Database Assignment for Senior Data Platform Engineer

Amarja Shivraj Pawar

https://github.com/Amarja20/Data-Modelling-Healthcare

Task 1: Database Schema Optimization

Schema Overview

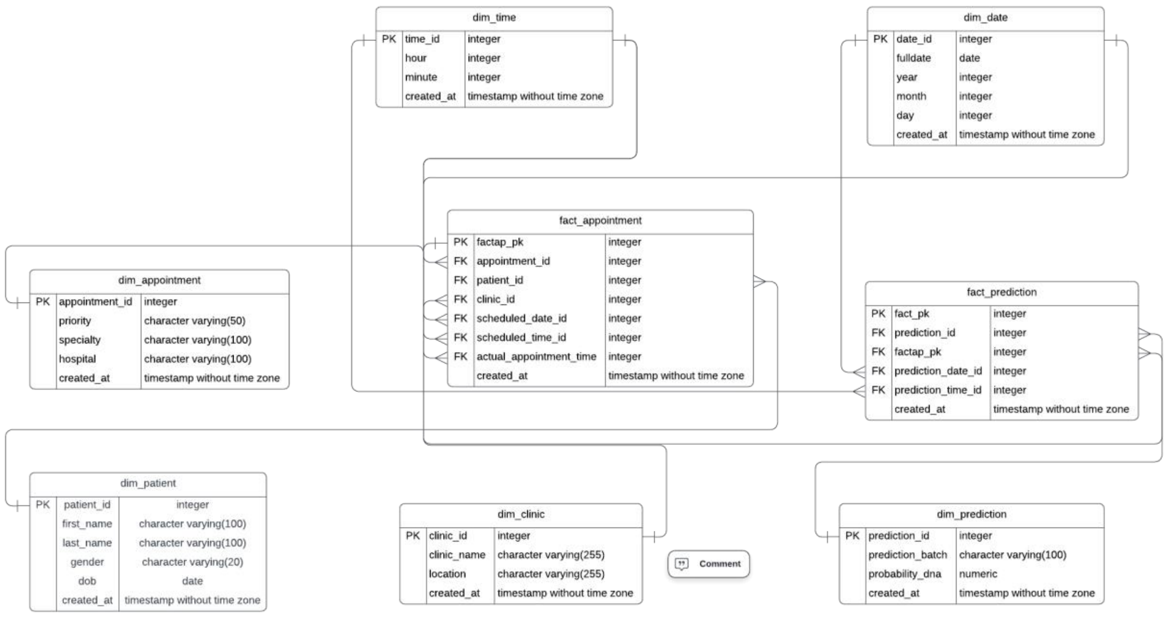
dim_time
PK time_id integer
hour integer
minute integer
created_at timestamp without time zone
dim_date
PK date_id integer
fulldate date
year integer
month integer
day integer
created_at timestamp without time zone
fact_appointment
PK factap_pk integer
FK appointment_id integer
FK patient_id integer
FK clinic_id integer
FK scheduled_date_id integer
FK scheduled_time_id integer
FK actual_appointment_time integer
created_at timestamp without time zone
dim_appointment
PK appointment_id integer
priority character varying(50)
specialty character varying(100)
hospital character varying(100)
created_at timestamp without time zone
fact_prediction
PK fact_pk integer
FK prediction_id integer
FK factap_pk integer
FK prediction_date_id integer
FK prediction_time_id integer
created_at timestamp without time zone
dim_patient
PK patient_id integer
first_name character varying(100)
last_name character varying(100)
gender character varying(20)
dob date
created_at timestamp without time zone
dim_clinic
PK clinic_id integer
clinic_name character varying(255)
location character varying(255)
created_at timestamp without time zone
Comment
dim_prediction
PK prediction_id integer
prediction_batch character varying(100)
probability_dna numeric
created_at timestamp without time zone

Fig: ER diagram showcasing the dimensional modelling approach

In my proposed methodology, I adopted a dimensional modelling approach based on star schema model where data is distributed into facts and dimensional tables. The Fact tables hold measurable data and Dimension tables contain descriptive attributes related to facts. Moreover, for analytical purposes, I implemented the one big table (OBT) data modelling technique based on the Medallion Architecture. This approach consolidates multiple dimension tables into a single, comprehensive table, thereby simplifying query operations. This approach reduces joins and aggregations resulting in better performance for large-scale analytics and machine learning tasks.

