# Project guide, and questions answers:

This project is a web application that processes page view data to provide insights into visitor sessions on websites. The main version of the application is built using Streamlit for the web interface and Pandas for data processing. The data is provided as CSV files, and the application supports queries to determine the number of sessions, the median session length, and the number of unique visited sites for a given visitor (Including 2 bonus queries as well).

Table of Contents

[Project guide, and questions answers: 1](#_Toc169116030)

[Project Structure: 1](#_Toc169116031)

[Streamlit Application: 1](#_Toc169116032)

[Data Loading and Processing: 2](#_Toc169116033)

[Query Functions: 3](#_Toc169116034)

[How to Set Up, Run, and Use the Project: 3](#_Toc169116035)

[Setup Instructions: 3](#_Toc169116036)

[Using the Application 4](#_Toc169116037)

[Streamlit Configuration 4](#_Toc169116038)

[Local version notebook 4](#_Toc169116039)

[Scaling the Solution 5](#_Toc169116040)

[About the Pyspark solution: 6](#_Toc169116041)

[Advantages of doing the processing in Pyspark: 7](#_Toc169116042)

[Explanation of Databricks Widgets and Integration with Slack: 7](#_Toc169116043)

[Using the Databricks notebook: 8](#_Toc169116044)

[Space and Time Complexity 8](#_Toc169116045)

[Testing 9](#_Toc169116046)

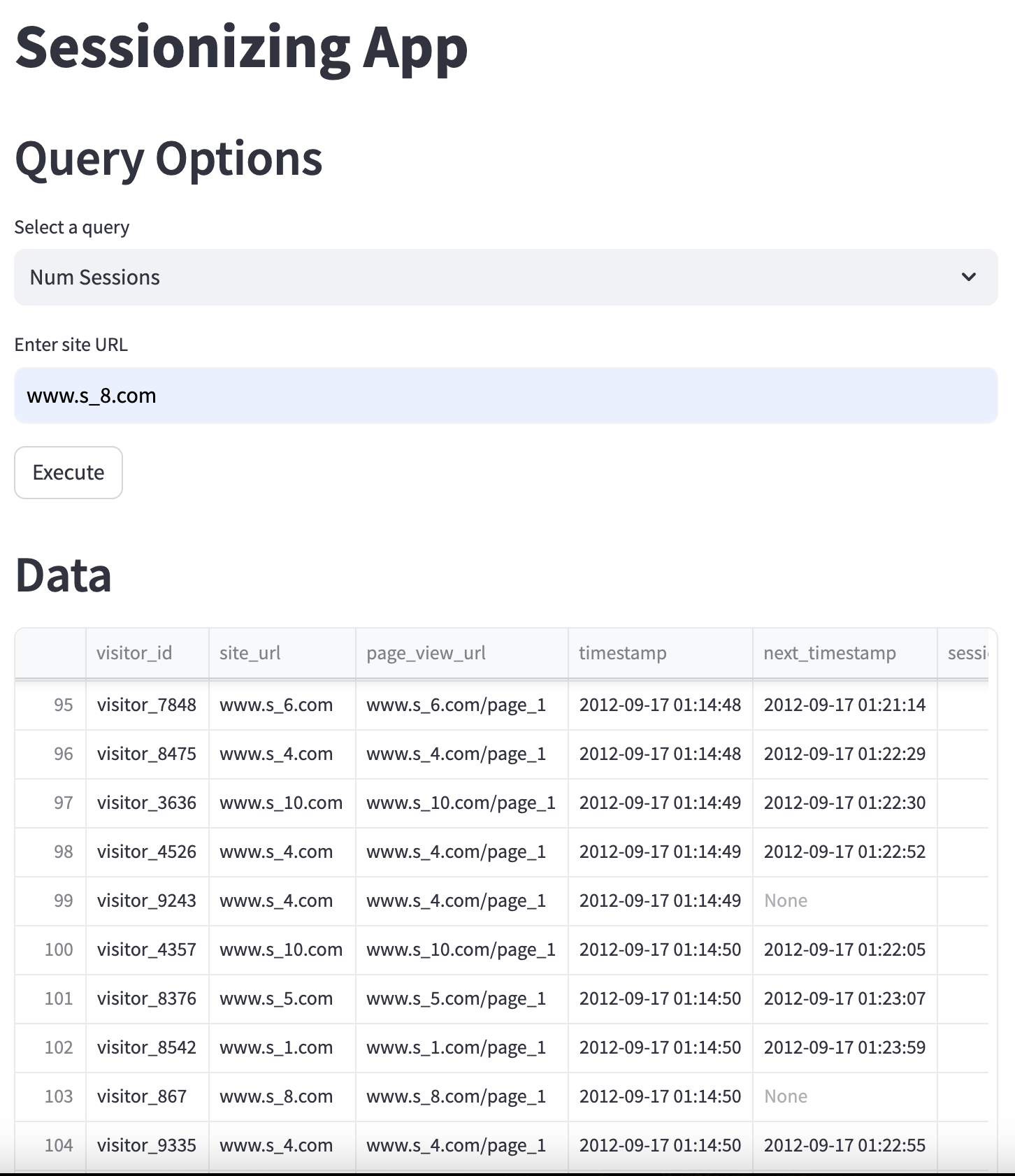
## Project Structure:

