**Problem 1:**

For this problem, since we have weather data from multiple stations and spanning several years, I have designed a relational database schema. Given that the data includes information about temperature and precipitation, I have created a table to store information from all weather stations for all years, as specific information about the weather stations or their locations has not been provided in the dataset.

I'm using PostgreSQL for this exercise. The code to setup and establish connection with the PostgreSQL database can be found in src/connect\_database.py and the code to create and setup tables can be accessed from src/models.py. Here's a basic outline of the schema:

**Table:**

**weather\_data:**

* + **Purpose:** Stores individual weather data records.
  + **Columns:**
    - **id:** Unique identifier for each weather data record.
    - **date:** Date of the weather observation.
    - **station\_id:** Identifier for the weather station where the data was recorded.
    - **max\_temp:** Maximum temperature recorded on the given date (in degrees Celsius).
    - **min\_temp:** Minimum temperature recorded on the given date (in degrees Celsius).
    - **precipitation:** Amount of precipitation recorded on the given date (in centimeters).
  + **Constraints:**
    - **id** is the primary key.
    - **date** cannot be null.
    - **station\_id** cannot be null.
    - Unique constraint on (**station\_id**, **date**) to ensure no duplicate records for the same station and date.

**Problem 2:**

I have used Python to write code to read the raw text files containing weather data, parse them, update data to the correct units, handle missing values, and insert the records into the PostgreSQL database. I have used the ORM provided by SQLAlchemy to interact with the database. The code can be found in src/ingest\_data.py and is designed to handle duplicate entries, where multiple rows with same data will be avoided when code is run more than once. The code also logs the start and end times, as well as the number of records ingested, as shown below.

While ingestion, the following log output is generated:

A screen shot of a computer

Description automatically generated

If code is re-run, and duplicates are detected but not ingested, the following log output is generated:

A screen shot of a computer

Description automatically generated

**Problem 3:**

For this problem, I have used Python to calculate the required statistics (average max temperature, average min temperature, total precipitation) for every year and every weather station. This code can be found in src/weather\_statistics.py. Missing data has been ignored while calculation, and NULL was used for statistics that could not be calculated. A new data model was designed to store these calculated results, coded in Python, and can be found in src/models.py. I have used a similar relational schema with another table to store the statistics:

**Table:**

**weather\_statistics**:

* + **Purpose**: Stores aggregated weather statistics data.
  + **Fields**:
    - **id**: Unique identifier for each weather statistics record.
    - **year**: Year for which the statistics are calculated.
    - **station\_id**: Identifier for the weather station these statistics belong to.
    - **avg\_max\_temp**: Average maximum temperature for the year (in degrees Celsius).
    - **avg\_min\_temp**: Average minimum temperature for the year (in degrees Celsius).
    - **total\_precipitation**: Total precipitation for the year (in centimeters).
  + **Constraints**:
    - **id** is the primary key.
    - **year** cannot be null.
    - **station\_id** cannot be null.

**Problem 4:**

I have chosen Flask as the web framework to create a REST API. I have implemented two GET endpoints **/api/weather** and **/api/weather/stats** to retrieve weather data and statistics, respectively. These endpoints will accept query parameters for filtering by date and station ID, and the responses will be JSON-formatted and paginated. The code to setup and build the API can be found in src/app.py. Additionally, I have included Swagger documentation for the API, and created some unit tests which can be found in src/unit\_tests.py.

**The weather API:**

A screenshot of a computer

Description automatically generated

**GET endpoint /api/weather:**

Clients can filter by date and station ID as shown below.

A screenshot of a computer

Description automatically generated

The response is generated in the following format, paginated:

A black rectangular object with white border

Description automatically generated

**GET endpoint /api/weather/stats:**

Clients can filter by year and station ID as shown below.

A screenshot of a computer

Description automatically generated

The response is generated in the following format, paginated:

A black and white lines

Description automatically generated with medium confidence

**Extra Credit - Deployment:**

To deploy the API, database, and a scheduled version of the data ingestion code on AWS, I would use the following tools and AWS services:

* **Amazon EC2 (Elastic Compute Cloud)**: For deploying the Flask application. EC2 provides resizable compute capacity in the cloud and allows us to run applications on virtual servers.
* **Amazon RDS (Relational Database Service)**: For hosting the PostgreSQL database. RDS manages database administration tasks like backups, software patching, and scaling, allowing us to focus on application development.
* **AWS Lambda**: For the scheduled version of the data ingestion code. Lambda allows running code without provisioning or managing servers. We can trigger Lambda functions using AWS CloudWatch Events, which can be scheduled to run at specific intervals.
* **Amazon S3 (Simple Storage Service)**: For storing static files, such as uploaded CSV files or any other resources required by the application.
* **Amazon VPC (Virtual Private Cloud)**: For configuring a virtual network environment in which the EC2 instances and RDS database can operate securely.
* **AWS CloudFormation**: For automating the deployment process by defining infrastructure as code. CloudFormation templates can be used to define the resources and configurations required for the deployment.
* **AWS IAM (Identity and Access Management)**: For managing access to AWS services securely. IAM roles and policies can be configured to grant permissions to resources and actions.

**Deployment Approach:**

1. **Setup Amazon RDS:** We establish a PostgreSQL database instance using Amazon RDS, configuring security groups to control access to it.
2. **Deploy Flask Application on EC2:** We launch an EC2 instance, installing necessary dependencies like Python and Flask. The Flask application is then deployed on this instance, with security groups allowing inbound traffic on the required port. Optionally, Elastic Load Balancing (ELB) can be used for distributing traffic across multiple EC2 instances.
3. **Scheduled Data Ingestion with AWS Lambda:** The data ingestion code is written as a Lambda function. A CloudWatch Events rule is created to trigger this function on a scheduled basis (e.g., daily/hourly). Permissions are configured for the Lambda function to access resources like S3 for file storage and RDS for database interaction.
4. **Set Up Amazon S3 (if required):** An S3 bucket is created for storing static files or any resources needed by the application.
5. **Configure Networking with Amazon VPC:** A VPC is set up to isolate resources and control network access. Subnets, route tables, and security groups are configured to define network settings.
6. **Automate Deployment with AWS CloudFormation (optional):** A CloudFormation template is defined to provision and configure the required AWS resources. This template ensures consistent deployment across different environments.
7. **Manage Access with AWS IAM:** IAM roles and policies are created to grant necessary permissions to AWS resources. The principle of least privilege is followed to restrict access only to required resources and actions.
8. **Monitoring and Logging:** CloudWatch Alarms are set up to monitor EC2 instances, RDS database, and Lambda functions for performance and health metrics. CloudWatch Logs are configured to centralize logging and monitor application logs for errors and debugging.
9. **Scale and Maintain:** The application's performance is monitored regularly, scaling resources as needed to handle changes in traffic or workload. Regular updates and patches are applied to the application, database, and underlying infrastructure to ensure security and performance.