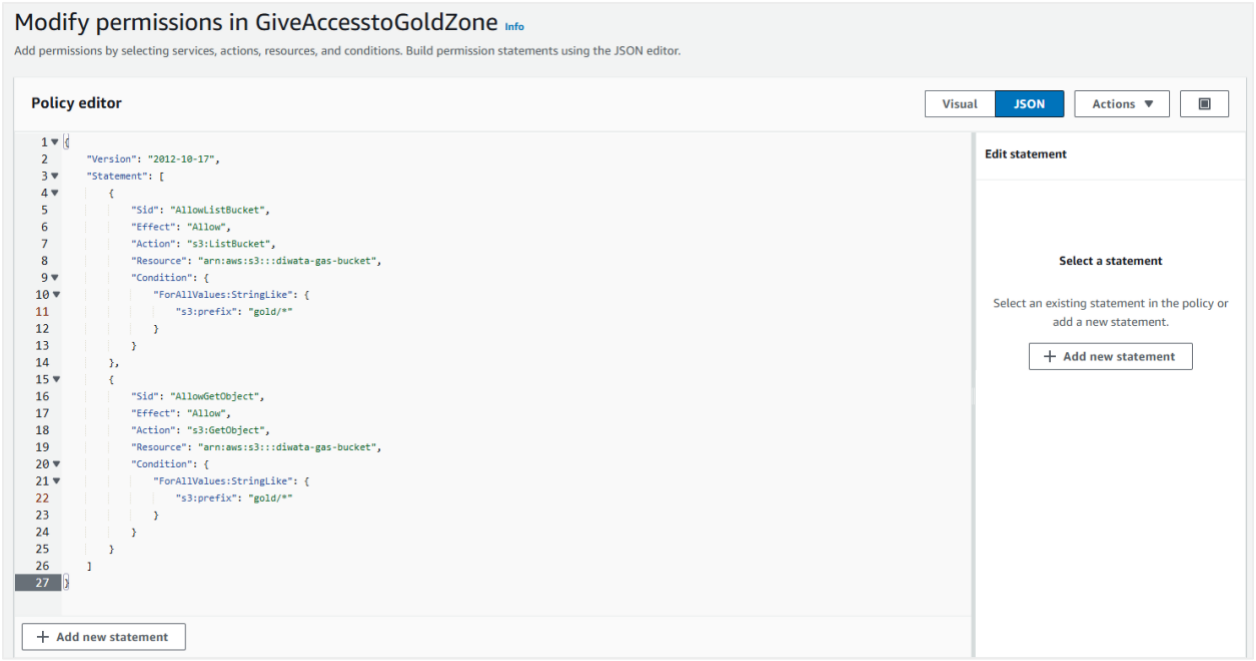


Figure 16. GiveAccesstoGoldZone Policy

Finally, business analysts in Diwata Gas are granted only permissions to list bucket contents and read or download objects from the gold zone as shown in **Figure 15**, where they can analyze curated datasets for business insights. They do not have write access to prevent any modifications to the data and allow them to concentrate on reporting and decision-making, while avoiding direct interaction with raw or sensitive data.

4.3 ETL (Extract-Transform-Load) Jobs

For the ETL workflows or jobs, Apache Airflow was used by the team given that (1) the data is processed by batch which Apache Airflow handles well, (2) its user interface allows for easy addition of connections and operation of created directed acyclic graphs (DAGs), (3) its Python back-end allows for easier customization and addition of several features into the DAGs, and (4) its modular structure allows for easy addition of new DAGs into the current workflow (Blanc, 2023; Franklin, 2021; Apache, 2024).

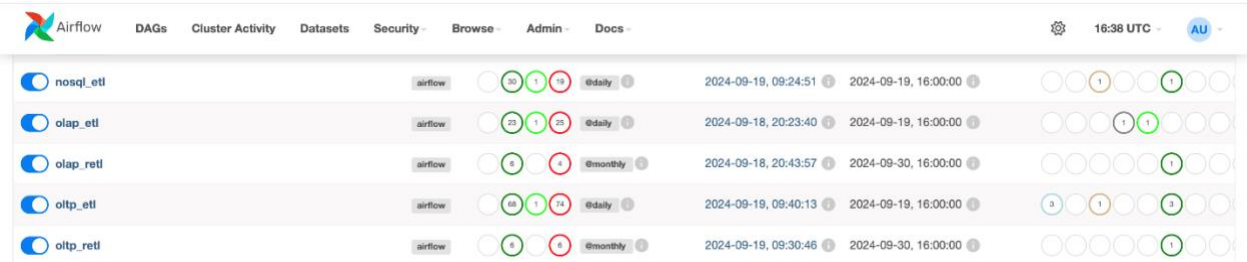


Figure 17. Apache Airflow User Interface

In the architecture, five DAGs (which are toggled on in the figure above) were created, and these are:

- Extract-Transform-Load (ETL) for OLTP
- Reverse Extract-Transform-Load (RETL) for OLTP
- Extract-Transform-Load (ETL) for OLAP
- Reverse Extract-Transform-Load (RETL) for OLAP
- Extract-Transform-Load (ETL) for NoSQL

Prerequisites for the DAGs

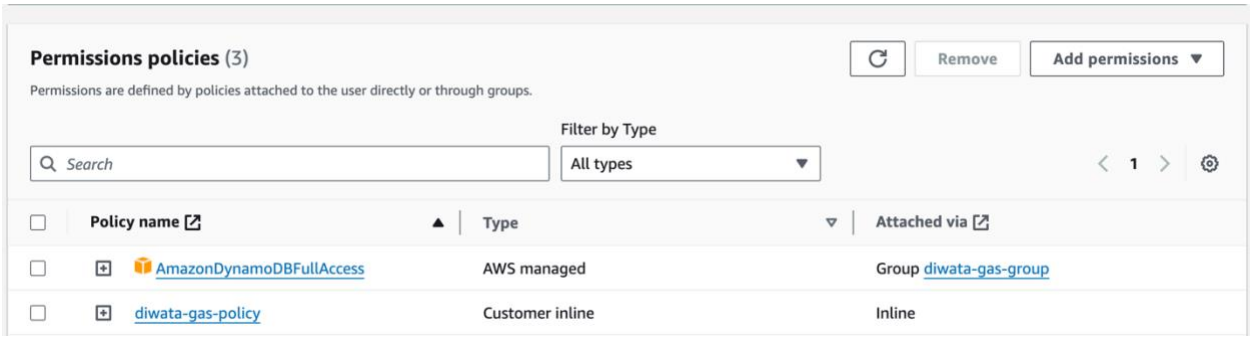


Figure 18. Permission Policies for Used IAM User

As seen in **Figure 18**, the connected IAM user for the Apache Airflow are provided the permission policies, namely, DynamoDBFullAccess policy and Diwata Gas Policy.

Service	Access level	Resource	Request condition
Application Auto Scaling	Limited: Read, Write	All resources	None
CloudWatch	Limited: List, Read, Write	Multiple	None
Data Pipeline	Limited: List, Read, Write	All resources	None
DynamoDB	Full access	All resources	None
DynamoDBAccelerator	Full access	All resources	None
EC2	Limited: List	All resources	None
IAM	Limited: List, Read, Write	All resources	Multiple
Kinesis	Limited: List, Read	All resources	None
KMS	Limited: List, Read	All resources	None
Lambda	Limited: List, Read, Write	All resources	None
Resource Group Tagging	Limited: Read	All resources	None
Resource Groups	Limited: List, Read, Write	All resources	None
SNS	Limited: List, Permissions management, Write	All resources	None

Figure 19. DynamoDBFullAccess Policy for Used IAM User

What is notable here would be the full access to the DynamoDB, which would allow for all transformations and operations done to the DynamoDB through the created DAG.

Figure 20. Diwata Gas Policy for Used IAM User

What is notable here are the S3 permissions, involving listing of the bucket, getting an object, putting an object, and deleting an object. This would prove most useful to all DAGs given that all involve listing of the contents of the bucket and retrieving and reading objects from the bucket, while one DAG requires copying and deleting objects from the bucket.

ETL for OLTP

```
from airflow import DAG
from datetime import datetime
from airflow.io.path import ObjectStoragePath
from airflow.decorators import task
from airflow.providers.postgres.hooks.postgres import PostgresHook
import pendulum
```

Figure 21. Libraries used for ETL for OLTP

Functions from the Airflow library which were used for this DAG are the DAG function, ObjectStoragePath function, task decorator, and the PostgresHook. The DAG function is used to initialize the DAG; the task decorator is used to designate a function as a DAG task. Moreover, the ObjectStoragePath is used to traverse and access the files of the data lake (S3 bucket), while the PostgresHook is used to connect to the OLTP database. Lastly, datetime and pendulum is used to create datetime objects, where pendulum is used to localize the datetime to the time zone of Manila, Philippines.

```
with DAG(dag_id='oltp_etl',
         schedule='@daily',
         start_date=pendulum.datetime(2024, 9, 19, tz="Asia/Manila"),
        ) as dag:
    import json
    import pandas as pd

    base = ObjectStoragePath("s3://s3_diwata_gas@diwata-gas-bucket/")
```

Figure 22. DAG details, other imported libraries, and base directory

As seen above, the ID of the DAG is “oltp_etl”, and it is scheduled daily. In order to adjust to the local time zone, the start date is set at a specific date in the “Asia/Manila” time zone. Moreover, other libraries which were imported include json and pandas, for easier data loading and manipulation. Lastly, the ObjectStoragePath function is used to load the S3 bucket, allowing for reading of the files within the bucket.

```python
@task
def customer_extract():
    """Extract customer file paths from S3 bucket.

    Returns
    -------
    customer_files : list
        List of customer file paths
    """
    customer_base = base / 'sensitive' / 'Customer'
    customer_files = [f for f in customer_base.iterdir() if f.is_file()]
    return customer_files
```

Figure 23. Customer File Paths Extraction Task

```python
@task
def log_extract():
    """Extract logistics file paths from S3 bucket.

    Returns
    -------
    log_files : list
        List of logistics file paths
    """
    log_base = base / 'landing' / 'Logistics'
    log_files = [f for f in log_base.iterdir() if f.is_file()]
    return log_files
```

Figure 24. Log File Paths Extraction Task

```python
@task
def order_extract():
    """Extract order file paths from S3 bucket.

    Returns
    -------
    order_files : list
        List of order file paths
    """
    order_base = base / 'landing' / 'Orders'
    order_files = [f for f in order_base.iterdir() if f.is_file()]
    return order_files
```

Figure 25. Order File Paths Extraction Task

Using these three tasks, the pathlib-based file paths for the customer information, logistics information, and order information are extracted, which would then be forwarded to other tasks for database loading purposes.

Figure 26. Postgres DB (OLTP) Airflow Connection

```
@task
def customer_load(customer_files):
    """Load or upsert customer data into OLTP.
    """
    hook = PostgresHook(postgres_conn_id="diwata_gas_oltp")
    conn = hook.get_conn()
    cursor = conn.cursor()
    lst_rows = []
    for filepath in customer_files:
        customer_json = json.loads(filepath.read_text())
        cursor.execute("""SELECT areaid FROM area WHERE region = %s
        AND province = %s AND city = %s AND barangay = %s""", (customer_json['AreaRegion'],
                                                               customer_json['AreaProvince'],
                                                               customer_json['AreaCity'],
                                                               customer_json['AreaBarangay']))
        area_id = cursor.fetchone()[0]
        cursor.execute("""
            SELECT salespersonid FROM salesperson WHERE salespersonname = %s""",
            (customer_json['SalesPersonName'],))
        salesperson_id = cursor.fetchone()[0]
        if area_id and salesperson_id:
            lst_rows.append((customer_json['CustomerName'],
                             customer_json['CustomerType'],
                             customer_json['CustomerAddress'],
                             area_id,
                             customer_json['ContactNumber'],
                             'Active',
                             salesperson_id))
    cursor.executemany("""
        INSERT INTO customer (customername, customertype, customeraddress, areaid, contactnumber, customerstatus, salespersonid)
        VALUES (%s, %s, %s, %s, %s, %s, %s)
        ON CONFLICT (customername)
        DO NOTHING""", lst_rows)
    conn.commit()
    cursor.close()
    conn.close()
```

Figure 27. Customer Data Loading Task

The collected file paths of customer data are first loaded as text and then loaded as a dictionary using the json library's loads function. Using the PostgresHook and the connection created in Airflow (shown in **Figure 27**), a connection to the OLTP along with its cursor is created, which would be used to find the referential IDs of the customer's area based on their region, province, city, and barangay and of the customer's assigned salesperson based on their salesperson's name. Afterwards, if both IDs are existent, all the imperative information as seen above are *upserted* into the Customer table, which means that a row which is similar to a row in the database (in this case, by customer name) would not be added, while the opposite is true for a row with no similarities in any row in the database. Afterwards, the changes are committed, and both the cursor and the connection are closed.

```python
@task
def doc_load(order_files):
    """Load or upsert document reference data for DynamoDB
    """
    hook = PostgresHook(postgres_conn_id="diwata_gas_oltp")
    conn = hook.get_conn()
    cursor = conn.cursor()
    for filepath in order_files:
        order_json = json.loads(filepath.read_text())
        cursor.execute("""
                INSERT INTO documentreference (documentreferencenumber, documentreferencetype)
                VALUES (%s, %s)
                ON CONFLICT (documentreferencenumber)
                DO NOTHING""", (order_json['DocumentReferenceNumber'],
                                order_json['DocumentReferenceType']))
        conn.commit()
    cursor.close()
    conn.close()
```

Figure 28. Document Reference Data Loading Task

Using the same process earlier, the order file paths are first loaded and the document reference information from the files are then *upserted* into the DocumentReference table.

```python
@task
def order_load(order_files):
    """Load or upsert order data into OLTP.
    """
    hook = PostgresHook(postgres_conn_id="diwata_gas_oltp")
    conn = hook.get_conn()
    cursor = conn.cursor()
    for filepath in order_files:
        try:
            order_json = json.loads(filepath.read_text())
            cursor.execute("""SELECT branchid FROM branch WHERE branchname = %s""", (order_json['BranchName'],))
            branch_id = cursor.fetchone()[0]
            cursor.execute("""SELECT customerid FROM customer WHERE customername = %s""", (order_json['OrderName'],))
            customer_id = cursor.fetchone()[0]
            cursor.execute("""SELECT paymentid FROM payment WHERE paymenttype = %s""", (order_json['PaymentType'],))
            payment_id = cursor.fetchone()[0]
            cursor.execute("""SELECT MAX(logisticsid) FROM logistics""")
            logistics_id = cursor.fetchone()[0]
            if logistics_id is None:
                logistics_id = 0
            cursor.execute("""SELECT MAX(orderid) FROM customerorder""")
            old_order_id = cursor.fetchone()[0]
            cursor.execute("""
            INSERT INTO customerorder (branchid, customerid, documentreferencenumber, orderdate, logisticsid, paymentid)
            VALUES (%s, %s, %s, %s, %s, %s)
            ON CONFLICT (branchid, customerid, documentreferencenumber, orderdate, paymentid)
            DO NOTHING""", (branch_id,
                            customer_id,
                            order_json['DocumentReferenceNumber'],
                            order_json['OrderDate'],
                            logistics_id + 1,
                            payment_id))
            conn.commit()
            cursor.execute("""SELECT MAX(orderid) FROM customerorder""")
            new_order_id = cursor.fetchone()[0]
            if old_order_id != new_order_id:
                lst_rows_ordercylinder = []
                cursor.execute("""
                    INSERT INTO logistics (driverid, deliverydate, pickupdate)
                    VALUES (%s, %s, %s)""", (None, None, None))
                conn.commit()
                for i, j in order_json['Prices'].items():
                    cursor.execute("""
                    INSERT INTO ordercylinder (orderid, serialno, price)
                    VALUES (%s, %s, %s)""", (new_order_id, i, j))
                    conn.commit()
        except Exception:
            continue
    cursor.close()
    conn.close()
```

Figure 29. Order Data Loading Task

Using the same process earlier, the order file paths are first loaded as dictionaries. Similarly, the branch ID, customer ID, and payment ID of the given order are referred to using its branch name, customer name, and payment type. Moreover, the latest logistics ID is also extracted wherein if the logistics ID is non-existent, the ID becomes 0. The latest order ID is also extracted, which would be used later to denote whether a row has been added to the database. All the information, including the extracted logistics ID which would be incremented by 1, would be *upserted* into the CustomerOrder database. Afterwards, if conditional which checks if a row has been added to the database returns True, the logistics table is added with a new row which is filled with null

values, and accordingly, the OrderCylinder table would be added with the newly created order ID and the serial numbers and prices of the tanks in the order.

```
@task
def log_load(log_files):
    """Load or upsert logistics data into OLTP.
    """
    hook = PostgresHook(postgres_conn_id="diwata_gas_oltp")
    conn = hook.get_conn()
    cursor = conn.cursor()
    lst_rows = []
    for filepath in log_files:
        log_json = json.loads(filepath.read_text())
        cursor.execute("""SELECT driverid FROM driver WHERE drivername = %s""", (log_json['DriverName'],))
        driver_id = cursor.fetchone()[0]
        if driver_id:
            if log_json['PickUpDate'] == "":
                log_json['PickUpDate'] = None
            lst_rows.append((driver_id, log_json['DeliveryDate'], log_json['PickUpDate'], log_json['LogisticsID']))
    cursor.executemany("""
        UPDATE logistics
        SET driverid = %s, deliverydate = %s, pickupdate = %s
        WHERE logisticsid = %s;""", lst_rows)
    conn.commit()
    cursor.execute("""
        UPDATE cylinder
        SET cylinderstatus = 'Available'
        FROM ordercylinder oc
        JOIN customerorder co
        ON oc.orderid = co.orderid
        JOIN logistics l
        ON co.logisticsid = l.logisticsid
        WHERE cylinder.serialno = oc.serialno
        AND l.deliverydate IS NOT NULL
        AND l.pickupdate IS NOT NULL""")
    conn.commit()
    cursor.execute("""
        UPDATE cylinder
        SET cylinderstatus = 'Unavailable'
        FROM ordercylinder oc
        JOIN customerorder co
        ON oc.orderid = co.orderid
        JOIN logistics l
        ON co.logisticsid = l.logisticsid
        WHERE cylinder.serialno = oc.serialno
        AND l.deliverydate IS NOT NULL
        AND l.pickupdate IS NULL""")
    conn.commit()
    cursor.close()
    conn.close()
```

Figure 30. Logistics Data Loading Task

While the same process for the PostgresHook was used again for this task, the driver ID of the logistics data row was referred to from the Driver table using the driver's name in the logistics data. Moreover, given that pick-up date may be null and is reflected as an empty string, the pick-up date is changed to a NoneType. Given these, the logistics row with the corresponding logistics ID is then updated with the newly updated delivery date and pick-up date. Lastly, the cylinders with both non-null delivery dates and pick-up dates (which meant that the cylinder has been picked up) are then updated with an

“Available” cylinder status in the Cylinder table, while the cylinders with a non-null delivery date and a null pick-up date (which meant that the cylinder is still with the client) is updated with an “Unavailable” cylinder status.

```
customer_files = customer_extract()
order_files = order_extract()
log_files = log_extract()
customer_load(customer_files) >> doc_load(order_files) >> order_load(order_files) >> log_load(log_files)
```

Figure 31. Order of DAGs in the Workflow

The order of the DAGs in the workflow would be extracting the file paths of the customer, order, and logistics files. The flow of the loading would start with the customers, moving on to the document references. The reason why the document references loading task is first done before the order loading task is because the document reference number in the DocumentReference table is referenced by the document reference number in the CustomerOrder table. This means that the document reference number to be added in the CustomerOrder table should already be present within the DocumentReference table, as to abide by the referencing constraint. Lastly, the reason why the logistics loading is done after the order loading is because the logistics loading task only aims to update the existing records in the Logistics table, which can be also produced by the order loading task. Hence, if the order loading were done after the logistics loading, rows which may be possibly updated would not be updated by the logistics loading task during the same day (wherein the update may happen the next day instead).

RETL for OLTP

```
from airflow import DAG
from datetime import datetime
from airflow.io.path import ObjectStoragePath
from airflow.decorators import task
from airflow.providers.postgres.hooks.postgres import PostgresHook
import pendulum
```

Figure 32. Imported Libraries for RETL for OLTP

The same functions of the Airflow library for the ETL for OLTP are used for the RETL for OLTP. Moreover, the pendulum library is used again for datetime localization purposes.

```
with DAG(dag_id='oltp_retl',
         schedule='@monthly',
         start_date=pendulum.datetime(2024, 9, 19, tz="Asia/Manila"),
        ) as dag:
    import json
    import pandas as pd
```

Figure 33. Additional DAG Information and Imported Libraries

The DAG ID for the task is “oltp_retl”, while the DAG is scheduled monthly. The start date is set in the local time zone, while the imported libraries are json and pandas for easier data loading, manipulation, and writing.

```
@task
def oltp_retl():
    """Reverse extract, transform, and load data for OLTP and data scientists,
    and store in S3 bucket (Work Zone).
    """
    # Customer Order
    base = ObjectStoragePath("s3://s3_diwata_gas@diwata-gas-bucket/work/")

    hook = PostgresHook(postgres_conn_id="diwata_gas_oltp")
    conn = hook.get_conn()
    cursor = conn.cursor()
    date_today = pendulum.now("Asia/Manila").subtract(months=1).strftime("%B") + '_' + pendulum.now("Asia/Manila").strftime("%Y")
    first_day = pendulum.now("Asia/Manila").subtract(months=1).start_of('month')
    last_day = pendulum.now("Asia/Manila").subtract(months=1).end_of('month')

    cursor.execute("""SELECT co.orderid, co.logisticsid, co.orderdate, oc.serialno, oc.price,
                   c.productid, c.cylinderstatus, p.producttype,
                   p.productcategory, cu.customerstatus, cu.customerid,
                   a.region, a.province, a.city, a.barangay,
                   b.branchname
                   FROM CustomerOrder co
                   JOIN branch b ON b.branchid = co.branchid
                   JOIN customer cu ON cu.customerid = co.customerid
                   JOIN area a ON cu.areaid = a.areaid
                   JOIN ordercylinder oc ON co.orderid = co.orderid
                   JOIN cylinder c ON oc.serialno = c.serialno
                   JOIN product p ON c.productid = p.productid
                   WHERE orderdate >= %s AND orderdate <= %s""",
                  (first_day, last_day))
    lst_desc = [desc[0] for desc in cursor.description]
    tuple_order = cursor.fetchall()
    df = pd.DataFrame(tuple_order, columns=lst_desc)

    path = base / f"CustomerOrder_{date_today}.parquet"

    with path.open("wb") as file:
        df.to_parquet(file)
```

Figure 34. OLTP RETL Task (Customer Order RETL Sub-task)

In the OLTP RETL task, it is subdivided into the customer order and the logistics RETL sub-tasks. In the Customer Order RETL sub-task, the ObjectStoragePath is used to save the Customer Order data within the past month in the work zone, using an SQL query which involves getting the Customer Order data within the past month, joined with other relevant tables seen above. In order to save these data to a parquet file, the query result and

its corresponding columns are inserted into a pandas dataframe and then saved as a parquet file using a created file path from the ObjectStoragePath and the to_parquet method of the pandas dataframe.

```
# Logistics
cursor.execute("""
SELECT logisticsid, Logistics.driverid, deliverydate, pickupdate,
    restrictionno
    FROM Logistics
    JOIN driver ON Logistics.driverid = driver.driverid
    WHERE deliverydate >= %s AND deliverydate <= %s""",
          (first_day, last_day))
lst_desc = [desc[0] for desc in cursor.description]
tuple_logistics = cursor.fetchall()
df = pd.DataFrame(tuple_logistics, columns=lst_desc)

path = base / f"Logistics_{date_today}.parquet"

with path.open("wb") as file:
    df.to_parquet(file)
```

Figure 35. OLTP RETL Task (Logistics RETL Sub-task)

In the logistics RETL sub-task, the logistics table is only joined with the driver table where the data is filtered using the delivery date (where the delivery date would be within the past month only). The same process would be applied to save the table as a parquet file within the work zone.

