Stanford Snyder Lab – Challenge

Background

The context behind this task is to develop portions of a software app that can ETL wearable data and create useful analysis / visualizations on-the-fly for clinical trials. We are a team of software engineers, building useful tools for researchers to run their own clinical trials using wearable devices. The core of this challenge is rooted in:

1. Developing quality, distributed software for clinical trials with up to n=1,000 participants
2. Creating dashboards for visualizing wearable data intuitively
3. Providing extensible data formats / schema that allow integrations with multiple wearable devices

For this challenge, the Fitbit Charge** is the wearable of choice, which provides data at varying resolutions (highest being 1 second intervals). The target data format is intraday, time-series data. Assume data from multiple participants needs to be replicated onto Stanford's servers (hosted on a cloud provider), transformed or otherwise post-processed into readily accessible formats, then accessed by a web app for visualizations. This assumes a 1-to-many relationship (1 clinician trial manager, many participants).

Note:
**Although this challenge focuses on Fitbit Charge data, the ingestion system must be designed for future expansion to other wearables (e.g., Apple Watch, Oura Ring, MyFitnessPal, etc.). It should be modular and extensible, allowing you to plug in new, client-specific data formats in the future. This is described in point 3.

Challenge Instructions

Preface

During the challenge duration, go through as many tasks as possible. **Please create a github repo for the tasks and include a README with an explanation of the design and any code.** You can choose to make it public, or if you are concerned with making it public (for any reason), please provide access to the repo to Alexander (alrojo) and Kyu (kyuhur2).

The README should include information regarding what was built as well as why a certain decision was made. Please adhere to industry standards when contributing to the repo (e.g. clean code, pull requests on github, squash merging when merging branches into main, utilize basic software design principles but don't over engineer, etc).

Functions, variable names, file names, etc are listed in `Courier New` font. Technologies (e.g. databases, containerization tech, etc) are highlighted in **bold**. The challenge requirements of each task are written generally as a loose guide–please feel free to tweak requirements as needed.

Task 0.a: Data Volume Estimation

The following tasks are based on data estimations for running medium to large scale clinical trials across multiple months. Data estimations are very important as it affects design decisions and feasibility of this service. Everything built here should be run locally (i.e. not on the cloud). The purpose of this task is to conduct back-of-the-envelope data volume calculations to determine 1) what the size of the problem is and 2) whether there may be any issues that could arise with the volume of data we're potentially dealing with.

Answer the following questions.

1. Assume you are pulling 4 metrics from the Fitbit (`heart_rate`, `steps`, `distance`, `spo2`) at a 1 second resolution for 1 year**.
    a. How many data points is this for n=1? n=1,000? n=10,000?
    b. How many data points is this for 1 year? 2 years? 5 years?
2. To store 1 data point, how many bytes of data should this take?
    a. For n=1,000 / 2 years / 3 metrics at 1 second resolutions, how many bytes of data will this take if uncompressed (e.g. stored in native **PostgreSQL**)?
    b. A compressed time-series database (e.g. **TimescaleDB**) is commonly quoted to be able to compress data by >90% (a, b). Assuming a conservative compression rate of 80%, how much data is this?
        i. How can time-series databases compress data by such a large %? In what cases would time-series data not be compressed by very much? Would health data (e.g. heart rate, sleep, physical activity data, etc) extracted from a Fitbit be a good or bad example of data that can be compressed significantly? Why or why not?
3. For questions 1 and 2, we assumed 3 metrics at a 1 second resolution. In reality, depending on the metric, the resolution can vary considerably. Look into the Fitbit Web API explorer. What may be some useful metrics to run a physical activity affecting sleep study?
    a. List the metrics and their highest frequency (e.g. 1 sec, 1 min, 5 min, 15 min, etc)
    b. What is the actual volume of data that is produced for n=1,000 for a study duration of 1 year?
    c. What is the compressed volume of data for above? (e.g. assume 80% compression)
4. When retrieving time-series data from a database, it may be too expensive an operation to access all the data at the finest resolution.
    a. How would you solve this issue? What would you do to make queries "less expensive"? (Hint: this may come at an added data storage cost)
5. The above assumes a single server / machine. While modern day servers are able to store and process high volumes of data at high capacities, vertical scaling has its limits.
    a. First, what would be a feasible limit for vertically scaling this? List CPU, memory, and hard disk limitations on one machine.

b. If we were to scale this service horizontally, what considerations would be important? How would you resolve queries across multiple machines? Remember, this is not being built on the cloud, so you would physically have multiple machines next to each other.

