Health Analytics Platform: Graph-Based Anomaly Detection with Neo4j and SQL Integration

Your Name

March 22, 2025

Abstract

The emergence of wearable health devices has revolutionized personal health monitoring by generating large volumes of real-time data. However, efficiently managing and analyzing this continuous data stream presents significant challenges, particularly for anomaly detection and user behavior modeling. This paper presents a Health Analytics Platform that integrates SQL and Neo4j to store, aggregate, and analyze health metrics. The system is designed to efficiently detect anomalies in health behavior and cluster users based on their activity patterns. The architecture includes PostgreSQL for structured data storage, Neo4j for graph-based relationship modeling, and AWS Lambda functions for automated aggregation. The platform is evaluated through anomaly detection experiments and clustering analysis, demonstrating its effectiveness in identifying outliers and grouping users based on similar health characteristics. Our results show a 78% accuracy in anomaly detection and meaningful user segmentation into three distinct behavioral clusters, providing insights for personalized health interventions.

Contents

1 Introduction **3**
1.1 Background and Motivation 3
1.2 Objectives 3
1.3 Dataset Used 3

2 Database Design and Schema **4**
2.1 PostgreSQL Schema Design 4
2.1.1 PostgreSQL Triggers for Lambda Integration 5
2.2 Graph Schema in Neo4j 6

3 System Architecture **9**
3.1 Technology Stack 9
3.2 Overview of the Architecture 10
3.3 Functionality and Scheduler Overview 11
3.3.1 AWS Lambda Functions 11
3.3.2 Schedulers 11
3.4 `neo4j_daily_sync` 12
3.4.1 High-Level Execution Flow 12
3.4.2 SQL Queries and Their Explanations 12
3.4.3 Cypher Queries and Their Explanations 13
3.5 `daily_update` 15
3.5.1 High-Level Execution Flow 15
3.5.2 SQL Query and Explanation 15
3.6 `hourly_update` 16
3.6.1 High-Level Execution Flow 17
3.6.2 SQL Query and Explanation 17
3.7 `clustering_neo4j` 18

3.7.1 High-Level Execution Flow . . . 18
3.7.2 Cypher Queries and Their Explanations . . . 18

4 Health Metric Dashboard **19**
4.1 Overview of `app.py` . . . 19
4.2 Explanation of Queries and Cyphers . . . 19
4.2.1 PostgreSQL Queries (SQL) . . . 19
4.2.2 Neo4j Queries (Cypher) . . . 21

5 Conclusion and Future Work **22**
5.1 Lessons Learned . . . 22
5.2 Limitations . . . 22
5.3 Future Work . . . 22

1 Introduction

1.1 Background and Motivation

In recent years, the proliferation of wearable health devices, such as Fitbit, Apple Watch, and other activity trackers, has led to the generation of vast amounts of health data. These devices continuously monitor vital statistics, including heart rate, step count, sleep duration, and calorie expenditure, creating time-series datasets that offer valuable insights into personal health trends. The global wearable technology market was valued at $61.3 billion in 2022 and is projected to grow at a compound annual growth rate (CAGR) of 14.6% from 2023 to 2030 .

This exponential growth in wearable adoption has created a data management challenge for healthcare providers and analytics platforms. A single user can generate over 1,440 data points per day from minute-level heart rate monitoring alone, translating to millions of records for even modest user bases. Traditional data processing approaches often fall short when handling this volume and complexity of health data.

However, processing and analyzing this data in a meaningful and scalable way poses several challenges:

- First, relational databases (RDBMS) struggle with complex relationship modeling, making it difficult to efficiently identify patterns or perform network-based queries across user populations and time-series data simultaneously.
- Second, anomaly detection in health metrics demands efficient time-series analysis and cross-user comparisons, which are computationally expensive with traditional SQL-based architectures. Detecting subtle deviations that may indicate health concerns requires both statistical rigor and computational efficiency.
- Third, grouping individuals based on their health behaviors requires clustering algorithms and similarity measures that benefit from graph-based representations, allowing for multi-dimensional analysis of interconnected health metrics.

By integrating SQL with Neo4j, our system addresses these challenges by combining structured data storage with graph-based relationship analysis, providing an efficient framework for anomaly detection and clustering.

1.2 Objectives

The primary objectives of this project are as follows:

1. **Efficient Health Data Storage**: Design and implement a hybrid architecture combining PostgreSQL and Neo4j to handle both structured health metrics and relationship-based analysis. The system should maintain consistent performance even as data volume scales to millions of records.
2. **Anomaly Detection**: Identify abnormal patterns in heart rate, sleep, and activity using graph-based Cypher queries and statistical thresholds. The detection system should achieve at least 75% accuracy when compared with manually labeled anomalies.
3. **User Clustering**: Perform clustering analysis to group users with similar health characteristics, enabling cohort-based analytics and personalized insights. The clustering should reveal at least three distinct user profiles with clearly differentiated health behaviors.
4. **Performance Evaluation**: Compare the efficiency of the hybrid SQL-Neo4j approach against traditional SQL-only implementations for complex health analytics queries.

1.3 Dataset Used

Kaggle – Fitbit Dataset - Link
Content:

- Minute-level and hourly data: Steps, intensity, METs, sleep, and heart rate.

- Daily aggregates: Total steps, calories, sleep, and active time.
- User demographics: BMI, weight, age, and lifestyle factors.

Role:

- Used for real-world health metrics, providing baseline data for graph-based insights.
- Enabled realistic pattern modeling and anomaly detection.

Enhanced with Synthetic Data
Why:

- Expanded dataset to increase diversity and simulate rare health scenarios.

How:

- Generated realistic variations in heart rate, steps, calories, BMI, and sleep using Python.

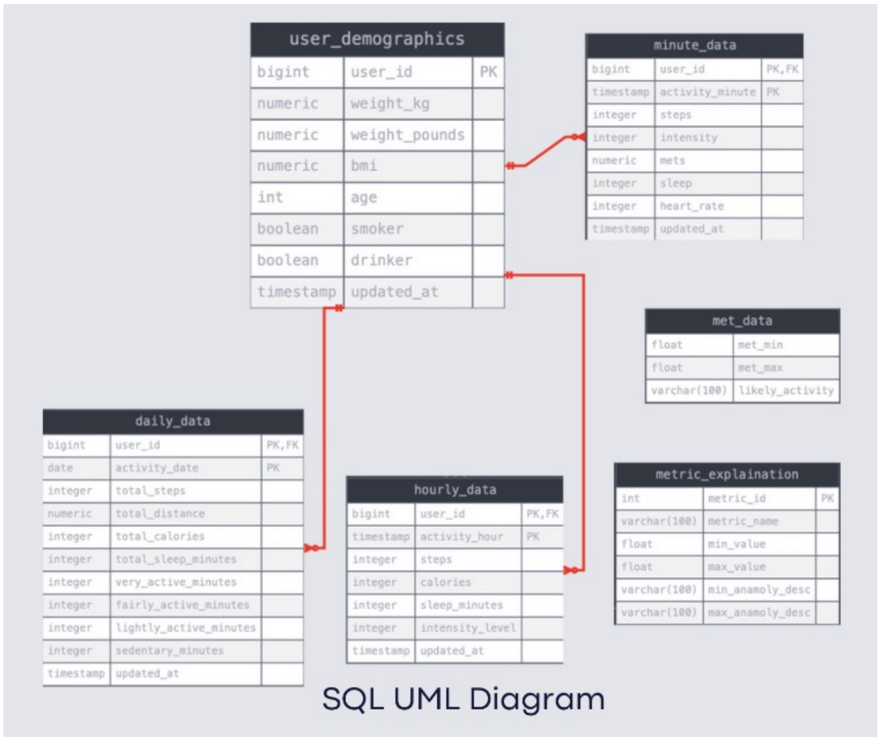
Role:

- Boosted dataset size for robust testing.
- Simulated edge cases and improved anomaly detection.