ETL for OLAP

```
from airflow import DAG
from datetime import datetime
from airflow.io.path import ObjectStoragePath
from airflow.decorators import task
from airflow.providers.postgres.hooks.postgres import PostgresHook
from airflow.providers.amazon.aws.hooks.redshift_sql import RedshiftSQLHook
import pendulum
```

Figure 36. OLAP ETL DAG Imported Libraries

Similar to the previous tasks, the same Airflow functions were also imported. However, in addition to these functions, RedshiftSQLHook is also imported, given that this ETL task would be dealing with an OLAP database.

```
with DAG(dag_id='olap_etl',
         schedule='@daily',
         start_date=pendulum.datetime(2024, 9, 19, tz="Asia/Manila"),
        ) as dag:
```

Figure 37. OLAP ETL DAG Details

The DAG ID is “olap_etl”, and it is scheduled daily. The start date is also set using Pendulum’s datetime.

```
@task
def dim_etl():
    """Extract, transform, and load data for OLAP dimensions.
    """
    # Extract
    hook = PostgresHook(postgres_conn_id="diwata_gas_oltp")
    conn = hook.get_conn()
    cursor = conn.cursor()
    cursor.execute("""SELECT c.customerid,
        c.customername, c.customertype, s.salespersonname AS salespersonname
        FROM customer c JOIN salesperson s ON c.salespersonid = s.salespersonid""")
    customer_dim = cursor.fetchall()
    cursor.execute("""SELECT branchid, branchname, branchmanager FROM branch""")
    branch_dim = cursor.fetchall()
    cursor.execute("""SELECT productid, producttype, productcategory FROM product""")
    product_dim = cursor.fetchall()
    cursor.execute("""SELECT areaid, region, province, city, barangay FROM area""")
    area_dim = cursor.fetchall()
    cursor.execute("""SELECT cylindertypeid, cylindertype, cylindervolume FROM cylindertype""")
    cylindertype_dim = cursor.fetchall()
    cursor.execute("""SELECT paymentid, paymenttype, paymentdescription FROM payment""")
    payment_dim = cursor.fetchall()
    cursor.execute("""SELECT driverid, drivername, hiredate, licenseno, restrictionno FROM driver""")
    driver_dim = cursor.fetchall()
```

Figure 38. Dimension ETL Task (Extraction Stage)

The PostgresHook is used to connect to the OLTP database where using the several SQL queries, dimensions are extracted from the tables within the OLTP database.

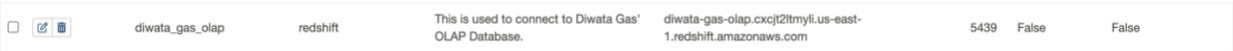


Figure 39. OLAP Airflow Connection

```
# Transform and Load
hook2 = RedshiftSQLHook(redshift_conn_id='diwata_gas_olap')
conn2 = hook2.get_conn()
cursor2 = conn2.cursor()

cursor2.execute("""DELETE FROM customerdimension""")
conn2.commit()
cursor2.executemany("""INSERT INTO customerdimension
(customerid, customername, customertype, salespersonname)
VALUES (%s, %s, %s, %s)""", customer_dim)
conn2.commit()

cursor2.execute("""DELETE FROM branchdimension""")
conn2.commit()
cursor2.executemany("""INSERT INTO branchdimension
(branchid, branchname, branchmanager)
VALUES (%s, %s, %s)""", branch_dim)
conn2.commit()

cursor2.execute("""DELETE FROM productdimension""")
conn2.commit()
cursor2.executemany("""INSERT INTO productdimension
(productid, producttype, productcategory)
VALUES (%s, %s, %s)""", product_dim)
conn2.commit()

cursor2.execute("""DELETE FROM areadimension""")
conn2.commit()
cursor2.executemany("""INSERT INTO areadimension
(areaid, region, province, city, barangay)
VALUES (%s, %s, %s, %s, %s)""", area_dim)
conn2.commit()

cursor2.execute("""DELETE FROM cylindertypedimension""")
conn2.commit()
cursor2.executemany("""INSERT INTO cylindertypedimension
(cylindertypeid, cylindertype, cylindervolume)
VALUES (%s, %s, %s)""", cylindertype_dim)
conn2.commit()

cursor2.execute("""DELETE FROM paymentdimension""")
conn2.commit()
cursor2.executemany("""INSERT INTO paymentdimension
(paymentid, paymenttype, paymentdescription)
VALUES (%s, %s, %s)""", payment_dim)
conn2.commit()

cursor2.execute("""DELETE FROM driverdimension""")
conn2.commit()
cursor2.executemany("""INSERT INTO driverdimension
(driverid, drivername, hiredate, licenseno, restrictionno)
VALUES (%s, %s, %s, %s, %s)""", driver_dim)
conn2.commit()

cursor2.close()
conn2.close()
cursor.close()
conn.close()
```

Figure 40. Dimension ETL Task (Transformation and Load Phases)

The RedshiftSQLHook is used in conjunction with the OLAP Airflow connection (**in Figure 40**). Using this hook, each dimension is deleted fully and replaced with the existing dimensions extracted from the OLTP.

```
@task
def fact_etl():
    """Extract, transform, and load data for OLAP Fact Tables.
    """
    hook = PostgresHook(postgres_conn_id="diwata_gas_oltp")
    conn = hook.get_conn()
    cursor = conn.cursor()
    hook2 = RedshiftSQLHook(redshift_conn_id='diwata_gas_olap')
    conn2 = hook2.get_conn()
    cursor2 = conn2.cursor()

    # Fact Sales
    cursor.execute("""SELECT to_char(c.orderdate, 'YYYYMMDD'), c.customerid, c.branchid,
        c.paymentid, a.areaid, cylinder.productid,
        cylinder.cylindertypeid, COUNT(cylinder.serialno) AS quantity,
        SUM(o.price) AS revenue
        FROM customerorder c
        JOIN branch b
        ON c.branchid = b.branchid
        JOIN area a
        ON a.areaid = b.areaid
        JOIN ordercylinder o
        ON c.orderid = o.orderid
        JOIN cylinder
        ON o.serialno = cylinder.serialno
        GROUP BY c.orderdate, c.customerid, c.branchid,
        c.paymentid, a.areaid, cylinder.productid, cylinder.cylindertypeid""")

    new_fact_sales = cursor.fetchall()
```

Figure 41. Fact ETL Task (Fact Sales Extraction and Transformation)

Both hooks are again initialized, and using the PostgresHook, series of joins, and aggregations, the data for the Sales Fact Table is extracted.

```
    cursor2.execute("""CREATE temp TABLE temporary_staging_area (LIKE salesfacttable)""")
    conn2.commit()
    cursor2.executemany("""INSERT INTO temporary_staging_area (dateid, customerid, branchid, paymentid,
    areaid, productid, cylindertypeid, quantity, revenue)
    VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)""", new_fact_sales)
    conn2.commit()
    cursor2.execute("""SELECT dateid, customerid, branchid, paymentid,
    areaid, productid, cylindertypeid, quantity, revenue FROM salesfacttable""")
    current_fact_sales = cursor2.fetchall()
    cursor2.executemany("""DELETE FROM temporary_staging_area WHERE dateid = %s AND
    customerid = %s AND branchid = %s AND paymentid = %s AND areaid = %s
    AND productid = %s AND cylindertypeid = %s AND quantity = %s AND revenue = %s""", current_fact_sales)
    conn2.commit()
    cursor2.execute("""SELECT dateid, customerid, branchid, paymentid,
    areaid, productid, cylindertypeid, quantity, revenue FROM temporary_staging_area""")
    new_fact_sales = cursor2.fetchall()
    cursor2.executemany("""INSERT INTO salesfacttable (dateid, customerid, branchid, paymentid,
    areaid, productid, cylindertypeid, quantity, revenue)
    VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)""", new_fact_sales)
    conn2.commit()
    cursor2.execute("""DROP TABLE temporary_staging_area""")
    conn2.commit()
```

Figure 42. Fact ETL Task (Fact Sales Loading)

Using the RedshiftSQLHook, a temporary staging area following the schama of the sales fact table is created. The new data is inserted into this temporary staging area, where the data in the current sales fact table is then deleted from this temporary staging area. Through this, only new data is added from the temporary staging area to the sales fact table. Afterwards, the temporary staging area is then dropped to preserve space in the OLAP database.

```
# Fact Inventory
cursor.execute("""SELECT productid, cylindertypeid,
    sum(case when cylinderstatus = 'Available' then 1 else 0 end) as AvailableQuantity,
    sum(case when cylinderstatus = 'Unavailable' then 1 else 0 end) as UnavailableQuantity
    FROM cylinder
    GROUP BY productid, cylindertypeid""")

new_fact_inventory = []
for i in cursor.fetchall():
    new_fact_inventory.append((datetime.strftime(pendulum.now('Asia/Manila').date(), '%Y%m%d'),) + i)

cursor2.execute("""CREATE temp TABLE temporary_staging_area (LIKE inventoryfacttable)""")
conn2.commit()
cursor2.executemany("""INSERT INTO temporary_staging_area (dateid, productid, cylindertypeid, availablequantity,
unavailablequantity) VALUES (%s, %s, %s, %s, %s)""", new_fact_inventory)
conn2.commit()
cursor2.execute("""SELECT dateid, productid, cylindertypeid, availablequantity,
unavailablequantity FROM inventoryfacttable""")
current_fact_inventory = cursor2.fetchall()
cursor2.executemany("""DELETE FROM temporary_staging_area WHERE dateid = %s AND
productid = %s AND cylindertypeid = %s AND availablequantity = %s
AND unavailablequantity = %s""", current_fact_inventory)
conn2.commit()
cursor2.execute("""SELECT dateid, productid, cylindertypeid, availablequantity,
unavailablequantity FROM temporary_staging_area""")
new_fact_inventory = cursor2.fetchall()
cursor2.executemany("""INSERT INTO inventoryfacttable (dateid, productid, cylindertypeid, availablequantity,
unavailablequantity) VALUES (%s, %s, %s, %s, %s)""", new_fact_inventory)
conn2.commit()
cursor2.execute("""DROP TABLE temporary_staging_area""")
conn2.commit()
```

Figure 43. Fact ETL Task (Fact Inventory ETL)

In extracting the data for the Inventory fact table, an SQL query involving aggregations is sent to the OLTP database, and each row of the query results is appended with the current time (in the “Asia/Manila” time zone). Afterwards, a similar process involving the temporary staging area is done to *upsert* the Inventory Fact Table.

```
# Fact Logistics
cursor.execute("""
SELECT to_char(co.orderdate, 'YYYYMMdd') AS dateid, co.customerid,
    b.areaid, c.productid,
    c.cylindertypeid, l.driverid,
    sum(case when l.deliverydate = co.orderdate then 1 else 0 end) as deliveryquantity,
    sum(case when l.pickupdate = co.orderdate then 1 else 0 end) as pickupquantity
    FROM customerorder co
    JOIN ordercylinder oc ON co.orderid = oc.orderid
    JOIN cylinder c ON oc.serialno = c.serialno
    JOIN logistics l ON co.logisticsid = l.logisticsid
    JOIN branch b ON co.branchid = b.branchid
    GROUP BY dateid, co.customerid, b.areaid, c.productid, c.cylindertypeid, l.driverid""")

new_fact_logistics = cursor.fetchall()

cursor2.execute("""CREATE temp TABLE temporary_staging_area (LIKE logisticsfacttable)""")
conn2.commit()
cursor2.executemany("""INSERT INTO temporary_staging_area (dateid, customerid, areaid, productid,
cylindertypeid, driverid, deliveryquantity, pickupquantity
) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)""", new_fact_logistics)
conn2.commit()
cursor2.execute("""SELECT dateid, customerid, areaid, productid,
cylindertypeid, driverid, driverid, deliveryquantity, pickupquantity FROM logisticsfacttable""")
current_fact_logistics = cursor2.fetchall()
cursor2.executemany("""DELETE FROM temporary_staging_area WHERE dateid = %s AND
customerid = %s AND areaid = %s AND productid = %s AND cylindertypeid = %s AND (driverid = %s)
OR (driverid IS NULL AND %s IS NULL) AND deliveryquantity = %s AND pickupquantity = %s""", current_fact_logistics)
conn2.commit()
cursor2.execute("""SELECT dateid, customerid, areaid, productid,
cylindertypeid, driverid, deliveryquantity, pickupquantity FROM temporary_staging_area""")
new_fact_logistics = cursor2.fetchall()
cursor2.executemany("""INSERT INTO logisticsfacttable (dateid, customerid, areaid, productid,
cylindertypeid, driverid, deliveryquantity, pickupquantity
) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)""", new_fact_logistics)
conn2.commit()
cursor2.execute("""DROP TABLE temporary_staging_area""")
conn2.commit()
conn2.commit()
cursor2.close()
conn2.close()
cursor.close()
conn.close()
```

Figure 44. Fact ETL Task (Fact Logistics ETL)

In extracting the data for the Logistics Fact Table, an SQL query involvign joins and aggregations is done, and using the same process beforehand involving a temporary staging area, the new data is *upserted* into the logistics fact table. A significant thing to note here would be the removal of rows with null driver IDs, which is done in the transformations within the temporary staging area.

```
dim_etl()
fact_etl()
```

Figure 45. Sequence of Tasks Within OLAP ETL DAG

The dimension ETL is first done before the fact ETL, however, there is no issue with regards to the ordering of both, since neither affects the other.

RETL for OLAP

```
from airflow import DAG
from datetime import datetime
from airflow.io.path import ObjectStoragePath
from airflow.decorators import task
from airflow.providers.amazon.aws.hooks.redshift_sql import RedshiftSQLHook
import pendulum
```

Figure 46. Imported Libraries for RETL for OLAP DAG

The RETL for OLAP uses the same functions and libraries as the ETL for OLAP.

```
with DAG(dag_id='olap_retl',
         schedule='@monthly',
         start_date=pendulum.datetime(2024, 9, 19, tz="Asia/Manila"),
        ) as dag:
```

Figure 47. RETL for OLAP DAG Details

The RETL for OLAP DAG is identified by the ID, “olap_retl”, and it is scheduled monthly. Moreover, the start date is localized using the pendulum datetime function in the “Asia/Manila” time zone.

```python
@task
def olap_retl():
    """Reverse extract, transform, and load data for OLAP and data analysts, and store
    in S3 bucket (Gold Zone).
    """
    import pandas as pd

    # Sales Fact Table
    base = ObjectStoragePath("s3://s3_diwata_gas@diwata-gas-bucket/gold/")

    hook = RedshiftSQLHook(redshift_conn_id='diwata_gas_olap')
    conn = hook.get_conn()
    cursor = conn.cursor()
    date_today = pendulum.now("Asia/Manila").subtract(months=1).strftime("%B") + '_' + pendulum.now("Asia/Manila").strftime("%Y")
    first_day = pendulum.now("Asia/Manila").subtract(months=1).start_of('month')
    last_day = pendulum.now("Asia/Manila").subtract(months=1).end_of('month')

    cursor.execute("""SELECT d.dateid, customerid, branchid, paymentid,
    areaid, productid, cylindertypeid, quantity, revenue FROM salesfacttable s JOIN datedimension d
    ON s.dateid = d.dateid
    WHERE date >= %s AND date <= %s""",
                (first_day, last_day))
    try:
        tuple_sales = cursor.fetchall()
        df = pd.DataFrame(tuple_sales, columns=['dateid', 'customerid', 'branchid', 'paymentid',
        'areaid', 'productid', 'cylindertypeid', 'quantity', 'revenue'])

        path = base / f"SalesFactTable_{date_today}.csv"

        with path.open("wb") as file:
            df.to_csv(file)
    except Exception:
        pass
```

Figure 48. OLAP RETL Task (Sales Fact Table)

Using the ObjectStoragePath, the pendulum datetime function, and the RedshiftSQLHook, the sales fact table for the previous month is saved as a csv file. This is done by extracting the data within the past month through a filtering SQL query which also joins the sales fact table with the date dimension. Afterwards, by converting the query result into a pandas data frame, it is then saved as a csv file into a created file path in the gold zone.

```python
    # Inventory Fact Table
    cursor.execute("""SELECT d.dateid, productid, cylindertypeid, availablequantity, unavailablequantity
    FROM inventoryfacttable i JOIN datedimension d
    ON i.dateid = d.dateid
    WHERE date >= %s AND date <= %s""",
              (first_day, last_day))
    try:
        tuple_inventory = cursor.fetchall()
        df = pd.DataFrame(tuple_inventory, columns=['dateid', 'productid', 'cylindertypeid', 'availablequantity',
        'unavailablequantity'])

        path = base / f"InventoryFactTable_{date_today}.csv"

        with path.open("wb") as file:
            df.to_csv(file)
    except Exception:
        pass
```

Figure 49. OLAP RETL Task (Inventory Fact Table)

Using the same process as the Sales Fact Table, the Inventory Fact Table is also then saved as a csv file in the gold zone.

```
# Logistics Fact Table
cursor.execute("""SELECT d.dateid, customerid, areaid, productid,
cylindertypeid, driverid, deliveryquantity, pickupquantity FROM logisticsfacttable l JOIN datedimension d
ON l.dateid = d.dateid
WHERE date >= %s AND date <= %s""", (first_day, last_day))
try:
    tuple_logistics = cursor.fetchall()
    df = pd.DataFrame(tuple_logistics, columns=['dateid', 'customerid', 'areaid', 'productid',
    'cylindertypeid', 'driverid', 'deliveryquantity', 'pickupquantity'])

    path = base / f"LogisticsFactTable_{date_today}.csv"

    with path.open("wb") as file:
        df.to_csv(file)

except Exception:
    pass
```

Figure 50. OLAP RETL Task (Logistics Fact Table)

Using the same process as the Sales and Inventory Fact Tables, the Logistics Fact Table is also then saved as a csv file in the gold zone.

ETL for NoSQL

```
from airflow import DAG
from datetime import datetime
from airflow.io.path import ObjectStoragePath
from airflow.decorators import task
from airflow.providers.amazon.aws.hooks.dynamodb import DynamoDBHook
from boto3.dynamodb.conditions import Key
import pendulum
```

Figure 51. NoSQL ETL DAG Imported Libraries and Functions

The imported libraries involve the libraries and functions in the other DAGs. However, other libraries and functions used in this DAG are the DynamoDBHook from Airflow and the Key from boto3 which is used in DynamoDB queries.

```
with DAG(dag_id='nosql_etl',
         schedule='@daily',
         start_date=pendulum.datetime(2024, 9, 19, tz="Asia/Manila"),
        ) as dag:
    import json
    import pandas as pd
    import re
    import boto3

    base1 = ObjectStoragePath("s3://s3_diwata_gas@diwata-gas-bucket/landing/CustomerComplaint/")
    base2 = ObjectStoragePath("s3://s3_diwata_gas@diwata-gas-bucket/landing/CustomerResolution/")
```

Figure 52. NoSQL DAG Details, other imported libraries, and base paths

For the NoSQL DAG, the DAG ID is “nosql_etl”, and it is scheduled daily. Moreover, the start date is localized in the “Asia/Manila” time zone, and the json, pandas, regex, and boto3 libraries are also used for this DAG. Lastly, two base paths are used which are namely, the CustomerComplaint and CustomerResolution paths.

```
    @task
    def nosql_etl():
        """Extract, transform, and load data for DynamoDB.
        """
        dynamodb = boto3.resource('dynamodb')
        diwata_gas = dynamodb.Table('DiwataGasComplaints')
        nosql_files = [f for f in base1.iterdir() if f.is_file()]
        s3 = boto3.resource('s3')
        for files in nosql_files:
            nosql_json = json.loads(files.read_text())
            try:
                if len(diwata_gas.query(KeyConditionExpression=Key('SerialNo').eq(nosql_json['ComplaintSerialNo']))['Item']) == 0:
                    complaint_id = 1
                else:
                    complaint_id = 2
            except Exception:
                complaint_id = len(diwata_gas.query(KeyConditionExpression=Key('SerialNo').eq(nosql_json['ComplaintSerialNo']))['Items']) + 1
            diwata_gas.put_item(
                Item={
                    'SerialNo': nosql_json['ComplaintSerialNo'],
                    'ComplaintID': complaint_id,
                    'CustomerName': nosql_json['CustomerComplaintName'],
                    'Date': datetime.strftime(datetime.strptime(nosql_json['ComplaintDateTime'], "%Y-%m-%dT%H:%M"), '%Y-%m-%d'),
                    'Time': datetime.strftime(datetime.strptime(nosql_json['ComplaintDateTime'], "%Y-%m-%dT%H:%M"), '%H:%M:00'),
                    'CustomerContactNumber': nosql_json['CustomerComplaintContactNumber'],
                    'ChiefComplaints': set(nosql_json['ChiefComplaint'].split(', ')),
                    'PictureS3Path': nosql_json['picture'],
                    'CustomerReviews': nosql_json['ComplaintDetails']
                })
            s3.Object("diwata-gas-bucket", f"landing/CustomerComplaint/Processed/{files.name}").copy_from(
            CopySource=f"diwata-gas-bucket/landing/CustomerComplaint/{files.name}")

            s3.Object("diwata-gas-bucket", f"landing/CustomerComplaint/{files.name}").delete()
```

Figure 53. NoSQL ETL Task

In this task, boto3 is used to acquire the DynamoDB resource of the AWS account. Moreover, this DynamoDB resource is used to extract the specific table for this architecture, which would be the DiwataGasComplaints table. In adding the new data, each file path is listed from the CustomerComplaint path, and to create an incremental Complaint ID, it is checked whether there is an

existing Complaint ID for the given serial number, wherein if there is none, the Complaint ID for the new item is set at 1, while if there is an existing Complaint ID, it is then incremented by 1 for the new item.

In adding the item, aside from the values which are directly placed into the database, some transformations are done for the others. For the date and time, both are extracted using the datetime's strptime function from a single datetime value from the given data. Lastly, for the Chief Complaints, given that it is in string form, it is first split by ", ", and then it is placed within a set. After adding these new items, through the S3 resource from the boto3 library, the files where the data are extracted from are then moved into a folder entitled "Processed". Afterwards, these files are deleted from the source folder, to avoid duplications during the successive running of this DAG.

```
@task
def nosql_update():
    """Update data for DynamoDB.
    """
    dynamodb = boto3.resource('dynamodb')
    diwata_gas = dynamodb.Table('DiwataGasComplaints')
    nosql_files = [f for f in base2.iterdir() if f.is_file()]
    s3 = boto3.resource('s3')
    for files in nosql_files:
        nosql_json = json.loads(files.read_text())
        if nosql_json['ForReplacement'] == 'Yes':
            replace = True
        else:
            replace = False
        try:
            diwata_gas.update_item(
                Key={
                    'SerialNo': nosql_json['SerialNo'],
                    'ComplaintID': int(nosql_json['ComplaintID'])
                },
                UpdateExpression="set IsReplaced = :r, MaintenanceComments = :m",
                ExpressionAttributeValues={
                    ':r': replace, ':m': nosql_json['ResComment']
                }
            )
            s3.Object("diwata-gas-bucket", f"landing/CustomerResolution/Processed/{files.name}").copy_from(
            CopySource=f"diwata-gas-bucket/landing/CustomerResolution/{files.name}")

            s3.Object("diwata-gas-bucket", f"landing/CustomerResolution/{files.name}").delete()
        except Exception:
            s3.Object("diwata-gas-bucket", f"landing/CustomerResolution/Error/{files.name}").copy_from(
            CopySource=f"diwata-gas-bucket/landing/CustomerResolution/{files.name}")

            s3.Object("diwata-gas-bucket", f"landing/CustomerResolution/{files.name}").delete()
```

Figure 54. NoSQL Update Task

The same logic from the previous DAG is done to load the DynamoDB table. Moving on, in iterating through each new data file, given that the "ForReplacement" attribute from the file is in string format, it is then converted into Boolean format. In updating an item, given a key-pair of the serial number

and an int-cast complaint ID, the “IsReplaced” and “MaintenanceComments” attributes are then replaced with the new values from the data file. Afterwards, the processed files are then transferred to the “Processed” folder, while files which ended up with an error are transferred to the “Error” folder, for inspection by the resident data engineer.

```
nosql_etl() >> nosql_update()
```

Figure 55. Sequence of NoSQL DAG Workflow

Given that the loaded files from the NoSQL ETL Task may be updated with the NoSQL Update Task, it is then logical to start first with the ETL task before the update task as seen above.

4.4. Online Transaction Processing (OLTP) Data Base: AWS RDS PostgreSQL

The OLTP database is built using PostgreSQL on AWS RDS, chosen for its high availability, which is crucial since order information is directly linked to the company's primary revenue source. According to an AWS article, PostgreSQL offers essential features like automatic failovers and the redundancy necessary for manufacturing enterprises to protect critical data (AWS, n.d.). On top of this, the database is equipped to manage the substantial volume of order data originating from multiple branches.