**Note: For the Fitbit, only heart rate is available at a 1 second resolution and most other health metrics will be available at a 1 minute resolution. This calculation is just for estimation purposes (and potential calculations for other wearable devices that may offer 1 second resolution for other metrics) For now however, these considerations are outside the scope of this challenge.

Task 0.b: Data Extraction

For this task, there are a few resources you should familiarize yourself with. You are provided with
the following resource:

- Fitbit Charge 6 – Wearipedia Colab Guide

Here, you will extract data by using the colab notebook above. This mimics what a clinical trial manager may do–for now, we will just extract the simulated data for 1 user (click on the "synthetic" checkbox under the Authentication and Authorization section)**.

- If running the colab online, start by installing the wearipedia package with `!pip install wearipedia` in the notebook (before `import wearipedia`)
- If running locally, setup a virtual environment in Python, then download the wearipedia package (as well as other dependencies)

Afterwards, running the 3. Data Extraction code blocks will generate data for breathing rate, active zone minute, activity, heart rate, heart rate variability, and spo2. Here, we are most interested in heart rate (hr), and the synthetic data will look like so:

```
[{'heart_rate_day': [{'activities-heart': [{'dateTime': '2024-01-01',
      'value': {'customHeartRateZones': [],
       'heartRateZones': [{'caloriesOut': np.float64(2877.06579),
         'max': 110,
         'min': 30,
         'minutes': np.int64(21600),
         'name': 'Out of Range'},
        {'caloriesOut': np.int64(15400),
         'max': 136,
         'min': 110,
         'minutes': np.int64(14400),
         'name': 'Fat Burn'},
        {'caloriesOut': np.int64(58600),
         'max': 169,
         'min': 136,
         'minutes': np.int64(28800),
         'name': 'Cardio'},
        {'caloriesOut': np.int64(65800),
         'max': 220,
         'min': 169,
         'minutes': np.int64(21600),
         'name': 'Peak'}],
       'restingHeartRate': 58}}],
    'activities-heart-intraday': {'dataset': [{'time': '00:00:00',
       'value': np.float64(80.13596370336198)},
      {'time': '00:00:01', 'value': np.float64(80.13596370336198)},
      {'time': '00:00:02', 'value': np.float64(81.13596370336198)},
      {'time': '00:00:03', 'value': np.float64(80.13596370336198)},
      …
]
```

Regardless of `START_DATE` or `END_DATE`, the Wearipedia generates 30 days of data, from 2024-01-01 to 2024-01-30. Restructure the data as needed for your own purposes (it could be good to define a data structure that rearranges the data for ingestion). This synthetic data structure closely mirrors what real data from a Fitbit may look like. Other data (i.e. br, azm, activity, hrv, and spo2) should also be utilized for this challenge but heart rate data is the most important as it is the only data source with 1 second resolution provided by a Fitbit.

The following tasks should be built based on this synthetic heart rate data as the core source of data.

**Note: If you have a Fitbit device, you can try extracting your own data (instructions available in the Fitbit Charge 6 – Wearipedia Colab Guide provided above) and finishing the challenge with your data. Note that you may have to share your data with the challenge administrators (or anyone if you make the project public). If you are uncomfortable with this, we recommend proceeding with the synthetic data method.

Task 1: Ingestion / Write Flow

Build a daily, delta-load data pipeline that pulls Fitbit data into a locally hosted, time-series database.

1. Provision local resources
    - Use `docker compose` to spin up a time-series database of choice (e.g. **TimescaleDB** a PostgreSQL extension for time-series) alongside a secondary container for the ingestion script with a cron scheduler
        i. While **TimescaleDB** is the example of choice, other time-series databases such as InfluxDB or Prometheus can be used
            1. Use the official image from DockerHub
            2. Mount a volume for persistence (so the data is not lost to container crashes or restarts)
        ii. Regardless of which database is used, please explain the advantages of time-series databases (specifically the one utilized in your solution) in the `README`
    - Configure a cron scheduler to run the ingestion script jobs once daily inside a Docker (e.g. `0 1 * * *`)
2. Implement ingestion logic
    - Write and implement a script (e.g. written in **Python / Node.js**) that reads a locally-stored, `last-run` timestamp and fetches new intraday Fitbit data via the Fitbit API and writes it into the local **TimescaleDB** hypertable
    - Configure **TimescaleDB** locally using **Docker** and create a `raw_data` hypertable partitioned by timestamp to store metrics
    - Provide a `Dockerfile` for the ingestion environment (Python + cron) and a `docker-compose.yml` that coordinates the ingestion and **TimescaleDB** container
3. Deliverables checklist
    - A working `docker compose` setup that runs an app that ingests data from the Fitbit API once daily (cron -> **Python / Node.js** -> Fitbit API -> **TimescaleDB**)
    - `README.md` detailing configuration, schema, instructions

Task 2: Access / Read Flow

Build a data flow model from a locally hosted time-series database (e.g. **TimescaleDB**) into a local dashboard app for analysis and visualization.