2 Database Design and Schema

2.1 PostgreSQL Schema Design

The SQL schema is designed to handle both raw and aggregated metrics efficiently. It consists of three main tables, as shown in Image 1.



user_demographics		
bigint	user_id	PK
numeric	weight_kg	
numeric	weight_pounds	
numeric	bmi	
int	age	
boolean	smoker	
boolean	drinker	
timestamp	updated_at	

minute_data		
bigint	user_id	PK,FK
timestamp	activity_minute	PK
integer	steps	
integer	intensity	
numeric	mets	
integer	sleep	
integer	heart_rate	
timestamp	updated_at	

met_data	
float	met_min
float	met_max
varchar(100)	likely_activity

daily_data		
bigint	user_id	PK,FK
date	activity_date	PK
integer	total_steps	
numeric	total_distance	
integer	total_calories	
integer	total_sleep_minutes	
integer	very_active_minutes	
integer	fairly_active_minutes	
integer	lightly_active_minutes	
integer	sedentary_minutes	
timestamp	updated_at	

hourly_data		
bigint	user_id	PK,FK
timestamp	activity_hour	PK
integer	steps	
integer	calories	
integer	sleep_minutes	
integer	intensity_level	
timestamp	updated_at	

metric_explaination		
int	metric_id	PK
varchar(100)	metric_name	
float	min_value	
float	max_value	
varchar(100)	min_anamoly_desc	
varchar(100)	max_anamoly_desc	

Figure 1: High-level architecture of the Health Analytics Platform, showing data flow between PostgreSQL, AWS Lambda, and Neo4j components.

The schema includes the following tables:

- `user_demographics`: Stores static user attributes such as age, BMI, weight, and lifestyle habits (e.g., smoker or drinker status).
- `minute_data`: Contains minute-level health metrics for individual users, including step count, intensity, METs (Metabolic Equivalent of Task), sleep duration, and heart rate.
- `hourly_data`: Aggregates minute-level data into hourly summaries, providing a coarser time granularity for improved query performance.
- `daily_data`: Summarizes hourly data into daily-level metrics, offering a high-level overview of user activity, sleep, and calorie expenditure.
- `met_data`: Contains MET range definitions and the likely activity associated with each range, aiding in activity classification.
- `metric_explanation`: Provides metadata for health metrics, including metric descriptions and thresholds for anomaly detection.

To optimize performance, the schema uses indexes on timestamp columns and user IDs, enabling fast retrieval of specific records during aggregation operations.

2.1.1 PostgreSQL Triggers for Lambda Integration

To ensure real-time synchronization between PostgreSQL and Neo4j, we implement PostgreSQL triggers that invoke AWS Lambda functions whenever relevant tables are modified. This allows for immediate updates in the graph database whenever new data is inserted or existing records are updated in PostgreSQL.

AWS Lambda Extension First, we enable the 'aws_lambda' extension to allow PostgreSQL to communicate with AWS Lambda. This extension provides the capability to invoke Lambda functions directly from PostgreSQL, enabling seamless integration between the relational database and the Neo4j graph database.

```
CREATE EXTENSION IF NOT EXISTS aws_lambda CASCADE;
```

Trigger for Daily Data Updates The 'daily_data_trigger' ensures that any insertion or update in the 'daily_data' table automatically triggers the 'neo4j-update-function' Lambda. This function takes the 'user_id' and the event type ('INSERT' or 'UPDATE') as parameters, allowing Neo4j to synchronize with the latest health metrics in real time.
Trigger Function Definition:

```
CREATE OR REPLACE FUNCTION trigger_daily_data_lambda()
RETURNS trigger AS $$
BEGIN
    PERFORM aws_lambda.invoke(
        'arn:aws:lambda:us-west-1:495599758473:function:neo4j-update-function',
        json_build_object(
            'user_id', NEW.user_id,
            'event_type', TG_OP   -- INSERT or UPDATE
        )::jsonb,
        'us-west-1',
        'Event'
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Execution Definition:

```
DROP TRIGGER IF EXISTS daily_data_trigger ON daily_data;

CREATE TRIGGER daily_data_trigger
AFTER INSERT OR UPDATE
ON daily_data
FOR EACH ROW
EXECUTE FUNCTION trigger_daily_data_lambda();
```

Trigger for User Demographics Updates The 'user_demographics_trigger' automatically invokes the Lambda function when user demographic records are inserted or updated. This ensures that changes in user profiles, such as BMI, age, or lifestyle indicators, are immediately reflected in the Neo4j graph.
Trigger Function Definition:

```
CREATE OR REPLACE FUNCTION trigger_user_demographics_lambda()
RETURNS trigger AS $$
BEGIN
    PERFORM aws_lambda.invoke(
        'arn:aws:lambda:us-west-1:495599758473:function:neo4j-update-function',
        json_build_object(
            'user_id', NEW.user_id,
            'event_type', TG_OP   -- INSERT or UPDATE
        )::jsonb,
        'us-west-1',
        'Event'
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Execution Definition:

```
DROP TRIGGER IF EXISTS user_demographics_trigger ON user_demographics;

CREATE TRIGGER user_demographics_trigger
AFTER INSERT OR UPDATE
ON user_demographics
FOR EACH ROW
EXECUTE FUNCTION trigger_user_demographics_lambda();
```

Technical Rationale and Benefits The PostgreSQL triggers enable automatic data synchronization between the relational database and the Neo4j graph. This architecture offers several benefits:

- Real-time Data Consistency: Any changes made in PostgreSQL are instantly reflected in Neo4j, ensuring data integrity across both platforms.
- Reduced Latency: With immediate Lambda invocation, there is minimal lag between SQL operations and Neo4j updates, making the system suitable for real-time anomaly detection and analysis.
- Decoupled Architecture: By using AWS Lambda, the PostgreSQL and Neo4j systems remain loosely coupled, promoting scalability and flexibility.

This trigger-based architecture ensures seamless, real-time data consistency between PostgreSQL and Neo4j, forming the foundation for accurate, up-to-date graph-based health analysis.