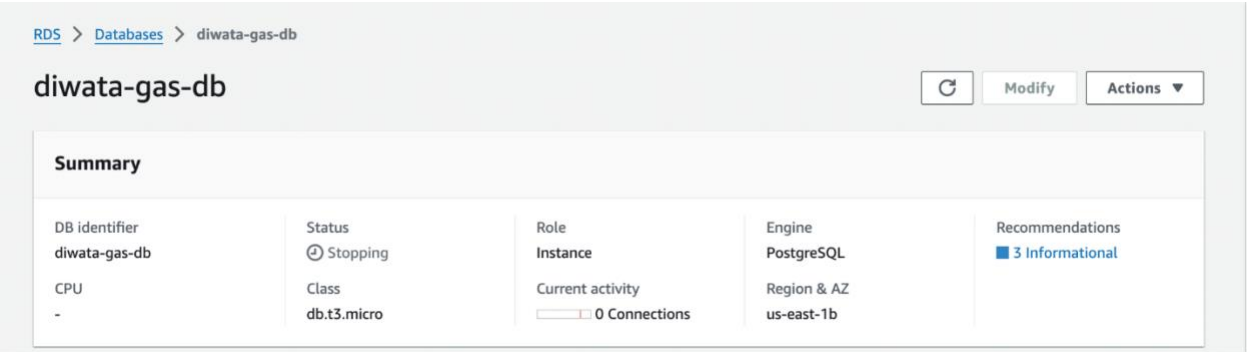


Figure 56. Summary Details of AWS RDS Database

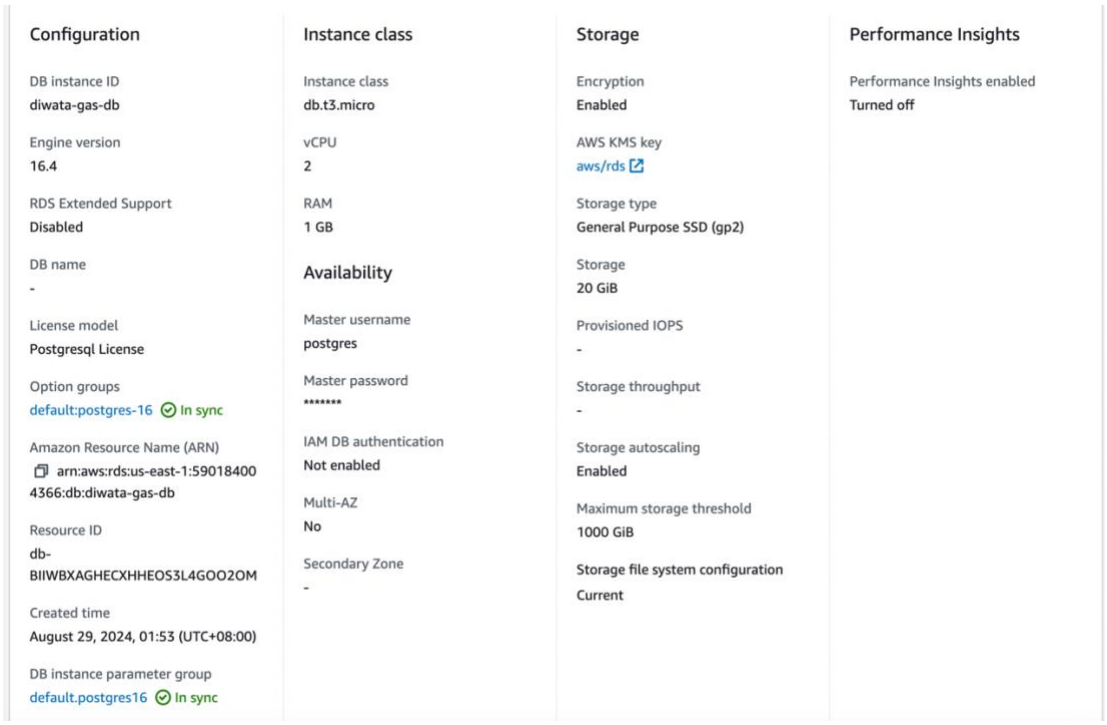


Figure 57. Configuration Details of AWS RDS Database

Figures 56 and 57 outline the specifications for the AWS RDS setup used to create the OLTP database. Key details include the use of PostgreSQL as the database engine on a db.t3.micro instance type. This configuration includes 2 vCPUs, a storage capacity of 20 gigabytes, and 1 gigabyte of RAM.

```
[ ]: %sql CREATE USER jj;

[25]: %sql ALTER USER jj WITH PASSWORD :password;

Running query in 'postgresql://postgres:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com'

[25]:

[26]: %sql CREATE DATABASE diwatagas_oltp_db OWNER jj;

Running query in 'postgresql://postgres:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com'
```

Figure 58. Set of Codes to Create Diwata Gas Corporation's OLTP Database

The OLTP database was set up using Python on an EC2 instance, which is linked to AWS RDS. The first step involves creating a user and granting this user access to PostgreSQL. Following that, this user is employed to establish the database. **Figure 58** illustrates the code required to perform these actions.

```
%sql \l
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Name	Owner	Encoding	Collate	Ctype	Access privileges
diwatagas_oltp_db	jj	UTF8	en_US.UTF-8	en_US.UTF-8	None
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	None
rdsadmin	rdsadmin	UTF8	en_US.UTF-8	en_US.UTF-8	rdsadmin=CTc/rdsadmin
template0	rdsadmin	UTF8	en_US.UTF-8	en_US.UTF-8	=c/rdsadmin rdsadmin=CTc/rdsadmin
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres postgres=CTc/postgres

Figure 59. List of All Databases on Postgres

To verify the creation of the database, the P-SQL command '\l' was executed to list all databases. As shown in **Figure 59**, 'diwatagas_oltp_db' was successfully created.

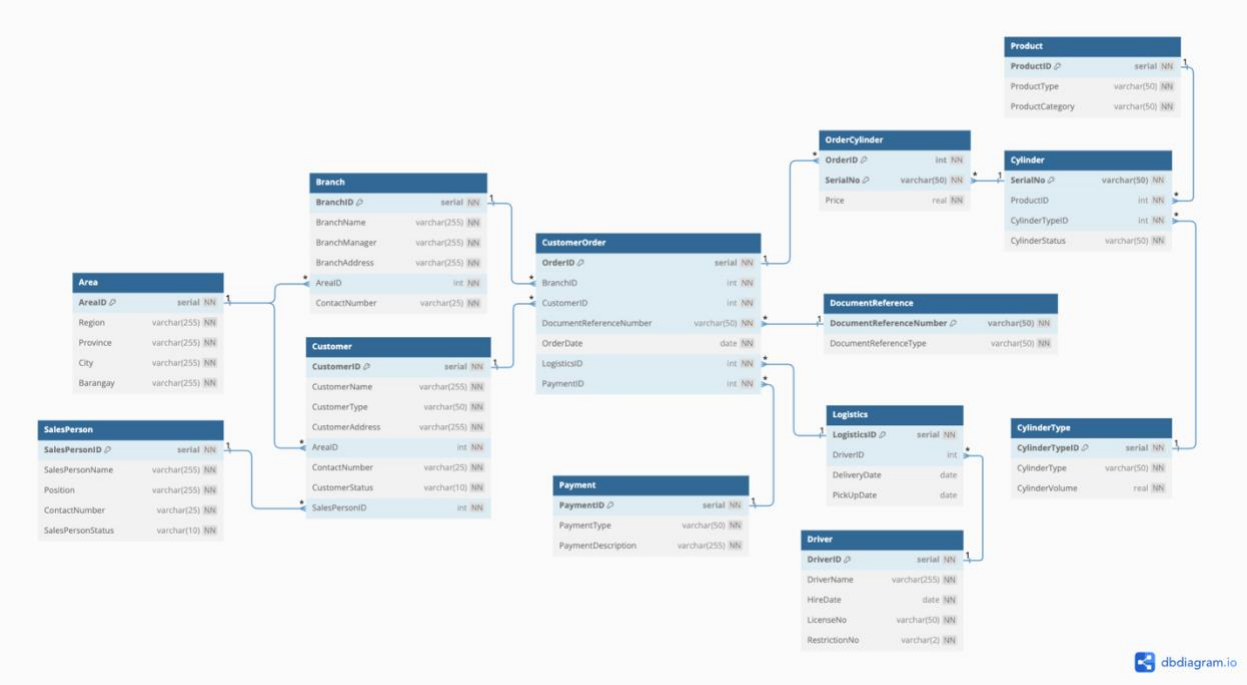


Figure 60. Online Transaction Processing (OLTP) Database Schema

The OLTP database consists of 13 distinct tables, which are normalized to improve organizational clarity and facilitate easier maintenance. **Figure 60** displays the schema of Diwata Gas Corporation's OLTP database. To provide a detailed understanding, each table is elaborated upon, including the SQL commands used for their setup and example data:

1. **Area table.** The branch and customer tables refer to the area table to identify their respective locations, connecting through the 'AreaID' which is the primary key of the area table. This table delineates the specific locations of branches and customers across various geographical tiers in the Philippines—region, province, city, and barangay.

```
%%sql
CREATE TABLE IF NOT EXISTS Area (
    AreaID SERIAL PRIMARY KEY NOT NULL,
    Region VARCHAR(255) NOT NULL,
    Province VARCHAR(255) NOT NULL,
    City VARCHAR(255) NOT NULL,
    Barangay VARCHAR(255) NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 61. SQL Script for Creating the Area Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'area';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

5 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	area	areaid	1	nextval('area_areaid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	area	region	2	None	NO	character varying	255
diwatagas_oltp_db	public	area	province	3	None	NO	character varying	255
diwatagas_oltp_db	public	area	city	4	None	NO	character varying	255
diwatagas_oltp_db	public	area	barangay	5	None	NO	character varying	255

Figure 62. Table Schema of the Area Table

Figures 61 and 62 display the SQL codes and outputs used for creating the area table and displaying its schema. The primary key 'AreaID' utilizes the serial data type. All additional attributes are of the varchar type, designed to hold up to 255 characters, and are mandated to be non-nullable as per the constraints established in the code.

areaid	region	province	city	barangay
1	Metro Manila	NCR	Quezon City	Barangay 1
2	Metro Manila	NCR	Quezon City	Barangay 2
3	Metro Manila	NCR	Quezon City	Barangay 3
4	Metro Manila	NCR	Manila	Barangay 1
5	Metro Manila	NCR	Manila	Barangay 2
6	Metro Manila	NCR	Manila	Barangay 3

Figure 63. Sample Data for the Area Table

To demonstrate the functionality of the area table, synthetic data was created using various regions, provinces, and cities in the Philippines as samples. For each unique combination of these three geographic levels, three barangays were generated following the format ‘Barangay X’.

2. **SalesPerson table.** This table is designated to record all sales personnel of Diwata Gas Corporation. It is indirectly linked to the orders table through the customer table, since each customer is managed by a salesperson. A unique identifier, 'SalesPersonID', acts as the primary key, with several key details such as name, position, contact, and status directly associated with it.

```
%%sql
CREATE TABLE IF NOT EXISTS SalesPerson (
    SalesPersonID SERIAL PRIMARY KEY NOT NULL,
    SalesPersonName VARCHAR(255) NOT NULL,
    Position VARCHAR(255) NOT NULL,
    ContactNumber VARCHAR(25) NOT NULL,
    SalesPersonStatus VARCHAR(10) NOT NULL
);
```

```
Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'
```

Figure 64. SQL Script for Creating the SalesPerson Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'salesperson';
```

```
Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'
5 rows affected.
```

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	salesperson	salespersonid	1	nextval('salesperson_salespersonid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	salesperson	salespersonname	2	None	NO	character varying	255
diwatagas_oltp_db	public	salesperson	position	3	None	NO	character varying	255
diwatagas_oltp_db	public	salesperson	contactnumber	4	None	NO	character varying	25
diwatagas_oltp_db	public	salesperson	salespersonstatus	5	None	NO	character varying	10

Figure 65. Table Schema of the SalesPerson Table

Figures 64 and 65 illustrate the SQL commands and outputs used to create and outline the schema of the salesperson table. The primary key, 'SalesPersonID', is of the serial data type. All other attributes are varchar type and are required to be non-nullable according to the code's constraints. 'SalesPersonName' and 'Position' can contain up to 255 characters, whereas 'ContactNumber' and 'SalesPersonStatus' are limited to 25 and 10 characters respectively.

salespersonid	salespersonname	position	contactnumber	salespersonstatus
1	Bonnie Powers	Sales Executive	(848)481-7756x4465	Inactive
2	Daniel Bullock	Sales Specialist	+1-424-869-3739x76346	Inactive
3	Mark Jensen	Sales Specialist	7389376989	Active
4	Christine Mcgee	Sales Consultant	(790)426-8912	Active
5	Angela Jackson	Regional Sales Manager	882-785-8051x633	Active
6	Colleen Sosa	Sales Executive	(762)433-3762x93924	Active
7	Carlos Mathews	Sales Executive	(975)389-3620	Active
8	Andrea Brown	Sales Consultant	772-914-3031	Active
9	Stacy Davis	Account Executive	(275)806-9345	Active
10	Brent Thompson	Sales Representative	(305)548-1549x504	Inactive
11	Larry Ball	Sales Representative	001-368-952-2088x05549	Active
12	Benjamin Phillips	Sales Representative	474.462.8532	Active

Figure 66. Sample Data for the SalesPerson Table

To illustrate the functionality of the salesperson table, synthetic data was generated (**See Figure 66**). Using a Python package called Faker, random names and contact numbers were created. Additionally, lists of sales positions and statuses (either active or inactive) were compiled, and selections were made randomly from these lists to populate the table.

3. **Branch table.** This table acts as a central hub for all branches of Diwata Gas Corporation, helping to track which orders are processed at each location. Each branch is uniquely identified by a surrogate key, 'BranchID', which serves as the primary key. The table records branch names, manager details, and contact numbers for straightforward identification. Additionally, it includes an 'AreaID' foreign key to link each branch to broader geographic data fields.

```
%%sql
CREATE TABLE IF NOT EXISTS Branch (
    BranchID SERIAL PRIMARY KEY NOT NULL,
    BranchName VARCHAR(255) NOT NULL,
    BranchManager VARCHAR(255) NOT NULL,
    BranchAddress VARCHAR(255) NOT NULL,
    AreaID INTEGER REFERENCES Area NOT NULL,
    ContactNumber VARCHAR(25) NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 67. SQL Script for Creating the SalesPerson Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'branch';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

6 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	branch	branchid	1	nextval('branch_branchid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	branch	areaid	5	None	NO	integer	None
diwatagas_oltp_db	public	branch	branchname	2	None	NO	character varying	255
diwatagas_oltp_db	public	branch	branchmanager	3	None	NO	character varying	255
diwatagas_oltp_db	public	branch	branchaddress	4	None	NO	character varying	255
diwatagas_oltp_db	public	branch	contactnumber	6	None	NO	character varying	25

Figure 68. Table Schema of the Branch Table

The SQL commands and their corresponding outputs used to create and illustrate the schema of the branch table are shown in **Figures 67 and 68**. 'BranchID' is designated as the primary key and is of the serial data type. All other attributes are defined as non-nullable in accordance with the constraints, and are mostly of the varchar type, except for 'AreaID', which is an integer and references the area table. The fields for branch name, manager, and address can hold up to 255 characters, whereas the 'ContactNumber' is restricted to 25 characters.

```
%%sql
SELECT *
FROM Branch;
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

9 rows affected.

branchid	branchname	branchmanager	branchaddress	areaid	contactnumber
1	Diwata Marikina	Taylor George	Block 12 Lot 36 Banyan Grove Phase 2, Matumtum Highway, Marikina	16	908.403.1353x01078
2	Diwata QC	Candace Turner	9490 Earth Extension, Quezon City	1	641.532.3528x37985
3	Diwata Pampanga	Lauren Phillips	8978 James Street, San Fernando, Pampanga	19	001-632-879-0205x1267
4	Diwata Laguna	Jesse English	B05 L34 Hydra Road, Oliva Cove Phase 2, Santa Rosa, Laguna	73	+1-500-397-5035x7754
5	Diwata Cagayan	Rebecca King	3704 West Extension, Cagayan	130	5979568391
6	Diwata Baguio	Thomas Wallace	B01 L19 Bouganvilla Estates 1, Baticulin Road, Baguio City	163	710-584-2659x758
7	Quezon City	Cher Howards	Mabalo Street, Quezon City	1	(425)436-4131
8	Pasig City	Sher Hill	Vang Condominium, Pasig City	3	(425)436-4131
9	Makati City	Rex Laylo	Laylo Compound, Makati City	6	(425)436-4131

Figure 69. Sample Data for the Branch Table

Synthetic data was generated to demonstrate the functionality of the table (**See Figure 69**). The branch names were manually crafted by the team, while the branch manager, address, and contact numbers were generated using the Faker package. The 'AreaID' values were randomly selected to ensure they correspond with those in the area table.

4. **Customer table.** This table presents a comprehensive list of the company's customers, both active and inactive, distinguishable by the 'CustomerStatus' field. It links to the orders table, facilitating the identification of who placed each order. Each customer is assigned a

unique surrogate ID, which serves as the primary key. The table captures essential details such as the customer's status, name, type, address, and contact information. Like the branch table, it includes an 'AreaID' that specifies the sales area associated with the order and a 'SalesPersonID' to indicate the company representative managing the customer's account.

```
%%sql
CREATE TABLE IF NOT EXISTS Customer (
    CustomerID SERIAL PRIMARY KEY NOT NULL,
    CustomerName VARCHAR(255) NOT NULL,
    CustomerType VARCHAR(50) NOT NULL,
    CustomerAddress VARCHAR(255) NOT NULL,
    AreaID INTEGER REFERENCES Area NOT NULL,
    ContactNumber VARCHAR(25) NOT NULL,
    CustomerStatus VARCHAR(10) NOT NULL,
    SalesPersonID INTEGER REFERENCES SalesPerson NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 70. SQL Script for Creating the Customer Table

```
%%sql
ALTER TABLE customer ADD CONSTRAINT unique_customer UNIQUE (customername);
```

Figure 71. SQL Script to Add a Unique Constraint on CustomerName

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'customer';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

8 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	customer	salespersonid	8	None	NO	integer	None
diwatagas_oltp_db	public	customer	areaid	5	None	NO	integer	None
diwatagas_oltp_db	public	customer	customerid	1	nextval('customer_customerid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	customer	customeraddress	4	None	NO	character varying	255
diwatagas_oltp_db	public	customer	customerstatus	7	None	NO	character varying	10
diwatagas_oltp_db	public	customer	contactnumber	6	None	NO	character varying	25
diwatagas_oltp_db	public	customer	customername	2	None	NO	character varying	255
diwatagas_oltp_db	public	customer	customertype	3	None	NO	character varying	50

Figure 72. Table Schema of the Customer Table

The SQL commands and their resulting outputs that were used to establish and describe the schema of the customer table are depicted in **Figures 70, 71, and 72**. 'CustomerID' serves as the primary key and utilizes the serial data type. All other attributes are set as non-nullable according to the defined constraints, and are generally of the varchar type, except for the 'AreaID' and 'SalesPersonID' columns, which are integers and link to the area and salesperson tables. The fields for customer name and address can accommodate up to 255 characters,

while customer type, contact, and status are limited to 50, 25, and 10 characters respectively. Only the customer name is set to unique, meaning it cannot be repeated.

customerid	customername	customertype	customeraddress	areaid	contactnumber	customerstatus	salespersonid
1	Western Company Inc._0	Branch	25066 Patrick Well Suite 160, Tanudan	212	001-809-402-7859x9787	Active	45
2	Thompson Finance Inc._1	Branch	372 James Islands, Baliuag	42	9065397477	Active	28
3	Reeves City Services Inc._2	Dealer	7823 Santos Landing, Guagua	34	469.852.3891x9518	Active	38
4	Cunningham Mining Corporation_3	Dealer	0798 Perry Crossing, Los Baños	88	396.384.9473	Active	46
5	Taylor Summit Silver Finance Limited_4	Dealer	8982 Martin Terrace Suite 865, Tarlac City	56	393.574.1281	Active	100
6	Goodwin Services Corporation_5	Branch	25186 Steven Throughway Suite 057, Biñan	84	654-344-5882	Active	15
7	Metro Shipping Inc._6	Branch	75652 Michelle Avenue Apt. 512, Bokod	180	+1-453-773-3114	Active	25
8	DP Enterprise Inc._7	Dealer	02379 Steven Forges Apt. 159, Malolos	38	001-615-763-0904x783	Active	6
9	OW Construction Corporation_8	Branch	756 Shannon Lane, Cebu City	217	829.628.0297	Active	42
10	Northern Equities Inc._9	Direct Customer	3408 Jason Rest Suite 266, Tuguegarao	129	517-404-7259x60956	Active	61
11	FNX Hotel Inc._10	Branch	9851 Garner Island Suite 881, Pasil	216	369-308-2788x371	Active	97
12	Adams Dragon Liberty Development Corporation_11	Direct Customer	97625 Angela Tunnel, San Jose del Monte	47	979-996-9166	Active	100
13	NYO Empire Foods Corporation_12	Branch	977 Terri Tunnel, San Simon	28	+1-306-857-5958x0029	Active	99
14	Triple Morning Silver Construction Inc._13	Direct Customer	6533 Stewart Parkway Suite 375, Itogon	170	2093341164	Active	1
15	LF Empire Millennium Hotel Corporation_14	Direct Customer	773 Matthew Keys Suite 460, Angeles	22	(574)858-4401	Active	89
16	QPJN Summit Enterprise Corporation_15	Branch	876 Silva Ranch, Aparri	131	(273)870-5940x5011	Active	8

Figure 73. Sample Data for the Customer Table

Figure 73 shows sample data for the customer table. The data was synthetically generated using the Faker package to produce names, addresses, and contact numbers for customers. 'AreaID' and 'SalesPersonID' were chosen at random from the available IDs in their corresponding tables. The 'CustomerType' can be a branch, dealer, or direct customer, while the 'CustomerStatus' is categorized as either active or inactive. Additionally, to test the customer registration form on the web application, fake data was inputted and added to the table.

5. **Payment table.** The payment table is designed to record various payment methods available for settling customer orders. Each method is uniquely identified by a 'PaymentID', which serves as the primary key. This table also includes non-key attributes such as 'PaymentType' and 'PaymentDescription', which name and describe each payment method respectively.

```
%%sql
CREATE TABLE IF NOT EXISTS Payment (
    PaymentID SERIAL PRIMARY KEY NOT NULL,
    PaymentType VARCHAR(50) NOT NULL,
    PaymentDescription VARCHAR(255) NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 74. SQL Script for Creating the Payment Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'payment';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

3 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	payment	paymentid	1	nextval('payment_paymentid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	payment	paymenttype	2	None	NO	character varying	50
diwatagas_oltp_db	public	payment	paymentdescription	3	None	NO	character varying	255

Figure 75. Table Schema of the Payment Table

Figures 74 and 75 display the SQL commands and the outputs used for setting up and detailing the schema of the payment table. The 'PaymentID' is the primary key and is of the serial data type. All additional attributes are required to be non-nullable as per the constraints and are of the varchar type. The 'PaymentType' and 'PaymentDescription' fields can hold up to 50 and 255 characters, respectively.

```
%%sql
SELECT *
FROM Payment;
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

6 rows affected.

paymentid	paymenttype	paymentdescription
1	Mobile Wallet	Payments made through mobile wallet apps.
2	Cash	Direct cash transactions.
3	Bank Transfer	Direct transfers from bank accounts.
4	GCash	Payments made through GCash
5	Credit Card	Payments made through credit card
6	Debit Card	Payments made through debit card

Figure 76. Sample Data for the Payment Table

As seen in **Figure 76**, there are six types of payment methods that customers can use to settle their purchases.

6. **Driver table.** This table contains comprehensive details about the company's fleet. The 'DriverID', which serves as the primary key, is associated with specific information including the driver's name, hire date, license number, and restriction number.

```
%%sql
CREATE TABLE IF NOT EXISTS Driver (
    DriverID SERIAL PRIMARY KEY NOT NULL,
    DriverName VARCHAR(255) NOT NULL,
    HireDate DATE NOT NULL,
    LicenseNo VARCHAR(50) NOT NULL,
    RestrictionNo VARCHAR(2) NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 77. SQL Script for Creating the Driver Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'driver';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

5 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	driver	driverid	1	nextval('driver_driverid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	driver	hiredate	3	None	NO	date	None
diwatagas_oltp_db	public	driver	drivername	2	None	NO	character varying	255
diwatagas_oltp_db	public	driver	licenseno	4	None	NO	character varying	50
diwatagas_oltp_db	public	driver	restrictionno	5	None	NO	character varying	2

Figure 78. Table Schema of the Driver Table

Figures 77 and 78 showcase the SQL commands and outputs used to configure and illustrate the schema of the driver table. 'DriverID' acts as the primary key and uses the serial data type. All other attributes must be non-nullable according to the established constraints and are of the varchar type, except for the 'HireDate' attribute, which is of the Date data type. The varchar attributes include the driver's name, license number, and restriction number, which can accommodate up to 255, 50, and 2 characters, respectively.