1. Star Schema

Appointment Table: In the original schema, the Appointment table contained both transactional data (event-driven) and descriptive details in a single structure. To optimise this structure, I divided the Appointment table into a Fact Table FACT_Appointment and a Dimension Table DIM_Appointment.

Prediction Table: Similar to the Appointment table the prediction table in the original model, contained both the prediction event data and descriptive details in a single structure. The tables were then divided into Fact table FACT_Prediction and Dimension table DIM_Prediction.

1.1 Fact Tables

1. FACT_APPOINTMENT: This table hold all the measurable details of the appointments. It includes foreign keys to the Dimensional tables such as DIM_APPOINTMENT,

DIM_PATIENT, DIM_CLINIC, DIM_DATE, and DIM_TIME tables, which contain additional context for each appointment.
2. FACT_PREDICTION: This table hold all the measurable details of the predictions including foreign keys of dimensional tables. The fact table references to Dimensional tables such as DIM_PREDICTION, DIM_DATE, and DIM_TIME tables, which further describe the prediction event.

1.2 Dimension Tables

The dimension tables store descriptive attributes and enable filtering and aggregation of data within the fact tables. The key dimensions are described below:

1. DIM_APPOINTMENT: Contains metadata related to the appointment, such as priority, specialty, and hospital information. This dimension is referenced by the FACT_APPOINTMENT table to provide detailed appointment attributes.
2. DIM_PREDICTION: Stores prediction metadata such as batch information and DNA probability scores. This table is referenced by the FACT_PREDICTION table to provide prediction-specific details.
3. DIM_PATIENT: Contains patient demographic information, such as first_name, last_name, gender, and dob. This dimension is linked to FACT_APPOINTMENT to provide patient-specific context.
4. DIM_CLINIC: Stores the clinic's metadate information, such as clinic name and location. This dimension supports better organization of appointment data across multiple clinics.
5. DIM_TIME: This table holds detailed information regarding the time of day, including hour and minutes.
6. DIM_DATE: This table is designed to store detailed calendar information, including the full date, year, month, and day.

We separated DIM_TIME and DIM_DATE to ensures efficient querying and aggregation of time-series data across both appointments and predictions. As they are referenced multiple times in the FACT_APPOINTMENT and FACT_PREDICTION tables. This approach avoiding duplication of the same date or time entries for multiple records (primary key of time and date). For example, if 100 appointments are scheduled at 10:00 AM on the 2024/07/02, instead of storing the same time and date 100 times in the fact table, we store a single reference to the DIM_TIME and DIM_DATE tables. This improves storage efficiency and simplifies the process of performing complex, time-based analyses, such as calculating the total number of appointments in a specific month.

1.3. Fact Table Benefits

We have stored only numeric values in the fact tables, which provide key advantages such as:

1. Reduced Storage Overhead: Numeric foreign keys require less storage than full descriptive data (e.g., storing an integer ID rather than a patient name string like "John Doe" for patients). This reduces the overall size of the fact tables, especially as the volume of data grows.
2. By moving common descriptive fields (e.g., clinic, patient, date, and time) to dimension tables, we avoid repeatedly storing the same information in the fact tables, significantly reducing the storage usage. By referencing dimension tables, we can quickly retrieve relevant information without scanning large volumes of data.
3. A fact table can consist of billions of records. If new dimensions are introduced, we can easily incorporate it by adding the foreign key to the fact table.

2. One Big Table

2.1 Approach

In order to optimise the data architecture for analytical purposes, I implemented the One Big Table (OBT) methodology which is an extension to the star scheme. We merged relevant data from fact and dimension tables from the traditional star schema into single , comprehensive, flattened table: one related to Unified_Appointments and another Unified_Predictions. This approach denormalises the data. The data is typically loaded during an automated off-peak hours ETL process.