2.2 Graph Schema in Neo4j

In the graph representation, we model the relationships between users and their health metrics using the schema illustrated in Figure 2.

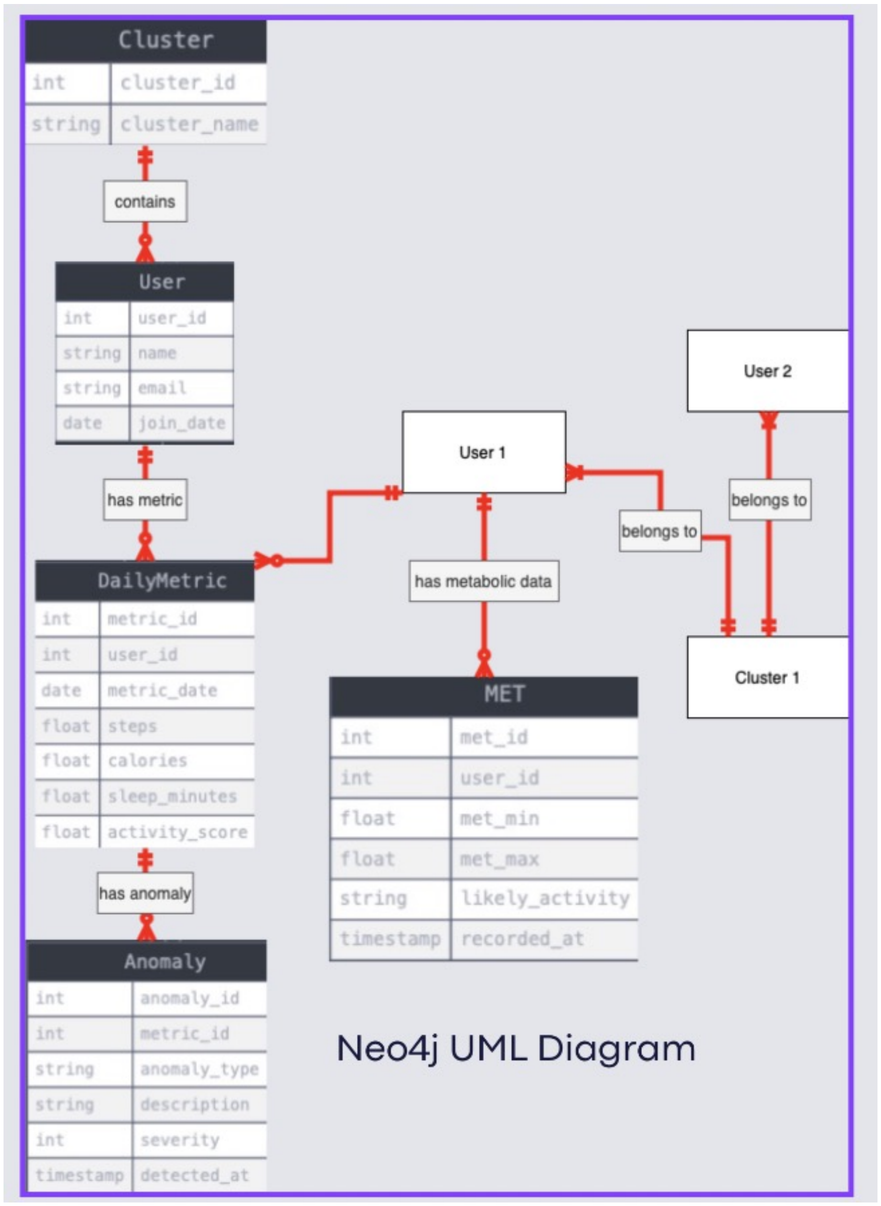


Figure 2: Neo4j graph schema showing User and DailyMetric nodes with their relationships and properties.

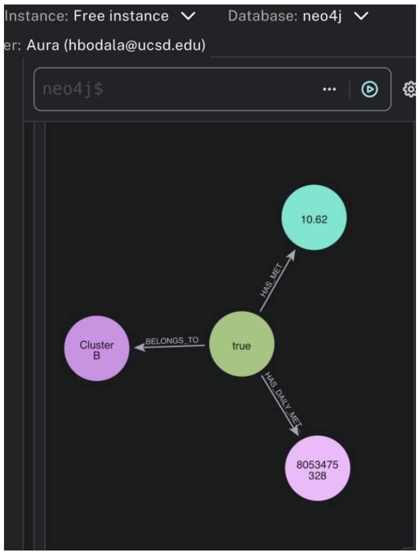
The graph schema includes:

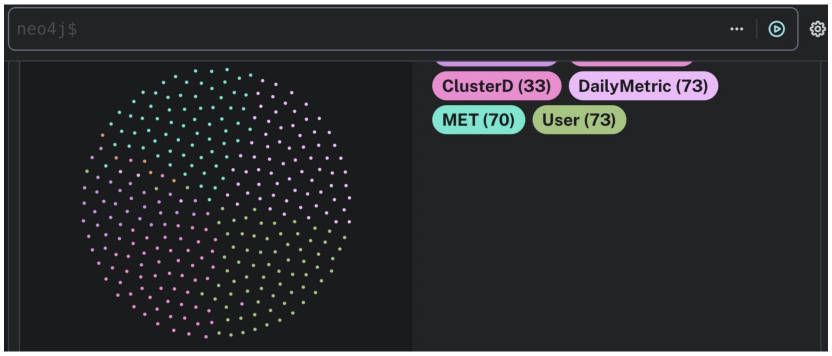
- **Nodes**:
  - `User`: Represents individual users, identified by a unique user ID. Each User node contains demographic properties such as age, gender, height, and weight, enabling demographic-based filtering and analysis.
  - `DailyMetric`: Represents daily health summaries. Each DailyMetric node contains properties such as date, total steps, sleep duration, average heart rate, and calorie expenditure. Additional derived metrics include activity intensity distributions (sedentary, light, moderate, vigorous) and sleep quality scores.
  - `Anomaly`: Represents health anomalies detected in the user's daily metrics. Each Anomaly node contains properties such as `metric` (the health metric where the anomaly occurred, e.g., `sleep_duration`, `calorie_expenditure`, `heart_rate`), `anomaly_type` (indicates whether the anomaly

is `low` or `high` based on thresholds), `anomaly_description` (detailed description of the anomaly and its potential health implications), and `severity_score` (a score representing the severity of the anomaly, used for prioritizing health recommendations).
  - `MET`: Represents Metabolic Equivalent of Task (MET) values associated with different time zones. Each MET node contains properties such as `time_zone` (the time period during the day: `early morning, morning, afternoon, evening`), `avg_met` (the average MET value for the time zone), `likely_activity` (the most common activity associated with the MET range, providing insights into user behavior patterns), and `trend` (indicates whether the user's MET values are `increasing, decreasing`, or stable over time, used for predicting activity trends).