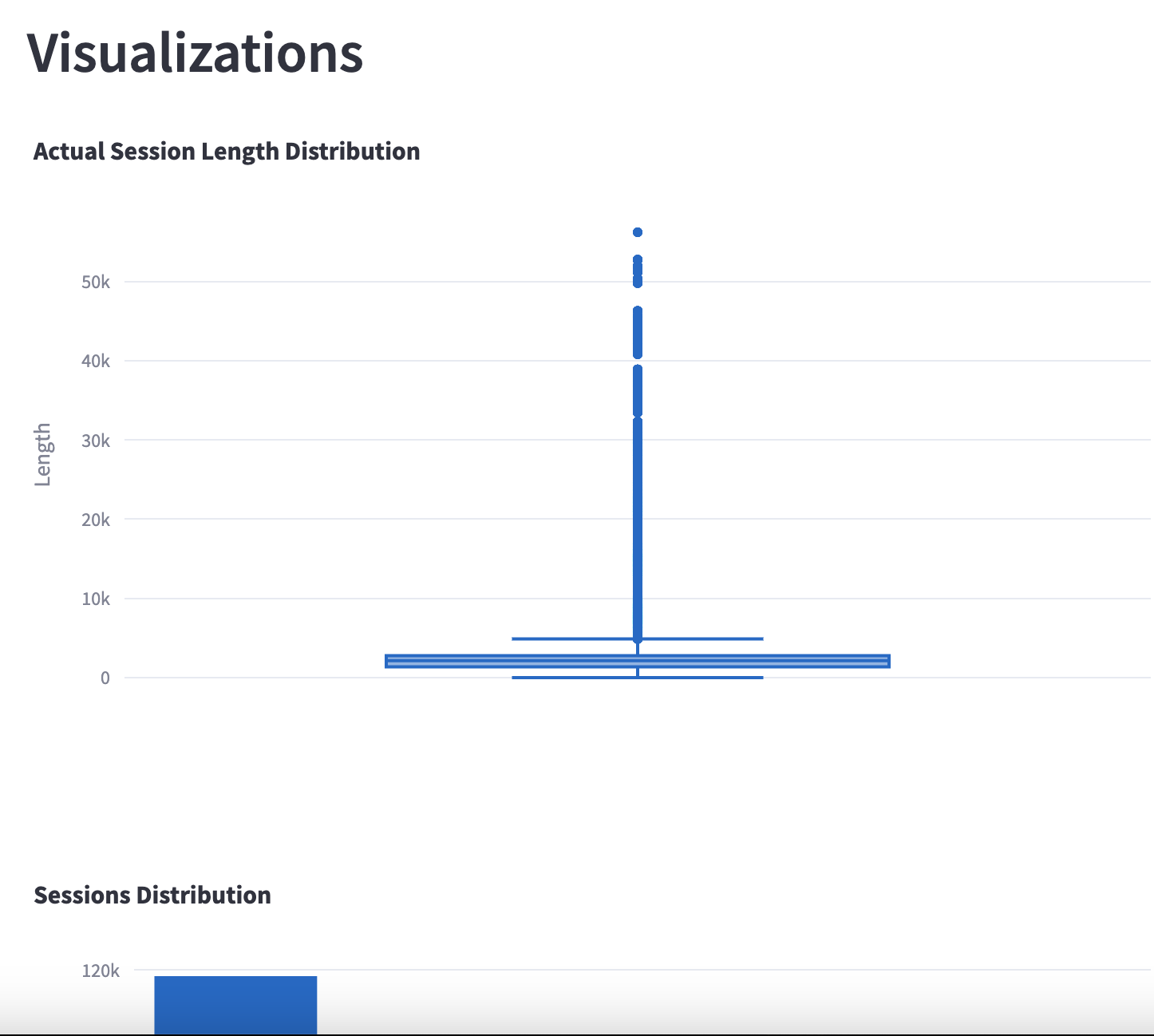
1. app.py: The main Streamlit application script.
2. utils.py: Contains utility functions for data processing and query execution.
3. test\_sessionizing\_app.py: Unit tests for the application.
4. environment.yml: Conda environment configuration file.
5. streamlit\_config/config.toml: Streamlit configuration file.
6. Given files from SW (3 data files, instructions, and expected results).
7. A folder with extra versions of the code:
   1. Pyspark version written in Databricks for large scale (using S3), will not work on your computer, but can be reviewed.
   2. Jupyter notebook local version, used to learn and preprocess the data before the final scripts. This notebook will also work with local user insertion of the query names and needed inputs from inside the notebook.