1. Unified_Appointments Table: This table combines all relevant data related to appointments, including fact data (e.g., factap_pk, created_at_fact) and dimension attributes (e.g., appointment_id, priority, patient_id, clinic_id, scheduled_date_id, scheduled_time_id, actual_appointment_time_id).
2. Unified _ Predictions Table: Similarly, this table integrates prediction-related data alongside appointment keys, facilitating seamless predictive analytics.

2.2 One Big Table Benefits and how it addresses bottlenecks

1. Reduces Join operations: This approach avoids joining multiple tables as all relevant data is consolidated in one table, this makes analytical queries much faster as there is no need to navigate through multiple tables.
2. OBT reduces the complexity of the data model by minimizing the number of tables.
3. OBT includes pre-aggregated data which reduces the need for on-the-fly calculations. This helps seed up querying when working on large dataset.

3. Use Cases Addressed by the new schema

3.1 Identifying the best performing Clinics based on the delay in the Actual Appointment Time and scheduled time.

Identifying clinics with a delay between the actual appointment time and the scheduled appointment time is crucial for optimizing healthcare resources. In the updated schema, this is done by comparing the scheduled_time_id and actual_appointment_time in the FACT_APPOINTMENT table, both of which reference DIM_TIME. Clinics that have an appointment with a delay of more than 10 minutes are flagged as having delayed appointments. To determine the best-performing clinic, we calculate the number of delayed appointments for each clinic by using the clinic_id field in FACT_APPOINTMENT, which references DIM_CLINIC. Clinics with the least number of delayed appointments can then be identified as the best-performing.

Using star schema

```sql
 SELECT
    c.clinic_id,
    c.clinic_name,
    COUNT(a.factap_pk) AS delayed_appointments
FROM
    FACT_APPOINTMENT a
JOIN
    DIM_CLINIC c ON a.clinic_id = c.clinic_id
JOIN
    DIM_TIME scheduled_time ON a.scheduled_time_id = scheduled_time.time_id
```

```
JOIN
    DIM_TIME actual_time ON a.actual_appointment_time = actual_time.time_id
WHERE
    (actual_time.hour * 60 + actual_time.minute) -
    (scheduled_time.hour * 60 + scheduled_time.minute) > 10
GROUP BY
    c.clinic_id, c.clinic_name
ORDER BY
    delayed_appointments ASC;
```

Using OBT

```
 SELECT
    clinic_id,
    clinic_name,
    COUNT(factap_pk) AS delayed_appointments
FROM
    Unified_Appointment
WHERE
    (actual_hour * 60 + actual_minute) -
    (scheduled_hour * 60 + scheduled_minute) > 10
GROUP BY
    clinic_id, clinic_name
ORDER BY
    delayed_appointments ASC;
```

3.2 Predicting Missed Appointments Based on Clinic Location

Geographical barriers can play a significant role in whether patients attend appointments. By linking clinic locations with predicted no-shows, healthcare providers can understand if certain clinics are more prone to missed appointments due to accessibility issues. In the new schema, missed appointments based on clinic location can be predicted by joining the DIM_CLINIC, DIM_PREDICTION, and FACT_APPOINTMENT tables. The location field in DIM_CLINIC and probability_dna in DIM_PREDICTION allow for aggregating missed appointments (e.g., probability_dna > 0.85) by clinic.

Using Star schema

```
SELECT
    dc.clinic_name,
    dc.location,
    COUNT(fp.fact_pk) AS missed_appointments
FROM
    FACT_APPOINTMENT fa
JOIN
    DIM_CLINIC dc ON fa.clinic_id = dc.clinic_id
JOIN
    FACT_PREDICTION fp ON fa.factap_pk = fp.factap_pk
JOIN
    DIM_PREDICTION dp ON fp.prediction_id = dp.prediction_id
WHERE
    dp.probability_dna > 0.85
GROUP BY
    dc.clinic_name,
    dc.location
ORDER BY
    missed_appointments DESC;
```

3.3 Tracking Patient No-Shows Over Time

With the new schema we can track patient no-shows over time, we can leverage the DIM_DATE and DIM_TIME tables along with FACT_APPOINTMENT and FACT_PREDICTION to analyze the pattern of missed appointments across different time periods (e.g., daily, monthly, or yearly).