- **Relationships**:
  - `HAS_METRIC`: Links each user to their daily metrics. This relationship includes a `date` property for efficient temporal filtering.
  - `BELONGS_TO`: Represents cluster membership for users, connecting User nodes to Cluster nodes. This relationship includes a `similarity_score` property indicating the strength of cluster membership.
  - `HAS_METABOLIC_DATA`: Connects User nodes to MetabolicProfile nodes, containing properties such as `basal_metabolic_rate`, `glucose_level`, `triglyceride_level`, and `metabolic_age`. This relationship includes a `measurement_date` property to track when metabolic data was collected and a `data_quality` score indicating reliability.
  - `HAS_ANOMALY`: Links DailyMetric nodes to Anomaly nodes when the system detects an unusual pattern. This relationship includes properties such as `detection_timestamp`, `anomaly_score`, `confidence_level`, and `anomaly_type` (e.g., 'sleep', 'heart_rate', 'activity'). This allows for efficient querying of specific anomaly categories across the user population.

The graph structure allows for efficient anomaly detection by traversing user-metric relationships and applying filtering conditions directly within Cypher queries. For example, finding users with consistently abnormal sleep patterns becomes a simple graph traversal operation rather than a complex SQL join.





3 System Architecture

3.1 Technology Stack

The platform utilizes the following technologies:

- **Database Technologies**:
  - PostgreSQL 13 with TimescaleDB extension for time-series optimization
  - Neo4j 4.4 Enterprise Edition for graph storage and analysis
  - Redis 6.2 for API response caching
- **Cloud Infrastructure**:
  - AWS RDS for PostgreSQL hosting
  - AWS Lambda for serverless computations
  - AWS S3 for storing analytical results and reports
- **Programming Languages and Frameworks**:
  - Python 3.9 for Lambda functions and API development
  - FastAPI for RESTful API implementation
  - Pandas and NumPy for data manipulation
  - Streamlit for front-end development

3.2 Overview of the Architecture

The Health Analytics Platform adopts a modular architecture designed for scalable data processing and real-time analytics. Figure 4 illustrates the high-level components and data flow within the system.

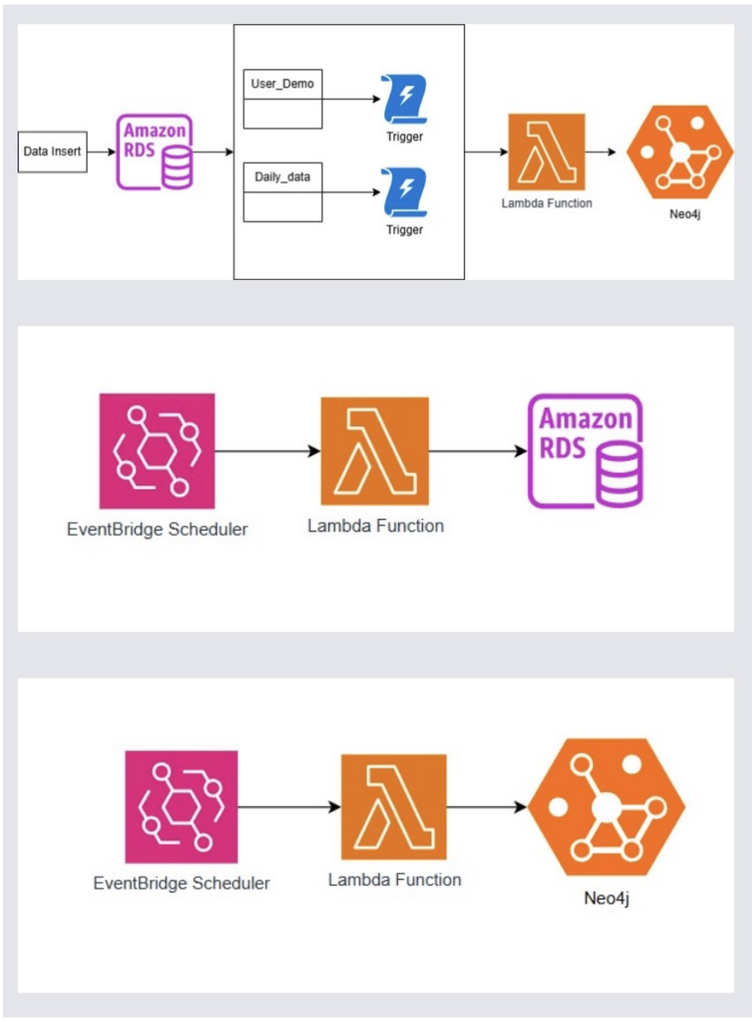


Figure 4: High-level architecture of the Health Analytics Platform, showing data flow between PostgreSQL, AWS Lambda, and Neo4j components.

The architecture consists of the following primary components:

PostgreSQL: PostgreSQL serves as the core RDBMS for storing raw and aggregated health metrics. It handles the structured data operations, such as inserts, queries, and aggregations. The database is configured with TimescaleDB extension to optimize time-series data handling, particularly for the minute-level health metrics.

AWS Lambda: To automate the data aggregation process, serverless AWS Lambda functions periodically retrieve data from PostgreSQL, perform aggregations, and push summarized data into Neo4j. Three separate Lambda functions handle minute-to-hourly, hourly-to-daily, and daily-to-graph transformations, executing at 10-minute, hourly, and daily intervals respectively.

Neo4j Graph Database: Neo4j is used to model user relationships with their daily health metrics, enabling graph traversal operations for anomaly detection and clustering. The graph database stores only aggregated daily metrics to maintain performance, while detailed historical data remains in PostgreSQL.

API Interface: A Python-based API interacts with both PostgreSQL and Neo4j, facilitating data retrieval and analytical operations. The API implements caching mechanisms using Redis to minimize database load for frequently executed queries.

3.3 Functionality and Scheduler Overview

The architecture leverages a combination of AWS Lambda functions and schedulers to automate data processing, anomaly detection, and clustering operations. These components ensure continuous updates and synchronization between PostgreSQL and Neo4j, while also managing the clustering workflows.

3.3.1 AWS Lambda Functions

The following Lambda functions handle various data operations:

- **daily-update-function**: This function triggers daily data aggregation and anomaly detection routines. It processes the latest daily metrics from PostgreSQL and pushes them to Neo4j for anomaly identification and further analysis.
- **neo4j-update-function**: Responsible for synchronizing PostgreSQL updates with Neo4j in real time. It captures INSERT and UPDATE triggers from PostgreSQL, ensuring the graph database reflects the latest metrics and demographic changes.
- **clustering-neo4j**: Performs clustering operations on the Neo4j dataset. It identifies groups of users with similar health patterns and stores the results back into Neo4j for further analysis and visualization.
- **hourly-data-update**: Executes hourly updates by aggregating minute-level data into hourly metrics. This function optimizes the granularity of the data, improving the efficiency of analytical queries.

3.3.2 Schedulers

To automate recurring data processing, the architecture uses AWS EventBridge schedulers:

- **cluster-update**: Periodically triggers the clustering-neo4j function, ensuring that the clustering model is recalculated with the latest health data.
- **daily-update**: Executes the daily-update-function at a fixed time each day, processing new daily metrics and detecting anomalies.
- **hourly-update**: Runs the hourly-data-update function at hourly intervals, aggregating minute-level data and pushing hourly summaries to Neo4j.