## Streamlit Application:

The Streamlit application (`app.py`) provides a user interface for querying the data. It includes the following features:

1. A dropdown menu to select the query type.
2. Text input fields for `site\_url` and `visitor\_id`.
3. Buttons to execute the queries and display the results.
4. Visualizations of the data, including a box plot of session lengths and a bar chart of session counts.
5. Displays a message indicating if CSV files were loaded, which files were loaded, and if columns were consistent across all files.
6. Returns an error message if the `visitor\_id` or `site\_url` entered by the user does not exist in the data.





* After investigating possible solutions, I decided to go with Streamlit as it is looking better than the basic GUI of running python through CLI with Argparse or similar libraries, or using a more complex Rest API.  
  Streamlit fits perfectly for our needs.   
  Though there were some challenges with it as well, as the errors and alerts from Streamlit are dull and hard to understand. So most logics were written before I wrapped everything with Streamlit.

### Data Loading and Processing:

The `load\_and\_process\_data` function in `utils.py` is responsible for loading and processing the CSV files. It performs the following steps:

1. Checks if there are any CSV files in the folder.
2. Verifies column consistency across all CSV files.
3. Loads all CSV files into Pandas DataFrames.
4. Concatenates the DataFrames and removes duplicates and null values (There were none in the given files).
5. Converts the `timestamp` column to datetime format.
6. Sorts the DataFrame by `timestamp`, `visitor\_id`, `site\_url`, and `page\_view\_url`.
7. Calculates the `next\_timestamp` for each page view.
8. Calculates the `session\_length` based on the `next\_timestamp`.
9. Identifies new sessions based on a 30-minute threshold.
10. Assigns a unique `session\_id` to each session.
11. Calculates the actual session length as the difference between the first and last timestamp in each session.
12. Returns the processed DataFrame, a message indicating column consistency, and the names of the loaded files.

### Query Functions:

The following query functions are defined in `utils.py`:

1. `num\_sessions(df\_filtered, site\_url)`: Returns the number of sessions for a given site URL.
2. `median\_session\_length(df\_filtered, site\_url)`: Returns the median session length for a given site URL.
3. `num\_unique\_visited\_sites(df\_filtered, visitor\_id)`: Returns the number of unique visited sites by a given visitor.
4. `session\_page\_sequence(df\_filtered, visitor\_id, site\_url, session\_id)`: Returns the sequence of pages visited within a session.
5. `most\_visited\_pages(df\_filtered, site\_url)`: Returns the most visited pages for a given site URL.

### How to Set Up, Run, and Use the Project:

Prerequisites – Anaconda or Miniconda installed on your system.

