Contents

Question 1

Assume you are pulling 4 metrics from the Fitbit (heart_rate, steps, distance, $\mathbf{SpO_2}$) at a 1-second resolution for 1 year.

We assume that each metric provides readings every second. We will ignore the fact that no other metric apart from heart rate gives resolution up to one second for now.

$$\begin{aligned}
\text{Total seconds in a year} &= 365 \times 24 \times 60 \times 60 = 31,536,000 \text{ seconds}\\
\text{Data points per metric} &= 31,536,000\\
\text{Number of metrics} &= 4\\
\text{Total data points (n=1)} &= 31,536,000 \times 4 = 12,61,44,000
\end{aligned}$$

Question 1.a

How many data points is this for $n = 1$? $n = 1{,}000$? $n = 10{,}000$?

- For $n = 1$:
  Total Data Points = **12,61,44,000**
- For $n = 1,000$:
  Total Data Points = **12,61,44,00,000**
- For $n = 10,000$:
  Total Data Points = **1,26,14,40,00,000**

Question 1.b

How many data points is this for 1 year? 2 years? 5 years?

We assume $n = 1$

- For 1 year:
  Total Data Points = **12,61,44,000**
- For 2 years:
  Total Data Points = $2 \times 12,61,44,000 =$ **25,22,88,000**
- For 5 years:
  Total Data Points = $5 \times 12,61,44,000 =$ **63,07,20,000**

Question 2

To store 1 data point, how many bytes of data should this take?
Assume the schema to be structured as follows:

- `user_id` $\rightarrow$ `integer`
- `metric_type` $\rightarrow$ `smallint`
- `timestamp` $\rightarrow$ `timestamp with time zone`
- `value` $\rightarrow$ `real`

Considering PostgreSQL's typical storage sizes:

- `integer` = 4 bytes
- `smallint` = 2 bytes
- `timestamp with time zone` = 8 bytes
- `real` = 4 bytes

Total storage per data point: 4 + 2 + 8 + 4 = **18 bytes**

Question 2.a

For $n = 1,000$ users, 2 years, and 3 metrics at 1-second resolution, how many bytes of data will this take if uncompressed (e.g., stored in native PostgreSQL)?

- 1 year $= 31,536,000$ seconds or data points
- So, 2 years $= 2 \times 31,536,000 = 63,072,000$ data points
- For $1,000$ users and 3 metrics:

$$\text{Total data points} = 63,072,000 \times 1,000 \times 3 = 1,89,21,60,00,000$$
$$\text{Total size (in bytes)} = 1,89,21,60,00,000 \times 18 = 34,05,88,80,00,000 \text{ bytes}$$

which is equivalent to almost 3.097 TB of uncompressed data.

Question 2.b

A compressed time-series database (e.g., TimescaleDB) is commonly quoted to compress data by more than 90%. Assuming a conservative compression rate of 80%, how much data is this?

$$\begin{aligned}\text{Compressed size} &= 34,05,88,80,00,000 \times (1-0.8)\\ &= 34,05,88,80,00,000 \times 0.2\\ &= 6,81,17,76,00,000 \text{ bytes}\end{aligned}$$

which is equivalent to almost 634.396 GB or 0.619 TB.

Question 2.b.i

How can time-series databases compress data by such a large percentage? 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?

There are several algorithms which are used to compress the time-series database. Some of the commonly used ones are discussed below:

- **Delta Encoding:** Instead of storing the actual value of a data point, the difference from the previous data point is stored. As a result, the required number of bits stored is significantly reduced.
- **Simple 8b encoding:** Although Delta encoding now stores a very small value, the way database works is that the field will still take up the same amount of bytes. (An integer in PostgreSQL will take 4 byte regardless of what is stored). To tackle this, this encoding method is uses, where integer is stored as an array of blocks. And each block is set in such way that each integer in that block is represented as the minimum number of bits to represent the largest integer in that block.

  Example:

  Say the records are like: 11, 256, 45, 456, 445, 89

  They would be represented as: [11, 45, 89], [256, 456, 445]
- **Run Length Encoding or RLE:** In case there are several repeating large values appearing in any field, instead of storing them as it is, the repeating values are represented with a very small value.

