Hello Fresh Data Processing Pipeline

In the HelloFresh Data Processing assignment, the focus was primarily on data processing. Task 1 involved utilizing Apache Spark and Python to read, preprocess, and store data in an optimized structure. For this task, I leveraged Azure Databricks as the environment for running Spark jobs and used Azure Blob Storage for file storage.

To organize the data processing pipeline, I created three distinct folders within Azure Blob Storage: Input, Processed, and Final Output. In the first step, data was read from the Input folder using the Azure Blob Storage ABFS (Azure Blob File System) driver in Spark. I then applied various transformation and preprocessing logic to clean and structure the data. After processing, the data was saved in Parquet format to the Processed folder, which is optimized for storage and query performance.

For Task 2, I read the Parquet files from the Processed folder and applied additional filtering based on the task's specific requirements. Finally, the filtered data was exported and stored in the Final Output folder in CSV format, making it ready for further analysis or reporting.

This approach ensured efficient data handling, optimized storage, and seamless processing using Azure Databricks and Azure Blob Storage.

Requirements: Microsoft Azure account for accessing the Databricks and Azure Blob Storage, Jupyter notebook for testing, Docker desktop, Visual Studio Code

NOTE: To access the data using S3 instead of Azure Blob Storage following changes are needed:

1. Set up AWS S3 Credentials

You need to configure AWS credentials for accessing the S3 bucket. This can be done in a few different ways:

- Using AWS IAM Roles (in an environment that supports it, e.g., AWS EMR or EC2): This is the most secure option if running on AWS.
- Using AWS Access Keys: In case you're running Spark on a local machine or a non-AWS environment, you will need to set your AWS Access Key ID and Secret Access Key.

2. Configuring AWS Credentials in Spark

```
spark.conf.set("spark.hadoop.fs.s3a.access.key", "<AWS_ACCESS_KEY_ID>")

spark.conf.set("spark.hadoop.fs.s3a.secret.key", "<AWS_SECRET_ACCESS_KEY>")

spark.conf.set("spark.hadoop.fs.s3a.endpoint", "s3.amazonaws.com")  # Or another region's endpoint if needed
```

3. Accessing S3 Bucket Files

Instead of using the Azure Blob Storage path, you will need to change the paths to point to the S3 bucket. The format for accessing files in S3 from Spark is s3a://<bucket-name>/<file-path>.

```
input_path = "s3a://<bucket-name>/path/to/files/*.json"
```

```
data = spark.read.json(input_path)
```

Task 1:

The task required using Apache Spark and Python to read source data, perform necessary preprocessing, and persist (write) it in a manner that ensures optimal structure and performance for further processing. The source data, in the form of event logs, is stored in the Input folder within Azure Blob Storage.

Execution Process:

1. **Reading the Source Data:**

    - The source event data was accessed from the Input folder in Azure Blob Storage using the Azure Blob File System (ABFS) driver, which allows Spark to efficiently read from the storage.

```
input_path = "abfss://input@storageaccountda.dfs.core.windows.net/*.json"
```

    - The data was loaded into a Spark DataFrame for further processing.

    Input Data: The provided input files are in JSON format and contain the following columns: **name**, **ingredients**, **description**, **URL**, **Image URL**, **cooktime**, **preptime**, **recipeYield**, and **datePublished**.

2. **Data Cleaning:**

    - Conversion from ISO to Minutes (Cooktime, Preptime):
      A custom function iso_to_minutes was implemented to convert ISO 8601 duration strings (e.g., "PT1H30M") into minutes. The function efficiently parses the input string, extracts hours and minutes, and returns the total duration in minutes. - It also includes error handling, logging any issues with invalid inputs.

      Example:
      Input: 'PT1H30M'

This input represents 1 hour and 30 minutes.
Output: 90 (1 hour * 60 minutes + 30 minutes)

- Filling Zero/Null Values:
  For cooktime and preptime columns, any zero or null values were replaced with the median value of the respective column to ensure consistent data for analysis.