### Setup Instructions:

1. Clone the Repository (Or download and unzip the project folder instead):

git clone <repository\_url>

cd Home\_Assignment\_Sessionizing

2. Create and Activate the Conda Environment:

conda env create -f environment.yml

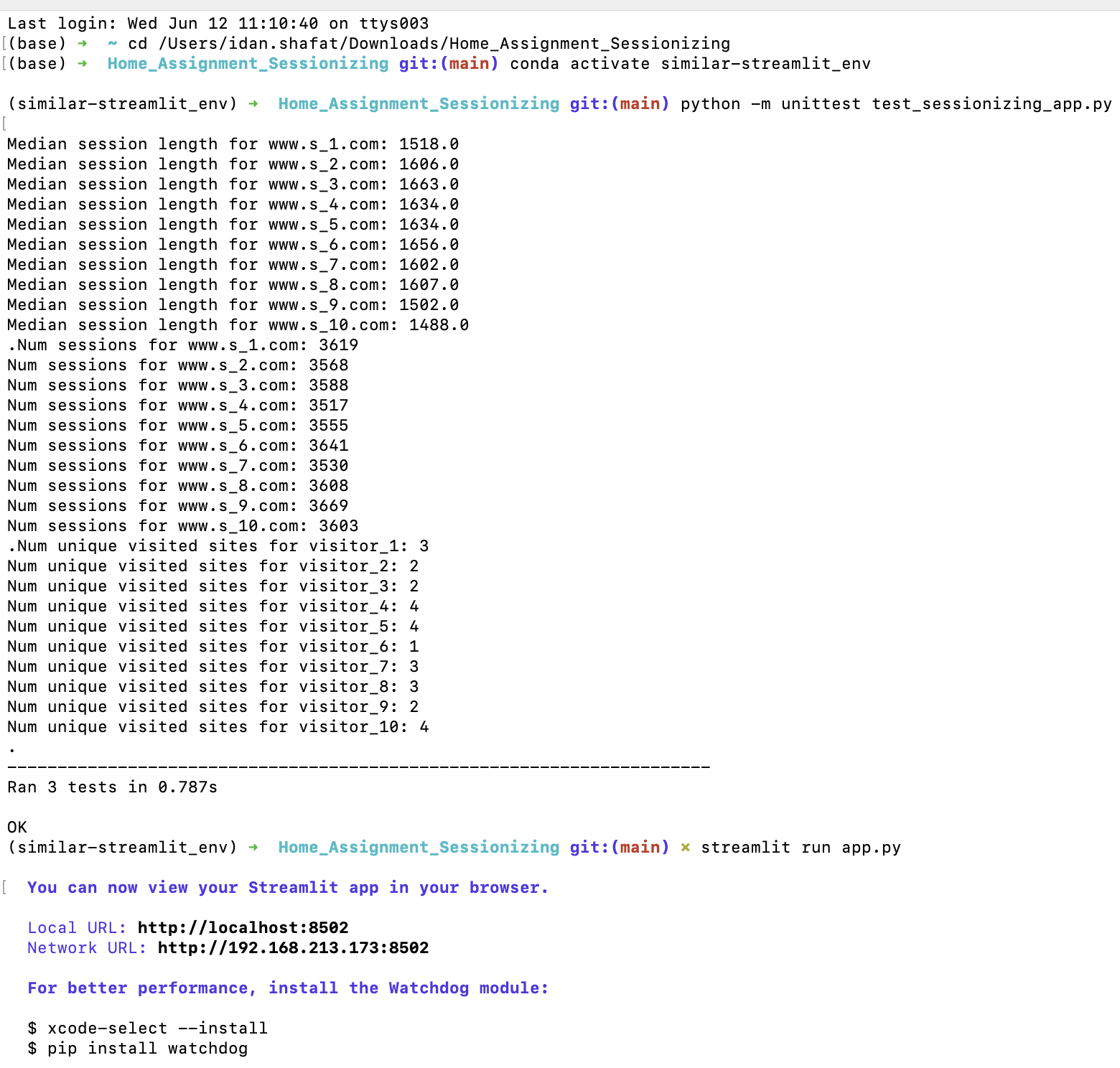
conda activate similar-streamlit\_env

3. Run Unit Tests:

python -m unittest test\_sessionizing\_app.py

4. Run the Streamlit Application:

streamlit run app.py



* If you are not using Pandas, lets set the environment with venv -
  + Install venv if python version velow 3.3
  + Create a virtual environment while inside the directory, using CLI:  
    ‘python3 -m venv.venv’
  + Activate environment
    - Mac: source.venv/bin/activate
    - Windows: .venv\Scripts\activate
  + Create a requirements.txt file listing the packages you need:
    - Pandas, plotly, streamlit
  + Install all packages from the file:
    - python -m pip install -r requirements.txt

### Using the Application

1. The commands will open the Streamlit application in your web browser (default URL: `http://localhost:8501`).
2. Select a query type from the dropdown menu.
3. Enter the required input values (`site\_url` or `visitor\_id`).
4. Click the "Execute" button to run the query and display the results (Utils notebook uses a decorator to Cache the data so that multiple executes are faster and lighter).