1. Provision local resources
    - Take the **TimescaleDB** service from the previous task and load Fitbit data into the database. Approximately ~1 month of data should suffice
    - On container start, connect via `psql` or an init script to create the `raw_data` table and convert it into a hypertable to define the schema
        i. If you prefer **InfluxDB**, define a bucket with appropriate retention settings
        ii. For **Prometheus**, set up a scrape target on your exporter
2. Build the app layer
    - Extracting data from **TimescaleDB**
        i. Build the backend infrastructure to connect the dashboard backend to **TimescaleDB** via a **PostgreSQL** client (e.g., `pg` in **Node.js** / `psycopg2` in **Python**) and run SQL queries
    - Building the backend app
        i. Provide a simple HTTP endpoint (e.g. built with **Express.js** / **FastAPI** / **Flask** / **Django**) that the frontend can call to fetch time-series data–queries should directly hit your **TimescaleDB** database
        ii. Endpoint should send queries for a given `start_date`, `end_date`, `user_id` (participant), and `metric`
            1. There should be error / fallback flows for when a parameter is not provided
            2. Multi-user or multi-metric queries as well as database optimizations and retrieve strategies is further discussed in Task 3–simple `start_date`, `end_date`, `user_id`, `metric` queries should be supported at this stage
    - Building the frontend app (e.g. built in **React** / **Vue** / etc)
        i. Allows selecting `start_date`, `end_date`, `user_id`
        ii. Calls the backend endpoint and renders a line chart of the returned data
        iii. Does not need complex features—focus on demonstrating the data flow (this can be expanded upon in Task 4)
3. Deliverables checklist
    - A working app launchable with a `docker compose` that serves data from the locally hosted **TimescaleDB** database
    - `README.md` detailing configuration, schema, instructions

Task 3: Optimizing Design for Multi-Year / Multi-User queries

By now, you should have a working app that pulls data from the Fitbit API into a locally hosted time-series database as well as an app utilized to Build a data flow model from a locally hosted time-series database (e.g. **TimescaleDB**) into a local dashboard app for analysis and visualization. Memory is a limited resource as this solution is managed locally and not on the cloud so you are limited to return queries of data that do not exceed a fixed amount that is typical for on-prem servers (e.g. ~128 GB). Remember that on-prem should also consider whatever data is required to load the OS as well as other apps (so ultimately the solution will need to use less memory).

While you will not run into this issue with the data provided (as you have only 1 user and maybe ~1 month of data), this may become an issue for clinical trials that have n>100. The best solution will implement a multitude of the following strategies.

1. Database-side optimizations
    - Create aggregates in **TimescaleDB** so that queries over large time spans hit precomputed summaries rather than raw rows (e.g. 1-minute, 1-hour, 1-day) in the `raw_data` hypertable
        i. This will ultimately minimize raw data scans
        ii. Aggregates should be created daily, at ingestion time (e.g. `data_1m`, `data_1h`, `data_1d`, etc)
        iii. Based on the `start_date` and `end_date` parameters, automatically resolve the interval to decide which table to hit (e.g. `raw_data`, `data_1m`, `data_1h`, `data_1d`, etc)
2. [Optional] Data retrieval strategies
    - Pagination / Chunked fetching
        i. For queries exceeding a certain amount of data (e.g. large number of users, multiple months or years of data, etc), paginated by time window and stream results to the app
        ii. Release memory after processing each chunk before loading the next
    - Use columnar format (e.g. `parquet`)
        i. Stores statistics per column and compresses the data
3. Deliverables checklist
    - Integrate database-side optimizations for multiple downsampled formats during ingestion
    - Rethink data retrieval strategies for large queries (and change the query logic on the app accordingly)

Task 4: Creating a Dashboard for Analysis / Visualization

Previously, in Task 2, you should have built a simple dashboard for analysis and visualization. Now, build a more detailed dashboard that allows for more analysis and visualizations in depth. Beyond supporting the basic functionality to look up multiple users' fitbit data for an extended amount of time, there are no requirements. A project for inspiration can be found here.

