Analytics Engineer Practical Interview

Github repo analysis based on issue label

Summary - What will you find here:

A Comprehensive Guide for Data Analytics Implementation

1- Data Flow Solution:

General data flow drawing:

- Data Extraction: Python script using `requests` library to interact with GitHub's API. (Attachment)
- Data Warehouse/DB: Comparisons, recommendations and an example of integration for simple integration with various data flow tools.
- Data Processing & Orchestration: Apache Airflow setting
- Visualization: Tableau, Power BI, or Looker characteristics for easy integration with SQL databases and user-friendly visualization capabilities. Since those are all paid tools, the visualizations here are built in python using Deepnote.

2- Process:

- Develop a Python script or application that uses GitHub API to fetch issues information from a specified repository. The script should accept the repository name as an input argument.
- Extract data such as issue title, creation date, labels, and any other relevant information.
- Transform the data as required to fit the data storage schema, focusing on issue labels and creation dates.

3- Data Warehouse/DB Setup

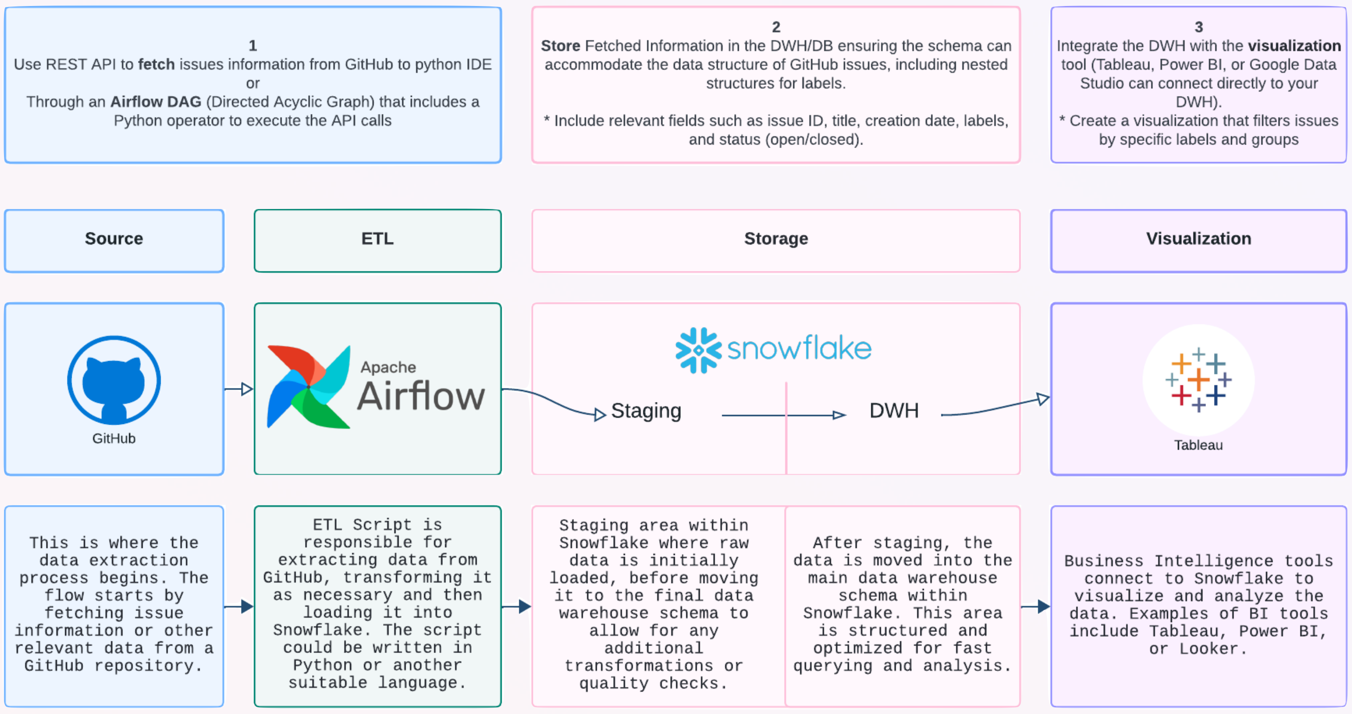
- Design a simple schema to store issues data, ensuring it supports querying issues by labels and creation dates.
- Consider tables for `Issues` and `Labels`, with a many-to-many relationship since an issue can have multiple labels and a label can be associated with multiple issues.
- Document the setup and usage instructions for the data pipeline and visualization dashboard.