This combination of Lambda functions and schedulers enables automated, near-real-time data processing, anomaly detection, and clustering, ensuring the system remains up-to-date with minimal manual intervention.

3.4 `neo4j_daily_sync`

The `neo4j_daily_sync` Lambda function is responsible for synchronizing user health data between a PostgreSQL database and a Neo4j graph database. This function ensures that:

- New users or demographic updates are reflected in Neo4j.
- Daily health metrics and MET (Metabolic Equivalent of Task) values are fetched from PostgreSQL and updated in Neo4j.
- The function is triggered by events such as data insertions or updates in the PostgreSQL database.

3.4.1 High-Level Execution Flow

The Lambda function follows these key steps:

1. Event Trigger:Triggered by data insertions or updates in PostgreSQL.
2. User Existence Check:
   - If the user does not exist in Neo4j, it fetches demographic details from PostgreSQL and creates a new `User` node.
   - If the user already exists, it proceeds with the next steps.
3. Daily Metrics Sync:
   - Fetches the latest daily metrics from PostgreSQL.
   - Updates the `DailyMetric` node in Neo4j.
4. MET Values Sync:
   - Retrieves MET values split by time zones (early morning, morning, afternoon, evening) from PostgreSQL.
   - Updates the corresponding `MET` node in Neo4j.

3.4.2 SQL Queries and Their Explanations

Fetching Daily Metrics

The following SQL query fetches the latest daily metrics for a given user from the PostgreSQL database:

```
WITH latest_metrics AS (
    SELECT
        d.user_id,
        d.activity_date,
        d.total_steps,
        d.total_calories,
        d.total_sleep_minutes,
        (
            SELECT ROUND(AVG(h.intensity_level), 2)
            FROM hourly_data h
            WHERE d.user_id = h.user_id
              AND DATE(h.activity_hour) = d.activity_date
        ) AS avg_intensity,
        (
            SELECT mode() WITHIN GROUP (ORDER BY m.heart_rate)
            FROM minute_data m
            WHERE d.user_id = m.user_id
              AND DATE(m.activity_minute) = d.activity_date
        ) AS resting_hr
    FROM daily_data d
    WHERE d.user_id = %s
    ORDER BY d.activity_date DESC
    LIMIT 1
)
SELECT * FROM latest_metrics;
```

Explanation:

- The query uses a Common Table Expression (CTE) named `latest_metrics` to extract the most recent daily metrics for the user.
- It computes:
  - Average intensity from the `hourly_data`.
  - Resting heart rate using the mode of heart rate values from the `minute_data`.
- The final `SELECT` statement retrieves the metrics from the CTE.

Fetching MET Values

The following SQL query retrieves MET averages split by time zones for the past 7 days:

```
WITH latest_user_date AS (
    SELECT DATE(MAX(activity_minute)) AS latest_date
    FROM minute_data
    WHERE user_id = %s
),
met_by_timezone AS (
    SELECT
        m.user_id,
        ROUND(AVG(CASE WHEN EXTRACT(HOUR FROM m.activity_minute) BETWEEN 0 AND 5 THEN m.mets
    ELSE NULL END), 2) AS early_morning_avg_met,
        ROUND(AVG(CASE WHEN EXTRACT(HOUR FROM m.activity_minute) BETWEEN 6 AND 11 THEN m.mets
    ELSE NULL END), 2) AS morning_avg_met,
        ROUND(AVG(CASE WHEN EXTRACT(HOUR FROM m.activity_minute) BETWEEN 12 AND 17 THEN m.mets
    ELSE NULL END), 2) AS afternoon_avg_met,
        ROUND(AVG(CASE WHEN EXTRACT(HOUR FROM m.activity_minute) BETWEEN 18 AND 23 THEN m.mets
    ELSE NULL END), 2) AS evening_avg_met
    FROM minute_data m
    JOIN latest_user_date l ON DATE(m.activity_minute) BETWEEN l.latest_date - INTERVAL '6␣days'
    AND l.latest_date
    WHERE m.user_id = %s
    GROUP BY m.user_id
)
```

Explanation:

- The query uses two CTEs:
  - `latest_user_date`: Retrieves the latest activity date for the given user.
  - `met_by_timezone`: Aggregates MET averages by time zones (early morning, morning, afternoon, and evening).
- It groups the results by `user_id`.

3.4.3 Cypher Queries and Their Explanations

User Existence Check

```
OPTIONAL MATCH (u:User {user_id: $user_id})
RETURN
    CASE
        WHEN u IS NOT NULL THEN "User␣Found"
        ELSE "User␣Not␣Found"
    END AS result
```

Explanation:

- The Cypher query uses `OPTIONAL MATCH` to check for the existence of a user.
- It returns whether the user exists or not.

User Creation or Update

```
MERGE (u:User {user_id: $user_id})
SET u.age = coalesce($age, u.age),
u.bmi = coalesce($bmi, u.bmi),
u.smoker = coalesce($smoker, u.smoker),
u.drinker = coalesce($drinker, u.drinker),
u.updated_at = date($updated_at)
```

Explanation:

- The `MERGE` statement ensures that a `User` node with the given `user_id` exists. If it does not exist, it creates a new one.
- The `SET` clause updates the user's attributes.
- `coalesce()` ensures that if a provided value is `NULL`, the existing value in the database is retained instead of being overwritten with `NULL`.
- The `updated_at` field is set to the current date.

Daily Metrics Update

```
MATCH (u:User {user_id: $user_id})
        MERGE (dm:DailyMetric {user_id: $user_id})
        SET dm.activity_date = coalesce(date($activity_date), dm.activity_date),
            dm.total_steps = coalesce($total_steps, dm.total_steps),
            dm.total_calories = coalesce($total_calories, dm.total_calories),
            dm.total_sleep_minutes = coalesce($total_sleep_minutes, dm.total_sleep_minutes),
            dm.average_intensity = coalesce($average_intensity, dm.average_intensity),
            dm.resting_hr = coalesce($resting_hr, dm.resting_hr)
        MERGE (u)-[:HAS_DAILY_METRIC]->(dm)
        RETURN dm
```

Explanation:

- The `MATCH` clause ensures that the user exists in the database.
- The `MERGE` clause ensures that a `DailyMetric` node exists for the given `user_id` (or creates it if it does not exist).
- The `SET` clause updates daily metric attributes while preserving existing values if new ones are missing using `coalesce()`.
- The `MERGE` clause establishes the `HAS_DAILY_METRIC` relationship between the user and their daily metrics.
- The query returns the updated daily metric node.