4. Optimization Strategies for transactional and analytical stores

4.1 Partitioning on Clinic_id, and date

I implemented composite partitioning on two keys clinic_id, date to improve query performance by reducing the amount of data that is scanned. By partitioning by **clinic** first and then by **month** (using scheduled_month), we ensured that queries focusing on a specific month within a clinic would only scan the relevant monthly partition.

```
CREATE TABLE Unified_Appointment (
    factap_pk SERIAL PRIMARY KEY,
    created_at_fact TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    appointment_id INT,
    priority VARCHAR(50),
    specialty VARCHAR(100),
    hospital VARCHAR(100),
    created_at_appointment TIMESTAMP,
    patient_id INT,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    gender VARCHAR(20),
    dob DATE,
    created_at_patient TIMESTAMP,
    clinic_id INT,
    clinic_name VARCHAR(255),
    location VARCHAR(255),
    created_at_clinic TIMESTAMP,
    scheduled_date_id INT,
    scheduled_fulldate DATE,
    scheduled_year INT,
    scheduled_month INT,
    scheduled_day INT,
    scheduled_created_at TIMESTAMP,
```

```sql
    scheduled_time_id INT,
    scheduled_hour INT,
    scheduled_minute INT,
    scheduled_time_created_at TIMESTAMP,
    actual_appointment_time_id INT,
    actual_hour INT,
    actual_minute INT,
    actual_time_created_at TIMESTAMP
) PARTITION BY RANGE (scheduled_fulldate);

CREATE TABLE Unified_Appointment_clinic_1_202401 PARTITION OF Unified_Appointment
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01')
PARTITION BY LIST (clinic_id);

CREATE TABLE Unified_Appointment_clinic_1_202402 PARTITION OF Unified_Appointment
FOR VALUES FROM ('2024-02-01') TO ('2024-03-01')
PARTITION BY LIST (clinic_id);

CREATE TABLE Unified_Appointment_clinic_1_202401 PARTITION OF Unified_Appointment_202401
FOR VALUES IN (101);  -- Assuming clinic_id 101

CREATE TABLE Unified_Appointment_clinic_2_202401 PARTITION OF Unified_Appointment_202401
FOR VALUES IN (102);  -- Assuming clinic_id 102

CREATE TABLE Unified_Appointment_clinic_1_202402 PARTITION OF Unified_Appointment_202402
FOR VALUES IN (101);

CREATE TABLE Unified_Appointment_clinic_2_202402 PARTITION OF Unified_Appointment_202402
FOR VALUES IN (102);
```

For Example, calculating the total number of appointments for **Clinic 1** in **January 2024**:

```sql
  SELECT COUNT(*)
    FROM Unified_Appointment
    WHERE clinic_id = 101
    AND scheduled_fulldate BETWEEN '2024-01-01' AND '2024-01-31';
```

4.2 Indexing on foreign keys

To optimize the performance of your transactional store, I have implemented the below indexing strategy.

```sql
    -- Indexes on FACT_APPOINTMENT
CREATE INDEX idx_fact_appointment_appointment_id ON FACT_APPOINTMENT(appointment_id);
CREATE INDEX idx_fact_appointment_clinic_id ON FACT_APPOINTMENT(clinic_id);
CREATE INDEX idx_fact_appointment_scheduled_date_id ON FACT_APPOINTMENT(scheduled_date_id);

    -- Indexes on FACT_PREDICTION
CREATE INDEX idx_fact_prediction_prediction_id ON FACT_PREDICTION(prediction_id);
CREATE INDEX idx_fact_prediction_factap_pk ON FACT_PREDICTION(factap_pk);
```