```
%%sql
SELECT *
FROM Driver;
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

43 rows affected.

driverid	drivername	hiredate	licenseno	restrictionno
1	JJ Ramoso	2023-05-08	N83-61-180847	CE
2	PM Medina	2023-05-08	N83-61-180847	CE
3	RG Laylo	2023-05-08	N83-61-180847	CE
4	Justin Williams	2012-01-25	N12-79-993628	D
5	Reginald Lewis	2021-06-23	N67-55-496758	C
6	Robert Odonnell	2015-05-22	N28-11-684899	CE
7	Anne Taylor	2020-02-02	N39-03-100137	B2

Figure 79. Sample Data for the Driver Table

Driver data was synthetically added by the team, either through manual input or by using a script to automatically generate synthetic drivers (**See Figure 79**). License numbers adhere to the format used in the Philippines. The restriction numbers indicate the types of trucks that drivers are authorized to operate, although these numbers are fictional and do not correspond to actual codes on drivers' licenses.

7. **Logistics table.** This table is linked to the 'CustomerOrder' table through the 'LogisticsID', which assigns a specific driver to deliver and pick up each order. The primary key of the table is the 'LogisticsID'. Associated attributes include the 'DriverID' representing the driver, along with the scheduled 'DeliveryDate' and 'PickupDate' for each order.

```
%%sql
CREATE TABLE IF NOT EXISTS Logistics (
    LogisticsID SERIAL PRIMARY KEY NOT NULL,
    DriverID INTEGER,
    DeliveryDate DATE,
    PickUpDate DATE
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 80. SQL Script for Creating the Logistics Table

```
%sql ALTER TABLE logistics ADD CONSTRAINT driver_unique UNIQUE (driverid, deliverydate, pickupdate);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 81. SQL Script to Add a Unique Constraint on DeliveryID, DeliveryDate, and PickUpDate

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'logistics';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'
4 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	logistics	logisticsid	1	nextval('logistics_logisticsid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	logistics	driverid	2	None	YES	integer	None
diwatagas_oltp_db	public	logistics	deliverydate	3	None	YES	date	None
diwatagas_oltp_db	public	logistics	pickupdate	4	None	YES	date	None

Figure 82. Table Schema of the Logistics Table

Figures 80, 81, and 82 display the SQL commands and the resulting outputs for setting up and detailing the schema of the logistics table. 'LogisticsID' serves as the primary key and is of the serial data type. All other attributes are nullable to accommodate situations where details such as driver assignment, delivery date, and pick-up date are not immediately inputed by the logistics team. 'DriverID' is an integer type, while 'DeliveryDate' and 'PickUpDate' are both of the date data type.

Moreover, combinations of ‘DriverID’, 'DeliveryDate', and 'PickUpDate' should be unique except when all are null.

logisticsid	driverid	deliverydate	pickupdate
6	None	None	None
7	None	None	None
8	None	None	None
9	None	None	None
10	None	None	None
11	None	None	None
12	None	None	None
4	1	2024-08-30	2024-09-02
1	1	2024-09-30	2024-09-03
5	2	2024-08-29	None
2	2	2024-08-31	2024-09-05
3	3	2024-08-30	None

Figure 83. Sample Data for the Logistics Table

Figure 83 displays the logistics table filled with synthetic data. As mentioned earlier, it is possible for certain fields to contain null values when orders are pending assignment of delivery details. Generally, drivers and delivery dates are assigned concurrently, whereas pick-up dates are scheduled later as per customer instructions. Moreover, it is important to highlight that 'DriverID', serving as a foreign key, must match an existing ID in the driver table.

8. **Product table.** The ‘Product’ table catalogs all the gas types available for purchase. Each type of gas is uniquely identified by a 'ProductID', which serves as the primary key. Dependent non-key attributes in this table include the name and category of the 17 gas products manufactured and distributed by the company.

```
%%sql
CREATE TABLE IF NOT EXISTS Product (
    ProductID SERIAL PRIMARY KEY NOT NULL,
    ProductType VARCHAR(50) NOT NULL,
    ProductCategory VARCHAR(50) NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 84. SQL Script for Creating the Product Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'product';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

3 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	product	productid	1	nextval('product_productid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	product	producttype	2	None	NO	character varying	50
diwatagas_oltp_db	public	product	productcategory	3	None	NO	character varying	50

Figure 85. Table Schema of the Product Table

Figures 84 and 85 showcase the SQL commands and their respective outputs used to establish and describe the schema of the product table. 'ProductID' is the primary key and is of the serial data type. All other non-key attributes are non-nullable and are of the varchar data type, each capable of accommodating up to 50 characters.

productid	producttype	productcategory
1	Acetylene (C2H2 I.G.)	Industrial
2	Acetylene (C2H2)	Industrial
3	Argon (200 BARS - ARGO51)	Industrial
4	Argon (Ar HP)	Industrial
5	Argon (Ar UHP)	Industrial
6	Argon (Ar)	Industrial
7	Argon (ARGO40)	Industrial

Figure 86. Sample Data for the Product Table

Figure 86 shows a section of the product table which contains 17 types of gas products offered by the company. These gases are classified as either industrial or medical, depending on their intended use.

9. **CylinderType table.** The 'CylinderType' table stores information about the five different cylinder sizes available. Each size is uniquely identified by a 'CylinderTypeID', which serves as the primary key. The table also includes non-key attributes such as the name of the cylinder and its volume capacity in liters.

```sql
%%sql
CREATE TABLE IF NOT EXISTS CylinderType (
    CylinderTypeID SERIAL PRIMARY KEY NOT NULL,
    CylinderType VARCHAR(50) NOT NULL,
    CylinderVolume REAL NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 87. SQL Script for Creating the CylinderType Table

```sql
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'cylindertype';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

3 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	cylindertype	cylindertypeid	1	nextval('cylindertype_cylindertypeid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	cylindertype	cylindervolume	3	None	NO	real	None
diwatagas_oltp_db	public	cylindertype	cylindertype	2	None	NO	character varying	50

Figure 88. Table Schema of the CylinderType Table

Figures 87 and 88 illustrate the SQL commands and the outputs used for creating and describing the schema of the CylinderType table. 'CylinderTypeID' serves as the primary key and uses the serial data type. All additional non-key attributes are set to be non-nullable. The cylinder type is specified as a varchar with a maximum capacity of 50 characters, while the cylinder volume is recorded in real format to accommodate decimal values.

cylindertypeid	cylindertype	cylindervolume
1	Oversized	60.0
2	Standard	40.0
3	Bantam	30.0
4	Medium	15.0
5	Flask Type	10.0

Figure 89. Sample Data for the CylinderType Table

Figure 89 displays the five cylinder sizes available from the company, detailing the name and volume in liters for each cylinder type.

10. **Cylinder table.** This table is a central repository for all the company's cylinder assets. The 'SerialNo', serving as the primary key, is a natural key directly linked to a physical label on each cylinder, rather than being generated by the system. Cylinder sizes are classified by 'CylinderTypeID', and each cylinder is capable of containing only one type of product, as specified by 'ProductID'. The status of each cylinder, whether available or unavailable, is defined by 'CylinderStatus'.

```
%%sql
CREATE TABLE IF NOT EXISTS Cylinder (
    SerialNo VARCHAR(50) PRIMARY KEY NOT NULL,
    ProductID INTEGER REFERENCES Product NOT NULL,
    CylinderTypeID INTEGER REFERENCES CylinderType NOT NULL,
    CylinderStatus VARCHAR(10) NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 90. SQL Script for Creating the Cylinder Table

```
%%sql
ALTER TABLE Cylinder
ALTER COLUMN CylinderStatus TYPE VARCHAR(50);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 91. SQL Script to Alter the Cylinder Column to Handle a Higher Maximum Character Count

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'cylinder';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

4 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	cylinder	productid	2	None	NO	integer	None
diwatagas_oltp_db	public	cylinder	cylindertypeid	3	None	NO	integer	None
diwatagas_oltp_db	public	cylinder	serialno	1	None	NO	character varying	50
diwatagas_oltp_db	public	cylinder	cylinderstatus	4	None	NO	character varying	50

Figure 92. Table Schema of the Cylinder Table

Figures 90, 91, and 92 display the SQL commands and their outputs for setting up and outlining the schema of the Cylinder table. 'SerialNo' is designated as the primary key and uses the varchar data type with a maximum capacity of 50 characters, similar to the non-key attribute 'CylinderStatus'. Both 'ProductID' and 'CylinderTypeID' are integer types and link to their respective tables. Additionally, all non-key attributes are configured to be non-nullable.

serialno	productid	cylindertypeid	cylinderstatus
S20-184221	16	5	Available
x28-625067	11	4	Available
A67-730814	2	2	Available
q25-408851	10	2	Available
u72-068397	3	1	Available
k66-879598	5	1	Available
W23-340576	11	1	Available
g15-582070	10	3	Available
v46-712308	4	3	Available
A34-347046	1	3	Available
b25-671067	9	3	Available
D72-922999	3	2	Available
k87-877371	1	1	Available
N87-734666	16	3	Available

Figure 93. Sample Data for the Cylinder Table

Figure 93 presents a sample from the cylinder table. Each cylinder is identified by a serial number and can contain any of the 17 gas products, fitting into one of the 5 available cylinder sizes. The status of these cylinders is marked as either available or unavailable. Unavailable cylinders require either refilling or repair.

11. **DocumentReference table.** For each customer order, a physical document is required to confirm its fulfillment. The 'DocumentReference' table stores these documents, which are linked to the orders. The documents, typically a delivery receipt or a sales invoice, have unique reference numbers issued by the Philippine Bureau of Internal Revenue. These numbers serve as natural keys and are designated as the primary keys of the table. The only non-key attribute in this table is the 'DocumentReferenceType'.

```
%%sql
CREATE TABLE IF NOT EXISTS DocumentReference (
    DocumentReferenceNumber VARCHAR(50) PRIMARY KEY NOT NULL,
    DocumentReferenceType VARCHAR(50) NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 94. SQL Script for Creating the DocumentReference Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'documentreference';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

2 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	documentreference	documentreferencenumber	1	None	NO	character varying	50
diwatagas_oltp_db	public	documentreference	documentreferencetype	2	None	NO	character varying	50

Figure 94. Table Schema of the DocumentReference Table

Figures 93 and 94 show the SQL commands and the corresponding outputs used to establish and define the schema of the DocumentReference table. 'DocumentReferenceNumber' serves as the primary key and is of the varchar data type, capable of holding up to 50 characters, similar to the non-key attribute 'DocumentReferenceType'. Both columns are set to be non-nullable.

documentreferencenumber	documentreferencetype
GH07OE1GIO12VX	DeliveryReceipt
DL08O22FIO12LM	DeliveryReceipt
KP25100FIO12MN	DeliveryReceipt
GH07OE1GIO1SAVX	DeliveryReceipt
RGL599FIO12FB	SalesInvoice
XL599FIO12DL	DeliveryReceipt
BF508FIO12DS	DeliveryReceipt
IG25100FIO12SF	DeliveryReceipt

Figure 95. Sample Data for the DocumentReference Table

Figure 95 presents sample data from the DocumentReference table. This table includes a 'DocumentReferenceNumber' which, for demonstration purposes, is randomly generated and does not reflect the actual numbering sequence used by the Philippine Bureau of Internal Revenue. The documents listed, such as delivery receipts or sales invoices, act as verification for the delivery of customer orders.

12. **CustomerOrder table.** Each row in this table represents an order transaction, identified by a unique primary key. Additional transaction details are linked through foreign keys for the branch, customer, logistics, and payment. The table also records the order date and a document reference number, which connects to an external table for further details.

The customer table is structured to store and associate essential information for each transaction.

```
%%sql
CREATE TABLE IF NOT EXISTS CustomerOrder (
    OrderID SERIAL PRIMARY KEY NOT NULL,
    BranchID INTEGER REFERENCES Branch NOT NULL,
    CustomerID INTEGER REFERENCES Customer NOT NULL,
    DocumentReferenceNumber VARCHAR(50) REFERENCES DocumentReference NOT NULL,
    OrderDate DATE NOT NULL,
    PaymentID INTEGER REFERENCES Payment NOT NULL,
    LogisticsID INTEGER NOT NULL
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 96. SQL Script for Creating the CustomerOrder Table

```
%sql ALTER TABLE customerorder ADD CONSTRAINT document_unique UNIQUE (branchid, customerid, documentreferencenumber, orderdate, paymentid);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 97. SQL Script to Add a Unique Constraint on a Combination of BranchID, CustomerID, DocumentReferenceNumber, OrderDate, and PaymentID

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'customerorder';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'
7 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	customerorder	logisticsid	7	None	NO	integer	None
diwatagas_oltp_db	public	customerorder	branchid	2	None	NO	integer	None
diwatagas_oltp_db	public	customerorder	customerid	3	None	NO	integer	None
diwatagas_oltp_db	public	customerorder	orderid	1	nextval('customerorder_orderid_seq'::regclass)	NO	integer	None
diwatagas_oltp_db	public	customerorder	orderdate	5	None	NO	date	None
diwatagas_oltp_db	public	customerorder	paymentid	6	None	NO	integer	None
diwatagas_oltp_db	public	customerorder	documentreferencenumber	4	None	NO	character varying	50

Figure 98. Table Schema of the CustomerOrder Table

Figures 96, 97, and 98 display the SQL commands and their corresponding outputs used to establish and define the schema of the CustomerOrder table. 'OrderID' is the primary key and is designated as the serial data type. The fields 'BranchID', 'CustomerID', 'PaymentID', and 'DocumentReferenceNo' function as foreign keys and are also primary keys in their respective tables. The first three are of integer type, while 'DocumentReferenceNo' is a varchar that supports up to 50 characters. The 'OrderDate' is the sole attribute not serving as a foreign key and is of date data type. Rows should be a unique set or combination of BranchIDs, Customer IDs, Document Reference Number,s Order Dates, and Payment IDs.

orderid	branchid	customerid	documentreferencenumber	orderdate	paymentid	logisticsid
1	7	201	GH07OE1GIO12VX	2024-08-15	2	1
2	7	201	DL08O22FIO12LM	2024-08-28	4	2
3	7	201	GH07OE1GIO12VX	2024-08-29	2	3
4	7	201	GH07OE1GIO12VX	2024-09-01	2	4
5	9	203	KP25100FIO12MN	2024-08-28	4	5
6	9	203	KP25100FIO12MN	2024-08-29	2	6
7	8	204	RGL599FIO12FB	2024-08-30	2	7
8	9	204	XL599FIO12DL	2024-08-31	6	8

Figure 99. Sample Data for the CustomerOrder Table

Figure 99 provides a sample from the CustomerOrder table. Unique order IDs and order dates were generated, while identifiers for branches, customers, document references, payments, and logistics were randomly selected from their respective tables using a Python script to populate the CustomerOrder table.

13. **OrderCylinder table.** This bridge table connects cylinders to specific customer orders, particularly when multiple cylinders are bought under one 'OrderID'. The table uses 'OrderID' and 'SerialNo' as its primary keys, which also act as foreign keys. Each record includes a price, which reflects the cost of cylinders for each distinct order. Given the company's unique pricing policy, which varies prices based on customer and market conditions, it is prudent not to associate the price directly with the cylinder's serial number in the cylinder table, but rather to manage pricing through this bridge table.

```
%%sql
CREATE TABLE IF NOT EXISTS OrderCylinder (
    OrderID INTEGER REFERENCES CustomerOrder NOT NULL,
    SerialNo VARCHAR(50) REFERENCES Cylinder NOT NULL,
    Price REAL NOT NULL,
    PRIMARY KEY(OrderID, SerialNo)
);
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Figure 100. SQL Script for Creating the OrderCylinder Table

```
%%sql
SELECT *
FROM information_schema.columns
WHERE table_name = 'ordercylinder';
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

3 rows affected.

table_catalog	table_schema	table_name	column_name	ordinal_position	column_default	is_nullable	data_type	character_maximum_length
diwatagas_oltp_db	public	ordercylinder	orderid	1	None	NO	integer	None
diwatagas_oltp_db	public	ordercylinder	price	3	None	NO	real	None
diwatagas_oltp_db	public	ordercylinder	serialno	2	None	NO	character varying	50

Figure 101. Table Schema of the OrderCylinder Table

Figures 100 and 101 present the SQL commands and their results used to set up and describe the schema of the OrderCylinder table. This bridge table's primary composite key consists of 'OrderID' and 'SerialNo', both of which are foreign keys linking to their respective tables. 'OrderID' is an integer type, while 'SerialNo' is a varchar type with a capacity of 50 characters. The only non-key attribute is 'price'. Notably, all columns in this table are configured to be non-nullable.

orderid	serialno	price
1	Y76-917600	25000.0
1	h22-534522	11000.0
2	L22-621685	11000.0
2	K24-041119	29000.0
3	Y76-917600	25000.0
3	h22-534522	11000.0

Figure 102. Sample Data for the OrderCylinder Table

Figure 102 showcases a sample from the OrderCylinder table, illustrating that a single 'OrderID' can be associated with multiple serial numbers. This highlights the capability of the table to track multiple cylinders within a single order transaction.

```
%sql \dt
```

Running query in 'postgresql://jj:***@diwata-gas-db.cdi6iccimjq9.us-east-1.rds.amazonaws.com/diwatagas_oltp_db'

Schema	Name	Type	Owner
public	area	table	jj
public	branch	table	jj
public	customer	table	jj
public	customerorder	table	jj
public	cylinder	table	jj
public	cylindertype	table	jj
public	documentreference	table	jj
public	driver	table	jj
public	logistics	table	jj
public	ordercylinder	table	jj
public	payment	table	jj
public	product	table	jj
public	salesperson	table	jj

Figure 103. List of All Tables on the 'diwatagas_oltp_db' Database

Figure 103 confirms the successful creation of all 13 tables in the database, as evidenced by the output of the JupySQL command '\dt'.

4.5 Online Analytics Processing Data Base (OLAP): AWS Redshift

AWS Redshift was chosen for the creation of the company's OLAP database / data warehouse due to its efficient data handling capabilities, particularly for analytics. It offers rapid query responses and includes robust security features essential for protecting company data. Additionally, AWS Redshift is cost-effective compared to competitors like Snowflake, as noted by Tobin (2023). It also supports customization to accommodate data from various sources and handle multiple data formats, including JSON and Parquet.

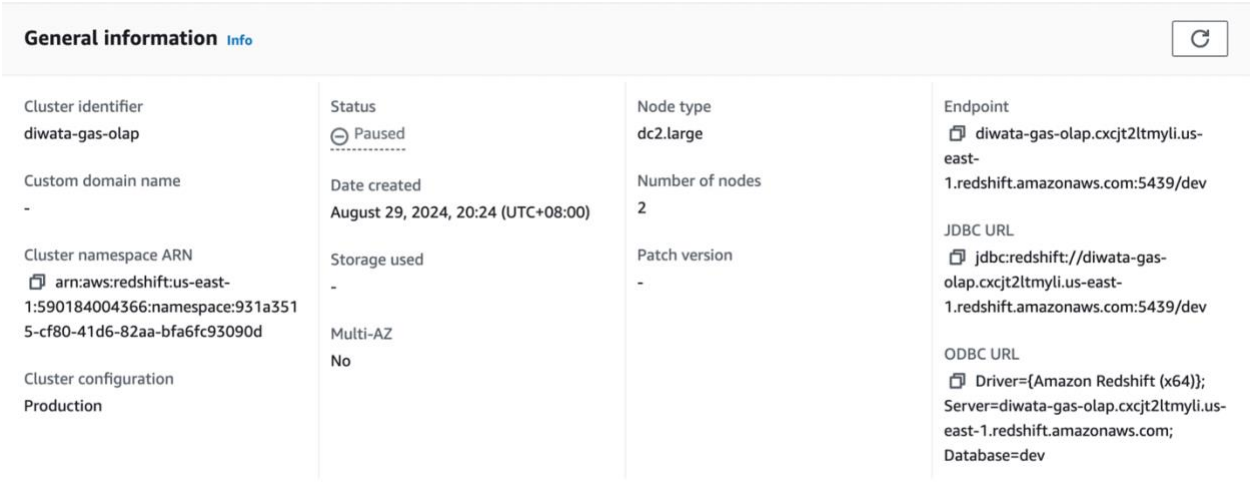


Figure 104. General Information Details of AWS Redshift Cluster Created

Figure 104 shows that the Redshift cluster, named 'diwata-gas-olap,' consists of 2 nodes of the dc2.large type.

```
%sql CREATE DATABASE diwatagas_olap_db OWNER jj;
```

```
Running query in 'redshift://awsuser:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/dev'
```

Figure 105. Code to Create Diwata Gas Corporation's OLAP Database

The data warehouse for the company on Redshift was created using the same user. As depicted in **Figure 105**, the user named 'jj' was responsible for creating the database 'diwatagas_olap_db'.

```
%%sql
SELECT * FROM pg_database;
```

```
Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'
7 rows affected.
```

datname	datdba	encoding	datistemplate	datallowconn	datlastsysoid	datvacuumxid	datfrozenxid	dattablespace
awsdatacatalog	1	6	False	False	102368	924	924	0
diwatagas_olap_db	101	6	False	True	102368	924	924	1663
template0	1	6	True	False	102368	924	924	1663
dev	1	6	False	True	102368	0	0	1663
padb_harvest	1	6	False	True	102368	0	0	1663
sys:internal	1	6	False	True	102368	924	924	1663
template1	1	6	True	True	102368	924	924	1663

Figure 106. List of Databases on the Redshift Cluster 'diwata-gas-olap'

Figure 106 confirms that the OLAP database named 'diwatagas_olap_db' has been successfully created using the code provided in the previous figure.

OLAP Dimensions

Eight dimensions will be employed across three fact tables. While these dimensions share similarities with certain tables in the OLTP database, they are optimized to retain key attributes that enhance analytical capabilities. Each dimension is detailed below:

1. **DateDimension table.** This dimension is designed to assist in periodically filtering facts using unique identifiers such as date, day, month, year, day of the week, and quarter. It also facilitates analysis that considers weekends and holidays, as it includes specific attributes for these within the dimension.

```
%%sql
CREATE TABLE IF NOT EXISTS DateDimension (
    DateID INTEGER NOT NULL UNIQUE,
    Date DATE NOT NULL,
    Day SMALLINT NOT NULL,
    Month SMALLINT NOT NULL,
    Year SMALLINT NOT NULL,
    DayOfWeek VARCHAR(9) NOT NULL,
    Quarter SMALLINT NOT NULL,
    IsHoliday BOOL NOT NULL,
    IsWeekend BOOL NOT NULL
)
DISTSTYLE KEY DISTKEY (DateID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 107. SQL Script for Creating the Date Dimension

```
%sqlcmd columns -t datedimension
```

name	type	nullable	default	autoincrement	comment	info
dateid	INTEGER	False	None	False	None	{'encode': 'az64'}
date	DATE	False	None	False	None	{'encode': 'az64'}
day	SMALLINT	False	None	False	None	{'encode': 'az64'}
month	SMALLINT	False	None	False	None	{'encode': 'az64'}
year	SMALLINT	False	None	False	None	{'encode': 'az64'}
dayofweek	VARCHAR(9)	False	None	False	None	{'encode': 'lzo'}
quarter	SMALLINT	False	None	False	None	{'encode': 'az64'}
isholiday	BOOLEAN	False	None	False	None	{}
isweekend	BOOLEAN	False	None	False	None	{}

Figure 108. Table Schema of the DateDimension Table

Figure 107 presents the SQL script for creating the date dimension table in the database, and **Figure 108** illustrates the table's schema. The 'DateID' is established as the primary key, offering a unique identifier for each date, configured as an integer that is non-nullable and unique. All other attributes are set as non-nullable: day, month, year, and quarter are defined using the smallint data type; the date itself uses the date data type; the day of the week is specified as a varchar with a limit of 9 characters; boolean attributes denote whether the date is a holiday or weekend. The distribution style is centered around 'DateID' to facilitate the team's querying of data based on specific dates.

