Analysis of Safecast radiation data

# Data Acquisition

## **1. Description of the Data Source**

We initially planned to use Safecast’s publicly available REST API ([API Documentation](https://api.safecast.org/home)). However, we quickly encountered several limitations with this approach.

### **Challenges with REST API:**

* **Limited Results Per Request**: Each request to the API returns only 25 results. Given our requirement for extensive data, this would necessitate numerous requests, significantly slowing down the data acquisition process and burdening the server.
* **Lack of Pagination**: The API does not support pagination, complicating the retrieval of continuous data. To gather data for specific periods, we would have had to make sequential queries within very small time windows (e.g., a few hours). This approach was impractical due to the variability in measurement frequency—some days have no measurements, while others have tens of thousands.
* **Unnecessary Data**: The API responses include extraneous information such as user IDs, measurement IDs, and station IDs. Our interest is solely in the radiation values relative to location (latitude, longitude, height), date, and measurement unit.

Due to these limitations, we decided to abandon the REST API approach. Fortunately, the Safecast documentation also mentions the possibility of accessing a Postgres database instance, which suited our needs perfectly.

## **2. Selecting the Right Data Source**

Upon realizing the limitations of the REST API, we contacted a member of the Safecast team to gain access to their Postgres database. This direct access allowed us to efficiently download large volumes of relevant data, significantly streamlining our data acquisition process. Despite being faster and more efficient than the API, this method also had its limitations, which we will discuss in more detail later in this article.

## **3. Process of Data Acquisition from the Database**

### **Initial Database Inspection**

Initially, we used the DBeaver tool to inspect the database, searching for the tables and columns of interest. Given the vast amount of data, any request that fetched data without limiting the number of results took a long time and often failed. These failures were due to the database state changing during the transaction, making it impossible to complete the request. To address this, we needed a method to fetch data in appropriately large chunks to minimize the number of queries, while ensuring the data retrieval process was swift enough to complete before new data entries caused the transaction to fail.

### **First Approach: Fixed Window Size**

Our first approach involved setting a fixed window size, fetching data from periods of several months, starting from 2010 and moving to the present date. Data before 2010 was fetched in a single request due to its smaller volume. However, this method proved inadequate as the number of measurements per month increased over time, leading to failures when querying data from more recent periods.

### **Improved Approach: Adaptive Window Size and Retry Mechanism**

To automate the process of fetching data in smaller chunks, we implemented an algorithm featuring an adaptive window size and multiple retry attempts. If a query failed, it was retried up to a retryLimit of three times, after which the window size was halved (decreaseFactor), starting from an initial window size of 100 days (initialIncrement). The window size could not be reduced below one day (minIncrement). Data fetched was saved to separate CSV files to ensure continuity in case the process needed to restart from a specific point.

### **Problems with Python implementation**

#### Initially, we used Python for data retrieval, given its popularity in data analysis. However, the psycopg2 library, the most common PostgreSQL adapter in Python, could not handle the exception thrown when a query was rejected by the database. Attempts to handle this error in Python were unsuccessful, as the exception occurred within the library code, making it unmanageable.

#### Here's the key part of the Python code where we encountered issues with error handling:

def execute\_query(conn, condition, file\_suffix):

"""Execute a query and save the results to a CSV file if non-empty, using the connection provided."""

with conn.cursor() as cursor:

query = f"""

SELECT

m.device\_id,

m.unit,

m.location,

m.height,

m.captured\_at::date AS measurement\_day,

AVG(m.value) AS average\_value

FROM

public.measurements AS m

WHERE

{condition} AND (m.device\_id IS NOT NULL OR m.height IS NOT NULL)

GROUP BY

m.device\_id, m.unit, m.location, m.height, measurement\_day

ORDER BY

m.device\_id;

"""

try:

cursor.execute(query)

logger.debug(f"Query executed: {query}")

results = cursor.fetchall()

if results:

df = pd.DataFrame(results)

if not df.empty:

filename = os.path.join(DATA\_DIR, f"{FILE\_PREFIX}\_{file\_suffix}.csv")

df.to\_csv(filename, index=False)

logger.info(f"Results saved to {filename}.")

else:

logger.info("No data to save. Data frame was empty.")

else:

logger.info("Query returned no data.")

return True

except psycopg2.Error as e:

logger.error(f"Failed to execute query: {e}")

return False

Due to the problems we encountered with Python, we switched to Go, which supports multithreading. If the same issue happens, the idea was to execute queries in separate threads, and if a query failed, the thread would be terminated and a new one spawned, allowing the main program to continue running without manual restarts, a limitation of Python’s single-threaded nature.