3. Data Transformation:

- Number Extraction from RecipeYield:
  The function process_recipe_yield was created to extract numeric values from the RecipeYield column, which contains varying formats like "12 dozen", "serves 4", "about 5", and fractional values like "2 1/2". - This function normalizes the recipeYield values into a consistent numeric format, handling different formats and returning the yield in servings or portions.

  Examples:
  Input: '2 dozen'
  Output: 24 (since 2 dozen = 2 * 12 = 24)
  Input: '1 1/2'
  Output: 1.5 (converted from the fraction "1 1/2")

- Filling Zero/Null Values:
  Any zero or null values in the recipeYield column were replaced with the median value of the column to ensure completeness and consistency of the data.

4. Loading:

- After cleaning and transforming the data, the final dataset was written to the Processed folder in Azure Blob Storage.
- The data was stored in Parquet format, which is optimized for performance and can be efficiently used for further analysis in Task 2.

Task 2:

The preprocessed dataset from Task 1 needs to be further processed using Apache Spark and Python. This involves:

- Filtering the dataset to include only recipes that contain "beef" as an ingredient.
- Calculating the average total cooking time for each difficulty level (easy, medium, hard).
- Persisting the results in CSV format, with two columns: difficulty and avg_total_cooking_time.

Execution Process:

1. **Reading the Preprocessed Data:**
   - First, the preprocessed dataset from Task 1, which is stored in the Processed folder as Parquet files in Azure Blob Storage, will be read into a Spark DataFrame.

2. **Filtering Recipes Containing "Beef":**
   - The dataset will be filtered to include only rows where the ingredients column contains the word "beef".

3. **Calculating Total Cooking Time:**
   - The total cooking time is calculated by summing the cooktime and preptime columns. Since both columns are in ISO 8601 duration format, we will first convert them to minutes using the iso_to_minutes function defined earlier.

4. **Calculating Average Cooking Time per Difficulty Level:**
   - The dataset will be grouped by the difficulty column, and the average total cooking time will be calculated for each group.

5. **Persisting the Results:**
   - The final dataset, which contains the difficulty and avg_total_cooking_time columns, will be written to the **Output** folder in Azure Blob Storage as a CSV file.

```
  output_path=
"abfss://loggingoutputnew@storageaccountda.dfs.core.windows.net/output.csv"
```

Descriptive Analysis for Missing Values:

Using distribution in the context of handling missing data means understanding how the values for a column (like cookTime or prepTime) are spread out or distributed. It helps to determine the best way to handle missing values based on the characteristics of the data.

1.a. Distribution Analysis for cookTime_minutes (Input File 1: recipes-000):

- Count: 484 observations
- Mean: 23.20 minutes
- Standard Deviation (std): 38.08 minutes
- Minimum (min): 0 minutes
- 25th Percentile (25%): 0 minutes
- Median (50%): 15 minutes
- 75th Percentile (75%): 30 minutes

- Maximum (max): 420 minutes

Key Observations:

1. Presence of Zero Values:
    - Similar to cookTime_minutes, 25% of the records have 0 as their preparation time, potentially representing missing or invalid data.
    - Actionable Insight: These values should be handled similarly by imputation or removal.
2. Skewed Distribution:
    - The positive skew is evident here as well, with the median (15 minutes) being lower than the mean (23.20 minutes).
    - This distribution indicates that some recipes require unusually long preparation times, influencing the mean.
3. Outliers:
    - The maximum preparation time of 420 minutes also stands out as a potential outlier, requiring verification or treatment to prevent distortion in analysis.

Proposed Solution:

- Replace zero preparation times with the median (15 minutes) to address missing or invalid data.

1.b. Distribution Analysis for prepTime_minutes (Input File 1: recipes-000):

- Count: 484 observations
- Mean: 31.84 minutes
- Standard Deviation (std): 105.90 minutes
- Minimum (min): 0 minutes
- 25th Percentile (25%): 0 minutes
- Median (50%): 15 minutes
- 75th Percentile (75%): 20 minutes
- Maximum (max): 1440 minutes

Key Observations:

1. Presence of Zero Values:
    - Similar to cookTime_minutes, 25% of the records have a preparation time of 0 minutes, as indicated by the 25th percentile being 0.
    - These zero values are likely missing or invalid data, as most recipes would require some preparation time.
    - Actionable Insight: Imputation or exclusion of these values is necessary for accurate analysis.
2. Skewed Distribution:

    - The median preparation time (15 minutes) is much lower than the mean (31.84 minutes), indicating a positively skewed distribution.
    - This skewness suggests the presence of a few recipes with extraordinarily long preparation times that pull the mean upward.
3. Outliers:
    - The maximum preparation time of 1440 minutes (24 hours) is a significant outlier. It is highly unusual for a recipe to require such a long preparation time and warrants further investigation.
    - Actionable Insight: These extreme values should be validated and either capped or removed if deemed erroneous.

