**Detailed Documentation**

**Use Case 1**

**Assumptions**

1. The data is available on a daily or weekly basis in a folder named dataset.
2. The task is to extract relevant information from the dataset and store it in Snowflake.
3. No CI/CD pipeline is configured.
4. The Snowflake table has been created using the CreateTableScript.sql.
5. Requirements are installed using the requirements.txt file.

**Overview**

This script processes a large JSON file containing restaurant data for various cities and converts it into a CSV format. The script uses the ijson library for efficient, incremental parsing of large JSON files. The output CSV file contains restaurant details with fields separated by a pipe (|) delimiter. The script also includes error handling to ensure robustness.

**Dependencies**

* Python 3.x
* ijson library
* time library (for measuring script runtime)
* PySpark
* Snowflake Connector for Python
* Snowflake JDBC and Spark Snowflake Connector

**Installation of Dependencies**

pip install -r requirements.txt

**High-Level Architecture**

**A diagram of a file

Description automatically generated**

**Step-by-Step Processing**

1. **Data Availability**: Data is available in the dataset folder with the name data.json.
2. **JSON Data Conversion**: A Python file named **jsonDataConversion.py** processes this data and converts it into CSV format (with delimiter |). This step is necessary because the JSON structure is inconsistent. Some JSON objects are structured, while some keys are actual values needed in the final data (e.g., city name). This step takes approximately 42 seconds to complete.
3. **Data Processing and Loading**: Once the data is processed into CSV format, it is processed through Spark and loaded into the corresponding Snowflake table using a Python file named **processRestaurentData.py.**

**Enhancements and Explanations**

**jsonDataConversion.py:**

1. **Data Transformation:**
   * **Added normalize\_text function to ensure consistent text formatting.**
   * **Stripped white spaces from strings.**
2. **Data Integrity:**
   * **Processes data in the order it reads from the JSON file, maintaining data inflow order.**
   * **Used logging to ensure any errors are recorded.**
3. **Performance Optimization:**
   * **Added logging for better monitoring and error tracking.**
   * **Efficiently reads the JSON file line by line using ijson.**
4. **Documentation:**
   * **Added comments in the code for better understanding.**

**processRestaurentData.py:**

To run the processRestaurentData.py file, use the following command:

spark-submit --packages net.snowflake:snowflake-jdbc:3.13.22,net.snowflake:spark-snowflake\_2.12:2.12.0-spark\_3.4 processRestaurentData.py

**Explanations**

1. **Logging: Added logging to track the pipeline's performance and errors.**
2. **Data Normalization and Cleaning: Included steps to handle missing values and standardize text fields.**
3. **Error Handling: Wrapped the main code in a try-except block to handle errors gracefully.**
4. **Snowflake Configuration: Used a configuration file for all Snowflake-related information.**
5. **Performance Optimization: Optimized data cleaning and transformation steps. Implemented Snowflake truncate and load process.**

**Notes**

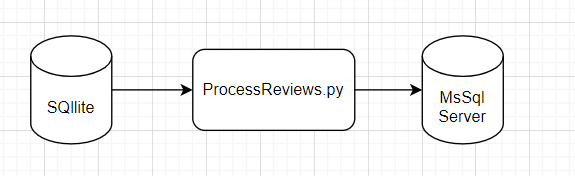
* **Ensure the PySpark version matches your Spark setup.**
* **snowflake-connector-python: Ensure compatibility with the other libraries.**
* **ijson: Used for parsing JSON data incrementally.**

**Requirement 2**

**Assumptions**

1. Data is present in the local system and pushed into a local MS SQL system.
2. Data processing options are provided here.
3. The script can be scheduled based on requirements using orchestration tools like Apache Airflow, SnapLogic, or cron jobs.
4. Requirements are installed using the requirements.txt file.

**High-Level Architecture**

****

**Explanation**

1. Bulk Insert with executemany: This significantly improves performance compared to inserting each row individually.
2. Data Normalization: Ensures consistent date formats, removes duplicates, and handles missing values.
3. Logging: Logs progress and any errors that occur during the process.
4. Configuration: Database connection parameters are read from a configuration file for flexibility.
5. Error Handling: Provides error handling for database connections and data processing steps.
6. Table Creation: Checks if the table exists and creates it if it doesn’t.

**This script is designed to run as a standalone Python script but can be adapted to run as a scheduled job using tools like cron or Apache Airflow for automation.**