### **Go Implementation for Data Retrieval**

The Go implementation allowed us to efficiently and reliably fetch the data needed. Below are Go code snippets used for this process, focusing on the key parts.

#### **Setting Up the Environment and Connecting to the Database**

First, we set up the environment and established a connection to the database:

package main

import (

"database/sql"

"fmt"

"os"

"time"

"github.com/joho/godotenv"

\_ "github.com/lib/pq"

)

const (

filePrefix = "measurements"

dataDirectory = "../data/chunks"

retryLimit = 3

decreaseFactor = 2

minIncrement = 1

initialIncrement = 180

)

func init() {

if err := godotenv.Load(); err != nil {

logError("Error loading .env file")

}

}

func connectDB() \*sql.DB {

connStr := fmt.Sprintf("user=%s password=%s dbname=%s host=%s port=%s sslmode=require",

os.Getenv("USER"), os.Getenv("PASSWORD"), os.Getenv("DATABASE"),

os.Getenv("HOST"), os.Getenv("PORT"))

db, err := sql.Open("postgres", connStr)

if err != nil {

logError("Failed to connect to database: %v", err)

return nil

}

logSuccess("Successfully connected to the database.")

return db

}

This code sets up the environment using environment variables and establishes a connection to the PostgreSQL database.

#### **Executing Queries with Error Handling**

Next, we implemented the function to execute queries, handle errors, and save the results:

func executeQuery(db \*sql.DB, condition string, fileSuffix string) bool {

query := fmt.Sprintf(`

SELECT device\_id, unit, location, height, captured\_at::date AS measurement\_day, AVG(value) AS average\_value

FROM public.measurements

WHERE %s AND (device\_id IS NOT NULL OR height IS NOT NULL)

GROUP BY device\_id, unit, location, height, measurement\_day

ORDER BY measurement\_day;`,

condition)

rows, err := db.Query(query)

if err != nil {

logError("Failed to execute query: %v", err)

return false

}

defer rows.Close()

var data [][]string

headers := []string{"Device ID", "Unit", "Location", "Height", "Measurement Day", "Average Value"}

for rows.Next() {

var (

deviceID sql.NullInt64

unit, location sql.NullString

height sql.NullFloat64

measurementDay sql.NullTime

averageValue sql.NullFloat64

)

if err := rows.Scan(&deviceID, &unit, &location, &height, &measurementDay, &averageValue); err != nil {

logError("Failed to scan row: %v", err)

return false

}

record := []string{

nullInt64ToString(deviceID),

nullStringToString(unit),

nullStringToString(location),

nullFloat64ToString(height),

nullTimeToString(measurementDay),

nullFloat64ToString(averageValue),

}

data = append(data, record)

}

if len(data) > 0 {

var fileName = fmt.Sprintf("%s\_%s.csv", filePrefix, fileSuffix)

if err := writeCSV(data, headers, dataDirectory, fileName); err != nil {

return false

}

} else {

logInfo("No data found for the current query.")

}

return true

}

This function executes the SQL query, processes the results, and saves them to a CSV file. It also handles any errors that occur during query execution.

#### **Managing Retries and Adaptive Window Size**

Lastly, we implemented the main function to manage retries and adapt the window size for data fetching:

func main() {

db := connectDB()

if db == nil {

return

}

defer db.Close()

startDate := time.Date(2010, 1, 1, 0, 0, 0, 0, time.UTC)

currentDate := startDate

interval := initialIncrement

for currentDate.Before(time.Now()) {

nextDate := currentDate.AddDate(0, 0, interval)

condition := fmt.Sprintf("captured\_at BETWEEN '%s' AND '%s'", currentDate.Format("2006-01-02"), nextDate.Format("2006-01-02"))

fileSuffix := fmt.Sprintf("%s\_to\_%s", currentDate.Format("2006-01-02"), nextDate.Format("2006-01-02"))

logInfo("Executing query for interval: %s to %s", currentDate.Format("2006-01-02"), nextDate.Format("2006-01-02"))

retries := retryLimit

for retries > 0 {

if executeQuery(db, condition, fileSuffix) {

break

}

retries--

logWarning("Retrying... %d retries left.", retries)

if retries == 0 {

if interval > minIncrement {

interval = max(interval/decreaseFactor, minIncrement)

logWarning("Reducing interval due to errors. New interval: %d", interval)

retries = retryLimit

} else {

logError("Failed to execute query after multiple retries at minimum interval. Exiting loop.")

return

}

}

}

currentDate = nextDate

}

}

func max(a, b int) int {

if a > b {

return a

}

return b

}

This function manages the query execution process, adjusting the window size and handling retries to ensure efficient and reliable data retrieval.