- **XOR based compression:** This is similar to Delta encoding, but for floating point values. Here, the the result of XOR of a corresponding record's value and its previous record's value is stored.
- **Dictionary Compression:** If there exists common/reappearing values in the db, a dictionary is created with each unique value, and in the db, index of that corresponding data in the dictionary is stored.

In what cases would time-series data not be compressed by very much?

The above encoding techniques will mostly be applicable if the data point values are slowly changing.

If the records are very random/diverse, then their difference might not actually be represented. with smaller amount of bits than it would have required to store their absolute value, which nullifies the point of Delta encoding, Same thing applicable for XOR based compression.

Similarly, randomness will also affect the Simple 8b encoding and Dictionary compression as there will be very little presence of repeating values.

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?

Usually different types of health metrics like heart rate, activity levels, SpO2 remain consistent throughout the time and doesn't fluctuate that much. And since the resolution is small; 1 sec, even if there is sudden change, the change appears gradually in the dataset.

In my opinion, data extracted from Fitbit can be compressed at a very high ratio.

Question 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?

Question 3.a

List the metrics and their highest frequency (e.g., 1 sec, 1 min, 5 min, 15 min, etc.)

- **Active Zone Minutes Intraday**
  - fatBurnActiveZoneMinutes
  - cardioActiveZoneMinutes
  - peakActiveZoneMinutes
  - **activeZoneMinutes**

  Highest resolution: *1 minute*

- **Activities Intraday**
  - level
  - mets
  - **value**

  Highest resolution: *1 minute*

- **Breathing Rate Intraday**
  - lightSleepSummary
  - deepSleepSummary
  - remSleepSummary
  - **fullSleepSummary**

  Highest resolution: *per sleep*

- **Heart Rate Intraday**
  - value : customHeartRateZone : caloriesOut

  - value : customHeartRateZone : max
  - value : customHeartRateZone : min
  - value : customHeartRateZone : minutes
  - value : customHeartRateZone : name
  - value : HeartRateZone : caloriesOut
  - value : HeartRateZone : max
  - value : HeartRateZone : min
  - value : HeartRateZone : minutes
  - value : HeartRateZone : name
  - value : restingHeartRate
  - **dataset : value**

  Highest resolution: *1 second*

- **HRV Intraday**
  - hrv : minutes : value : rmssd
  - hrv : minutes : value : coverage
  - hrv : minutes : value : hf
  - hrv : minutes : value : lf

  Highest resolution: *every 5 minutes during sleep*

- **SpO2 Intraday**
  - **Value**

  Highest resolution: *every 5 minutes during sleep*

Note: *While detailed information is available under each metric, it is possible to work with a single value for each metric, which are marked in* ***bold****.*

Question 3.b

What is the actual volume of data that is produced for n=1,000 for a study duration of 1 year?

Some of the metrics provide some detailed values, but for the sake of this task, we will ignore them and use the single "summary" value that is also provided when relevant metrics' API is called. Exception is the HRV Intraday, where we would have to use 4 values.

The API documentation states the sleep period should be minimum 3 hours at a stable position. For this calculation, we will assume that the participant sleeps once each day, for 6 hours.

- **Active Zone Minutes Intraday**
  Total minutes in a year $= 365 \times 24 \times 60 = 525,600$
  For $n = 1000$, total data points $= 525,600 \times 1000 = 525,600,000$
  Each data point $= 18$ bytes
  Total size $= 525,600,000 \times 18 =$ **9,460,800,000** bytes

- **Activities Intraday**
  Total minutes in a year $= 365 \times 24 \times 60 = 525,600$
  For $n = 1000$, total data points $= 525,600 \times 1000 = 525,600,000$
  Each data point $= 18$ bytes
  Total size $= 525,600,000 \times 18 =$ **9,460,800,000** bytes

- **Heart Rate Intraday**
  Total seconds in a year $= 365 \times 24 \times 60 \times 60 = 31,536,000$
  For $n = 1000$, total data points $= 31,536,000 \times 1000 = 31,536,000,000$
  Each data point $= 18$ bytes
  Total size $= 31,536,000,000 \times 18 =$ **567,648,000,000** bytes

- **Breathing Rate Intraday**
  Total hours slept in a year $= 365 \times 6 = 2,190$ hours
  For $n = 1000$, total data points $= 2,190 \times 1000 = 2,190,000$
  Each data point $= 18$ bytes
  Total size $= 2,190,000 \times 18 =$ **39,420,000** bytes