```
%%sql
SELECT *
FROM DateDimension;
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

365 rows affected.

dateid	date	day	month	year	dayofweek	quarter	isholiday	isweekend
20240103	2024-01-03	3	1	2024	Wednesday	1	False	False
20240109	2024-01-09	9	1	2024	Tuesday	1	False	False
20240112	2024-01-12	12	1	2024	Friday	1	False	False
20240115	2024-01-15	15	1	2024	Monday	1	False	False
20240120	2024-01-20	20	1	2024	Saturday	1	False	True
20240121	2024-01-21	21	1	2024	Sunday	1	False	True
20240125	2024-01-25	25	1	2024	Thursday	1	False	False
20240130	2024-01-30	30	1	2024	Tuesday	1	False	False
20240206	2024-02-06	6	2	2024	Tuesday	1	False	False
20240208	2024-02-08	8	2	2024	Thursday	1	False	False

Figure 109. Sample Data for the DateDimension Table

The date dimension table, shown in **Figure 109**, is pre-populated with sample data from the year 2024 to demonstrate its functionality.

2. **CustomerDimension table.** The customer dimension is crucial for generating sales and logistics monitoring reports. It enables the tracking of sales and revenue for each customer and determines which orders were specifically delivered by certain drivers. The primary key of this table is 'CustomerID', which stores data such as the customer's name, type, and their assigned salesperson.

```
%%sql
CREATE TABLE IF NOT EXISTS CustomerDimension (
    CustomerID INTEGER NOT NULL UNIQUE,
    CustomerName VARCHAR(255) NOT NULL,
    CustomerType VARCHAR(50) NOT NULL,
    SalesPersonName VARCHAR(255) NOT NULL
)
DISTSTYLE KEY DISTKEY (CustomerID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 110. SQL Script for Creating the Customer Dimension

```
%sqlcmd columns -t customerdimension
```

name	type	nullable	default	autoincrement	comment	info
customerid	INTEGER	False	None	False	None	{'encode': 'az64'}
customername	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}
customertype	VARCHAR(50)	False	None	False	None	{'encode': 'lzo'}
salespersonname	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}

Figure 111. Table Schema of the CustomerDimension Table

Figure 110 features the SQL script for constructing the customer dimension table, while **Figure 111** depicts its schema. This dimension table, derived from the customer table in the OLTP database, maintains similar constraints and data types but is streamlined for analytics. The distribution style is centered around 'CustomerID' to facilitate the team's querying of data based on specific customers.

customerid	customername	customertype	salespersonname
2	Thompson Finance Inc._1	Branch	Kathleen Campbell
4	Cunningham Mining Corporation_3	Dealer	Laura Rojas
7	Metro Shipping Inc._6	Branch	Theodore Mason
9	OW Construction Corporation_8	Branch	Jordan Martin
10	Northern Equities Inc._9	Direct Customer	Jose Young
16	QPJN Summit Enterprise Corporation_15	Branch	Andrea Brown
18	Prime Enterprise Corporation_17	Branch	Jordan Martin
19	Quad Empire Sun Hotel Inc._18	Dealer	Jennifer Robles
25	MP King Banking Corporation_24	Direct Customer	Brandi Mason
29	Pacific Crown Silver Finance Inc._28	Branch	Laura Rojas

Figure 112. Sample Data for the DateDimension Table

Figure 112 illustrates the synthetic data generated for the date dimension in the OLAP database. The sample data was created using the same method as that for the customer table in the OLTP.

3. **BranchDimension table.** The branch dimension is utilized specifically by the sales monitoring fact table to assess the sales and revenue generated by each branch, providing insights into the performance of individual branches.

```
%%sql
CREATE TABLE IF NOT EXISTS BranchDimension (
    BranchID INTEGER NOT NULL UNIQUE,
    BranchName VARCHAR(255) NOT NULL,
    BranchManager VARCHAR(255) NOT NULL
)
DISTSTYLE KEY DISTKEY (BranchID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 113. SQL Script for Creating the Branch Dimension

```
%sqlcmd columns -t branchdimension
```

name	type	nullable	default	autoincrement	comment	info
branchid	INTEGER	False	None	False	None	{'encode': 'az64'}
branchname	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}
branchmanager	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}

Figure 114. Table Schema of the BranchDimension Table

Figure 113 presents the SQL script used to create the branch dimension table, and **Figure 114** displays its schema. Derived from the OLTP database's branch table, this dimension table retains similar constraints and data types but is optimized for analytical purposes. The distribution style is centered around 'BranchID' to facilitate the team's querying of data based on specific branches.

branchid	branchname	branchmanager
5	Diwata Cagayan	Rebecca King
1	Diwata Marikina	Taylor George
6	Diwata Baguio	Thomas Wallace
3	Diwata Pampanga	Lauren Phillips
8	Pasig City	Sher Hill
2	Diwata QC	Candace Turner
4	Diwata Laguna	Jesse English
7	Quezon City	Cher Howards
9	Makati City	Rex Laylo

Figure 115. Sample Data for the BranchDimension Table

Figure 115 displays synthetic data for the branch dimension in the OLAP database, created using the same methodology as that used for the branch table in the OLTP database.

4. **AreaDimension table.** The area dimension represents the geographical location of customers, independent of branch locations. It facilitates filtering at various geographic levels within the Philippines, specifically by region, province, city, and barangay.

```
%%sql
CREATE TABLE IF NOT EXISTS AreaDimension (
    AreaID INTEGER NOT NULL UNIQUE,
    Region VARCHAR(255) NOT NULL,
    Province VARCHAR(255) NOT NULL,
    City VARCHAR(255) NOT NULL,
    Barangay VARCHAR(255) NOT NULL
)
DISTSTYLE KEY DISTKEY (AreaID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 116. SQL Script for Creating the Area Dimension

```
%sqlcmd columns -t areadimension
```

name	type	nullable	default	autoincrement	comment	info
areaid	INTEGER	False	None	False	None	{'encode': 'az64'}
region	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}
province	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}
city	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}
barangay	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}

Figure 117. Table Schema of the AreaDimension Table

Figure 116 presents the SQL script for creating the area dimension table, and **Figure 117** shows its schema. Originating from the OLTP database's area table, this dimension table maintains similar constraints and data types, tailored for enhanced analytics.

areaid	region	province	city	barangay
5	Metro Manila	NCR	Manila	Barangay 2
15	Metro Manila	NCR	Pasig	Barangay 3
22	Central Luzon	Pampanga	Angeles	Barangay 1
23	Central Luzon	Pampanga	Angeles	Barangay 2
24	Central Luzon	Pampanga	Angeles	Barangay 3
27	Central Luzon	Pampanga	Mabalacat	Barangay 3
28	Central Luzon	Pampanga	San Simon	Barangay 1
33	Central Luzon	Pampanga	Arayat	Barangay 3
35	Central Luzon	Pampanga	Guagua	Barangay 2
39	Central Luzon	Bulacan	Malolos	Barangay 3

Figure 118. Sample Data for the AreaDimension Table

Figure 118 illustrates the synthetic data generated for the area dimension in the OLAP database, using a similar approach to that employed for the area table in the OLTP database.

5. **CylinderTypeDimension table.** This dimension categorizes cylinder sizes to facilitate tracking sales, inventory, and the logistics of

deliveries and pickups. It helps in analyzing the sales, availability, and movements of cylinders of different sizes.

```
%%sql
CREATE TABLE IF NOT EXISTS CylinderTypeDimension (
    CylinderTypeID INTEGER NOT NULL UNIQUE,
    CylinderType VARCHAR(50) NOT NULL,
    CylinderVolume real
)
DISTSTYLE KEY DISTKEY (CylinderTypeID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 119. SQL Script for Creating the CylinderType Dimension

```
%sqlcmd columns -t cylindertypedimension
```

name	type	nullable	default	autoincrement	comment	info
cylindertypeid	INTEGER	False	None	False	None	{'encode': 'az64'}
cylindertype	VARCHAR(50)	False	None	False	None	{'encode': 'lzo'}
cylindervolume	REAL	True	None	False	None	{}

Figure 120. Table Schema of the CylinderTypeDimension Table

Figure 119 displays the SQL script utilized to generate the cylinder type dimension table, while **Figure 120** depicts its structural layout. Derived from the cylinder type table in the OLTP database, this dimension table adheres to analogous constraints and utilizes similar data types, but just optimized for analytical uses.

cylindertypeid	cylindertype	cylindervolume
5	Flask Type	10.0
1	Oversized	60.0
2	Standard	40.0
4	Medium	15.0
3	Bantam	30.0

Figure 121. Sample Data for the AreaDimension Table

Figure 121 illustrates that the composition of the area dimension mirrors that of the area table in the OLTP database.

6. **ProductDimension table.** The product dimension, similar to the cylinder type dimension, helps facilitate sales, inventory, and logistics tracking, but it is filtered based on gas products this time. It can help

determine how much sales is generated per gas product as well their availabilities and movements.

```
%%sql
CREATE TABLE IF NOT EXISTS ProductDimension (
    ProductID INTEGER NOT NULL UNIQUE,
    ProductType VARCHAR(50) NOT NULL,
    ProductCategory VARCHAR(50) NOT NULL
)
DISTSTYLE KEY DISTKEY (ProductID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 122. SQL Script for Creating the Product Dimension

```
%sqlcmd columns -t productdimension
```

name	type	nullable	default	autoincrement	comment	info
productid	INTEGER	False	None	False	None	{'encode': 'az64'}
producttype	VARCHAR(50)	False	None	False	None	{'encode': 'lzo'}
productcategory	VARCHAR(50)	False	None	False	None	{'encode': 'lzo'}

Figure 123. Table Schema of the ProductDimension Table

Figure 122 shows the SQL script used to create the product dimension table, and **Figure 123** outlines its schema. This dimension table, based on the product table from the OLTP database, maintains similar constraints and uses comparable data types, however, it is refined for analytical purposes.

productid	producttype	productcategory
1	Acetylene (C2H2 I.G.)	Industrial
6	Argon (Ar)	Industrial
11	Medical Compressed Air (MC-AIR)	Medical
14	Nitrogen (N2 HP)	Industrial
5	Argon (Ar UHP)	Industrial
15	Nitrogen (N2)	Industrial
2	Acetylene (C2H2)	Industrial
4	Argon (Ar HP)	Industrial

Figure 124. Sample Data for the ProductDimension Table

Figure 124 shows that the composition of the product dimension is the same as the product table in the OLTP database.

7. **PaymentDimension table.** The payment dimension is exclusively utilized to segment the sales fact table. This level of detail is crucial for analyzing sales quantities and revenues, as it allows the finance team to verify all cash and Gcash transactions by examining the sales reports broken down by payment method.

```
%%sql
CREATE TABLE IF NOT EXISTS PaymentDimension (
    PaymentID INTEGER NOT NULL UNIQUE,
    PaymentType VARCHAR(50) NOT NULL,
    PaymentDescription VARCHAR(255) NOT NULL
)
DISTSTYLE KEY DISTKEY (PaymentID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 125. SQL Script for Creating the Payment Dimension

```
%sqlcmd columns -t paymentdimension
```

name	type	nullable	default	autoincrement	comment	info
paymentid	INTEGER	False	None	False	None	{'encode': 'az64'}
paymenttype	VARCHAR(50)	False	None	False	None	{'encode': 'lzo'}
paymentdescription	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}

Figure 126. Table Schema of the PaymentDimension Table

Figure 125 presents the SQL script for constructing the payment dimension table, while **Figure 126** provides a view of its schema. Originating from the payment table in the OLTP database, this dimension table adheres to the same constraints and utilizes similar data types but is optimized for analysis.

paymentid	paymenttype	paymentdescription
1	Mobile Wallet	Payments made through mobile wallet apps.
6	Debit Card	Payments made through debit card
5	Credit Card	Payments made through credit card
2	Cash	Direct cash transactions.
4	GCash	Payments made through GCash
3	Bank Transfer	Direct transfers from bank accounts.

Figure 127. Sample Data for the PaymentDimension Table

Figure 127 illustrates that the structure of the payment dimension mirrors that of the payment table in the OLTP database.

8. **DriverDimension table.** The driver dimension is vital for the logistics fact table, as it tracks the number of deliveries and pick-ups each driver completes, which is essential for determining incentives. This dimension stores details such as each driver's name, license number, hire date, and restriction numbers, all linked to the primary key, 'DriverID'.

```
%%sql
CREATE TABLE IF NOT EXISTS DriverDimension (
    DriverID INTEGER NOT NULL UNIQUE,
    DriverName VARCHAR(255) NOT NULL,
    HireDate DATE NOT NULL,
    LicenseNo VARCHAR(50) NOT NULL,
    RestrictionNo VARCHAR(2) NOT NULL
)
DISTSTYLE KEY DISTKEY (DriverID)
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Figure 128. SQL Script for Creating the Driver Dimension

```
%sqlcmd columns -t driverdimension
```

name	type	nullable	default	autoincrement	comment	info
driverid	INTEGER	False	None	False	None	{'encode': 'az64'}
drivername	VARCHAR(255)	False	None	False	None	{'encode': 'lzo'}
hiredate	DATE	False	None	False	None	{'encode': 'az64'}
licenseno	VARCHAR(50)	False	None	False	None	{'encode': 'lzo'}
restrictionno	VARCHAR(2)	False	None	False	None	{'encode': 'lzo'}

Figure 129. Table Schema of the DriverDimension Table

Figure 128 displays the SQL script used to establish the driver dimension table, and **Figure 129** shows its schema. This dimension table, derived from the driver table in the OLTP, retains the original constraints and data types, while being tailored for analytical efficiency.

driverid	drivername	hiredate	licenseno	restrictionno
1	JJ Ramoso	2023-05-08	N83-61-180847	CE
6	Robert Odonnell	2015-05-22	N28-11-684899	CE
11	Matthew Rasmussen	2014-07-11	N30-24-994145	C
14	Ashley West	2012-06-20	N29-57-677042	BE
20	Whitney Brown	2020-11-24	N30-00-999861	C
32	Tyler Wilson	2011-05-12	N81-86-211766	CE
34	Jonathan Taylor	2008-01-26	N52-56-393215	CE
36	William Perry	2017-06-22	N85-48-668983	D
5	Reginald Lewis	2021-06-23	N67-55-496758	C
15	Amanda Arnold	2011-08-21	N61-34-982928	B2

Figure 130. Sample Data for the DriverDimension Table

Figure 130 depicts that the layout of the payment dimension is identical to the structure found in the payment table of the OLTP database.

Daily Sales Monitoring Report

The function of this OLAP database, as detailed in the design considerations table, is to track the company's sales performance daily by measuring both quantity sold and revenue generated.

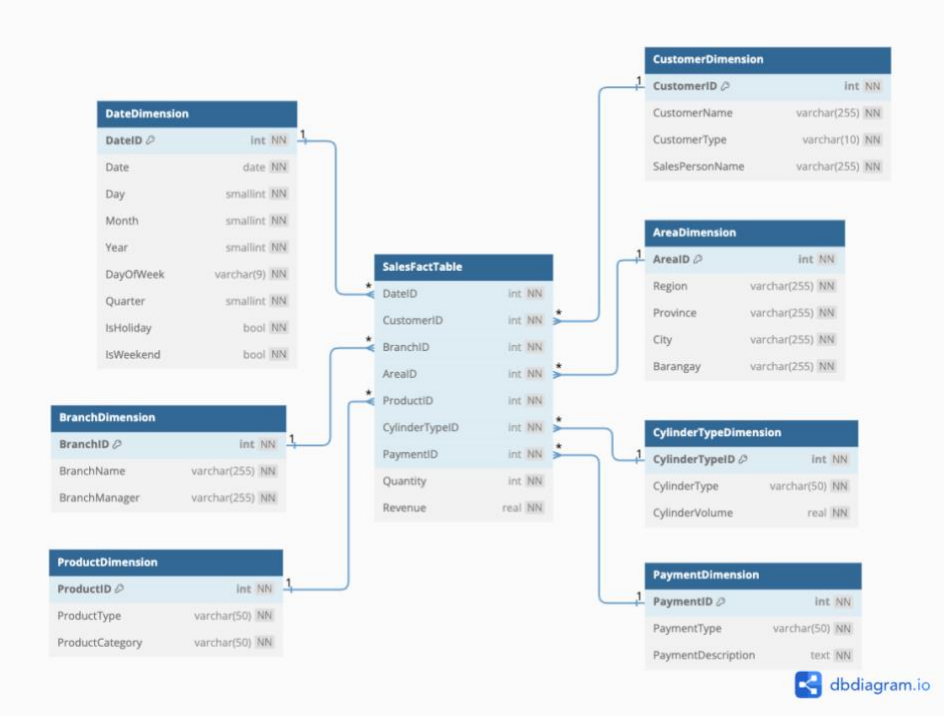


Figure 131. Schema for the Daily Sales Monitoring OLAP Database

To provide a visual understanding, **Figure 131** offers a schema of the fact table along with its associated dimensions and their interconnections. The grain of the table is defined by its purpose, with each row representing daily sales and revenue for each distinct combination of dimensions. For tracking sales, the table incorporates seven dimensions: date, customer, branch, area, cylinder type, product, and payment methods.

Sales Fact Table

DateID	CustomerID	BranchID	AreaID	ProductID	CylinderTypeID	PaymentID	Quantity	Revenue
20240104	14	5	148	7	2	1	78	79443.0
20240112	48	4	80	8	4	1	108	108056.0
20240113	78	5	146	2	4	3	95	140141.0

DIMENSIONS FACTS

Figure 132. Sales Fact Table Sample with Synthetic Data

The first seven columns of the sales fact table, depicted in **Figure 132**, serve as primary keys and are also foreign keys linked to various dimensions. These columns enable the filtering of data and the seamless merging of pertinent details from the dimension tables, thereby streamlining analytics. The data itself quantifies the sales in terms of quantity and revenue for a set of dimensions. For instance, the first-row details that 'CustomerID' 14, from 'AreaID' 148, bought 78 cylinders for a total of Php 79,443 of 'ProductID' 7 and 'CylinderTypeID' 2, purchased from 'BranchID' 5 on January 4, 2024.

For the SQL scripts used to generate the fact table, along with its schema and sample data, **please see Appendices 1 through 3**.

Daily Inventory Monitoring Report

The OLAP database dedicated to inventory monitoring offers regular analytics on the availability and unavailability of the company's cylinders.

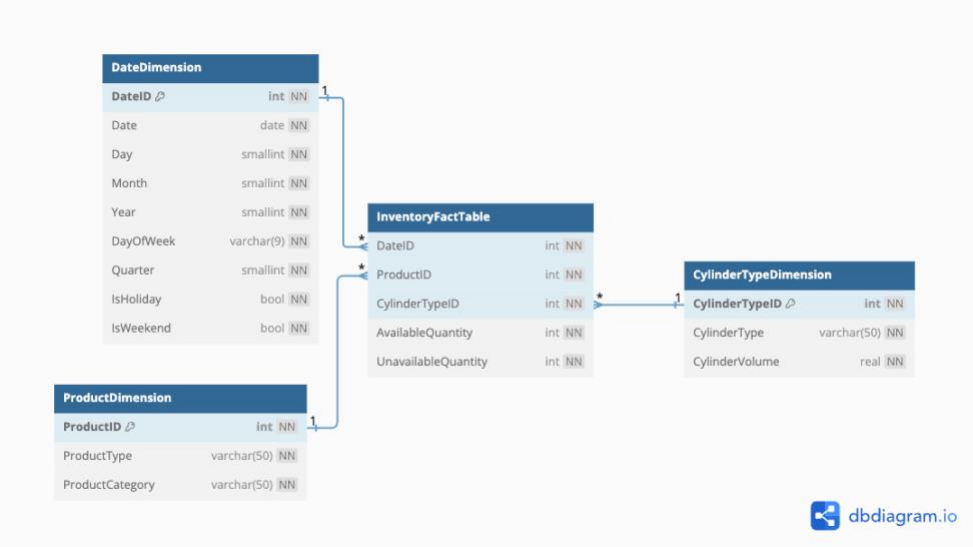


Figure 133. Schema for the Daily Inventory Monitoring OLAP Database

Figure 133 displays the schema of the inventory monitoring OLAP database. It features three dimension tables connected to the fact table: the date dimension, product dimension, and cylinder type dimension. The date dimension facilitates periodic filtering, whereas the product and cylinder type dimensions enable filtering of additive facts, specifically the available and unavailable quantities for different products and cylinder types.

Inventory Fact Table

DateID	ProductID	CylinderTypeID	AvailableQty	UnavailableQty
20240102	9	2	277	110
20240103	4	5	169	67
20240107	15	1	91	36

DIMENSIONS FACTS

Figure 134. Inventory Fact Table Sample with Synthetic Data

Exploring further into the inventory fact table, the initial three columns consist of dimensions, and the subsequent two columns record the facts. To illustrate the grain of the fact table using the first row as an example: on January 2, 2024, there were 277 available and 110 unavailable cylinders for ‘ProductID’ 9 and ‘CylinderTypeID’ 2.

For the SQL scripts used to generate the fact table, along with its schema and sample data, please see **Appendices 4 through 6.**

Daily Logistics Monitoring Report

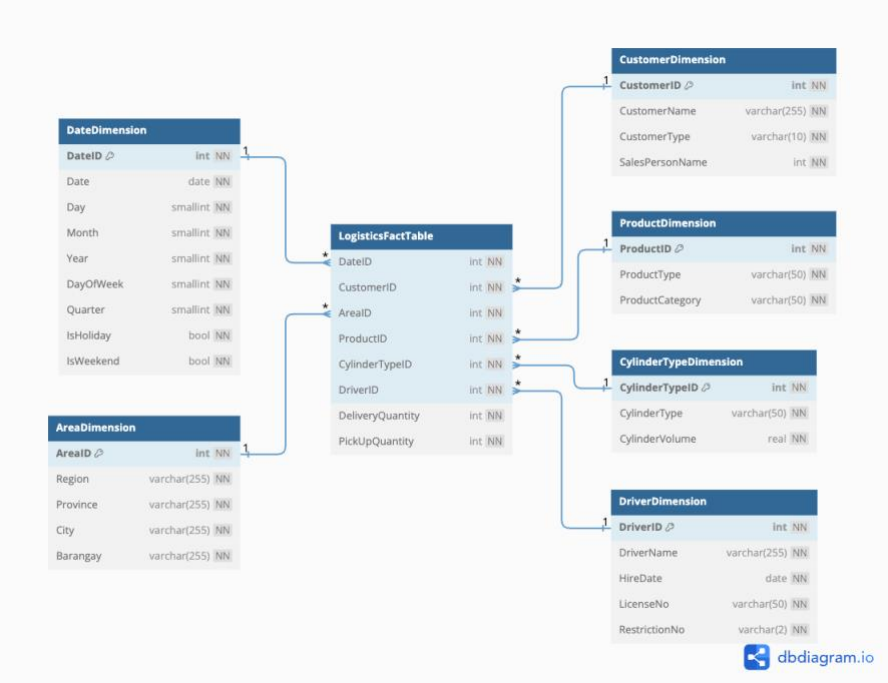


Figure 135. Schema for the Daily Logistics Monitoring OLAP Database

The schema for the logistics monitoring fact table, shown in Figure 135, includes seven dimensions: date, customer, area, product, cylinder type, and driver. These dimensions are used to filter and aggregate the additive facts, which are the quantities of deliveries and pickups.

Logistics Fact Table

DateID	CustomerID	AreaID	ProductID	CylinderTypeID	DriverID	DeliveryQuantity	PickUpQuantity
20240104	68	116	5	2	2	261	115
20240104	56	61	8	3	16	271	94
20240104	22	83	11	1	17	155	114

DIMENSIONS FACTS

Figure 136. Logistics Fact Table Sample with Synthetic Data

In the sample logistics fact table shown in Figure X, the first six columns represent dimensions and the last two columns represent facts. To explain the granularity using the first row as an example: on January 4, 2024, 'DriverID' 2 made 261 deliveries and 115 pick-ups for 'ProductID' 5 and 'CylinderTypeID' 2 to 'CustomerID' 68 located in 'AreaID' 116.

For the SQL scripts used to generate the fact table, along with its schema and sample data, please see **Appendices 7 through 9.**

Appendix 10 confirms the successful creation of all dimensions and fact tables in the database, as evidenced by the output of the JupySQL command '\dt'.

4.6 NoSQL Database: DynamoDB

The team used DynamoDB over other NoSQL databases due to it being fully managed by AWS, its key-value storage which fits the use case of finding complaints by keys such as serial number and complaint ID, and its application for use cases not involving complex queries such as in complaints management (which heavily involves only querying the details of the complaints themselves, without involving any other dimensions) (Adams, 2022).

Each component of the Dynamo DB schema ensures that the data is stored and retrieved efficiently while also maintaining the flexibility required by the company in managing customer complaints and resolution.

SerialNo *(String)*	ComplaintID *(Number)*	ChiefComplaints	CustomerContactNumber	CustomerNa...	CustomerReviews
SDFSDF222	1	{"Gauge"}	09519632198	PM See	Hore
SDFSDF222	2	{"Odor"}	09519632198	PM See	Hore
SDFSDF222	3	{"Odor"}	09519632198	PM See	Hore
SDAKSL3012A	1	{"Gauge","Odor"}	09123456892	Christian Alis	There is an unusua...
SDAKSL3012A	2	{"Clogged","Unusu...	09119632198	Rex Gregor M L...	There is an unusua...
SDAKSL3012A	3	{"Gauge","Odor"}	09123456892	Christian Alis	There is an unusua...
SDAKSL3012A	4	{"Clogged","Unusu...	09119632198	Rex Gregor M L...	There is an unusua...
SDAKSL3012A	5	{"Gauge","Odor"}	09123456892	Christian Alis	There is an unusua...
SDAKSL3012A	6	{"Clogged","Unusu...	09119632198	Rex Gregor M L...	There is an unusua...
SDAKSL3012A	7	{"Gauge","Odor"}	09123456892	Christian Alis	There is an unusua...
SDAKSL3012A	8	{"Clogged","Unusu...	09119632198	Rex Gregor M L...	There is an unusua...
SDAKSL3012A	9	{"Gauge","Odor"}	09123456892	Christian Alis	There is an unusua...

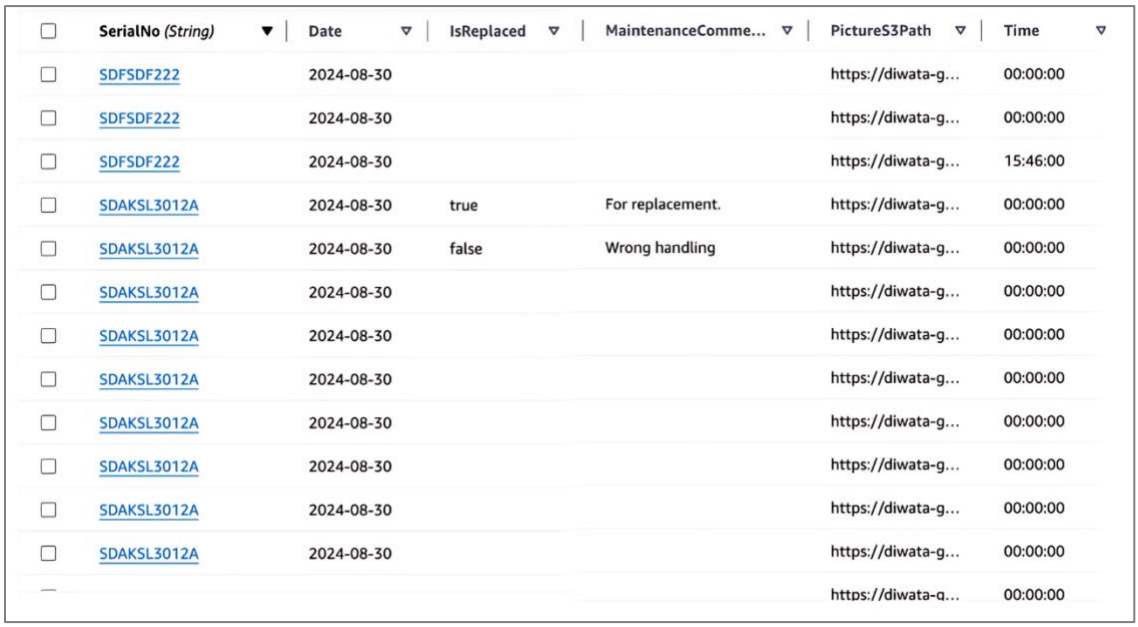
Figure 137. Dynamo DB Partition Key (SerialNo), Sort Key (ComplaintID), and Attributes

Partition Key : SerialNo

SerialNo, a unique string identifier for each cylinder tank, serves as the partition key to group all related complaints. This choice enables efficient and scalable querying of complaints for a specific cylinder tank, utilizing DynamoDB's partitioning to ensure fast access.

Sort Key: ComplaintID

The ComplaintID, a unique numeric identifier for each complaint, serves as the sort key in the DynamoDB schema. This design allows for efficient sorting and retrieval of complaints related to the same SerialNo in chronological order. It makes it easier to track the history of issues with cylinder tanks, aiding in diagnosing whether a new complaint is connected to a previous one. This approach supports faster and more accurate troubleshooting for products like cylinder tanks.



SerialNo *(String)*	Date	IsReplaced	MaintenanceComme...	PictureS3Path	Time
SDFSDF222	2024-08-30			https://diwata-g...	00:00:00
SDFSDF222	2024-08-30			https://diwata-g...	00:00:00
SDFSDF222	2024-08-30			https://diwata-g...	15:46:00
SDAKSL3012A	2024-08-30	true	For replacement.	https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30	false	Wrong handling	https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30			https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30			https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30			https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30			https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30			https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30			https://diwata-g...	00:00:00
SDAKSL3012A	2024-08-30			https://diwata-g...	00:00:00

Figure 138. Dynamo DB Partition Key, Local Secondary Index(Date), and Attributes

Local Secondary Index: Date

The Date-index enhances the schema by enabling efficient sorting and filtering of complaints based on the Date attribute. While the primary table uses SerialNo and ComplaintID, the Date-index allows querying by SerialNo and retrieving complaints in date order. This is especially useful for identifying issues within specific time frames. The Date-index adds flexibility for time-based queries, enabling faster analysis of complaint patterns and trends to improve operational efficiency and troubleshooting.

Chief Complaints

This attribute captures the primary issues reported by customers concerning the cylinder tank, such as problems with the gauge, unusual odors, or blockages. Grouping these complaints in one field enables efficient tracking of recurring issues, which in turn helps the support and maintenance teams address common problems swiftly.

CustomerContactNumber

To facilitate communication, the *CustomerContactNumber* attribute stores the phone number of the customer who reported the complaint. This ensures that the support team can easily reach out for further clarification or updates, streamlining the resolution process.

CustomerName

This attribute records the name of the person reporting the issue. This not only helps maintain personalized customer interactions but also aids in identifying recurring customers, which may signal unresolved or long-term product concerns.

CustomerReviews

This attribute provides description of the complaint, offering insight into customer sentiment and expectations. This feedback is valuable in understanding how the issue impacted overall satisfaction and contributes to product improvement and customer retention strategies.

IsReplaced

The IsReplaced attribute documents whether the cylinder tank or its parts were replaced as part of the resolution process. This information is needed to track the cost of replacements and ensuring that recurrent issues are addressed effectively by providing new components, thus avoiding frequent breakdowns of the same cylinder.

MaintenanceComments

During inspections, the maintenance team records their observations in the *MaintenanceComments* attribute. This helps with the technical diagnosis of the issue and provides a reference for future maintenance cases, helping to streamline the process for similar complaints.

PictureS3Path

Visual documentation of the issue is stored in the *Picture S3 Path* attribute, which holds links to images of the cylinder tank or its malfunction. These visuals assist both the support and maintenance teams in understanding the extent of the problem, facilitating faster diagnosis and resolution.

Time

Lastly, the *Time* attribute logs when the complaint was reported, enabling chronological tracking of the issue. This is essential for identifying complaint patterns, scheduling timely repairs, and ensuring prompt issue resolution. Combined with a date-index, it also allows for filtering complaints by specific time periods, further enhancing troubleshooting and analysis.

4.7 API Endpoints

In creating the API endpoints, AWS Lambda was used for hosting the functions of the APIs, while AWS API Gateway was used for exposing the AWS Lambda function, allowing for communication between the users and the databases and data lake. AWS Lambda was also used because of its serverless nature wherein additional compute resources may be allocated automatically when needed and monitoring and logging of the function usage may also be handled by the service itself (Amazon Web Services, 2024). On the other hand, AWS API Gateway was also used to easily create an API endpoint and set this endpoint as a trigger for the AWS Lambda function and to allow convenient setting of security measures through its AWS Web Application Firewall (WAF) integration which protects against web exploits, such as SQL injections (Amazon Web Services, 2024; Amazon Web Services, 2024).

In setting up the API endpoints, the following are considered:

- The Representational State Transfer (REST) protocol was used instead of an HTTP or SOAP protocol due to its scalability owing to its stateless nature (where servers can easily run faster due to not saving any state from one API call) and flexibility in its response's data structure which can be either JSON or XML similar to HTTP (as compared to SOAP which is only restricted to XML) (Alam-Naylor & Fateh, 2024; Marshall, 2024).
- HTTP methods for REST APIs include Get, Post, Put, and Delete which corresponds to the Create, Read, Update, and Delete (CRUD) operations, and for these APIs, all methods are used, wherein the Post method serves as the primary method for all APIs, which can still be further improved with a future iteration of the APIs, due to the inherent lack of caching with the Post method (which would serve as a problem for read requests) (W3Schools, n.d.).
- Lastly, while there is an inherent API key functionality in AWS API Gateway, the team decided instead to add an extra field in the request body of a user's API call, which would contain the API key. Given that this may lead to issues such as adversaries attempting to overwhelm the server by issuing several requests regardless if the API key is correct or not (due to AWS Lambda still processing these requests given that it still reads whether the API key is correct or not), further improvements regarding this issue may be done for the future iterations of the API endpoints.

Given these, the following endpoints were created:

- Diwata Gas DynamoDB API
- Diwata Gas S3 API
- Diwata Gas OLAP Query API

General Lambda Handler for All API Endpoints