Here are some key features that we would like to support:

1. Adherence overview
    a. Participant has no token yet
    b. Participant has not uploaded data in 48 hours
    c. Participant has low sleep upload (by threshold–TBD by you)
    d. Participant has less than 70% adherence overall (i.e. wear time)
2. Imputation method
    a. After study period concludes, missing data in between should be imputed reliably (simple imputation methods such as median / mean between two neighboring points is insufficient). Come up with a method to impute the data–but in a way that allows us to calculate with and without imputed values (i.e. keep track of which data points are imputed values)
3. Method of contacting participants by email (with the click of a button) who dip under certain thresholds
4. List of participants with click option when clicked, showcase specific health metrics utilizing the project mentioned above (https://github.com/arpanghosh8453/fitbit-grafana)

Task 5: Monitoring / Alerting

Build a monitoring and alerting system to ensure that your pipeline is observable and failures or performance issues trigger notifications. This should be done locally, by way of a monitoring and alerting stack with Docker Compose comprising of metrics collecting (e.g. **Prometheus**), visualizing (e.g. **Grafana**) and notification alerting (e.g. **AlertManager**) layers. Configure services to expose metrics (e.g. via a `/metrics` HTTP endpoint) and running host exporters (e.g. **Node Exporter** and **cAdvisor**). This architecture should be modular—each component can be relocated to a separate host by updating scrape targets and service URLs.

1. Provision local resources
    - Define services for **Prometheus**, **Grafana**, **Alertmanager** in `docker-compose.yml`
    - Mount host directories for **Prometheus** and **Grafana** data storage and expose standard ports so that each component can scrape or be accessed as needed
2. Implement monitoring logic
    - Integrate a **Prometheus** client library (e.g., `prom-client` for **Node.js** or `prometheus_client` for **Python**) to expose custom metrics on a `/metrics` endpoint
    - Deploy **Node Exporter** to collect CPU, memory, disk, and network metrics from the host, and cAdvisor (or Docker Daemon Exporter) to gather per-container resource usage metrics
    - In `prometheus.yml`, define `scrape_configs` for each target and set appropriate `scrape_interval` (e.g. 15-30 sec)
3. Configure AlertManager
    - Create a `rules.yml` specifying critical conditions under `rule_files:`
    - Define an `alertmanager.yml` with receiver (email SMTP endpoint to admin@wearipedia.com) and routing rules based on alert labels like `severity="critical"`
    - Configure inhibition and grouping in **Alertmanager** (e.g. suppress “warning” alerts when a “critical” alert is present)
4. Configure Grafana
    - Add **Prometheus** as a data source in **Grafana** (e.g. `http://prometheus:9090`), with no authentication or minimal creds
    - Use community dashboards (e.g. Node Exporter Full or Docker Monitoring) to visualize host / container metrics as well as ingestion latency / error rates
    - [Optional] Build simple custom panels for additional KPIs (e.g. ingestion throughput, error count) and set **Grafana** alerts to complement **AlertManager**
5. Deliverable checklist
    - Docker Compose Stack: A single docker-compose.yml that stands up **Prometheus**, **Grafana**, **AlertManager**, **Node Exporter**, and **cAdvisor** with configured volumes and networks to isolate and persist data
    - Services instrumented to expose Prometheus metrics on `/metrics`, with host/container exporters running alongside

- Validated Prometheus `rules.yml` and Alertmanager `alertmanager.yml` with at least two functional alerts (e.g., ingestion errors, high latency) and working email notifications (sent to admin@wearipedia.com)
- One or more Grafana dashboards showing application metrics (ingestion latency, error rate) and infrastructure metrics (CPU, memory)
- `README.md` describing how to run the stack, configure environment variables, and interpret dashboards / alerts

[Optional] Task 6: Horizontal scaling

For clinical trials between n=100 and n=1,000 spanning a few years of data, the above architecture will likely run with minimal issues with a reasonably powerful server (e.g. 16-32 GB vCPU, 128-256 GB RAM, 5-10 TB SSD). Expand on Task 0.a #5 by creating a diagram and which technologies would be required to horizontally scale this service.

Appendix

Fitbit API documentation:

- Fitbit Web API Explorer – Swagger UI
- Web API – Documentation
    - Refer to the Web API documentation for more up to date details