- **HRV Intraday**
  Total minutes of sleep per year $= 365 \times 6 \times 60 = 131,400$
  Data points per user (5-min interval) $= 131,400 \div 5 = 26,280$
  For $n = 1000$, total data points $= 26,280 \times 1000 = 26,280,000$
  Each data point has 3 additional values: $18 + (3 \times 4) = 30$ bytes
  Total size $= 26,280,000 \times 30 =$ **788,400,000** bytes

- **SpO$_2$ Intraday**
  Total minutes of sleep per year $= 365 \times 6 \times 60 = 131,400$
  Data points per user (5-min interval) $= 131,400 \div 5 = 26,280$
  For $n = 1000$, total data points $= 26,280 \times 1000 = 26,280,000$
  Each data point $= 18$ bytes
  Total size $= 26,280,000 \times 18 =$ **473,040,000** bytes

Total Data Volume (All 6 Metrics):

$$\begin{aligned}\text{Total (bytes)} &= 9,460,800,000 + 9,460,800,000 + 567,648,000,000 + 39,420,000 + 788,400,000 + \\ &= \mathbf{587{,}870{,}460{,}000\ bytes} \\ \text{Total (GB)} &= \frac{587,870,460,000}{1024^3} \approx \mathbf{547.50\ GB}\end{aligned}$$

Question 3.c

What is the compressed volume of data for above? (e.g. assume 80compression)?

$$\begin{aligned}\text{Compressed (bytes)} &= 587,870,460,000 \times (1 - 0.8) = \mathbf{117{,}574{,}092{,}000\ bytes} \\ \text{Compressed (GB)} &= \frac{117,574,092,000}{1024^3} \approx \mathbf{109.5\ GB}\end{aligned}$$

Question 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.

Question 4.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)

Based on the previous calculations, it can be comprehended that each query over the raw data could lead to potential scanning over millions of data points, making the operation very "expensive". There are several techniques that can be applied to optimize this, the most suitable ones are discussed below:

- **Partitioning and Sharding:** Data points can be partitioned based on attributes such as `timestamp`, `user_id`, or `metric`, and distributed across multiple database nodes. This allows queries to be processed in parallel, significantly reducing query latency. However, this approach introduces additional storage overhead due to replication and metadata management across nodes.

- **Indexing:** Create a index based on the attributes mentioned in the previous point. As a result, whenever a query is executed, the system knows exactly where to navigate. Indexing algorithm such as B-Tree or BRIN index can be applied here. To understand the improvement, consider the query: "Heart rate of participant X on timestamp Y". Previously it would scan through millions of records to find the relevant one, but now, it would just look through the indexing table and it will know exactly where to look for. This also comes with storage overhead as it is required to store the index table now.

- **Pre-Aggregation:** During this research period, it might be a very common case that one needs to perform aggregate functions frequently for visualization—for example, the activity summary of a participant in the last one month. As aggregations might require extra calculations, it further increases the query expense. To tackle this, we might store some of the results of aggregation queries beforehand, like the summary of heart rate of a participant over some timestamp, which will save the computational complexity. This approach, once again like the rest, comes with storage overhead. Time-series databases such as TimescaleDB offer built-in *continuous aggregates*, which automatically maintain and update these summaries in the background.

All of the techniques mentioned above come with storage overhead, typically around 2–3x. However, many teams and practitioners have adopted these optimizations and observed substantial improvements in query performance, making the trade-off between storage cost and computational expense a worthwhile one.

Question 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

Question 5.a

First, what would be a feasible limit for vertically scaling this? List CPU, memory, and hard disk limitations on one machine.

From the previous question, we found that 1,000 participants' data for 1 year will take approximately **547.5 GB (uncompressed)** and **109.5 GB (compressed)**.

So, each participant will require around **0.55 GB uncompressed** and **0.11 GB compressed** storage per year.

Also, number of data points generated for each participant per year:

$$\frac{0.55 \times 1024^3}{18 \times 1000} \approx 326,809 \text{ data points}$$

For this question, we will ignore query optimization (as discussed in Question 4) for simplicity. However, we consider the I/O constraints and compression overhead.