4- Visualization Solution

- Design a dashboard that includes a visualization showing the number of issues created each month, grouped by labels for the year 2023.
- Ensure the visualization is clear, interactive (when possible), and allows users to filter by specific labels or time periods.

Data Flow Solution: A simple Python script can serve as intermediator, using libraries like *requests* for API interaction and *pandas* for data manipulation. For more complex pipelines, a tool like **Apache Airflow** can be used for orchestration. Airflow allows scheduling, monitoring, and orchestrating complex data flows. Some other solutions are:

- **Integration Tools:** Tools like Apache NiFi, Talend, or cloud-based services like AWS Glue and Azure Data Factory can help in orchestrating data flow.
- **Real-Time Processing:** If real-time data processing is necessary, streaming platforms like Apache Kafka or cloud services like Amazon Kinesis.
- **ETL vs. ELT:** Decide whether you'll transform data before loading (ETL) or load data and then transform it (ELT). ETL is traditional and offers pre-processing benefits, while ELT leverages the power of modern data warehouses to transform data.



1
Use REST API to **fetch** issues information from GitHub to python IDE
or
Through an **Airflow DAG** (Directed Acyclic Graph) that includes a Python operator to execute the API calls

2
Store Fetched Information in the DWH/DB ensuring the schema can accommodate the data structure of GitHub issues, including nested structures for labels.

* Include relevant fields such as issue ID, title, creation date, labels, and status (open/closed).

3
Integrate the DWH with the **visualization** tool (Tableau, Power BI, or Google Data Studio can connect directly to your DWH).
* Create a visualization that filters issues by specific labels and groups

This is where the data extraction process begins. The flow starts by fetching issue information or other relevant data from a GitHub repository.

ETL Script is responsible for extracting data from GitHub, transforming it as necessary and then loading it into Snowflake. The script could be written in Python or another suitable language.

Staging area within Snowflake where raw data is initially loaded, before moving it to the final data warehouse schema to allow for any additional transformations or quality checks.

After staging, the data is moved into the main data warehouse schema within Snowflake. This area is structured and optimized for fast querying and analysis.

Business Intelligence tools connect to Snowflake to visualize and analyze the data. Examples of BI tools include Tableau, Power BI, or Looker.

DWH/DB Selection:

- **Cloud vs. On-premise:** Cloud-based solutions like Amazon Redshift, Google BigQuery, and Snowflake offer scalability and managed services. On-premise solutions might be preferred for control and security reasons but require more management.

- **SQL vs. NoSQL:** SQL databases are typically structured and offer ACID transactions, suitable for transactional data. NoSQL databases like MongoDB, Cassandra, offer scalability and flexibility for semi-structured or unstructured data.

- **Analytical vs. Operational:** DWHs like Redshift, BigQuery, are optimized for analytical workloads and complex queries over large datasets. Operational databases like MySQL, PostgreSQL, are optimized for CRUD operations and transactional integrity

For this scenario, a cloud-based data warehouse like Snowflake, Google BigQuery or Amazon Redshift could be ideal due to their scalability, ease of integration with data visualization tools, and support for semi-structured data (like JSON from GitHub API responses). Each DWH have specific advantages. Next, they have been ranked from 1 to 5, being 5 the maximum for each criteria. The rank was created having a startup that is starting to scale in mind.

	Amazon Redshift	Google BigQuery	Snowflake
Scalability and Performance:	Designed for high-performance analysis on large volumes of data and can scale horizontally by adding more nodes.	Serverless, automatically scales to meet query demands without the need for manual cluster management.	Unique architecture allows for instant, automatic scaling of compute resources without impacting storage.
	4	5	5
Data Integration Capabilities:	Integrates well with other AWS services and various data sources, but might require additional services like AWS Glue for complex integrations.	It easily integrates with various Google services and supports streaming data for real-time analysis.	Offers native connectors and support for a wide range of data formats and sources.
	4	5	4
Data Storage and Management:	Efficient storage with columnar storage and data compression techniques.	Manages storage automatically, optimizing for performance without user intervention.	Uses a columnar storage format that is optimized for speed and efficiency, with separate compute and storage scaling.
	4	5	5
Query Performance:	Generally high, with optimizations for complex queries and large datasets.	Extremely fast for large datasets due to its Dremel technology and serverless model.	High performance, with the ability to scale compute resources dynamically to meet query demands.
	4	5	5
Security and Compliance:	Offers robust security features, including encryption, IAM roles, and compliance with various standards.	Strong security features with encryption, IAM, and compliance with numerous standards.	Robust security with always-on encryption and compliance with many regulatory standards.
	5	5	5
Cost:	Offers a pricing model based on node types and the number of nodes. Can be cost-effective but may become expensive at scale.	Pay-as-you-go pricing based on the amount of data processed by queries and stored data, which can be cost-effective for varying workloads.	Usage-based pricing that separates storage and compute costs, offering flexibility and potential cost savings.
	3	4	4
Ease of Use:	User-friendly for those familiar with SQL and AWS, though there may be a learning curve for managing and optimizing clusters.	Very user-friendly with a minimal learning curve, especially for those already using Google Cloud Platform.	Highly user-friendly with a simple interface and minimal management overhead.
	4	5	5
Support and Community:	Strong support through AWS and an extensive community and documentation.	Comprehensive support and a large community, given Google's extensive presence in the cloud industry.	Strong support options and a growing community due to Snowflake's increasing popularity.
	5	5	4