```
def lambda_handler(event, context):
    '''Provide an event that contains the following keys:
      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
        operation being performed
    '''

    with open('api_key.txt') as f:
        api_key = f.read().strip()

    try:
        if event['api_key'] == api_key:
            pass
        else:
            return {
                'statusCode': 400,
                'body': 'Wrong API Key input'
            }
    except Exception:
        return {
                'statusCode': 400,
                'body': 'No API Key input'
            }

    operation = event['operation']
    try:
        payload = event['payload']
    except Exception:
        pass

    if operation in operations:
        try:
            return operations[operation](payload)
        except Exception:
            return operations[operation]()

    else:
        raise ValueError(f'Unrecognized operation "{operation}"')
```

Figure 139. Main Lambda Function (Similar for All API Endpoints)

As seen in **Figure 139**, in the main Lambda function of each API endpoint, there are conditional checks for the user's API key and given a list of operations, operations which either needs API payloads or not are done. Moreover, the function returns a ValueError if it does not recognize the operation in the request body of the user's API call.

Diwata Gas DynamoDB API

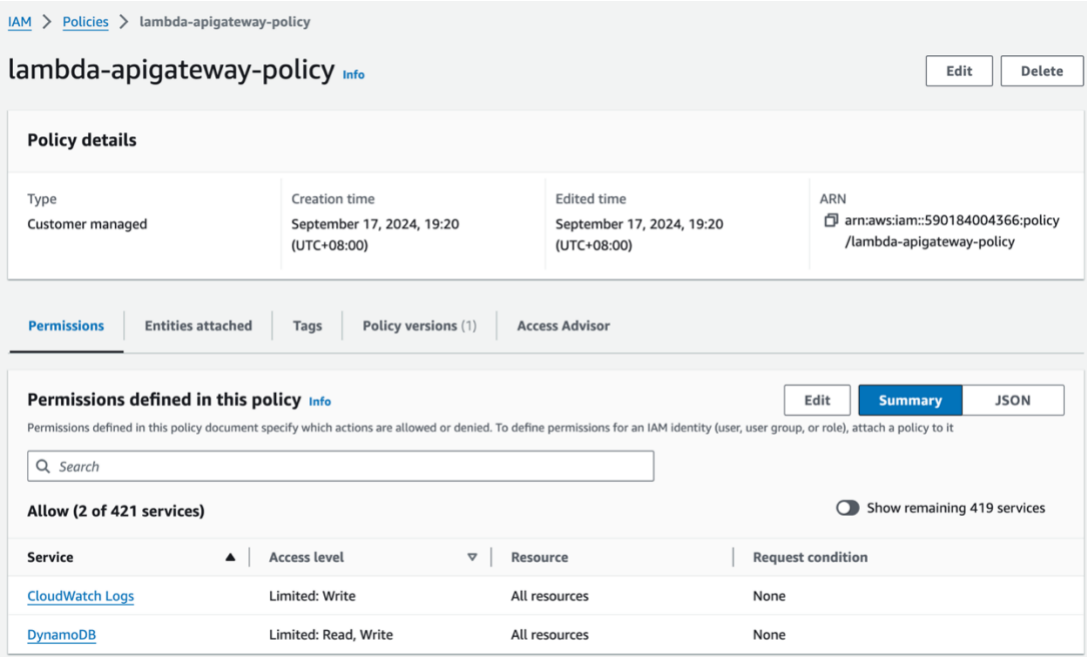


Figure 140. IAM Policy for Diwata Gas DynamoDB API

In creating the DynamoDB API, as seen in **Figure 140,** an IAM policy is created which (1) provides DynamoDB read and write access and (2) CloudWatch Logs write access (for the purpose of API metric logging, a requirement by AWS API Gateway). This was then placed inside an IAM role and attached to the AWS Lambda function for the API.

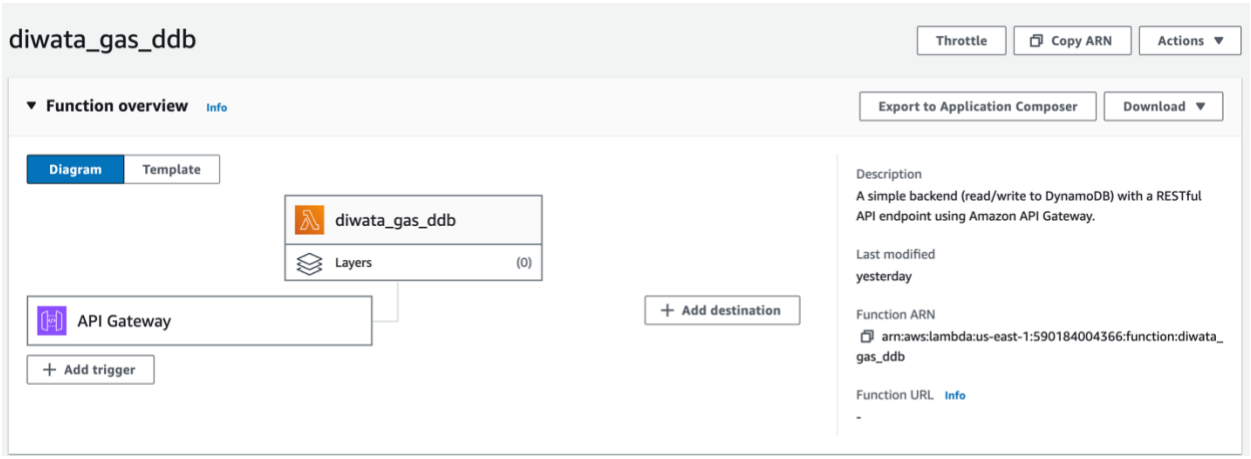


Figure 141. Diwata Gas DynamoDB Function Overview

```python
import boto3
import json
from decimal import Decimal

# Define the DynamoDB table that Lambda will connect to
table_name = "DiwataGasComplaints"

# Create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(table_name)
```

Figure 142. Imported Libraries and Other Misc. Variables

As seen in **Figure 141**, the Lambda function is triggered every time the API endpoint, through the AWS API Gateway, is called. Moreover, the boto3, json, and decimal libraries are imported for the function, and for the boto3 library, it is used to access the DynamoDB table as seen above.

```python
# Set Encoder
class SetEncoder(json.JSONEncoder):
    """
    A class module based from json.JSONEncoder
    named SetEncoder that encodes Python objects into
    JSON-serializable objects.

    Methods
    -------
    default(obj):
        Return JSON-serializable counterpart of Python object.
    """
    def default(self, obj):
        """Return JSON-serializable counterpart of Python object.

        Parameters
        ----------
        obj
            Python object

        Returns
        -------
        JSON-serializable object
        """
        if isinstance(obj, set):
            return list(obj)
        elif isinstance(obj, Decimal):
            return int(obj)
        return json.JSONEncoder.default(self, obj)
```

Figure 143. JSON Encoder for Non-JSON Serializable Python Objects

As seen in **Figure 143,** given that sets and Decimal objects which may found within the items returned by DynamoDB may not be serialized into JSON objects, an encoder class is created to handle these issues, which then convert these objects into JSON-serializable objects.

```
operations = {
    'scan': scan,
    'find': find,
    'query_item': query_item,
    'echo': echo,
}
```

Figure 144. DynamoDB API Operations

The operations involved in this API are the scan, find, and query item operations. The echo operation simply returns the payload given by the API caller.

```
# Define some functions to perform the operations
def scan():
    """Scan DynamoDB table.

    Returns
    -------
    dict
        Dict containing list of items
    """
    return json.loads(json.dumps(dynamo.scan()['Items'], cls=SetEncoder))
```

Figure 145. Scan Operation

As seen in **Figure 145,** in the scan operation, the DynamoDB table is scanned for all its items and encoded if needed using the created encoder. These are then loaded again into a JSON object, to be returned to the API caller.

```
def find(payload):
    """Find specific item using key-pair.

    Parameters
    ----------
    payload : dict
        API Object

    Returns
    -------
    dict
        Dict containing details of specific item
    """
    return json.loads(json.dumps(dynamo.get_item(Key=payload['Key']),
        cls=SetEncoder))
```

Figure 146. Find Operation

The find operation is used to find a specific item given the serial number and the complaint ID or the date. The key-pair is extracted from the payload input by the API caller.

```python
def query_item(payload):
    """Query specific detail about a cylinder.

    Parameters
    ----------
    payload : dict
        API Object

    Returns
    -------
    response : dict
        Dict containing list of a specific detail about a cylinder
    """
    if payload['Item'] == 'Date':
        return json.loads(json.dumps(dynamo.query(
            KeyConditionExpression='SerialNo = :serialno',
            ExpressionAttributeNames={'#D': 'Date'},
            ExpressionAttributeValues={':serialno': payload['SerialNo']},
            ProjectionExpression='#D'), cls=SetEncoder))
    elif payload['Item'] == 'Time':
        return json.loads(json.dumps(dynamo.query(
            KeyConditionExpression='SerialNo = :serialno',
            ExpressionAttributeNames={'#T': 'Time'},
            ExpressionAttributeValues={':serialno': payload['SerialNo']},
            ProjectionExpression='#T'), cls=SetEncoder))
    else:
        return json.loads(json.dumps(dynamo.query(
            KeyConditionExpression='SerialNo = :serialno',
            ExpressionAttributeValues={':serialno': payload['SerialNo']},
            ProjectionExpression=payload['Item'])['Items'], cls=SetEncoder))
```

Figure 147. Query Item Operation

By inputting the serial number and the desired information (such as date, time, e.g.), the API caller may be able to extract a list of specific information about a cylinder. Moreover, several conditionals are created to account for reserved words such as “Date” and “Time”, which requires specific handling in terms of the structure of their queries, where they are given placeholder names in the queries.

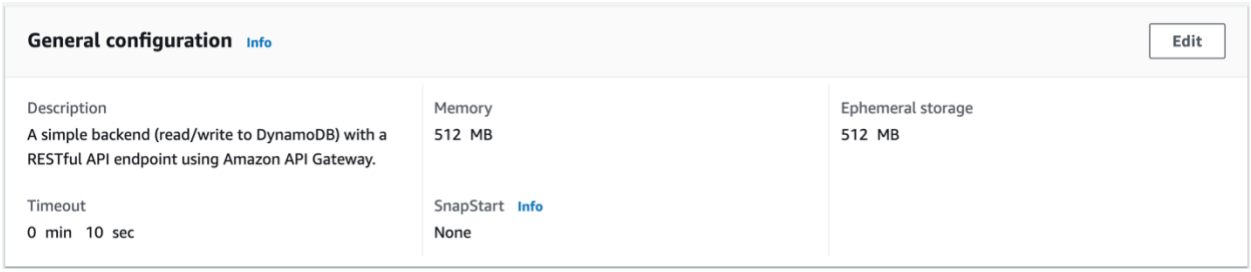


Figure 148. General Configuration of DynamoDB Function

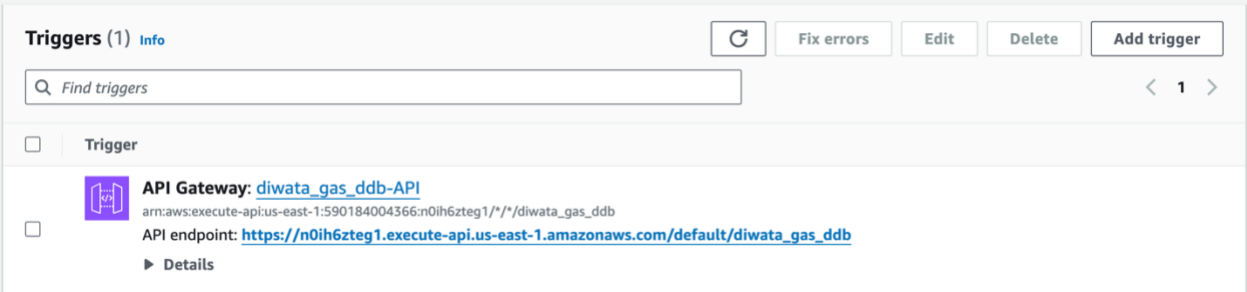


Figure 149. API Endpoint of DynamoDB API

The timeout is only ten seconds given that the queries immediately return results to the users, while the set memory is at 512 megabytes to handle possible API call spikes. Lastly, the API endpoint is shown above, which can be used to do all the read operations stated above.

Diwata Gas S3 API

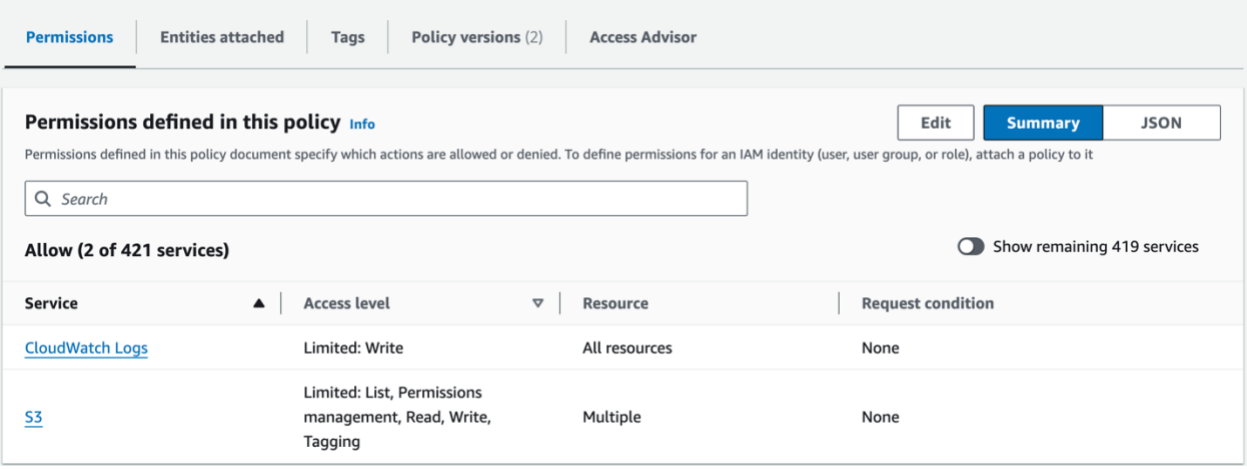


Figure 150. IAM Policy for Diwata Gas S3 API

As seen above, aside from the CloudWatch logging, S3 permissions including listing, permission management, reading, writing, and tagging are also included in the policy, which allows for extensive functionalities for the S3 API.

Figure 151. Diwata Gas S3 Function Overview

```
import json
import re
import boto3
from botocore.exceptions import ClientError
```

Figure 152. Imported Libraries for Diwata Gas S3 Function

Similarly, the json and boto3 libraries are imported, alongside the boto3's exceptions, which is the ClientError (which is used for the download operations of the S3 API). Also, the regex library is also imported to list file paths under specific data lake zones such as the work and gold zones.

```
    global s3
    s3 = boto3.resource('s3')
    global s3_2
    s3_2 = boto3.client('s3')

    global bucket
    bucket = s3.Bucket('diwata-gas-bucket')
```

Figure 153. Boto3 S3 Resource, Client, and Bucket

Using boto3, the following are used to access the S3 bucket and to create specific calls on the bucket using boto3's S3 client.