Assumed machine configuration

- **CPU:** Up to 64 cores
  (Assume each core can process up to 1M data points/second)
- **RAM:** 12 TB
- **Storage:** 200 TB SSD with 10 GB/s read speed

(Note: While it is possible to getter better configuration like 256 core CPU, 24TB Ram etc, I decided to go with a realistic and practical configuration)

Storage Constraints:

$$\text{Participants per year} = \frac{200 \times 1024}{0.11} \approx 1,861,818$$

This is equivalent to:

18,000 participants for 10 years

(Note: We are assuming compression is done on the fly, and compressed data is what is stored.)

RAM Constraints:
RAM must be able to handle uncompressed data:

$$\text{Participants per year} = \frac{12 \times 1024}{0.55} \approx 22,464$$

This means:

5,000 participants for 4 years, or 2,000 participants for 11 years

CPU Constraints:
(1) Uncompression Time:
Assume uncompression rate = 500 MB/s
Compressed data size for 5,000 participants over 4 years:

$$0.11 \times 5000 \times 4 \times 1024 = 2,252,800 \text{ MB}$$

Time to uncompress:

$$\frac{2,252,800}{500 \times 64} = 70.4 \text{ seconds}$$

(2) Data Processing Time:

$$\text{Data points} = 326,809 \times 5000 \times 4 = 6.56 \times 10^8$$

$$\text{Processing time} = \frac{656,175,560}{1,000,000 \times 64} \approx 10.25 \text{ seconds}$$

(3) Disk I/O Read Time:
Total data to load (compressed):

$$0.11 \times 5000 \times 4 = 2,200 \text{ GB}$$

$$\text{I/O time} = \frac{2200}{10} = 220 \text{ seconds}$$

Total query time:

$$70.4 + 10.25 + 220 \approx 300.65 \text{ seconds} \approx 5 \text{ minutes}$$

This is acceptable for most analytics workloads.

Note: If we apply query optimizations (e.g., pre-aggregates, indexing), this time can be significantly reduced. We ignore these for now to evaluate raw vertical limits.

Conclusion: Based on the above calculations, the assumed machine should be capable of handling data from **5,000 participants for 4 years** without hitting any major bottlenecks. Beyond this scale, we would likely need to move to a horizontally scalable architecture.

Question 5.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.

While scaling the system horizontally, the following considerations should be accounted for:

- **ACID Property:** One of the crucial aspects of database design. The distributed system must be able to maintain consistency of data, handle transaction failures as well as other types of failures like power outage or hardware failure. This is an in-general consideration whereas its components are discussed in the upcoming components.
- **Sharding Strategies and Load Balancing:** The data sharding, which is already mentioned in Question 4, should be done efficiently, ensuring that the load is distributed equally across all servers.
- **Fault Tolerance:** In case of any issue, there should be some mechanism to keep the system running. For example, if one of the machines goes down, the other servers should continue operation without hindrance. One way to achieve this is through data replication, although it comes with extra storage cost.
- **Transaction Management / Query Resolution:** Executing queries over a distributed system is more complex than doing so on a single machine. There should be mechanisms to properly handle indexing, maintain transactional consistency, and handle joins and foreign key constraints — all with minimal overhead. Since data is hosted across machines, query latency will naturally increase.

- **Infrastructure Consideration:** This mainly concerns network setup. The interconnect speed should be fast enough to support synchronized operations across machines.
- **Data Migration:** The system must support seamless addition or removal of nodes without data loss or service disruption.

How to resolve the queries across multiple machines?

The query optimization techniques mentioned in Question 4 are relevant here.

- **Proper Sharding Strategy:** Sharding strategy significantly affects query performance. For Fitbit-like data, sharding can be based on user ID, metric type, or time. The choice should align with the analytical requirements.
- **Implementing Distributed Transactions:** To maintain consistency among shards, distributed transactions are necessary. For example, the *Two-Phase Commit* protocol ensures all clusters either commit or rollback changes in the case of a failure.
- **Indexing and Load Balancing:** Indexes should be designed to optimize common queries, while ensuring the query and write loads are evenly distributed across clusters.
- **Using External Query Engines:** External engines like Presto or Apache Hive can handle data across distributed storage efficiently. These are optimized for auto-scaling, sharding, and query planning. Not all such engines support locally-hosted clusters, but solutions like **TimescaleDB's multi-node support** can offer built-in scaling, replication, and performance optimization capabilities.