Proposed Solution:

- Replaced zero preparation times with the median (15 minutes), as the data is skewed and the median provides a more robust estimate.

2.a. Distribution Analysis for cookTime_minutes (Input File 2: recipes-001):

- Count: 244 observations
- Mean: 37.68 minutes
- Standard Deviation (std): 63.37 minutes
- Minimum (min): 0 minutes
- 25th Percentile (25%): 10 minutes
- Median (50%): 20 minutes
- 75th Percentile (75%): 31.25 minutes
- Maximum (max): 540 minutes

Key Observations:

1. Presence of Zero Values:
    - The minimum value is 0, and while the 25th percentile is 10 minutes, this suggests some records still have zero cooking times.
    - These zeros may represent missing or invalid data, as cooking time is rarely expected to be zero.
    - Actionable Insight: Imputation or removal of these zero values is necessary to ensure the dataset's integrity.
2. Skewed Distribution:
    - The median cooking time (20 minutes) is lower than the mean (37.68 minutes), indicating a positively skewed distribution.
    - A few recipes with very long cooking times are pulling the mean upwards, which can distort analysis.
3. Outliers:
    - The maximum cooking time of 540 minutes (9 hours) is a significant outlier, as it is much higher than the 75th percentile (31.25 minutes).
    - Actionable Insight: This extreme value should be investigated for validity. If deemed erroneous, it can be capped or excluded from the analysis.

Proposed Solution:

- Replaced records with zero cooking times using the median value (20 minutes) rather than the mean, as the data is skewed.

2.b. Distribution Analysis for prepTime_minutes (Input File 2: recipes:001)

- Count: 244 observations
- Mean: 45.28 minutes
- Standard Deviation (std): 143.59 minutes
- Minimum (min): 0 minutes
- 25th Percentile (25%): 10 minutes
- Median (50%): 15 minutes
- 75th Percentile (75%): 20 minutes
- Maximum (max): 1440 minutes

Key Observations:

1. Presence of Zero Values:
    - A minimum value of 0 suggests there are recipes with no recorded preparation time.
    - As preparation time is rarely zero, these are likely missing or invalid entries.
    - Actionable Insight: Imputation or exclusion of zero values is necessary to improve data accuracy.
2. Skewed Distribution:
    - The median preparation time (15 minutes) is significantly lower than the mean (45.28 minutes), indicating a positively skewed distribution.
    - A few recipes with extremely long preparation times are inflating the mean.
3. Outliers:
    - The maximum preparation time is 1440 minutes (24 hours), which is an extreme outlier.
    - This value is far above the 75th percentile (20 minutes) and seems unrealistic for most recipes.

Proposed Solution:

- Replaced records with zero preparation times with the median value (15 minutes) as it is more robust in skewed data.

3.a Distribution Analysis for cookTime_minutes (Input File 3: recipes-002):

- Count: 314 observations
- Mean: 9.43 minutes

- Standard Deviation (std): 18.74 minutes
- Minimum (min): 0 minutes
- 25th Percentile (25%): 0 minutes
- Median (50%): 0 minutes
- 75th Percentile (75%): 10 minutes
- Maximum (max): 180 minutes

Key Observations:

1. High Proportion of Zero Values:
    - The minimum, 25th percentile, and median cooking times are all 0 minutes, indicating that more than 50% of the recipes have zero cooking time recorded.
    - These zero values could represent missing or invalid data and must be handled for reliable analysis.
    - Actionable Insight: These zero values need to be imputed or excluded, as they likely do not reflect actual cooking times.
2. Skewed Distribution:
    - The mean cooking time (9.43 minutes) is higher than both the median (0 minutes) and the 75th percentile (10 minutes), suggesting a positively skewed distribution.
    - A small number of recipes with longer cooking times are pulling the mean upward.
3. Outliers:
    - The maximum cooking time is 180 minutes (3 hours), which is much higher than the 75th percentile (10 minutes).
    - While this is not as extreme as in previous datasets, it is still worth validating to ensure accuracy.