```
operations = {
    'list_gold': list_gold,
    'list_work': list_work,
    'download_gold': download_gold,
    'download_work': download_work,
    'echo': echo,
}
```

Figure 154. S3 API Operations

Using the S3 API, the user may list all the files in the gold and work zone and download files in the same zones.

```
# Define some functions to perform the operations
def list_gold():
    """List gold files.

    Returns
    -------
    dict_gold : dict
        Dict containing list of gold files
    """
    dict_gold = {}
    dict_gold['Items'] = []
    for obj in bucket.objects.all():
        if re.match(r"gold/.+", obj.key):
            dict_gold['Items'].append(obj.key)
    return dict_gold

def list_work():
    """List work files.

    Returns
    -------
    dict_work : dict
        Dict containing list of work files
    """
    dict_work = {}
    dict_work['Items'] = []
    for obj in bucket.objects.all():
        if re.match(r"work/.+", obj.key):
            dict_work['Items'].append(obj.key)
    return dict_work
```

Figure 155. List Gold and List Work Operations

In both operations, using a regex logic which looks for the specific term of the zones, one may be able to find the list of objects within both zones.

```
def download_work(payload):
    """Provide presigned URL for S3 work object.

    Parameters
    ----------
    payload : dict
        API Payload

    Returns
    -------
    response : dict
        Dict containing presigned URL for S3 work object or error message
    """
    if payload['table'] == 'customer_order':
        try:
            response = s3_2.generate_presigned_url('get_object',
                        Params={'Bucket': 'diwata-gas-bucket',
                        'Key': f'work/CustomerOrder_{payload['month'].title()}_{payload['year']}.parquet'},
                        ExpiresIn=3600)
            return {
                'statusCode': 200,
                'body': response
            }
        except ClientError as e:
            return {
                'statusCode': 400,
                'body': json.dumps('Downloading failed...')
            }
```

```
    elif payload['table'] == 'logistics':
        try:
            response = s3_2.generate_presigned_url('get_object',
                        Params={'Bucket': 'diwata-gas-bucket',
                        'Key': f'work/Logistics_{payload['month'].title()}_{payload['year']}.parquet'},
                        ExpiresIn=3600)
            return {
                'statusCode': 200,
                'body': response
            }
        except ClientError as e:
            return {
                'statusCode': 400,
                'body': json.dumps('Downloading failed...')
            }
```

Figure 156. Download Work Operation

Using the download work operation, one can refer to either table, customer order and logistics, and get a pre-signed download URL which provides the user access to the file and expires after 3,600 seconds (1 hour), referring to the document with the specific month and date in the payload (AWS Boto3, 2024). In case of a ClientError, the user will receive a 400-status response with the message, “Downloading failed...”. In case the call is successful, the user will receive a status code of 200 and the pre-signed download URL.

```python
def download_gold(payload):
    """Provide presigned URL for S3 gold object

    Parameters
    ----------
    payload : dict
        API Payload

    Returns
    -------
    dict
        Dict containing presigned URL for S3 gold object or error message
    """
    if payload['fact_table'] == 'logistics':
        try:
            response = s3_2.generate_presigned_url('get_object',
                        Params={'Bucket': 'diwata-gas-bucket',
                        'Key': f'gold/LogisticsFactTable_{payload['month'].title()}_{payload['year']}.csv'},
                        ExpiresIn=3600)
            return {
                'statusCode': 200,
                'body': response
            }
        except ClientError as e:
            return {
                'statusCode': 400,
                'body': json.dumps('Downloading failed...')
            }

    elif payload['fact_table'] == 'sales':
        try:
            response = s3_2.generate_presigned_url('get_object',
                        Params={'Bucket': 'diwata-gas-bucket',
                        'Key': f'gold/SalesFactTable_{payload['month'].title()}_{payload['year']}.csv'},
                        ExpiresIn=3600)
            return {
                'statusCode': 200,
                'body': response
            }
        except ClientError as e:
            return {
                'statusCode': 400,
                'body': json.dumps('Downloading failed...')
            }
    elif payload['fact_table'] == 'inventory':
        try:
            response = s3_2.generate_presigned_url('get_object',
                        Params={'Bucket': 'diwata-gas-bucket',
                        'Key': f'gold/InventoryFactTable_{payload['month'].title()}_{payload['year']}.csv'},
                        ExpiresIn=3600)
            return {
                'statusCode': 200,
                'body': response
            }
        except ClientError as e:
            return {
                'statusCode': 400,
                'body': json.dumps('Downloading failed...')
            }
    else:
        return {
                'statusCode': 400,
                'body': json.dumps('No such file...')
            }
```

Figure 157. Download Gold Operation

The same logic as the download work operation can be found for the download gold operation, except the files being downloaded in this case would be fact tables (sales or inventory) from the gold zone.

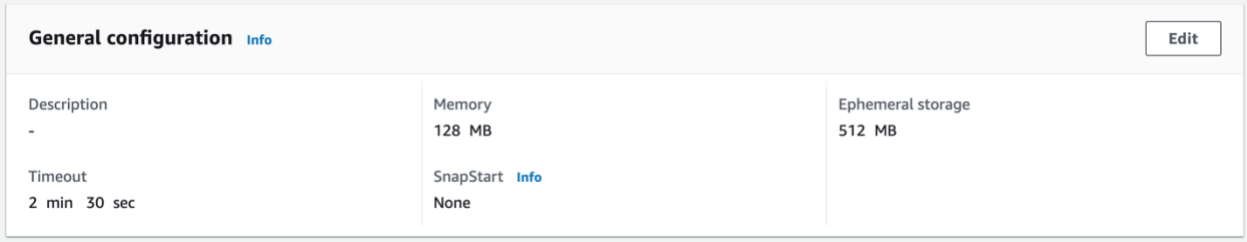


Figure 158. General Configuration of S3 API

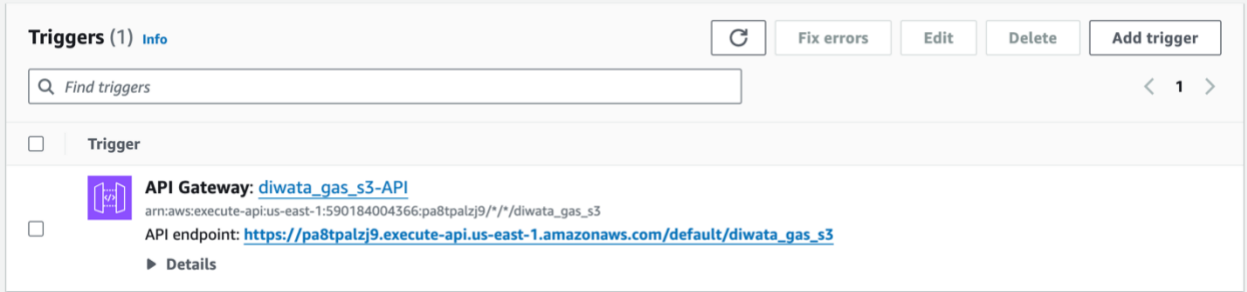


Figure 159. API Endpoint of S3 API

The timeout is set at 2 minutes and 30 seconds to account for long waiting times, especially with the creation of the pre-signed download URLs.

Diwata Gas OLAP Query API

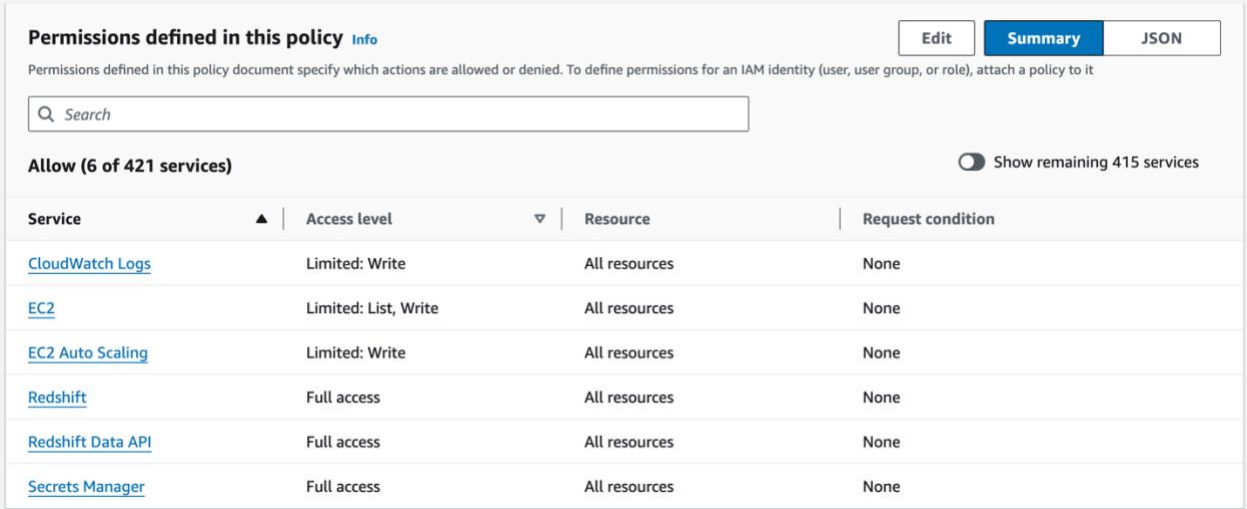


Figure 160. One of the IAM Policies of OLAP Query API

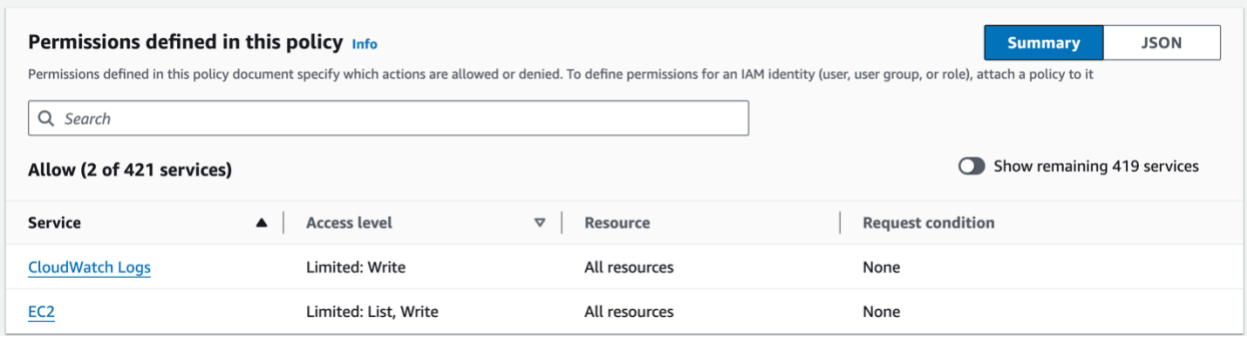


Figure 161. Other IAM Policy of OLAP Query API

Aside from the CloudWatch logging writing permission, full access is given to Redshift, Redshift Data API, and Secrets Manager, to easily handle calls to the Redshift API (provision of cluster credentials) and Redshift Data API (queries to Redshift clusters), and to handle secrets, which would serve as the function's credentials for its calls to the Redshift Data API. Lastly, permissions for EC2 and EC2 Autoscaling were added to account for within-VPC interactions between AWS Lambda and AWS Redshift.

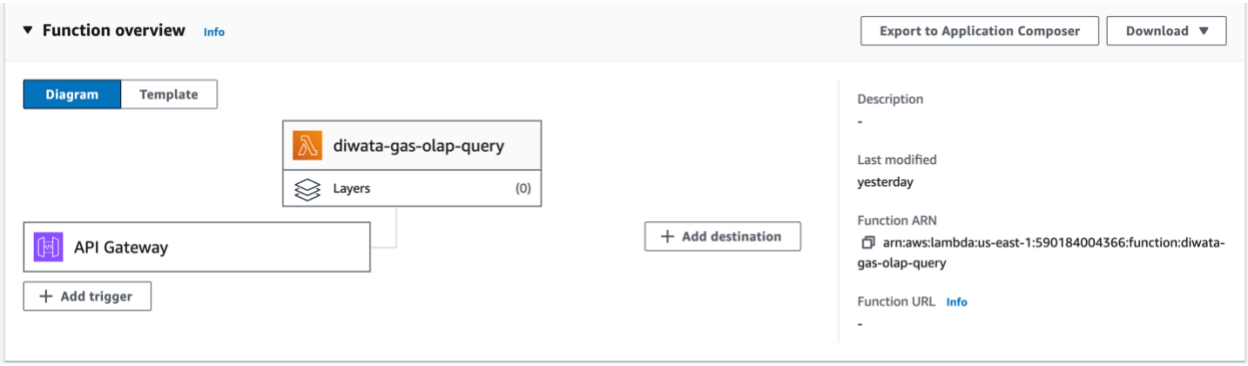


Figure 162. Function Overview of Diwata Gas OLAP Query API

```
import json
import boto3
import psycopg2
from decimal import Decimal
```

Figure 163. Imported Libraries

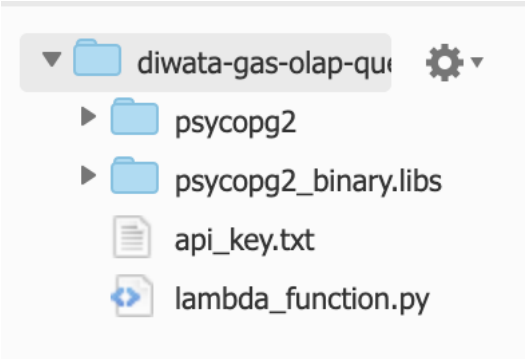


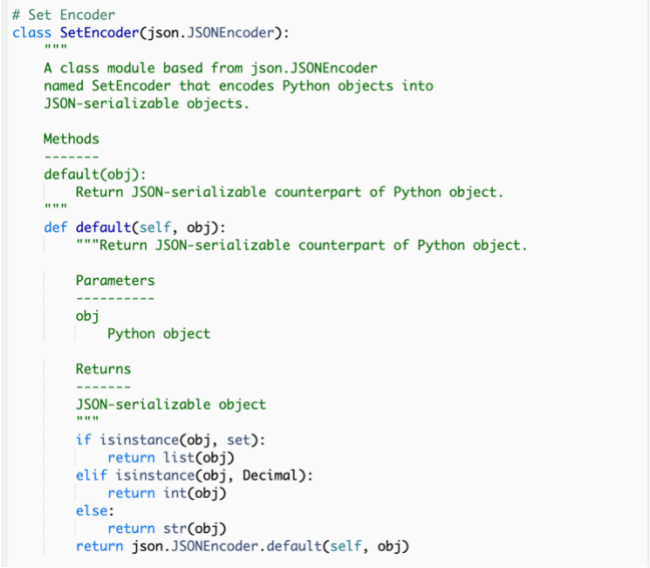
Figure 164. Folder of Diwata Gas OLAP Query Function

```
    global client
    client = boto3.client('redshift-data', region_name="us-east-1")

    client_2 = boto3.client('redshift', region_name="us-east-1")
    global cluster_creds
    cluster_creds = client_2.get_cluster_credentials(DbUser="jj",
                    DbName="diwatagas_olap_db",
                    ClusterIdentifier="diwata-gas-olap",
                    AutoCreate=False)
```

Figure 165. Redshift and Redshift Data Clients and Generated Redshift Cluster Credentials

Aside from the usual libraries imported in the Lambda function, the psycopg2 library is also imported, using an external folder which contains the library. This is because the Lambda service does not have psycopg2 within its repository of libraries or modules. Moreover, two clients are created for both the Redshift clusters and the Redshift Data API, and using the Redshift client, temporary cluster credentials are generated for the function.



```
# Set Encoder
class SetEncoder(json.JSONEncoder):
    """
    A class module based from json.JSONEncoder
    named SetEncoder that encodes Python objects into
    JSON-serializable objects.

    Methods
    -------
    default(obj):
        Return JSON-serializable counterpart of Python object.
    """
    def default(self, obj):
        """Return JSON-serializable counterpart of Python object.

        Parameters
        ----------
        obj
            Python object

        Returns
        -------
        JSON-serializable object
        """
        if isinstance(obj, set):
            return list(obj)
        elif isinstance(obj, Decimal):
            return int(obj)
        else:
            return str(obj)
        return json.JSONEncoder.default(self, obj)
```

Figure 166. JSON Encoder for Non-JSON Serializable Python Objects

As seen in **Figure 166**, aside from the sets and Decimal objects, in this function, the other non-serializable Python objects are just returned as strings, to streamline the JSON encoding.

```
operations = {
    'list_facttables': list_facttables,
    'list_dimensions': list_dimensions,
    'describe_table': describe_table,
    'query': query,
    'echo': echo,
}
```

Figure 167. OLAP Query API Operations

As seen in **Figure 167**, the following operations include listing the fact tables, listing the dimensions, describing specific tables, and querying from the OLAP database.

```
def list_dimensions():
    """List dimensions.

    Returns
    -------
    response : dict
        Dict containing list of dimensions
    """
    response = client.list_tables(
        ClusterIdentifier='diwata-gas-olap',
        Database='diwatagas_olap_db',
        SecretArn="arn:aws:secretsmanager:us-east-1:590184004366:secret:diwata-secret-6zcPUM",
        MaxResults=123,
        TablePattern="%dimension"
    )
    return response
```

Figure 168. List Dimensions Operation

As seen in **Figure 168**, the list dimensions operation lists all the dimensions within the Redshift database, through the Redshift client and the pattern "%dimension". This is also using the secret specific to the Redshift cluster.

```
def list_facttables():
    """List fact tables.

    Returns
    -------
    response : dict
        Dict containing list of fact tables
    """
    response = client.list_tables(
        ClusterIdentifier='diwata-gas-olap',
        Database='diwatagas_olap_db',
        SecretArn="arn:aws:secretsmanager:us-east-1:590184004366:secret:diwata-secret-6zcPUM",
        MaxResults=123,
        TablePattern="%facttable"
    )
    return response
```

Figure 169. List Fact Tables Operation

As seen in **Figure 169**, the list fact tables operation lists all the fact tables within the Redshift database, through the Redshift client and the pattern "%facttable". This is also using the secret specific to the Redshift cluster.

```python
def describe_table(payload):
    """Describe specific table.

    Parameters
    ----------
    payload : dict
        API Payload

    Returns
    -------
    dict
        Dict containing description of table.
    """
    response = client.describe_table(
        ClusterIdentifier='diwata-gas-olap',
        Database='diwatagas_olap_db',
        SecretArn="arn:aws:secretsmanager:us-east-1:590184004366:secret:diwata-secret-6zcPUM",
        MaxResults=123,
        Table=payload['table']
    )
    return {'ColumnList': [i['name'] for i in response['ColumnList']],
            'TableName': response['TableName']}
```

Figure 170. Describe Table Operation

As seen in **Figure 170,** the describe table operation provides the list of columns given a specific table name, using the Redshift client and the secret specific to the Redshift cluster.

```python
def query(payload):
    """Query from OLAP.

    Parameters
    ----------
    payload : dict
        API Payload

    Returns
    -------
    response : dict
        Dict containing query result or error message.
    """
    try:
        if payload['query'].startswith("SELECT"):
            DB_NAME = "diwatagas_olap_db"
            DB_PORT = 5439
            DB_HOST = "diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com"
            DB_USER = cluster_creds['DbUser']
            DB_PWD = cluster_creds['DbPassword']
            conn_string = "dbname='{}' port='{}' host='{}' user='{}' password='{}'".format(DB_NAME, DB_PORT, DB_HOST, DB_USER, DB_PWD)
            con = psycopg2.connect(conn_string)
            cur = con.cursor()
            cur.execute(payload['query'])
            lst_items = cur.fetchall()
            lst_desc = [desc[0] for desc in cur.description]
            con.close()
            return json.loads(json.dumps({'ColumnList': lst_desc,
                                          'Items': lst_items
            }, cls=SetEncoder))
        else:
            return {
                'statusCode': 400,
                'body': json.dumps('Request is not a query. Start your query with SELECT.')
            }
    except Exception as err:
        return str(err)
```

Figure 171. Query Operation

As seen in **Figure 171,** the query operation uses the psycopg2 library to connect to the Redshift cluster, using the credentials and other database information. The query in the payload is then executed, and the results are

returned as dictionary of the column names and the items from the query. Moreover, the dictionary is also encoded to account for the non-JSON serializable Python objects inside it. In case of an exception, to inform the user about the error, the exception message is returned as a string.

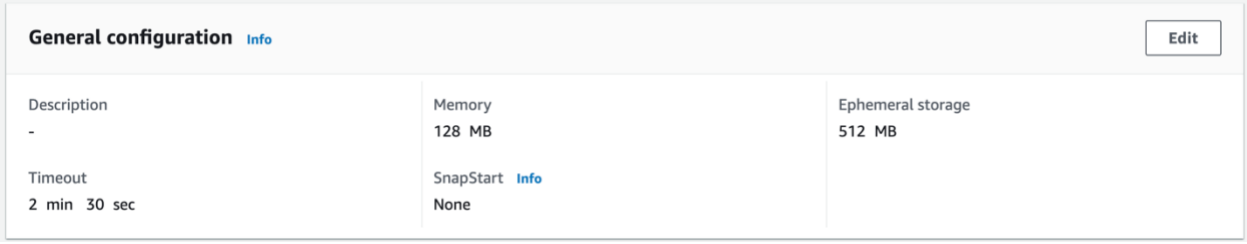


Figure 172. General Configuration of the OLAP Query API

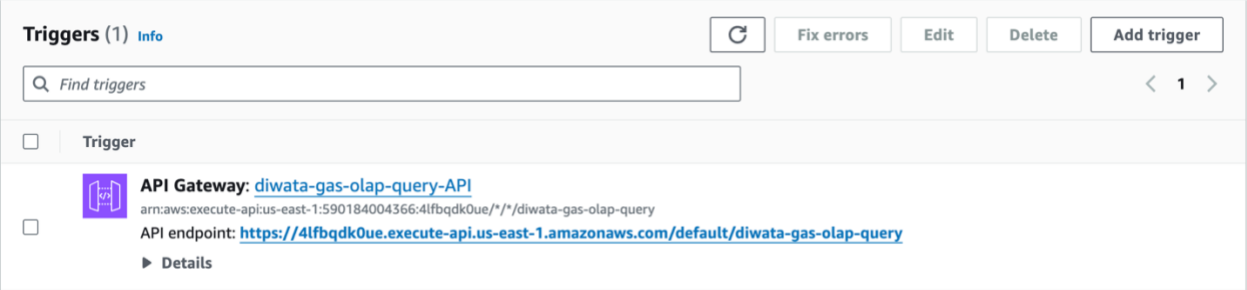


Figure 173. API Endpoint of the OLAP Query API

Similar configurations to the S3 API are done for the OLAP Query API.

API Sample Use Cases

Complaints IT Officer Use Case

A. Sample Complaints IT Officer Checking Complaints (Using DynamoDB API)

i. Checking all Complaints (Scan)

```
url = "https://n0ih6zteg1.execute-api.us-east-1.amazonaws.com/default/diwata_gas_ddb"
myobj = {'operation': 'scan', 'api_key': pin}
x = requests.post(url, json=myobj)
```
Last executed at 2024-09-19 17:53:59 in 2.02s

