Dashboard Therory

The provided code is a Streamlit-based dashboard designed for monitoring the execution of ETL jobs, analyzing their status, execution time, and dependencies. Here's a detailed line-by-line explanation of the code:

Imports

```
import streamlit as st
import pandas as pd
import altair as alt
import networkx as nx
import matplotlib.pyplot as plt
from query_utils import fetch_data

```

- `streamlit as st`: Streamlit is used to create the web dashboard interface.
- `pandas as pd`: Pandas is used for handling data in DataFrame format.
- `altair as alt`: Altair is a declarative statistical visualization library used for creating interactive charts.
- `networkx as nx`: NetworkX is used for creating and visualizing graphs (e.g., job dependency graphs).
- `matplotlib.pyplot as plt`: Matplotlib is used for drawing and displaying plots.
- `from query_utils import fetch_data`: This imports a custom utility function `fetch_data` that is likely used for querying data from an external source (e.g., a database).

Sample DataFrame for Testing

```
def get_sample_data():
    """Create a sample DataFrame for testing."""
    sample_data = {
        "job_id": ["001", "001", "001", "001", "001"] + ["002", "002", "002", "002", "002"] + ["003"] + ["004"],
        "business_date": ["2024-12-01", "2024-12-02", "2024-12-03", "2024-12-04", "2024-12-05"] + ["2024-12-01", "2024-12-02", "2024-12-03", "2024-12-04", "2024-12-05"] + ["2024-12-01"] + ["2024-12-01"],
        "status": ["Success", "Success", "Success", "Success", "Success"] + ["Success", "Success", "Success", "FAILED", "FAILED"] + ["FAILED"] + ["Success"],
        "start_time": ["2024-12-01 15:00:00", "2024-12-02 15:00:00", "2024-12-03 15:50:00", "2024-12-04 15:00:00", "2024-12-05 15:30:00"]  + ["2024-12-01 15:00:00", "2024-12-02 15:00:00", "2024-12-03 15:00:00", "2024-12-04 15:00:00", "2024-12-05 15:00:00"] + ["2024-12-01 18:00:00"] + ["2024-12-01 02:00:00"],
        "end_time": ["2024-12-01 15:10:00", "2024-12-02 15:15:00", "2024-12-03 16:10:00", "2024-12-04 15:12:00", "2024-12-05 15:41:00"] + ["2024-12-01 16:00:00", "2024-12-02 15:25:00", "2024-12-03 15:20:00", "2024-12-04 15:05:00", "2024-12-05 15:55:00"] + ["2024-12-05 19:01:00"] + ["2024-12-05 02:13:00"],
        "execution_time": [10, 15, 20, 12, 11] + [60, 25, 20, 5, 55] + [61] + [13]
    }
    return pd.DataFrame(sample_data)

```

- This function generates a sample dataset as a pandas DataFrame, which contains mock data on job executions, including job IDs, business dates, status, start/end times, and execution times. The data is used for testing purposes when the user is in "Development Mode."

Streamlit UI Setup

```
st.title("ETL JOB Execution Monitoring Dashboard")
```

```
2
```

- This sets the title of the Streamlit app.

```
dev_mode = st.sidebar.checkbox("Use Sample Data for Testing", value=False)

```

- A checkbox in the sidebar allows the user to switch between using sample data and querying real data (from a database).

```
tab1, tab2 = st.tabs(["Main Dashboard", "Job Dependency Graph"])

```

- This creates two tabs in the app: one for the main dashboard and one for the job dependency graph.

Main Dashboard (tab1)

```
with tab1:
    # Query Options
    QUERY_OPTIONS = {
        "By Business Date": "SELECT * FROM your_table WHERE business_date = ?",
        "By Business Date and Job ID": "SELECT * FROM your_table WHERE business_date = ? AND job_id = ?",
        "By Business Date Range": "SELECT * FROM your_table WHERE business_date BETWEEN ? AND ?"
    }

```

- The `QUERY_OPTIONS` dictionary defines different types of SQL queries that can be made to a database to fetch job data based on the business date, job ID, or a date range.

`get_data` function

```
def get_data(query_type, params):
    if dev_mode:
        df = get_sample_data()
        ...
        return filtered_df if not filtered_df.empty else pd.DataFrame()  # Return empty DataFrame if no data
    else:
        query = QUERY_OPTIONS[query_type]
        data, columns = fetch_data(query, params)
        return pd.DataFrame(data, columns=columns) if data else pd.DataFrame()  # Return empty DataFrame if no
data

```

- This function fetches data based on the query type (`By Business Date`, `By Business Date and Job ID`, or `By Business Date Range`).
- If `dev_mode` is enabled, it uses the sample data generated by `get_sample_data()`. Otherwise, it executes a query using `fetch_data()` and returns the result.

Sidebar Inputs for the Main Dashboard

```
query_type = st.sidebar.radio("Select Query Type", list(QUERY_OPTIONS.keys()))

```

- This creates a radio button in the sidebar to choose the query type (e.g., by business date, job ID, etc.).

Data Fetching and Filtering

```
if st.sidebar.button("Fetch Data"):
    st.session_state.data = get_data(query_type, params)

```

- When the "Fetch Data" button is clicked, it fetches the data based on the selected query type and parameters, and stores it in `st.session_state.data`.

Data Processing and Display

```
if "data" in st.session_state and st.session_state.data is not None:
    data = st.session_state.data
    if data.empty:  # Check if the DataFrame is empty
        st.warning("No data available for the selected criteria. Please change your selection.")