FACT_APPOINTMENT Table:

1. appointment_id is a foreign key referencing DIM_APPOINTMENT, indexing appointment_id accelerates join operations between FACT_APPOINTMENT and DIM_APPOINTMENT improving the performance of appointment-related queries.
2. clinic_id: In case of DIM_CLINIC, indexing clinic_id speeds up queries that filter or aggregate data by clinic, essential for operational reports and performance tracking.
3. scheduled_date_id: As a foreign key to DIM_DATE, indexing scheduled_date_id improves the performance of date-based queries, such as tracking appointments over specific time periods.

FACT_PREDICTION Table:

1. prediction_id: This foreign key references DIM_PREDICTION
2. factap_pk: Linking to FACT_APPOINTMENT, indexing factap_pk improves the performance of joins between predictions and their corresponding appointments.

4. NoSQL integration

Using Apache Kafka and Apache Spark

1. API to Mongo DB

Data coming from third party APIs, which can have variable schemas is stored directly in MongoDB. As MongoDB supports changing and varied schemas, prediction data, and new feature data which has varied schemas can be stored.

1. MongoDB to Kafka topics

We can use Debezium that would help monitor the MongoDB collection for any changes such as inserts or updates. Whenever there is new data, Debezium publishes the changes to Apache Kafka topics. Kafka would act as a message bridge that streams the data to the processing layer.

2. Processing layer – Apache Kafka to Apache Spark

Apache spark subscribes to Kafka topics to read any incoming data. This data gets transformed for example, if the incoming data contains nested JSON structures, Spark job can flatten these nested fields into a tables so that they can be integration. Spark jobs can also performs join operations between the incoming data and the existing structured data in PostgreSQL. For example, the incoming data from Kafka contains new prediction results for patients, with fields such as a prediction_id, appointment_id, and a predicted health risk score. Apache Spark can use appointment_id to join this new data with the Appointment table in PostgreSQL. Data from

MongoDB that aligns with existing tables is loaded appropriately. This may include new patient information, appointment data, or prediction results.

Using AWS services

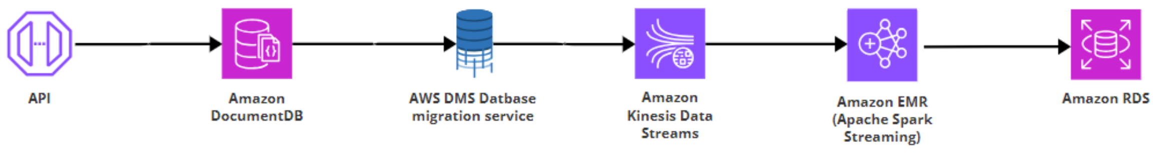
API
Amazon DocumentDB
AWS DMS Datbase migration service
Amazon Kinesis Data Streams
Amazon EMR (Apache Spark Streaming)
Amazon RDS

Fig: AWS architecture diagram for NoSQL integration with Amazon RDS

1. Data Ingestion to Amazon DocumentDB: Incoming data from third-party APIs is ingested into Amazon DocumentDB which is a MongoDB-compatible service that can store data with varied schemas such as variable JSON data from different APIs.

2. AWS Database Migration Service: Whenever there are any changes like inserts or updates to the data, the AWS Database migration Service can be used to detect and capture the change in the DocumentDB. DMS monitors the DocumentDB collections and detects data changes as any.

3. Amazon Kinesis: The DMS then publishes these changes to Amazon Kinesis Data Streams. Each change event from DocumentDB for example if an appointment is updated that becomes an event in a Kinesis data stream. Kinesis Data Streams acts as the streaming layer that captures and makes the data available in real time for processing to Apache Spark.

4. Real-Time Processing with Apache Spark Streaming: Amazon EMR is used to run Apache Spark Streaming. Spark job will consume data from Amazon Kinesis Data Streams in real time. This provides a continuous ingestion of change events from DocumentDB. Similar to the previous approach the spark job will read the JSON data and flatten it to create tables. The data will also be cleaned to handle any missing values or data type conversions if any.