Proposed Solution:

- Replaced the zero cooking times with the 75th percentile value (10 minutes) since this percentile captures a more realistic lower bound for non-zero cooking times.

3.b. Distribution Analysis for prepTime_minutes (Input File 3: recipes-002):

- Count: 314 observations
- Mean: 11.14 minutes
- Standard Deviation (std): 31.54 minutes
- Minimum (min): 0 minutes
- 25th Percentile (25%): 0 minutes
- Median (50%): 0 minutes
- 75th Percentile (75%): 10 minutes
- Maximum (max): 245 minutes

Key Observations:

1. High Proportion of Zero Values:
    - The minimum, 25th percentile, and median preparation times are all 0 minutes, indicating that more than 50% of the entries have no recorded preparation time.
    - These zero values could represent missing or invalid data, as preparation times are rarely zero for recipes.
    - Actionable Insight: Address these zero values through imputation or exclusion to improve the dataset's quality.
2. Skewed Distribution:
    - The mean preparation time (11.14 minutes) is higher than both the median (0 minutes) and the 75th percentile (10 minutes), signifying a positively skewed distribution.
    - A small number of recipes with lengthy preparation times are inflating the mean.
3. Outliers:
    - The maximum preparation time is 245 minutes (just over 4 hours), which is significantly larger than the 75th percentile value (10 minutes).
    - While less extreme than previously observed maximum values, these outliers still need validation for accuracy.

Proposed Solution:

- Replace zero preparation times with the 75th percentile value (10 minutes) since it is a realistic lower bound for preparation times.

Logging

The logging code in our script sets up logging functionality to record various stages of the Spark job, ensuring that any issues or milestones during execution are logged for tracking and troubleshooting.

Here's a breakdown of how the logging code works:

1. Basic Logging Setup

```
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
```

```
logger = logging.getLogger()
```

2. Setting Up Console and File Handlers

```
log_file = "/Workspace/Projects/spark-
application/logs/logfile.log"  # Specify log file path

# Creating a console handler for output to console
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)
console_formatter = logging.Formatter('%(asctime)s -
%(levelname)s - %(message)s')
console_handler.setFormatter(console_formatter)

# Creating a file handler for logging to a file
file_handler = logging.FileHandler(log_file)
file_handler.setLevel(logging.INFO)
file_formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')
file_handler.setFormatter(file_formatter)

# Adding both handlers to the logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)
```

- **Console Handler**: This part makes sure that log messages are shown in the console (the screen you see while running the program). It will show messages of level INFO and above (like WARNING, ERROR, and CRITICAL).
- **Formatter for Console Handler**: This part decides how the log messages will look in the console (e.g., showing the time, log level, and message).
- **File Handler**: This part saves log messages to a file called logfile.log. Like the console handler, it saves messages of level INFO and above.
- **Formatter for File Handler**: This part decides how the log messages will look in the saved file (same as in the console).
- **Add Console Handler**: This adds the part that shows the log messages on the console.
- **Add File Handler**: This adds the part that saves the log messages to the file.

3. **Logging During the Process**
   In various parts of the script, you use logger.info() and logger.error() to log messages based on the status of different steps of the ETL process.

   Example:

   a. Logging the success of setting up Azure Blob Storage account key:

```
logger.info("Azure Blob Storage account key set
successfully.")
```

4. Logging at Critical Stages

```
except Exception as e:
```

```
    logger.error(f"Error reading data from input path {input_path}: {e}")
```

5. Summary of Log Levels

- **INFO**: Used to log general information such as successful task completion (e.g., "Data read from input path").
- **ERROR**: Used when there’s an issue or an exception during execution (e.g., "Error reading data from input path").