**Detailed Project Description with Idea Flow – Suprith Reddy**

**Project Overview**

The **Weather Data Management System** project was developed to handle large volumes of weather data, ensuring it could be ingested, cleaned, stored, analyzed, and exposed via a REST API. The system was designed with a focus on scalability and flexibility, to enable querying and statistical analysis of the weather data.

The project was structured in four key phases: data modeling, data ingestion, data analysis, and REST API development. It utilized tools and technologies such as **Python**, **Flask**, **SQLAlchemy**, **PostgreSQL**, and **Pandas** to handle various aspects of the data pipeline. The system was tested locally using unit tests and then prepared for cloud deployment, considering both performance and security.

**Phases of Execution**

**Phase 1: Data Modeling**

The first step in the project was designing the database schema. The weather data is structured in a PostgreSQL database, optimized for querying and aggregating large datasets.

The database schema consists of:

* **station\_id**: A unique identifier for each weather station.
* **date**: The date the weather data was recorded (formatted YYYY-MM-DD).
* **max\_temp**: Maximum temperature recorded on that day, in Celsius.
* **min\_temp**: Minimum temperature recorded on that day, in Celsius.
* **precipitation**: Total precipitation in centimeters.

**Key Challenges:**

1. **Data integrity**: Ensuring the database schema supports the integrity of data (e.g., preventing null values where they are not allowed).
2. **Scalability**: Designing the schema for future expansion, enabling the storage of millions of weather records without affecting performance.
3. **Flexibility**: Keeping the model flexible enough to accommodate possible future changes, such as additional weather metrics.

**Solution:**

* We used **SQLAlchemy ORM** to abstract the database and interact with it in Python. SQLAlchemy allowed us to map database objects to Python objects, making it easier to manage relationships between tables and maintain code consistency.

**Phase 2: Data Ingestion**

This phase focused on ingesting the raw weather data from CSV files and storing it into the PostgreSQL database. The ingestion process involved:

* **Reading CSV files**: We used **Pandas** to efficiently read large datasets from multiple CSV files.
* **Data validation**: Before insertion, each record was checked for missing or malformed data.
* **Bulk insertion**: Once validated, the records were inserted into the PostgreSQL database using SQLAlchemy’s bulk insertion capabilities.

**Challenges:**

1. **Handling large datasets**: Inserting millions of records while ensuring that the process remains efficient.
2. **Data validation**: Ensuring data cleanliness and completeness before it is stored in the database.
3. **CSV consistency**: Different CSV files contained varying formats or had missing values, which had to be handled dynamically.

**Solution:**

* We implemented **bulk inserts** using SQLAlchemy to speed up the process.
* **Pandas** helped with **data validation** and cleaning before inserting the records, ensuring the database remained consistent.

A screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**Phase 3: Data Analysis**

Once the data was ingested, the analysis phase began. The goal here was to extract meaningful insights from the data using SQL queries and Python. This included:

* **Calculating average temperatures** over time.
* **Summing precipitation** across different weather stations and dates.
* **Filtering by station and date** to obtain specific data points.

**Challenges:**

1. **Efficient queries**: Some weather stations did not report data regularly, making it difficult to analyze consistent trends.
2. **Query optimization**: Queries needed to run efficiently over large datasets without causing performance bottlenecks.
3. **Handling missing data**: Some records were incomplete, requiring us to adjust queries to handle missing values gracefully.

**Solution:**

* We optimized queries using **indexes** on frequently queried fields like station\_id and date.
* **Pagination** was added to split large results into manageable chunks, improving performance and the API’s ability to handle large datasets.

A screenshot of a computer

Description automatically generated

**Phase 4: REST API Development**

The API, built using **Flask** and **Flask-RESTX**, exposed weather data through two key endpoints:

1. /api/weather: This endpoint retrieves raw weather data, with optional filters for station ID and date.
2. /api/weather/stats: This endpoint aggregates weather data, providing statistics like average temperatures and total precipitation per year and per station.

**Challenges:**

1. **Swagger and Flask-RESTX Issues**: Initially, we aimed to generate a **swagger.json** file manually to document the API. However, we encountered several issues when trying to configure Swagger with custom objects and complex responses. Serialization errors, missing fields, and improper type handling in **swagger.json** led to inconsistent API documentation.
2. **Error handling**: The API needed to provide clear error messages for invalid requests, missing data, or unexpected input.
3. **Pagination and Large Data**: Handling massive datasets required implementing pagination to ensure the API could serve data efficiently.