```

- This block processes the data fetched, ensuring it's not empty and displaying a warning message if it is.

Metrics Section

```
st.markdown("### Metrics")
col1, col2, col3, col4 = st.columns(4)
col1.metric("Total Records", len(filtered_data))

```

- Displays metrics (e.g., total records, success count, failed count, and average execution time) in a row of four columns.

Data Visualization (Execution Trend)

```
st.markdown("### Job Execution Trend (By Hour)")
hour_data = filtered_data.groupby("start_hour").size().reset_index(name="count")
bar_chart = alt.Chart(hour_data).mark_bar().encode(...)
st.altair_chart(bar_chart, use_container_width=True)

```

- This section visualizes the number of jobs executed by the hour of the day using an Altair bar chart.

Heatmaps

```
st.markdown("### Heatmap of Execution Times")
heatmap_data = filtered_data.pivot_table(index="job_id", columns="status", values="execution_time",
aggfunc="mean")
st.dataframe(heatmap_data)

```

- Creates heatmaps that show the average execution times, based on the job ID and status.

Job Dependency Graph (tab2)

```
with tab2:
    # Sample DataFrame for dependent jobs
    def get_dependent_jobs():
        """Create a DataFrame for dependent jobs."""
        dependent_jobs_data = {
            "job_id": ["001", "001", "001", "002", "003", "004", "005", "abc", "xyz"],
            "dependent_job_id": ["002", "004", None, None, "001", "005", None, None, "abc"],
        }
        return pd.DataFrame(dependent_jobs_data)

```

- This function creates a sample DataFrame with job dependencies (e.g., job "001" depends on job "002" and "004").

Dependency Graph Visualization

```
G = nx.DiGraph()
```

```
dependent_jobs = dependent_jobs_df.dropna(subset=["dependent_job_id"])  # Filter out rows with no dependencies

```

- This initializes a directed graph `G` and filters out rows without dependencies, ensuring only jobs with dependencies are considered.

```
fig, ax = plt.subplots(figsize=(10, 10))
nx.draw(G, pos, with_labels=True, node_size=3000, node_color="skyblue", font_size=10, font_weight="bold", ax=ax)

```

- This creates and visualizes the job dependency graph using NetworkX. The nodes represent jobs, and the directed edges represent dependencies.

Final Thoughts:

- This Streamlit app provides interactive tools for monitoring ETL job executions, including querying data, applying filters, visualizing job execution trends, and exploring job dependencies in a graph.

This code defines a Streamlit application to monitor ETL job execution and visualize job dependencies. It uses several Python libraries, including Pandas, Altair, NetworkX, and Matplotlib, to fetch, process, and display data related to job execution. Here's an overview of the structure and functionality:

Key Components:

1. **Main Dashboard (Tab 1)**:
   - **Query Options**: The user can filter data based on specific criteria such as business date, job ID, or a date range.
   - **Data Fetching**: Depending on the selected query type, the app either fetches data from a database or uses a sample dataset for testing purposes.
   - **Filtering and Display**:
     - The user can filter the data based on job ID, status (success or failed), and execution time.
     - Displays key metrics like the total number of records, success count, failure count, and average execution time.
     - Data visualization:
       - A bar chart shows job executions by hour.
       - A heatmap visualizes execution times for different job statuses or business dates.
2. **Job Dependency Graph (Tab 2)**:
   - **Job Dependencies**: A sample dataset of job dependencies is provided, where each job can have dependent jobs.
   - **Graph Configuration**: Users can configure the job dependency graph by selecting specific job IDs, highlighting dependencies, and setting the maximum depth of the graph.
   - **Dependency Graph Visualization**:
     - A directed graph is generated using NetworkX to visualize job dependencies.
     - The user can highlight dependencies and adjust the graph's depth to explore job relationships.

Explanation of Specific Parts:

- **Sample Data Generation**:
  - The `get_sample_data` and `get_dependent_jobs` functions generate sample datasets for testing purposes. These datasets simulate job execution records and job dependencies, respectively.
- **Dynamic Data Fetching**:

    - The function `get_data` is used to fetch filtered data based on user input, either from a sample dataset (when `dev_mode` is checked) or an external data source using a query function (`fetch_data`).
- **Interactive Components**:
    - The Streamlit app uses various components like `st.sidebar.radio`, `st.sidebar.date_input`, `st.sidebar.selectbox`, and `st.sidebar.slider` to allow users to interact with the app and filter data.
    - `st.download_button` lets users download the filtered data as a CSV file.
- **Visualizations**:
    - **Altair**: The execution trend is visualized using a bar chart to show the number of jobs executed each hour of the day.
    - **Matplotlib and NetworkX**: The job dependency graph is visualized using NetworkX's `spring_layout`, which positions nodes based on their connections, and Matplotlib to render the graph.

Potential Issues or Enhancements:

- **Database Integration**: The code uses `fetch_data` to fetch data from a database, but this is not implemented in the sample code. You'll need to implement or connect a database for real-world usage.
- **Scalability**: The app uses in-memory DataFrames, so it may not scale well for very large datasets. For large-scale systems, optimizations for data loading, processing, and visualization would be needed.

Further Improvements:

- **Error Handling**: Add more robust error handling for cases like invalid data, database connection failures, or empty inputs.
- **Real-Time Updates**: Implement a mechanism to fetch and display real-time updates for job execution or status changes.
- **Interactive Graph Features**: Enhance the dependency graph with features like zooming, hovering for additional info, or exporting the graph as an image.

This structure should provide a good foundation for building an ETL job monitoring and dependency visualization dashboard.