**Prerequisites:**

* **Snowflake Account**: Ensure you have access to a Snowflake account.
* **SnowSQL Client**: Install SnowSQL for command-line operations.
* **Raw Data Files**: Have the raw weather data files ready for upload.

**Problem 1:**

For this exercise, we’re working with weather data from multiple stations over several years. To store this data, I’ve designed a table that includes information on temperature and precipitation for all stations and years. We are using Snowflake/SnowSQL for this task. The code to create prerequisites[Database, schema, warehouse] and table to store the data can be found in the ‘***answers/create\_weather\_data\_table.sql’*** file.

**Explanation:**

* **Database and Schema:** The database is CORVETA\_AGRI and the schema is WEATHER.
* **Warehouse:** The LOAD\_WH warehouse is created if it doesn't exist.
* **Table Creation:**
  + wx\_id: An auto-incrementing primary key.
  + wx\_date: The date of the weather observation (non-nullable).
  + wx\_station\_id: Identifier for the weather station (non-nullable).
  + wx\_max\_temp: Maximum temperature recorded on the given date.
  + wx\_min\_temp: Minimum temperature recorded on the given date.
  + wx\_precipitation: Amount of precipitation recorded on the given date.
  + A unique constraint on the combination of date and station\_id to prevent duplicate records.

**Problem 2**:

DDL statements for this exercise can be found here “***answers\ingest\_weather\_data.sql***”.

For ingesting weather data into Snowflake:

1. **Setup**: Created a Snowflake stage to temporarily store raw data files and a logging table to record ingestion details.
2. **Process**: Utilized Snowflake’s MERGE statement within a stored procedure to ingest data. This process handles duplicates by ensuring no duplicate records are inserted.
3. **Efficiency**: Snowflake managed data ingestion efficiently:

* Raw to Stage: Completed < 20 seconds.
* Stage to Main Table: Completed in under 6 seconds.

1. **Logging**: The stored procedure logs the start and end times of the ingestion process, as well as the number of records ingested.
2. **File Handling**: Existing files are ignored upon repeated uploads, and only new files are processed.
3. **Results**: The procedure effectively manages duplicates and records ingestion metrics for monitoring.

A screenshot of a computer

Description automatically generated

Results from Raw files to STG\_Weather table

A screen shot of a computer

Description automatically generated

**Problem 3:**

DDL statement for this exercise can be found here “***answers\create\_weather\_stats.sql***”

For this exercise, I’ve used SnowSQL to calculate the required statistics (average max temperature, average min temperature, total precipitation) for every year and every weather station. Missing data has been ignored while calculation, and NULL was used for statistics that could not be calculated.

**Table creation:** New table to store the calculated statistics. This table will store:

* + ws\_year: Year of the observation
  + ws\_stationid:Weather station ID
  + ws\_avg\_max\_temp: Average maximum temperature
  + ws\_avg\_min\_temp: Average minimum temperature
  + ws\_total\_precipitation: Total accumulated precipitation

**Problem 4**:

I used Flask to build a REST API with two GET endpoints: /api/weather and /api/weather/stats. These endpoints let you filter by date and station ID, and they include pagination. You can find the code in ***weather\_stats\_API\app.py*** , Swagger documentation is set up, and I've included unit tests in ***weather\_stats\_API\test\_app.py***.

Weather API:

A close-up of a computer screen

Description automatically generated

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

A screenshot of a computer

Description automatically generated

Response:

A screenshot of a computer

Description automatically generated

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

A screenshot of a computer

Description automatically generated

Response:

A screenshot of a computer

Description automatically generated

Unit test results: Code in ***weather\_stats\_API\test\_app.py***

A black screen with white text

Description automatically generated

**Extra Credit - Deployment on AWS**

**Approach Overview**

For deploying my Flask-based weather API, Snowflake database, and data ingestion process on AWS, the best approach leverages Amazon Elastic Beanstalk, Amazon S3, AWS Lambda, and AWS Step Functions. This approach is optimal due to its simplicity, scalability, cost-effectiveness, and seamless integration with your existing Snowflake setup.

**1. Deploying the API**

* **Service**: **AWS Elastic Beanstalk**
* **Reason**: Elastic Beanstalk simplifies deploying and managing applications by automatically handling infrastructure provisioning, load balancing, and scaling. It supports Python and Flask, making it a good fit for my API.

**2. Deploying the Database**

* **Service**: **Snowflake (Already in Use)**
* **Reason**: Snowflake is already integrated into my setup, providing a powerful, scalable cloud data warehouse. I’d continue using it for data storage and analytics since it’s optimized for large-scale data operations.

**3. Scheduled Data Ingestion**

* **Service**: **AWS Lambda with AWS CloudWatch Events**
* **Reason**: AWS Lambda allows me to run my data ingestion code in a serverless environment, and CloudWatch Events can trigger these Lambda functions on a schedule. This setup is cost-effective and scales automatically with the workload.

**4. Supporting Services**

* **API Gateway**: To manage and route incoming requests to my Elastic Beanstalk environment.
* **S3 Bucket**: For storing raw data files before ingestion. This is especially useful if I need to upload or stage large amounts of data.

**Why This Approach?**

* **Seamless Integration**: Elastic Beanstalk and Lambda integrate well with other AWS services and provide automated scaling, which is ideal for handling varying loads.
* **Cost-Effective**: Using Lambda for scheduled tasks ensures that I only pay for the compute time I use, and Elastic Beanstalk’s management features reduce overhead costs.
* **Scalability**: Both Elastic Beanstalk and Snowflake scale easily, ensuring that my application and data processing can handle large amounts of traffic and data efficiently.

This setup leverages AWS’s managed services to simplify deployment, enhance scalability, and optimize costs, while keeping my existing Snowflake setup in place for robust data management.