### **Conclusion**

By transitioning from Python to Go, we significantly improved the efficiency and reliability of our data retrieval process. The multithreading capabilities of Go allowed us to handle query failures gracefully and continue fetching data without manual intervention, ultimately enabling us to gather the comprehensive datasets needed for our radiation analysis.

For the complete source code, please visit our [GitHub repository](https://github.com/MehowBd/ED-2024/tree/main/scripts/fetching).

# 

# Data Preprocessing

## **1. Limiting measurements and labeling sensors**

Many of the measurements collected from the sensors were repeated or showed minimal variation, especially when the sensors were stationary. To ensure the accuracy and clarity of our visualizations, we aimed to limit the data to one average value per day for each stationary sensor. If a sensor was moving (its longitude, latitude, or height changed beyond a set threshold), we labeled it as moving for that specific day.

### **Criteria for Sensor Movement**

* **Stationary Sensor**: A sensor was considered stationary if its location (latitude, longitude) did not change by more than 20 meters, and its height did not change by more than 5 meters.
* **Moving Sensor**: If the sensor’s location or height changed beyond these thresholds, it was labeled as moving.

### **Data Preprocessing Script**

We implemented a Python script to preprocess the data. The script reads data from multiple CSV files, aggregates measurements, determines whether sensors were moving or stationary, and outputs a cleaned dataset.

Here are the key parts of the preprocessing script, with detailed comments:

#### **Reading Data Chunks**

First, we read all the data chunks into a single DataFrame:

import os

import glob

import pandas as pd

from logger import setup\_logger

# Directory and file configuration

IN\_DIR = 'data/chunks'

FILE\_PATTERN = 'measurements\_\*.csv'

OUT\_DIR = 'data'

OUT\_FILE\_NAME = 'measurements\_preprocessed.csv'

logger = setup\_logger()

def read\_chunks(log\_every=10):

files = glob.glob(os.path.join(IN\_DIR, FILE\_PATTERN))

chunks = []

total\_files = len(files)

for i, file in enumerate(files):

chunk = pd.read\_csv(file)

chunks.append(chunk)

if i % log\_every == 0 and i != 0:

logger.info(f"Processed {i} of {total\_files} files.")

all\_data = pd.concat(chunks, ignore\_index=True)

logger.success(f"All chunks read. Total files processed: {total\_files}. Total measurements: {len(all\_data)}.")

return all\_data

df = read\_chunks()

This part of the script reads all CSV files matching the pattern, concatenates them into a single DataFrame, and logs the progress.

#### **Processing Data**

Next, we process the data to identify moving and stationary sensors:

from geopy.distance import geodesic

from shapely import wkb

GPS\_ACCURACY\_THRESHOLD = 20 # meters

HEIGHT\_MOVEMENT\_THRESHOLD = 5 # meters for height changes

def process\_data(df):

df['Location'] = df['Location'].apply(lambda x: wkb.loads(bytes.fromhex(x)))

df['Latitude'] = df['Location'].apply(lambda x: x.y)

df['Longitude'] = df['Location'].apply(lambda x: x.x)

# Handle non-finite 'Device ID' values

if df['Device ID'].isnull().any() or np.isinf(df['Device ID']).any():

logger.warning("Incorrect 'Device ID' values found. Removing affected rows.")

df = df.dropna(subset=['Device ID'])

# Convert Device ID to integer and set as index

df['Device ID'] = df['Device ID'].astype(int)

df.set\_index('Device ID', inplace=True)

aggregated\_data = df.groupby(['Device ID', 'Measurement Day']).agg({

'Unit': 'first',

'Latitude': 'median',

'Longitude': 'median',

'Height': 'median',

'Average Value': 'mean'

}).reset\_index()

# Temporary columns for movement calculations

aggregated\_data['Prev Latitude'] = aggregated\_data.groupby('Device ID')['Latitude'].shift(1)

aggregated\_data['Prev Longitude'] = aggregated\_data.groupby('Device ID')['Longitude'].shift(1)

aggregated\_data['Prev Height'] = aggregated\_data.groupby('Device ID')['Height'].shift(1)

# Calculate movement and height difference

aggregated\_data['Movement'] = aggregated\_data.apply(

lambda row: geodesic((row['Prev Latitude'], row['Prev Longitude']), (row['Latitude'], row['Longitude'])).meters if pd.notnull(row['Prev Latitude']) else 0,

axis=1

)

aggregated\_data['Height Difference'] = (aggregated\_data['Height'] - aggregated\_data['Prev Height']).abs()