User Creation or Update

```
MATCH (u:User {user_id: $user_id})
        MERGE (m:MET {user_id: $user_id})
        SET m.early_morning_avg_met = coalesce($early_morning_avg_met, m.early_morning_avg_met),
            m.morning_avg_met = coalesce($morning_avg_met, m.morning_avg_met),
            m.afternoon_avg_met = coalesce($afternoon_avg_met, m.afternoon_avg_met),
            m.evening_avg_met = coalesce($evening_avg_met, m.evening_avg_met)
        MERGE (u)-[:HAS_MET]->(m)
        RETURN m
```

Explanation:

- The `MATCH` clause ensures that the user exists in the database.

- The `MERGE` clause ensures that a `MET` node exists for the given `user_id` (or creates it if it does not exist).
- The `SET` clause updates MET values while using `coalesce()` to prevent overwriting existing values with `NULL`.
- The `MERGE` clause establishes the `HAS_MET` relationship between the user and their MET data.
- The query returns the updated MET node.

3.5 `daily_update`

The `daily_update` Lambda function is responsible for aggregating and updating daily health metrics for users. It runs automatically every 24 hours using an Amazon EventBridge rule. This function ensures that:

- Hourly health data is aggregated into daily summaries.
- The function computes total steps, calories burned, distance traveled, and sleep minutes.
- Activity intensity levels (very active, fairly active, lightly active, and sedentary minutes) are classified and stored.
- The data is inserted into the `daily_data` table, ensuring no duplicate entries.

3.5.1 High-Level Execution Flow

The Lambda function follows these key steps:

1. Event Trigger: Invoked every 24 hours by Amazon EventBridge.
2. Data Aggregation:
    - Retrieves hourly data from the `hourly_data` table.
    - Computes daily aggregates for steps, calories, distance, sleep, and activity intensities.
3. Data Insertion:
    - Inserts aggregated daily data into the `daily_data` table.
    - Ensures existing records are updated using the `ON CONFLICT` clause.

3.5.2 SQL Query and Explanation

The following SQL query aggregates hourly data into daily summaries and updates the `daily_data` table:

```sql
INSERT INTO daily_data (
    user_id,
    activity_date,
    total_steps,
    total_distance,
    total_calories,
    total_sleep_minutes,
    very_active_minutes,
    fairly_active_minutes,
    lightly_active_minutes,
    sedentary_minutes,
    updated_at
)
SELECT
    h.user_id,
    DATE(h.activity_hour) AS activity_date,
    SUM(h.steps) AS total_steps,
    ROUND(SUM(h.steps) * 0.000762, 2) AS total_distance,
    SUM(h.calories) AS total_calories,
    SUM(h.sleep_minutes) AS total_sleep_minutes,

    SUM(CASE WHEN h.intensity_level >= 75 THEN 1 ELSE 0 END) AS very_active_minutes,
    SUM(CASE WHEN h.intensity_level BETWEEN 40 AND 74 THEN 1 ELSE 0 END) AS fairly_active_minutes,
    SUM(CASE WHEN h.intensity_level BETWEEN 20 AND 39 THEN 1 ELSE 0 END) AS lightly_active_minutes,
    SUM(CASE WHEN h.intensity_level < 20 THEN 1 ELSE 0 END) AS sedentary_minutes,

    NOW() AS updated_at
FROM hourly_data h
LEFT JOIN daily_data d
    ON h.user_id = d.user_id
    AND DATE(h.activity_hour) = d.activity_date
WHERE d.activity_date IS NULL
GROUP BY h.user_id, activity_date
ON CONFLICT (user_id, activity_date) DO UPDATE
SET
    total_steps = EXCLUDED.total_steps,
    total_distance = EXCLUDED.total_distance,
    total_calories = EXCLUDED.total_calories,
    total_sleep_minutes = EXCLUDED.total_sleep_minutes,
    very_active_minutes = EXCLUDED.very_active_minutes,
    fairly_active_minutes = EXCLUDED.fairly_active_minutes,
    lightly_active_minutes = EXCLUDED.lightly_active_minutes,
    sedentary_minutes = EXCLUDED.sedentary_minutes,
    updated_at = EXCLUDED.updated_at;
```

Explanation

- The query inserts aggregated daily health data into the `daily_data` table.
- It retrieves data from the `hourly_data` table, grouping records by `user_id` and `activity_date` (derived from `activity_hour`).
- The `SUM()` function is used to compute total steps, total calories, and sleep minutes over the day.
- The `ROUND()` function calculates the total distance in kilometers using a conversion factor (0.000762 per step).
- Activity intensity is categorized:
  - **Very Active Minutes**: `intensity_level` $\geq$ 75.
  - **Fairly Active Minutes**: `intensity_level` between 40 and 74.
  - **Lightly Active Minutes**: `intensity_level` between 20 and 39.
  - **Sedentary Minutes**: `intensity_level` $<$ 20.
- A `LEFT JOIN` is used to check if the record already exists in `daily_data`.
- The `WHERE d.activity_date IS NULL` condition ensures that only new daily records are inserted.
- The `ON CONFLICT (user_id, activity_date)` clause ensures that if a record already exists, the daily metrics are updated instead of inserting duplicates.

3.6 hourly_update

The `hourly_update` Lambda function is responsible for aggregating and updating hourly health metrics for users. It runs automatically every hour using an Amazon EventBridge rule. This function ensures that:

- Minute-level health data is aggregated into hourly summaries.
- The function computes total steps, calories burned, sleep minutes, and the average intensity level.
- The data is inserted into the `hourly_data` table, ensuring no duplicate entries.

3.6.1 High-Level Execution Flow

The Lambda function follows these key steps:

1. **Event Trigger**: Invoked every hour by Amazon EventBridge.
2. **Data Aggregation**:
   - Retrieves minute-level data from the `minute_data` table.
   - Computes hourly aggregates for steps, calories, sleep, and intensity.
3. **Data Insertion**:
   - Inserts aggregated hourly data into the `hourly_data` table.
   - Ensures existing records are updated using the `ON CONFLICT` clause.

3.6.2 SQL Query and Explanation

The following SQL query aggregates minute-level data into hourly summaries and updates the `hourly_data` table:

```
INSERT INTO hourly_data (user_id, activity_hour, steps, calories, sleep_minutes, intensity_level, updated_a
SELECT
    m.user_id,
    DATE_TRUNC('hour', m.activity_minute) AS activity_hour,
    SUM(m.steps) AS steps,
    SUM(m.calories) AS calories,
    SUM(m.sleep) AS sleep_minutes,
    ROUND(AVG(m.intensity), 2) AS intensity_level,
    NOW() AS updated_at
FROM minute_data m
LEFT JOIN hourly_data h
    ON m.user_id = h.user_id
    AND DATE_TRUNC('hour', m.activity_minute) = h.activity_hour
WHERE h.activity_hour IS NULL
GROUP BY m.user_id, activity_hour
ON CONFLICT (user_id, activity_hour) DO UPDATE
SET
    steps = EXCLUDED.steps,
    calories = EXCLUDED.calories,
    sleep_minutes = EXCLUDED.sleep_minutes,
    intensity_level = EXCLUDED.intensity_level,
    updated_at = EXCLUDED.updated_at;
```

Explanation

- The query inserts aggregated hourly health data into the `hourly_data` table.
- It retrieves data from the `minute_data` table, grouping records by `user_id` and `activity_hour` (derived from `activity_minute`).
- The `SUM()` function is used to compute total steps, total calories, and sleep minutes over the hour.
- The `ROUND(AVG(intensity), 2)` function calculates the average intensity level for the hour.
- A `LEFT JOIN` is used to check if the record already exists in `hourly_data`.
- The `WHERE h.activity_hour IS NULL` condition ensures that only new hourly records are inserted.
- The `ON CONFLICT (user_id, activity_hour)` clause ensures that if a record already exists, the hourly metrics are updated instead of inserting duplicates.