5. Data Joins with RDS PostgreSQL: Apache Spark uses the JDBC connector to connect to Amazon RDS PostgreSQL. Spark performs join operations between incoming data from DocumentDB and existing tables in PostgreSQL. For example, adding context like patient gender to new appointment records.

6. Storing in PostgreSQL: After the data has been transformed, the results would be stored in the Amazon RDS. Spark uses JDBC connector to execute INSERT or UPDATE statements to store the transformed data. For example, If new prediction results are processed, they can be inserted into the Prediction table.

Here's a scenario that can be implements using Apache Kafka or AWS Kinesis. Assume we receive the following data from a 3rd party API in JSON format:

```
{
    "prediction_id": 1101,
        "prediction_batch": "batch_0012",
        "probability_dna": 0.85,
        "appointment_id": 2201,
        "created_at": "2024-10-11T12:45:00Z"
    }
```

This JSON data is stored in Amazon DocumentDB under the predictions collection. AWS DMS captures the new record in DocumentDB and streams it to Amazon Kinesis in real-time. Apache Spark which is running on Amazon EMR reads the data from Kinesis. Spark job then parses the JSON data and creates a DataFrame say predictions_df. Spark uses a JDBC connector to write the prediction data to PostgreSQL.

```
       # Convert JSON string to Spark DataFrame
        predictions_df =
spark.read.json(spark.sparkContext.parallelize([json.dumps(parsed_json)]))

        # PostgreSQL connection properties
        jdbc_url = "jdbc:postgresql://your-rds-endpoint:5432/your_database_name"
        jdbc_properties = {
            "user": "your_username",
            "password": "your_password",
            "driver": "org.postgresql.Driver"
        }

        # Insert new prediction records into DIM_PREDICTION table
        predictions_df.select("prediction_id", "prediction_batch", "probability_dna",
"created_at") \
        .write.jdbc(url=jdbc_url, table="DIM_PREDICTION", mode="append",
        properties=jdbc_properties)
```

Spark writes the new prediction data to the DIM_PREDICTION table in PostgreSQL. mode="append" would be used to make sure that the new data is added without overwriting existing records.

5. AWS Architecture

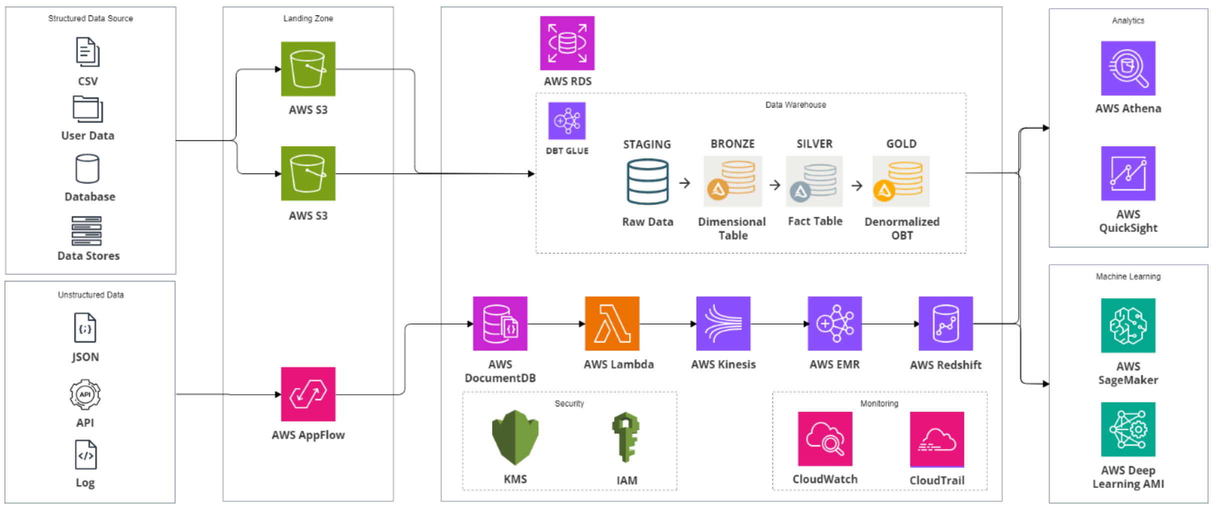

Fig: AWS Data Architecture for Data Ingestion, Processing, and Analytics Using Medallion Architecture