Is Journey Excellence within reach?

To facilitate a connection between a cloud data platform and Git (for version control), an environment for a tool or script that interacts with both systems can be setup by specifying connection details and defining some Git-related parameters for version control operations. Below is an example of a simple configuration file in YAML format.

Usage: This configuration file would be read by a script or application written in your language of choice (Python, Node.js, etc.). That code would handle the logic for connecting to Snowflake, performing operations, and then potentially using Git commands to version control relevant artifacts (e.g., SQL scripts, data models).

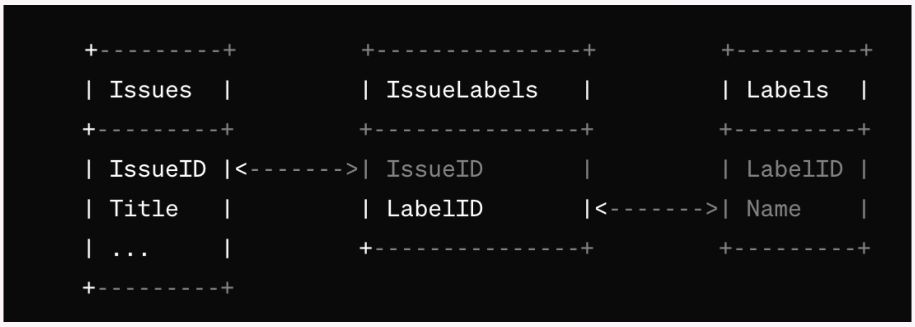
```yaml
snowflake:
  account: your_snowflake_account_identifier
  user: your_username
  password: your_password  database: your_database_name
  schema: your_schema_name
  warehouse: your_warehouse_name
  role: your_role

git:
  repository_url: https://github.com/your_username/your_repository.git
  branch: main
  commit_message: "Auto-commit from data pipeline"
  username: your_git_username
  password: your_git_password

```

- **Schema Overview:** To effectively store and query GitHub issues data, a relational database schema that captures the necessary details can be used. The schema includes tables for storing issues, labels, and a junction table to manage the many-to-many relationship between issues and labels (since one issue can have multiple labels and one label can be associated with multiple issues).

- **Entity-Relationship Diagram (ERD):**



```
+---------+          +--------------+          +---------+
| Issues  |          | IssueLabels  |          | Labels  |
+---------+          +--------------+          +---------+
| IssueID |<------->| IssueID       |          | LabelID |
| Title   |          | LabelID      |<------->| Name     |
| ...     |          +--------------+          +---------+
+---------+
```

1.**Issues Table:** Stores information about each issue.

   issue_id (Primary Key): Unique identifier for each issue.

   title: The title of the issue.

   creation_date: Date when the issue was created.

   status: The current status of the issue (e.g., open, closed).

   author: The username of the person who created the issue.

2. **Labels Table:** Contains the different labels that can be assigned to issues.

   label_id (Primary Key): Unique identifier for each label.

   label_name: The name of the label (e.g., bug, enhancement, documentation).

   label_description: The description assigned to the label

3. **IssueLabels Table:** A junction table to manage the many-to-many relationship between issues and labels.

   issue_id (Foreign Key): Links to the Issues table.

   label_id (Foreign Key): Links to the Labels table.

- **Visualization:** Some options are to use directly a Python/R/SQL IDEs, which requires previous programming language knowledge, or a business Intelligence tool. Some BI tools can be connect to the DWH directly to visualize and analyze the data, but they are in its vast majority paid tools.