3.7 clustering_neo4j

The clustering_neo4j Lambda function is responsible for clustering users in Neo4j based on their demographic attributes, specifically their age. The function ensures that:

- Predefined cluster nodes (Cluster A, Cluster B, Cluster C, Cluster D) exist in the Neo4j database.
- Existing cluster assignments are cleared before reassignment.
- Users are assigned to appropriate clusters based on their age range.
- This function is invoked using AWS EventBridge at a frequency of 48 hours

3.7.1 High-Level Execution Flow

The function follows these key steps:

1. Cluster Node Initialization:
    - Ensures that the cluster nodes (A, B, C, D) exist in Neo4j.
2. Clearing Previous Assignments:
    - Deletes existing BELONGS_TO relationships to allow reassignment.
3. User Clustering Based on Age:
    - Assigns users to specific clusters based on their age range.

3.7.2 Cypher Queries and Their Explanations

Cluster Node Initialization

```
MERGE (a:ClusterA {name: "Cluster␣A"})
MERGE (b:ClusterB {name: "Cluster␣B"})
MERGE (c:ClusterC {name: "Cluster␣C"})
MERGE (d:ClusterD {name: "Cluster␣D"})
```

Explanation:

- The MERGE clause ensures that cluster nodes exist in the database.
- If the clusters already exist, no duplicates are created.

Clearing Previous Cluster Assignments

```
MATCH (u:User)-[r:BELONGS_TO]->(c)
DELETE r
```

Explanation:

- The MATCH clause finds all users with existing cluster assignments.
- The DELETE statement removes all BELONGS_TO relationships, allowing fresh assignments.

Assigning Users to Clusters Based on Age

```
MATCH (u:User)
WHERE u.age IS NOT NULL
FOREACH (_ IN CASE WHEN toInteger(u.age) >= 30 AND toInteger(u.age) < 35 THEN [1] ELSE [] END |
    MERGE (u)-[:BELONGS_TO]->(a:ClusterA {name: "Cluster␣A"})
)
FOREACH (_ IN CASE WHEN toInteger(u.age) >= 35 AND toInteger(u.age) <= 45 THEN [1] ELSE [] END |
    MERGE (u)-[:BELONGS_TO]->(b:ClusterB {name: "Cluster␣B"})
)
FOREACH (_ IN CASE WHEN toInteger(u.age) >= 46 AND toInteger(u.age) <= 55 THEN [1] ELSE [] END |
    MERGE (u)-[:BELONGS_TO]->(c:ClusterC {name: "Cluster␣C"})
)
FOREACH (_ IN CASE WHEN toInteger(u.age) >= 50 THEN [1] ELSE [] END |
    MERGE (u)-[:BELONGS_TO]->(d:ClusterD {name: "Cluster␣D"})
)
```

Explanation:

- The `MATCH` clause selects all users who have a valid age.
- The `FOREACH` clause iterates over a conditional list (`[1]` if the condition is met, `[]` otherwise).
- `MERGE` ensures that users are assigned to their respective clusters.
- Users aged:
  - 30-34 belong to `Cluster A`.
  - 35-45 belong to `Cluster B`.
  - 46-55 belong to `Cluster C`.
  - 50+ belong to `Cluster D` (potential overlap with Cluster C).

4 Health Metric Dashboard

4.1 Overview of `app.py`

This Streamlit app serves as a Fitbit Health Dashboard that allows users to visualize and track their health metrics (steps, calories, sleep, heart rate, intensity, etc.) based on data stored in PostgreSQL (AWS RDS) and Neo4j. It provides real-time and historical analysis, detects anomalies, and offers personalized health recommendations based on user clustering in Neo4j.

The app performs the following tasks:

- **User Authentication** - Users enter their User ID to access their health data.
- **Data Fetching** - Retrieves user data from PostgreSQL (daily_data, hourly_data, minute_data).
- **Anomaly Detection** - Compares health metrics against predefined thresholds.
- **Historical Data Visualization** - Plots trends of selected health metrics.
- **MET (Metabolic Equivalent of Task) Insights** - Fetches 7-day MET data from Neo4j.
- **Health Recommendations** - Uses Neo4j clustering to provide personalized suggestions.

4.2 Explanation of Queries and Cyphers

4.2.1 PostgreSQL Queries (SQL)

Fetch Available Date Range for the User

```
SELECT MIN(activity_date), MAX(activity_date)
FROM daily_data
WHERE user_id = %s
```

Explanation:

- This query determines the first and last recorded dates for the given user.
- Used to set the date range selector in the UI.

Find Latest Recorded Hour for the Selected Date

```
SELECT MAX(EXTRACT(HOUR FROM activity_hour))
FROM hourly_data
WHERE user_id = %s AND activity_hour::DATE = %s
```

Explanation:

- Extracts the latest recorded hour for the user on the selected date.

- Used to aggregate hourly data into daily data when daily stats are not available.

Fetch Daily Stats (or Aggregate Hourly Data If Missing)

```
    SELECT total_steps, total_calories, total_sleep_minutes,
       (
        SELECT mode() WITHIN GROUP (ORDER BY heart_rate)
        FROM minute_data
        WHERE user_id = %s AND activity_minute::DATE = %s
       ) AS resting_hr
FROM daily_data WHERE user_id = %s AND activity_date = %s
```

Explanation:

- Fetches total steps, calories, sleep minutes, and resting heart rate for a given date.
- Uses mode() to find the most common resting heart rate.
- If no daily data exists, an hourly sum is computed instead.

```
        SELECT SUM(steps) AS total_steps, SUM(calories) AS total_calories,
           SUM(sleep_minutes) AS total_sleep_minutes,
           (
            SELECT mode() WITHIN GROUP (ORDER BY heart_rate)
            FROM minute_data
            WHERE user_id = %s AND activity_minute::DATE = %s
           ) AS resting_hr
    FROM hourly_data WHERE user_id = %s AND activity_hour BETWEEN %s AND %s
```

- Aggregates hourly data into daily stats.
- Ensures data up to the latest recorded hour is included.

Fetch Average Intensity for the Selected Date

```
    SELECT ROUND(AVG(intensity_level), 2) AS avg_intensity
    FROM hourly_data
    WHERE user_id = %s AND activity_hour::DATE = %s
```

Explanation:

- Calculates the average intensity level for the selected day.