**Solution**:

* We transitioned from a manually created **swagger.json** file to using **Flask-RESTX**, which automatically generated Swagger documentation for the API. Flask-RESTX simplified the process and provided interactive documentation through the /swagger/ endpoint, which was more reliable than the manual approach.
* **Pagination** was implemented to divide large datasets into pages.
* Error handling was improved with consistent status codes and clear error messages to guide users on invalid requests.

**Problems Faced During Implementation**

1. **Environment Setup Issues**:
   * Installing PostgreSQL and configuring it on Windows led to several issues, such as version conflicts and problems with pgAdmin drivers.
   * **Solution**: A detailed, step-by-step setup guide was created to streamline future installations.
2. **Python Dependency Issues**:
   * Installing and managing dependencies in the virtual environment (particularly issues with the pandas package) caused significant delays.
   * **Solution**: A stable virtual environment was created using venv, and consistent versions of required libraries were maintained in a requirements.txt file.
3. **Database Errors**:
   * There were issues with SQLAlchemy mappings, especially when dealing with data types such as Decimal and handling RowMapping objects in JSON responses.
   * **Solution**: These issues were resolved by converting SQLAlchemy objects to native Python types, such as converting Decimal to float.
4. **Swagger/Flask-RESTX Integration**:
   * We initially tried generating a **swagger.json** manually, which caused significant issues related to data serialization and improper handling of complex objects. Configuring custom fields for Swagger documentation was difficult, and the **swagger.json** file often missed key fields or generated errors.
   * **Solution**: Switching to **Flask-RESTX** for automatic Swagger generation helped solve these issues. Flask-RESTX provided a built-in Swagger UI, which offered a more reliable way of generating and testing API documentation dynamically.

A screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. **Data Analysis Logging**:
   * During the data analysis phase, tracking errors or abnormal behaviors during runtime (such as missing data, validation failures, or issues with specific weather stations) was difficult, leading to confusion about where problems occurred.
   * **Solution**: We created a dedicated data\_analysis.log file to log critical information during data analysis. The log records details such as:
     + **Start and end times** for analysis runs.
     + **Errors** encountered during data processing, including missing data or invalid values.
     + **Data validation failures** or discrepancies in records (e.g., missing temperatures or precipitation).
     + **Performance metrics**, such as how long each query took to execute.

Logging was crucial for:

* + **Debugging**: If any issues arose, the log provided a clear trail of what happened and where the failure occurred.
  + **Auditing**: The log allowed us to keep track of data processing workflows, making it easier to backtrack if something was missed or went wrong.
  + **Performance tuning**: By examining the time each query took, we were able to optimize queries and improve overall system performance.

**AWS Cloud Deployment Approach**

**API Hosting:**

* **AWS Elastic Beanstalk**: This service would be used to deploy and manage the Flask API. Elastic Beanstalk offers automatic scaling and load balancing, making it an ideal choice for our API.
* **AWS Lambda + API Gateway**: For serverless deployment, AWS Lambda could be used in conjunction with API Gateway to expose the API. This approach would reduce operational overhead and scale automatically with traffic.

**Database:**

* **Amazon RDS (PostgreSQL)**: The PostgreSQL database would be hosted on Amazon RDS, offering automatic backups, scaling, and security management.

**Data Ingestion:**

* **AWS Lambda**: The data ingestion script could be scheduled to run automatically using AWS Lambda, triggered by **Amazon CloudWatch Events** to run at specific intervals (e.g., daily ingestion).
* **AWS S3**: Raw weather data files could be stored in S3, and ingestion could be triggered when new files are uploaded using **S3 event triggers**.

**CI/CD (Continuous Integration and Deployment):**

* **AWS CodePipeline**: This would be used to automate deployments whenever new code is pushed to the repository.
* **AWS CodeBuild**: CodeBuild would compile and test the application before it’s deployed to production.

**Monitoring and Logging:**

* **Amazon CloudWatch**: CloudWatch would be used to monitor the API’s performance, track errors, and set alarms based on thresholds for metrics like memory usage or latency.
* **AWS CloudTrail**: For auditing and tracking API usage, CloudTrail would provide logging of every API request.

**Security and Authentication:**

* **AWS IAM**: IAM roles would control access to various AWS services, ensuring that the principle of least privilege is followed.
* **Amazon Cognito**: If user authentication is required, Cognito would manage API users, ensuring that only authorized users can access the data