- **Tableau -** Tableau turns complex data into simple visuals with an easy drag-and-drop feature. It's versatile, supports big data, and has a strong user community for support.
- **Microsoft Power BI -** is affordable and integrates well with Microsoft products. It's user-friendly, offers complex data analysis tools, and frequently updates with new features.
- **Qlik Sense -** Qlik Sense allows flexible data exploration with its unique associative engine, catering to intuitive insight discovery. It's scalable and integrates AI for smarter analysis.
- **Looker -** Looker simplifies data analysis with its web-based interface, making SQL queries accessible to all. It ensures consistent data metrics across teams and scales with business growth.

Analytics Engineer Practical Interview

Github repo analysis based on issue label

Attachments (code)

▶ Fetching data (Python and Airflow):

Step 1 - Generate a Personal Access Token (PAT) on GitHub:
1. Log in to GitHub.
2. Go to Settings > Developer settings > Personal access tokens.
3. Click on "Generate new token".
4. Give a descriptive name, select the scopes or permissions and click "Generate token" at the bottom.

▶ **Using Airflow -** Setting the DAG:

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
import requests
import pandas as pd
import sqlalchemy


GITHUB_TOKEN = 'your_github_token_here'
REPO_NAME = 'your_repo_name_here'  # Format: 'user_name/repo_name'
DATABASE_CONN_URI = 'your_database_connection_uri_here'

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2023, 1, 1),
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

## DAG Definition: Sets up the DAG with a start date, retry policy, and schedule interval.

dag = DAG(
    'github_issues_to_db',
    default_args=default_args,
    description='Fetch GitHub issues and store in DB',
    schedule_interval='@daily',  # Adjust as needed
    catchup=False  # Avoid backfilling past dates
)

## fetch_and_store_issues Function: This function uses the GitHub API to fetch issues from the
## specified repository and stores them in a SQL database.

def fetch_and_store_issues(ds, **kwargs):
    headers = {'Authorization': f'token {GITHUB_TOKEN}'}
    url = f'https://api.github.com/repos/{REPO_NAME}/issues'
    response = requests.get(url, headers=headers)
    issues = response.json()
    df_issues = pd.DataFrame(issues)
    engine = sqlalchemy.create_engine(DATABASE_CONN_URI)
    df_issues.to_sql('github_issues', engine, if_exists='append', index=False)
)

fetch_issues_task = PythonOperator(
    task_id='fetch_and_store_issues',
    provide_context=True,
    python_callable=fetch_and_store_issues,
    dag=dag
)
```

- **Using Python:** The requests library can be used to make an HTTP GET request to the GitHub API for accessing the issues of a specific repository.

```python
import requests
import pandas as pd

headers = {
    'Authorization': 'token {token_gh}',
    'Accept': 'application/vnd.github.v3+json'
}

# Constants
token_gh = ‘token_here'
user_name = 'DataDog'
repo_name = 'datadogpy'
labels_data = []

# Start with the first page
page = 1
per_page = 100

## The next lines iterates over each issue and its associated labels, extracting the desired information and appending it to a list of dictionaries (labels_data).
# Each dictionary in labels_data represents a row in the final DataFrame, containing information about the issue ID, issue title, label ID, label name, and label color.
# pd.DataFrame(labels_data) converts this list of dictionaries into a pandas DataFrame.

while True:
    url = f"https://api.github.com/repos/{user_name}/{repo_name}/issues?page={page}&per_page={per_page}"
    response = requests.get(url, headers)
    issues = response.json()

    # Break the loop if no issues are returned
    if not issues:
        break

    for issue in issues:
        for label in issue['labels']:
            labels_data.append({
                'issue_id': issue['id']
                , 'issue_title': issue['title']
                , 'status': issue['state']
                , 'created_at': issue['created_at']
                , 'updated_at': issue['updated_at']
                , 'closed_at': issue['closed_at']
                , 'label_id': label['id']
                , 'label_name': label['name']
                , 'label_description': label['description']
                , 'label_color': label['color']
            })
    # Increment the page number to fetch the next page in the next iteration
    page += 1

df_labels = pd.DataFrame(labels_data)
df_labels
```

- Storing the data: The fetched data can be stored in a database or a file system, depending on the scale and purpose. To do so, it is necessary to open a connection with the DB and send

```python
from sqlalchemy import create_engine
import os

user=os.environ["PRODUCTION_USER"]
password=os.environ["PRODUCTION_PASSWORD"]
host=os.environ["PRODUCTION_HOST"]
port=os.environ["PRODUCTION_PORT"]
database=os.environ["PRODUCTION_DATABASE"]

engine = create_engine(f'postgresql+psycopg2://{user}:{password}@{host}:{port}/{database}')

df.to_sql(‘file_name', engine, schema='schema_name', index=False, if_exists=‘append’)
```