1. Data Source

The data source consists of structured data and unstructured data. In this architecture, structured data sources CSV files, User data, Databases and data stores. The unstructured includes JSON files, API logs, text documents, and multimedia content.

2. Data Ingestion and Storage

1.1 Structured Data is stored in Amazon Simple Storage Service (S3) which serves as the central repository for structured data within this architecture. S3's scalable object storage capabilities enable efficient handling of large volumes of structured data. It provides a simple interface to store and retrieve any amount of data at any time. Use AWS S3 features like versioning, lifecycle policies, and encryption to enhance data security and governance. We can partition and organizing data within S3 based on the intended use cases for better data management and query performance.

1.2 Unstructured Data Ingestion via AWS AppFlow: AWS AppFlow facilitates the secure and efficient ingestion of unstructured data from external Software as a Service (SaaS) applications into AWS services. Unstructured data is sourced from applications and AWS AppFlow orchestrates the transfer of data from these SaaS applications to AWS, performing necessary transformations and validations during the process.

3. Real-time Data Processing Workflow: The real-time data processing layer integrates multiple AWS services to ensure efficient handling and transformation of data in real-time. The workflow involves the following components:

1. Amazon DocumentDB*:* Amazon DocumentDB is a managed NoSQL database service compatible with MongoDB, designed to store and manage unstructured data. It *stores* unstructured data ingested via AWS AppFlow in a document-based format. Providing robust querying capabilities to retrieve and index data for further processing.

2. AWS Lambda: AWS Lambda functions as an event-driven compute service that facilitates real-time data processing. Lambda functions are triggered by events such as data uploads to S3 or updates in DocumentDB. It performs lightweight transformations and filtering on incoming data and then routes processed data to subsequent services like Amazon Kinesis for real-time streaming.

3. Amazon Kinesis: Amazon Kinesis is a scalable real-time data streaming service that handles the ingestion and processing of streaming data. It receives data streams from AWS Lambda and enables real-time analytics and buffering of data for downstream processing. It feeds data into Amazon EMR for large-scale data transformations.

4. Amazon EMR (Big Data Processing): Amazon Elastic MapReduce (EMR) is a managed big data platform that leverages distributed computing frameworks such as Apache Spark and Hadoop. Executes large-scale data processing tasks, including cleaning, aggregation, and transformation. Also provides the computational power required to handle extensive datasets in parallel. It then prepares processed data for storage in Amazon Redshift or further analytical workflows.

5. Amazon Redshift (Data Warehousing): Amazon Redshift serves as the central data warehouse, enabling efficient querying and analysis of structured data. The processed and transformed data from Amazon EMR is stored in Amazon Redshift. It facilitates fast SQL-based queries for business intelligence and analytics. Also, it seamlessly connects with services like Amazon QuickSight for data visualization and Amazon SageMaker for machine learning model training.

4. Data Warehouse (Medallion Architecture)

In our data warehouse implementation, we adopted the Medallion Architecture utilizing AWS RDS PostgreSQL as the core database system. The architecture is structured into four distinct layers to ensure organized data flow and enhanced data quality.

1. Staging Layer: The Staging layer serves as the initial repository for raw data ingested from S3.
2. Bronze Layer: In the Bronze layer, the raw data from the Staging layer is transformed into dimensional tables.
3. Silver Layer: The Silver layer focuses on constructing fact tables, which capture transactional and event-based data. These fact tables aggregate and relate data from the dimensional tables and the provides a comprehensive view.
4. Gold Layer: The Gold layer is the culmination of the Medallion Architecture, featuring a denormalized, comprehensive table that integrates data from dimensional tables into a single, optimized structure. This denormalized table is designed for high-performance analytics and serves as the primary data source for advanced analytics and machine learning applications. By leveraging the denormalized Gold layer, data scientists and analysts can efficiently perform complex queries, generate insights, and develop predictive models without the overhead of navigating multiple tables.