```
df_complaints = pd.DataFrame(json.loads(x.text))
```
Last executed at 2024-09-19 17:53:59 in 9ms

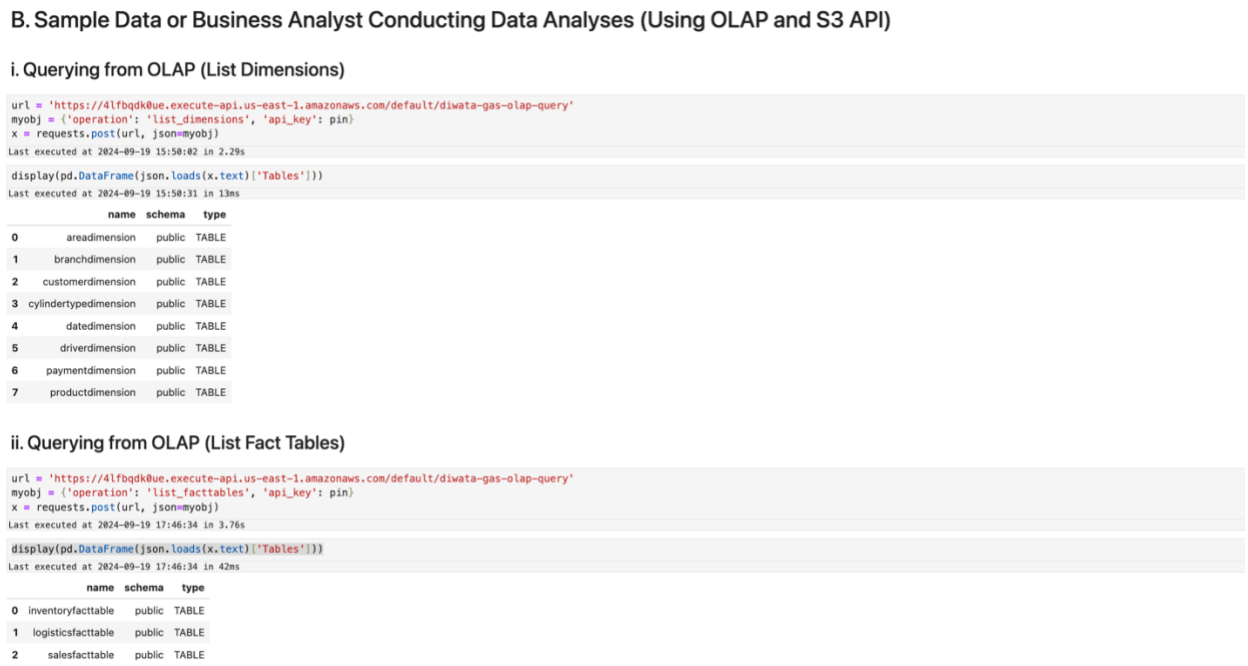
```
df_complaints.head(5)
```
Last executed at 2024-09-19 17:53:59 in 13ms

	Date	ChiefComplaints	ComplaintID	CustomerContactNumber	Time	SerialNo	PictureS3Path	CustomerName	CustomerReviews	IsReplaced	MaintenanceComments
0	2024-08-29	[]	1	09119632198	00:00:00	FKSAJF59FDKA	https://diwata-gas-bucket.s3.us-east-1.amazona...	John Cris Orenday	No cap	NaN	NaN
1	2024-08-30	[Odor, Gauge]	1	09123456892	00:00:00	SDAKSL3012A	https://diwata-gas-bucket.s3.us-east-1.amazona...	Christian Alis	There is an unusual noise.	True	For replacement.
2	2024-08-30	[Clogged, Unusual Noise]	2	09119632198	00:00:00	SDAKSL3012A	https://diwata-gas-bucket.s3.us-east-1.amazona...	Rex Gregor M Laylo	There is an unusual noise.	False	Wrong handling
3	2024-08-30	[Odor, Gauge]	3	09123456892	00:00:00	SDAKSL3012A	https://diwata-gas-bucket.s3.us-east-1.amazona...	Christian Alis	There is an unusual noise.	NaN	NaN
4	2024-08-30	[Clogged, Unusual Noise]	4	09119632198	00:00:00	SDAKSL3012A	https://diwata-gas-bucket.s3.us-east-1.amazona...	Rex Gregor M Laylo	There is an unusual noise.	NaN	NaN

Figure 174. Complaints IT Officer Use Case

As seen above, the IT officer for the Complaints Department can call the API to receive a table containing all the complaints.

Data or Business Analyst Use Case



B. Sample Data or Business Analyst Conducting Data Analyses (Using OLAP and S3 API)

i. Querying from OLAP (List Dimensions)

```
url = 'https://4lfbqdk0ue.execute-api.us-east-1.amazonaws.com/default/diwata-gas-olap-query'
myobj = {'operation': 'list_dimensions', 'api_key': pin}
x = requests.post(url, json=myobj)
```

Last executed at 2024-09-19 15:50:02 in 2.29s

```
display(pd.DataFrame(json.loads(x.text)['Tables']))
```

Last executed at 2024-09-19 15:50:31 in 13ms

	name	schema	type
0	areadimension	public	TABLE
1	branchdimension	public	TABLE
2	customerdimension	public	TABLE
3	cylindertypedimension	public	TABLE
4	datedimension	public	TABLE
5	driverdimension	public	TABLE
6	paymentdimension	public	TABLE
7	productdimension	public	TABLE

ii. Querying from OLAP (List Fact Tables)

```
url = 'https://4lfbqdk0ue.execute-api.us-east-1.amazonaws.com/default/diwata-gas-olap-query'
myobj = {'operation': 'list_facttables', 'api_key': pin}
x = requests.post(url, json=myobj)
```

Last executed at 2024-09-19 17:46:34 in 3.76s

```
display(pd.DataFrame(json.loads(x.text)['Tables']))
```

Last executed at 2024-09-19 17:46:34 in 42ms

	name	schema	type
0	inventoryfacttable	public	TABLE
1	logisticsfacttable	public	TABLE
2	salesfacttable	public	TABLE

Figure 175. Data or Business Analyst Use Case Part 1

As seen above, a data analyst can list the dimensions and fact tables from the OLAP, which would guide them towards their future OLAP queries.

iv. Querying from OLAP (Doing SQL Queries)

```
url = 'https://4lfbqdk0ue.execute-api.us-east-1.amazonaws.com/default/diwata-gas-olap-query'
myobj = {'operation': 'query',
         'payload':{
             'query': 'SELECT * FROM salesfacttable s JOIN productdimension p ON s.productid = p.productid'},
         'api_key': pin}
x = requests.post(url, json=myobj)
```

Last executed at 2024-09-19 15:59:54 in 1.44s

```
df_products = pd.DataFrame(json.loads(x.text)['Items'], columns=json.loads(x.text)['ColumnList'])
```

Last executed at 2024-09-19 15:59:56 in 17ms

```
display(df_products)
```

Last executed at 2024-09-19 16:00:02 in 29ms

	dateid	customerid	branchid	areaid	productid	cylindertypeid	paymentid	quantity	revenue	productid	producttype	productcategory
0	20240830	210	8	3	7	1	6	1	5900.0	7	Argon (ARGO40)	Industrial
1	20240830	210	8	3	4	3	2	1	15000.0	4	Argon (Ar HP)	Industrial
2	20240830	210	8	3	7	4	2	1	12000.0	7	Argon (ARGO40)	Industrial
3	20240830	204	8	3	2	2	2	1	7000.0	2	Acetylene (C2H2)	Industrial
4	20240830	204	8	3	16	2	2	1	6000.0	16	Nitrous Oxide (N2O)	Medical
5	20240828	203	9	6	9	2	4	1	18000.0	9	Carbon Dioxide (CO2)	Industrial
6	20240815	201	7	1	2	1	2	1	25000.0	2	Acetylene (C2H2)	Industrial
7	20240829	201	7	1	10	2	2	1	11000.0	10	Compressed Air (C-AIR)	Industrial
8	20240901	201	7	1	2	1	2	1	25000.0	2	Acetylene (C2H2)	Industrial
9	20240829	201	7	1	2	1	2	1	25000.0	2	Acetylene (C2H2)	Industrial
10	20240901	201	7	1	10	2	2	1	11000.0	10	Compressed Air (C-AIR)	Industrial
11	20240815	201	7	1	10	2	2	1	11000.0	10	Compressed Air (C-AIR)	Industrial
12	20240830	210	8	3	12	4	6	1	10800.0	12	Medical Carbon Dioxide (MCO2)	Medical
13	20240828	203	9	6	12	3	4	1	19000.0	12	Medical Carbon Dioxide (MCO2)	Medical
14	20240829	203	9	6	12	3	2	1	8900.0	12	Medical Carbon Dioxide (MCO2)	Medical
15	20240828	201	7	1	13	1	4	1	11000.0	13	Medical Oxygen (MO2)	Medical
16	20240830	208	9	6	3	3	2	1	7500.0	3	Argon (200 BARS - ARGO51)	Industrial
17	20240829	210	8	3	15	2	4	1	13500.0	15	Nitrogen (N2)	Industrial

Figure 176. Data or Business Analyst Use Case Part 2

As seen above, a data analyst can then query from the OLAP database, using an SQL query and the API, which could result in the table above.

v. Data Analysis and Visualizations using SQL Query

Summary Statistics

```
pd.DataFrame(df_products['revenue'].describe())
```

Last executed at 2024-09-19 16:06:41 in 14ms

	revenue
count	29.000000
mean	14137.931034
std	6598.344695
min	5000.000000
25%	8900.000000
50%	12000.000000
75%	19000.000000
max	29000.000000

Figure 177. Data or Business Analyst Use Case Part 3

As seen above, a data analyst can then produce summary statistics from the generated results from the query.

Product Type Visualization

```
df_products.groupby(['producttype'])['revenue'].mean().sort_values().plot.barh()
plt.xlabel('Average Revenue')
plt.title('Overall average revenue across product types')
plt.show()
```

Last executed at 2024-09-19 16:07:44 in 117ms

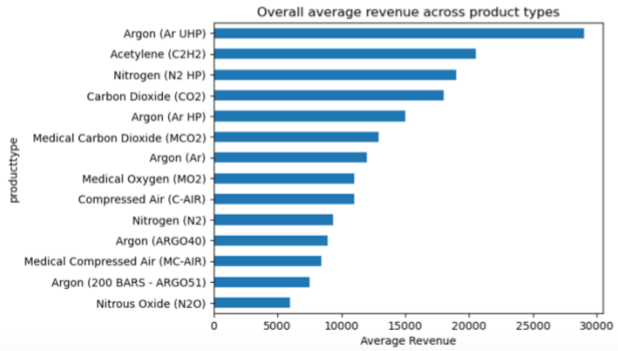


Figure 178. Data or Business Analyst Use Case Part 4

This could be further augmented by data visualizations such as the one above.

Data Scientist Use Case

C. Sample Data Scientist Conducting Data Modeling (Using S3 API)

i. Listing Work Files from S3 Bucket

```
url = 'https://pa8tpalzj9.execute-api.us-east-1.amazonaws.com/default/diwata_gas_s3'
myobj = {'operation': 'list_work', 'api_key': pin}
x = requests.post(url, json=myobj)
```

Last executed at 2024-09-19 16:09:52 in 5.92s

```
json.loads(x.text)
```

Last executed at 2024-09-19 16:09:53 in 8ms


```
{'Items': ['work/CustomerOrder_August_2024.parquet',
  'work/Logistics_August_2024.parquet']}
```


ii. Downloading Work Files from S3 Bucket

```
url = 'https://pa8tpalzj9.execute-api.us-east-1.amazonaws.com/default/diwata_gas_s3'
myobj = {'operation': 'download_work',
         'payload': {'table': 'logistics', 'month': 'August', 'year': 2024},
         'api_key': pin}
x = requests.post(url, json=myobj)
```

Last executed at 2024-09-19 16:53:34 in 4.74s

```
json.loads(x.text)
```

Last executed at 2024-09-19 16:53:34 in 7ms


```
{'statusCode': 200,
 'body': 'https://diwata-gas-bucket.s3.amazonaws.com/work/Logistics_August_2024.parquet?AWSAccessKeyId=ASIAYS2NVHMHNN4DZZNW&Signature=ujkKeMRUIwbhuNccypyVaEBQCAs%3D&x-amz-security-token=IQoJb3JpZ2luX2V
jECEaCXVzLWVhc3QtMSJHMEUCIQCam3kZRAy5kmWrpB8RG8xNBsOrgvNX%2BJrtVfTpnbgiQgIgZe1ALnkji5svUsJn0PhoaEvIduTMkladtO0Zehdtd0Yq7gIIWhAAGgw1OTAxODQwMDQzNjYiDBpeqC%2BhjupXT%2FqobirLAug25C28lt8KMYruWsZJQ0qa%2B9Pp
SUQcGxle2RmkHoyFjkiwSHrA%2Bg1q00%2BsFJ%2BaUMXO%2FIu0%2FiCU0a86GI3fa8Keg%2Byw%2BA%2B5z0MPHaFL2yFnnBD8w79tkhByxs%2FsTNQBzTPa17L2L9UafMeaf0tVv%2Bka7BdrOk9g0yWLHC3PATCASqvPKaFvgpaepTrc5mC%2FFJQBH%2BygHU%2F
cdyn6xNmuNEE3qvJYzJXQowuA3S4yJoXQkLsvlcRtzrDhHaTD9pCJXBNTjeAN5mtTqZ5Ypks2zOx4nXVes5KdKumxPus3%2B0nmgO3gfNrV7zV7I%2FC879LVG4YSQdFAydcDYRv4GkPZgIdfQNUaQIoytFCqgkoSUIKKTNlZHM6144O8yhA8lkAHPQXPA9pMFQGlWLpT
CKJOY%2B98fPX41aX0hW9IlFiuPkXRh8SmwIzlOYaM2MQw78mvtwY6ngGiATeCryb8sMzEtlV8szti0QJA1nUTETwRNvWtlNI86agaWnFsxluVKloJ6tAMOGxLCKK%2BnqPkrorxEMEF55QaPFrzHLgQUtIgL9WmshsJS4suuzTy4J%2FGldWyzyLylnI%2BEkd1GF8dU
nbELTeVDTkLcpUcVE44jIK%2FGQqIYUpVuY%2BQ3I%2FZbqUqTU1CNc0D%2BY67U%2B2XveE45V1zotKW9g%3D%3D&Expires=1726739614'}
```


Figure 179. Data Scientist Use Case Part 1

Using the S3 API, the data scientist can list work files and extract these work files, for data modeling purposes.

Figure 180. Data Scientist Use Case Part 2

After loading and pre-processing the data for modeling, the data scientist may then create regression models such as the one above, where the example above is used for predicting price based on the product type and client.

5. CONCLUSION AND RECOMMENDATION

To conclude, the team developed an end-to-end solution for Diwata Gas Corporation, which involved data ingestion through a web application which is then routed to an S3 bucket-based data lake. From the landing zone, these collected data either remain in the landing zone or are classified based on the other zones, namely, gold, work, and sensitive.

Using Apache Airflow DAGs, these data are then routed to the OLTP, OLAP, and NoSQL databases. Moreover, using these DAGs, historical data from the OLTP and OLAP databases are then saved back into the data lake either in the gold or work zone. Lastly, being an end-to-end solution, the users in the Diwata Gas Corporation are given API endpoints from which they can query from either the OLAP or NoSQL databases or from which they can download work or gold files from the data lake.

Given this solution, the team recommends the following recommendations:

- The web application may be improved by immediately channeling its collected data both to the OLTP and to the NoSQL databases, instead of first collecting it within the S3 bucket or data lake. These collected data may then be saved afterwards to the data lake. For the NoSQL database, in particular, it would be worth considering leveraging on DynamoDB's low-latency, fully managed database service. DynamoDB efficiently handles large volumes of structured or semi-structured data, allowing

for fast tracking and processing especially to customer complaint management, where low latency is critical for timely responses.

- The architecture may be further improved by using a data lakehouse instead, which would streamline the ETL process, instead of halving the process between the data lake and the data warehouse.
- For the workflow processing, stream processing through solutions such as Apache Flink may be looked into, as compared to relying on batch processing through Apache Airflow (Brunner, 2023).
- Moreover, for the NoSQL database, loading image data directly instead of the S3 paths may be looked into, to leverage the NoSQL database's capability of handling unstructured data.
- Lastly, the API endpoints can be further improved by aligning more with the REST API's principles (especially in terms of using each HTTP method more appropriately, as compared to assigning every operation to the POST method), and they may also be integrated within the web application, to provide a wholly end-to-end solution for the company.

6. REFERENCES

Adams, S. (2022, June 7). *MongoDB vs DynamoDB Head-to-Head: Which Should You Choose?* Rockset. https://rockset.com/blog/mongodb-vs-dynamodb-head-to-head-which-should-you-choose/

Alam-Naylor, S., & Fateh, D. (2024, February 13). *What is a REST API?* Contentful. https://www.contentful.com/blog/what-is-a-rest-api/

Apache. (2024). *GitHub - apache/airflow: Apache Airflow - A platform to programmatically author, schedule, and monitor workflows*. GitHub. https://github.com/apache/airflow

Blanc, D. (2023, October 30). *Data orchestration: A comparison of Airflow, Prefect, and Mage*. Medium. https://medium.com/@dblancbellido/data-orchestration-a-comparison-of-airflow-prefect-and-mage-71b6b9773bd3

Brunner, J. (2023, August 29). *Flink in Practice: Stream Processing Use Cases for Kafka Users*. Confluent. https://www.confluent.io/blog/apache-flink-stream-processing-use-cases-with-examples/?utm_medium=sem&utm_source=google&utm_campaign=ch.sem_br.nonbrand_tp.prs_tgt.kafka_mt.mbm_rgn.apac_lng.eng_dv.all_con.kafka-use-cases&utm_term=event%20streaming%20use%20cases&creative=&devi

ce=c&placement=&gad_source=1&gclid=Cj0KCQjwurS3BhCGARIsADdUH53Elz2Ne5I4hzRSM5spGqsuXwfct1WUg-rMy90c40baEH67vgXKXAMaAtg5EALw_wcB

Complaint management system schema design in DynamoDB. (2024). Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/data-modeling-complaint-management.html

Franklin, J. (2024, May 28). *Apache Airflow – When to use it, when to avoid it*. Upsolver. https://www.upsolver.com/blog/apache-airflow-when-to-use-it-when-to-avoid-it-while-building-a-data-lake

Marshall, M. (2024, April 22). *REST, SOAP, and HTTP APIs: What's the difference?* Qase Blog. https://qase.io/blog/rest-vs-soap-vs-http-api/

Muthiah, S. (2023, November 4). *Effortless global content delivery: Hosting a static website with Amazon S3, CloudFront, and Route 53*. https://www.linkedin.com/pulse/effortless-global-content-delivery-hosting-static-website-muthiah-2bq9c/

Presigned URLs - Boto3 1.35.24 documentation. (2024). https://boto3.amazonaws.com/v1/documentation/api/latest/guide/s3-presigned-urls.html

Use AWS WAF to protect your REST APIs in API Gateway. (2024). Amazon API Gateway. https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-control-access-aws-waf.html

What is AWS Lambda? (2024). AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

What is Amazon API Gateway? (2024). Amazon API Gateway. https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

What is PostgreSQL? (2024). Amazon Web Services, Inc. https://aws.amazon.com/rds/postgresql/what-is-postgresql/

W3Schools.com. (n.d.). https://www.w3schools.com/tags/ref_httpmethods.asp

What is NoSQL? Databases Explained | Google Cloud. (2024). Google Cloud. https://cloud.google.com/discover/what-is-nosql

Tobin, D. (2023, July 19). *AWS Redshift vs. the rest — What's the best data warehouse?* Integrate.io. https://www.integrate.io/blog/amazon-redshift-how-aws-redshift-compares-to-other-warehouse-data-solutions/

7. APPENDICES

Appendix 1. SQL Script for Creating the Sales Fact Table

```
%%sql
CREATE TABLE IF NOT EXISTS SalesFactTable (
    DateID INTEGER,
    CustomerID INTEGER,
    BranchID INTEGER,
    AreaID INTEGER,
    ProductID INTEGER,
    CylinderTypeID INTEGER,
    PaymentID INTEGER,
    Quantity INTEGER,
    Revenue REAL
)
DISTSTYLE KEY DISTKEY (CustomerID)
COMPOUND SORTKEY (DateID, BranchID, ProductID, CylinderTypeID);
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Appendix 2. Table Schema of the Sales Fact Table

```
%sqlcmd columns -t salesfacttable
```

name	type	nullable	default	autoincrement	comment	info
dateid	INTEGER	True	None	False	None	{}
customerid	INTEGER	True	None	False	None	{'encode': 'az64'}
branchid	INTEGER	True	None	False	None	{}
areaid	INTEGER	True	None	False	None	{'encode': 'az64'}
productid	INTEGER	True	None	False	None	{}
cylindertypeid	INTEGER	True	None	False	None	{}
paymentid	INTEGER	True	None	False	None	{'encode': 'az64'}
quantity	INTEGER	True	None	False	None	{'encode': 'az64'}
revenue	REAL	True	None	False	None	{}

Appendix 3. Sample Data for the Sales Fact Table

dateid	customerid	branchid	areaid	productid	cylindertypeid	paymentid	quantity	revenue
20240828	201	7	1	13	1	4	1	11000.0
20240815	201	7	1	2	1	2	1	25000.0
20240828	201	7	1	5	3	4	1	29000.0
20240829	201	7	1	10	2	2	1	11000.0
20240901	201	7	1	2	1	2	1	25000.0
20240829	201	7	1	2	1	2	1	25000.0
20240830	208	9	6	11	3	2	1	8900.0
20240830	208	9	6	3	3	2	1	7500.0
20240901	201	7	1	10	2	2	1	11000.0
20240815	201	7	1	10	2	2	1	11000.0

Appendix 4. SQL Script for Creating the Inventory Fact Table

```
%%sql
CREATE TABLE IF NOT EXISTS InventoryFactTable (
    DateID INTEGER,
    ProductID INTEGER,
    CylinderTypeID INTEGER,
    AvailableQuantity INTEGER,
    UnavailableQuantity INTEGER
)
DISTSTYLE KEY DISTKEY (ProductID)
COMPOUND SORTKEY (DateID, ProductID, CylinderTypeID);
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Appendix 5. Table Schema of the Inventory Fact Table

```
%sqlcmd columns -t inventoryfacttable
```

name	type	nullable	default	autoincrement	comment	info
dateid	INTEGER	True	None	False	None	{}
productid	INTEGER	True	None	False	None	{}
cylindertypeid	INTEGER	True	None	False	None	{}
availablequantity	INTEGER	True	None	False	None	{'encode': 'az64'}
unavailablequantity	INTEGER	True	None	False	None	{'encode': 'az64'}

Appendix 6. Sample Data for the Inventory Fact Table

dateid	productid	cylindertypeid	availablequantity	unavailablequantity
20240919	5	4	2	0
20240919	5	2	6	0
20240919	5	1	6	0
20240919	5	3	9	0
20240919	15	1	3	0
20240919	15	5	3	0
20240919	15	2	5	0
20240919	5	5	2	0
20240919	15	3	5	0
20240919	15	4	1	0

Appendix 7. SQL Script for Creating the Logistics Fact Table

```
%%sql
CREATE TABLE IF NOT EXISTS LogisticsFactTable (
    DateID INTEGER,
    CustomerID INTEGER,
    AreaID INTEGER,
    ProductID INTEGER,
    CylinderTypeID INTEGER,
    DriverID INTEGER,
    DeliveryQuantity INTEGER,
    PickUpQuantity INTEGER
)
DISTSTYLE KEY DISTKEY (CustomerID)
COMPOUND SORTKEY (DateID, DriverID);
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

Appendix 8. Table Schema of the Logistics Fact Table

```
%sqlcmd columns -t logisticsfacttable
```

name	type	nullable	default	autoincrement	comment	info
dateid	INTEGER	True	None	False	None	{}
customerid	INTEGER	True	None	False	None	{'encode': 'az64'}
areaid	INTEGER	True	None	False	None	{'encode': 'az64'}
productid	INTEGER	True	None	False	None	{'encode': 'az64'}
cylindertypeid	INTEGER	True	None	False	None	{'encode': 'az64'}
driverid	INTEGER	True	None	False	None	{}
deliveryquantity	INTEGER	True	None	False	None	{'encode': 'az64'}
pickupquantity	INTEGER	True	None	False	None	{'encode': 'az64'}

Appendix 9. Sample Data for the Logistics Fact Table

dateid	customerid	areaid	productid	cylindertypeid	driverid	deliveryquantity	pickupquantity
20240830	204	3	2	2	None	0	0
20240828	203	6	9	2	2	0	0
20240830	204	3	16	2	None	0	0
20240829	203	6	12	3	None	0	0
20240828	203	6	12	3	2	0	0
20240830	204	3	14	2	None	0	0
20240829	203	6	15	2	None	0	0
20240831	204	6	14	2	None	0	0
20240831	204	6	11	5	None	0	0
20240831	204	6	15	3	None	0	0

Appendix 10. List of All Tables on the ‘diwatagas_olap_db’ Database

```
%sql \dt
```

Running query in 'redshift://jj:***@diwata-gas-olap.cxcjt2ltmyli.us-east-1.redshift.amazonaws.com:5439/diwatagas_olap_db'

schema	name	type	owner
public	areadimension	table	jj
public	branchdimension	table	jj
public	customerdimension	table	jj
public	cylindertypedimension	table	jj
public	datedimension	table	jj
public	driverdimension	table	jj
public	inventoryfacttable	table	jj
public	logisticsfacttable	table	jj
public	paymentdimension	table	jj
public	productdimension	table	jj
public	salesfacttable	table	jj