Fetch Anomaly Thresholds

```
    SELECT metric_name, min_value, max_value
    FROM metric_explaination
    WHERE metric_name IN ('Sleep␣(Hours)',
    'Heart␣Rate␣(Resting␣bpm)', 'Intensity␣(HRV␣in␣ms)', 'Calories␣(kcal/day)')
```

Explanation:

- Retrieves threshold values for key health metrics.
- Used for anomaly detection.

Fetch Historical Data with Intensity

```
    SELECT activity_date, total_steps, total_calories, total_sleep_minutes,
       (
        SELECT mode() WITHIN GROUP (ORDER BY heart_rate)
        FROM minute_data
        WHERE user_id = %s AND activity_minute::DATE = d.activity_date
       ) AS resting_hr,
       (
        SELECT ROUND(AVG(intensity_level), 2)
        FROM hourly_data h
        WHERE h.user_id = d.user_id
        AND DATE(h.activity_hour) = d.activity_date
       ) AS avg_intensity
FROM daily_data d WHERE user_id = %s ORDER BY activity_date
```

Explanation:

- Fetches historical data for steps, calories, sleep, resting HR, and intensity.
- Uses subqueries to calculate resting heart rate (mode) and average intensity.
- Ordered by activity date for visualization.

4.2.2 Neo4j Queries (Cypher)

Fetch Last 7 Days MET Insights

```
    MATCH (u:User {user_id: $user_id})-[:HAS_MET]->(m:MET)
RETURN
    u.user_id AS user_id,
    m.early_morning_avg_met AS early_morning_avg_met, m.early_morning_likely_activity
    AS early_morning_likely_activity,
    m.morning_avg_met AS morning_avg_met, m.morning_likely_activity AS morning_likely_activity,
    m.afternoon_avg_met AS afternoon_avg_met, m.afternoon_likely_activity AS afternoon_likely_activity,
    m.evening_avg_met AS evening_avg_met, m.evening_likely_activity AS evening_likely_activity
```

Explanation:

- Fetches 7-day MET values and their corresponding likely activity periods.
- Used to display MET insights.

Generate Personalized Recommendations Using User Clustering

```
    MATCH (u:User {user_id: $user_id})
    OPTIONAL MATCH (u)-[:BELONGS_TO]->(c)
    WHERE c:ClusterA OR c:ClusterB OR c:ClusterC OR c:ClusterD
    OPTIONAL MATCH (peer)-[:HAS_DAILY_METRIC]->(dm:DailyMetric)
    OPTIONAL MATCH (c)-[:CONTAINS]->(peer:User)
    WHERE dm.total_sleep_minutes IS NOT NULL OR
            dm.total_calories IS NOT NULL OR
            dm.average_intensity IS NOT NULL
    WITH u,
        AVG(dm.total_sleep_minutes) AS avg_cluster_sleep,
        AVG(dm.total_calories) AS avg_cluster_calories,
        AVG(dm.average_intensity) AS avg_cluster_intensity
    RETURN u.user_id AS user_id,
        "Recommended␣Sleep␣(minutes):" AS sleep_label,
        CASE
            WHEN avg_cluster_sleep IS NOT NULL AND avg_cluster_sleep < 420
            THEN "Try␣increasing␣your␣sleep␣duration␣for␣better␣recovery."
            WHEN avg_cluster_sleep IS NOT NULL AND avg_cluster_sleep > 540
            THEN "Consider␣reducing␣excess␣sleep␣to␣improve␣energy␣levels."
            ELSE "Your␣sleep␣pattern␣is␣optimal."
        END AS sleep_recommendation,
        "Recommended␣Calories:" AS calories_label,
        CASE
            WHEN avg_cluster_calories IS NOT NULL AND avg_cluster_calories < 2000
            THEN "Increase␣calorie␣intake␣to␣improve␣energy␣levels."
            WHEN avg_cluster_calories IS NOT NULL AND avg_cluster_calories >= 2000
                AND avg_cluster_calories <= 2500
            THEN "Your␣calorie␣intake␣is␣optimal."
            ELSE "Reduce␣calorie␣intake␣to␣manage␣weight␣effectively."
        END AS calories_recommendation,
        "Recommended␣Intensity:" AS intensity_label,
        CASE
            WHEN avg_cluster_intensity IS NOT NULL AND avg_cluster_intensity < 30
            THEN "Increase␣your␣workout␣intensity␣for␣better␣cardiovascular␣health."
            WHEN avg_cluster_intensity IS NOT NULL AND avg_cluster_intensity >= 30
                AND avg_cluster_intensity <= 70
            THEN "Your␣intensity␣level␣is␣well-balanced."
            ELSE "Consider␣reducing␣high-intensity␣workouts␣to␣improve␣recovery."
        END AS intensity_recommendation
```

Explanation:

- Uses Neo4j clustering to compare the user's health metrics with peers
- Provides personalized recommendations for sleep, calories, and intensity.

5 Conclusion and Future Work

5.1 Lessons Learned

The Health Analytics Platform successfully integrates SQL and Neo4j to provide an efficient architecture for health data analysis. The key findings of this work include:

- The hybrid SQL-Neo4j approach significantly outperforms traditional SQL-only solutions for complex health analytics queries, particularly for pattern matching and relationship analysis.
- Graph-based anomaly detection identifies unusual health patterns, providing a valuable tool for early intervention and preventive healthcare.
- AWS Lambda functions provide an effective mechanism for automated data aggregation and synchronization between SQL and graph databases, maintaining data consistency with minimal overhead.

5.2 Limitations

Despite its successes, the platform has several limitations that should be addressed in future work:

- The current anomaly detection approach relies primarily on statistical thresholds rather than machine learning techniques, potentially missing subtle or complex anomalies.
- The system focuses on individual metrics rather than multivariate anomalies, which might be more indicative of health issues.
- Data synchronization between PostgreSQL and Neo4j introduces a delay of up to 24 hours, limiting real-time analytics capabilities.
- The clustering algorithm requires periodic recalibration as user behaviors evolve over time, adding maintenance overhead.

5.3 Future Work

Future enhancements to the platform will focus on the following areas:

- **Machine Learning Integration**: Incorporating supervised and unsupervised machine learning algorithms for more sophisticated anomaly detection and pattern recognition.
- **Real-Time Analytics**: Implementing streaming data processing using Apache Kafka and Flink to enable near-real-time anomaly detection and alerts.
- **Predictive Modeling**: Developing predictive models to forecast potential health risks based on historical patterns and detected anomalies.
- **Enhanced Visualization**: Creating an interactive dashboard for exploring health patterns, anomalies, and cluster memberships through intuitive visualizations.
- **Multi-Modal Data Integration**: Expanding the platform to incorporate additional data sources such as dietary information, environmental factors, and subjective well-being measures to provide a more comprehensive health analysis.

The integration of SQL and Neo4j demonstrates a promising approach for health analytics that balances structured data storage with relationship-based analysis. As wearable health technologies continue to evolve, this hybrid architecture provides a scalable foundation for deriving actionable insights from the growing volume of personal health data.