The integration of AWS Glue and dbt across these layers ensures a seamless and automated data transformation workflow. By implementing the Medallion Architecture within AWS RDS PostgreSQL, we achieve a scalable and maintainable data warehouse environment.

5. Monitoring

1. AWS CloudWatch is a comprehensive monitoring and observability service designed to provide real-time insights into the operational health and performance of AWS resources and applications. In the proposed architecture, CloudWatch plays a pivotal role by collecting and tracking metrics, logs, and events from various AWS services such as Amazon S3, AWS Lambda, Amazon Kinesis, Amazon EMR, and Amazon Redshift. By setting up custom dashboards and alarms, CloudWatch enables proactive monitoring of system performance, resource utilization, and application behavior. This facilitates the timely detection of anomalies, performance bottlenecks, and potential failures, thereby ensuring the reliability and efficiency of the data pipeline.

2. AWS CloudTrail is a service that provides comprehensive governance, compliance, and operational auditing by recording detailed logs of all API calls and user activities within an AWS environment. In the context of the proposed architecture, CloudTrail is used in tracking and auditing interactions with critical services such as Amazon S3, AWS AppFlow, Amazon DocumentDB, AWS Lambda, Amazon Kinesis, Amazon EMR, and Amazon Redshift. By capturing metadata about each API call, including the identity of the caller, time of the call, source IP address, and the parameters used, CloudTrail ensures full visibility into the actions performed within the data pipeline. This is essential for security monitoring, enabling the detection of unauthorized access, configuration changes, and potential security breaches.

5. Analytics and Machine Learning

The data store in redshift after real time processing and the relational data stored in AWS RDS is connected to Analytical and Machine Learning tools for visualization and training machine learning models.

1. Amazon Athena*:* Amazon Athena is a serverless interactive query service that allows for ad-hoc querying of data stored in Amazon S3 using standard SQL. By eliminating the need for infrastructure management, Athena enables rapid data exploration and analysis directly on data lakes.

2. Amazon QuickSight: Amazon QuickSight is a business intelligence (BI) service that provides data visualization and dashboarding capabilities. It is used for creating and sharing dashboards that offer real time insights. It easily connects to Amazon Redshift, Amazon Athena, and Amazon S3 for comprehensive data visualization. In our architecture, QuickSight is integrated to visualize the processed data stored in Redshift and AWS RDS, enabling stakeholders to gain actionable insights through intuitive and interactive dashboards.

3. Amazon SageMaker: Amazon SageMaker is a fully managed service that facilitates the building, training, and deployment of machine learning models at scale. It provides an integrated development environment (IDE) for data scientists. Streamlines the process of training models on large datasets and deploying them for inference. It also offers tools to monitor deployed models for performance and data quality.

6. Data Governance and Security

1. AWS Key Management Service (KMS): is a managed service that enables the creation, management, and control of cryptographic keys used to encrypt data across various AWS services. It ensures that sensitive information is securely protected both at rest and in transit, adhering to industry standards and compliance requirements. In our architecture, KMS is integrated to encrypt data stored in Amazon S3 and Amazon Redshift, providing robust data protection and ensuring that all critical data assets are safeguarded through encryption mechanisms.

2. AWS Identity and Access Management (IAM): is a fundamental service that provides fine-grained access control and secure management of AWS resources. IAM allows administrators to create and manage AWS users, groups, and roles, and define specific permissions to control who can access which resources and what actions they can perform. In our architecture, IAM is utilized to enforce the principle of least privilege, ensuring that only authorized users and services have access to critical components such as Amazon SageMaker and Amazon Redshift. This integration provides security by tightly controlling access and maintaining accountability through detailed permission settings.