# Determine status with height difference handling

aggregated\_data['Status'] = aggregated\_data.apply(

lambda row: 'Moving' if row['Movement'] > GPS\_ACCURACY\_THRESHOLD or (row['Height Difference'] > HEIGHT\_MOVEMENT\_THRESHOLD and pd.notnull(row['Height Difference'])) else 'Stationary',

axis=1

)

# Remove temporary calculation columns before final output

final\_data = aggregated\_data.drop(columns=['Prev Latitude', 'Prev Longitude', 'Prev Height', 'Movement', 'Height Difference'])

logger.success("Movement calculation and status assignment completed.")

return smooth\_status(final\_data)

df = process\_data(df)

This function processes the DataFrame to:

1. Convert location data from WKB format to latitude and longitude.
2. Aggregate daily measurements per sensor.
3. Calculate movement and height changes.
4. Label each sensor as moving or stationary based on the thresholds.

#### **Smoothing Status Transitions**

We then smooth the status transitions to ensure consistency:

def smooth\_status(df):

for device\_id, group in df.groupby('Device ID'):

statuses = group['Status'].values

for i in range(1, len(statuses) - 1):

if statuses[i-1] == statuses[i+1] and statuses[i] != statuses[i-1]:

statuses[i] = statuses[i-1]

df.loc[group.index, 'Status'] = statuses

logger.info(f"Status smoothed for Device ID {device\_id}")

logger.success("Status smoothing completed for all devices.")

return df

df = smooth\_status(df)

This function smooths the status transitions to handle intermittent status changes within short periods, providing more consistent labeling of moving and stationary sensors.

#### **Saving the Results**

Finally, we save the processed data to a CSV file:

def save\_results(df):

results\_file\_path = os.path.join(OUT\_DIR, OUT\_FILE\_NAME)

df.to\_csv(results\_file\_path, index=False)

logger.success(f"Results saved to {results\_file\_path}")

save\_results(df)

### **Conclusion**

The preprocessing script efficiently merges all fetched data chunks into a single file, processes the data to identify moving and stationary sensors, and smooths the status transitions for consistency. For the complete implementation, you can refer to our [GitHub repository](https://github.com/MehowBd/ED-2024/blob/main/scripts/preprocessing/process_measurements.py).

## 

## **2. Retrieving Missing Heights**

#### Many measurements in our dataset were missing height information, which is crucial for our analysis. To address this, we used the Google Elevation API to fetch missing heights based on the latitude and longitude of each measurement. The algorithm also incorporates caching to speed up the process and save API credits.

### **Key Parts of the Height Retrieval Script**

#### Here is the core part of the script, focusing on fetching missing heights with caching:

#### def fetch\_height(lat, lon, \*, cache = {}):

#### cache\_key = f"{lat},{lon}"

#### 

#### if cache\_key not in cache:

#### logger.info(f"Fetching height for location {lat}, {lon}.")

#### response = requests.get(f'{API\_URL}?locations={lat},{lon}&key={API\_KEY}')

#### 

#### if response.status\_code == 200:

#### results = response.json()['results']

#### if not results:

#### logger.warning(f"No results found for location {lat}, {lon}.")

#### return None

#### height = results[0]['elevation']

#### cache[cache\_key] = height

#### else:

#### logger.error(f"Failed to fetch height for location {lat}, {lon}.")

#### return None

#### else:

#### logger.info(f"Using cached height for location {lat}, {lon}.")

#### 

#### return cache[cache\_key]

#### 

#### def fetch\_missing\_heights(df):

#### missing\_heights = df['Height'].isnull()

#### if missing\_heights.any():

#### for i, row in df[missing\_heights].iterrows():

#### height = fetch\_height(row['Latitude'], row['Longitude'])

#### if height is not None:

#### df.at[i, 'Height'] = height

#### if i % LOG\_EVERY == 0 and i != 0:

#### logger.info(f"Processed {i} of {len(df[missing\_heights])} missing heights.")

#### else:

#### logger.success("No missing heights found.")

#### 

#### return df

This code fetches height data for locations with missing heights, using a cache to avoid redundant API requests and save credits.

For the complete implementation, you can refer to our [GitHub repository](https://github.com/MehowBd/ED-2024/blob/main/scripts/preprocessing/fetch_heights.py).

## **3. Cleaning up data**

# TODO

#### 

# 

# Visualization and Analysis

# TODO