### Streamlit Configuration

The `streamlit\_config/config.toml` file is optional and can be used to configure the Streamlit server settings. For example, it can specify the port on which the Streamlit application runs.

## Local version notebook

I also created a Jupyter notebook, placed in the other versions folder.

I used this notebook to investigate and learn the data before writing the application, pre process, EDA, and wrote the first versions of the functions and usage.

This notebook can receive users imput and will run the queries locally.



* Already when I made this version I saw how the session lengths distributes, and that there are some extremely long sessions in the data. Can those be people who forgot their computer on with the website open and not going to screensaver? Do we want to remove these as outliers? In the current version of the project I did not touch these.

## Scaling the Solution

To support large-scale input data, the following changes can be made:

* Distributed Computing: Use a distributed computing framework like Apache Spark to process large datasets.
* Data Storage: Store the data in a distributed file system like DBFS or a cloud storage service like AWS S3.
* Database Integration: Use a scalable database like Apache Cassandra or Amazon DynamoDB to store and query the data.
* Partitioning: Partition the data based on a suitable key (like timestamp, visitor\_id, or site\_url) to distribute the workload across multiple nodes.
* Broadcast Variables: Use broadcast variables for small lookup tables to reduce shuffling -   
  Broadcast variables in PySpark are read-only shared variables that are cached and available on all nodes in a cluster. They are used to efficiently distribute large data to all worker nodes, reducing the need for data shuffling and improving performance.   
  We can also use temp views to cache data in databricks if we need a subset of big, live data, so we will save the needed subset in our session lead to faster excecutions of tasks.
* Optimize Joins: Optimize join operations by using the correct join, repartitioning data, or using broadcast joins for smaller datasets.
* Lazy Evaluation: Leverage Spark's lazy evaluation model to optimize performance by minimizing unnecessary computations (Same as Streamlit) -   
  Meaning transformations on RDDs (Resilient Distributed Datasets) are not executed immediately, but recorded as a lineage of transformations to be applied when an action is called.   
  This allows Spark to optimize the execution plan and minimize unnecessary computations (saves time and compute power)
* Handling Data Skew with Salting (we can detect a stuck worker through Ganglia)
  + Data skew occurs in distributed systems (Spark) when some partitions have significantly more data than others, leading to performance bottlenecks and inefficient utilization of cluster resources.
  + Salting is a technique used to handle data skew by adding a random value to the join key. This helps distribute the data more evenly across partitions, reducing the chances of any single partition becoming a bottleneck and ensuring that all nodes in the cluster are utilized efficiently, leading to faster job completion.
    - Identify the Skewed Key: Determine which key is causing the data skew.
    - Add Salt: Add a random salt value to the skewed key.
    - Repartition: Repartition the data based on the salted key.
    - Perform the Join: Use the salted key for the join operation.
  + By following these steps (If needed), we can mitigate the issue of data skew and improve the performance of your Spark jobs in Databricks.

### About the Pyspark solution:

The provided code in the other versions folder uses PySpark (Python API of spark) to process and analyze web page view data in a high scale using cloud storage and clusters for compute systems.

If I would finalize the PySpark version, there would also be a utils notebook there so it will have the functions and stuff on the utils notebook ran by the main notebook.

We can use integrate this with several other technologies to make a complete, automated solution, some options for large scales of data are:

1. Using Airflow
   1. Author, schedule, and monitor workflows
   2. Define a Directed Acyclic Graph (DAG) for our workflow.
   3. Schedule tasks to run at specific intervals.
   4. Use Airflow operators to run Spark jobs, move data, and perform other tasks.
2. Using DBT (Data Build Tool)
   1. Enables data analysts and engineers to transform data in their warehouse more effectively, relevant for the filtered and modified version of the data we create for our function usage.
   2. Define models in SQL or Python.
   3. Run transformations on the data warehouse.
   4. Schedule DBT runs using a scheduler like Airflow.
3. Using Scheduled Databricks Jobs
   1. Databricks Jobs allows us to run notebooks or Python scripts on a schedule by time or other triggers, in addition to having a full control on the cluster that the job will be ran on.
   2. Monitor job runs, handle failures, and monitor cluster workers performance through Databricks workflows, UI, and Ganglia
4. Making a Databricks Dashboard and Alerts through Databricks SQL
   1. Databricks SQL allows you to create dashboards and set up alerts based on simple SQL queries, and triggered by schedule or other automations.

### Advantages of doing the processing in Pyspark:

1. In the project we wanted to maintain Chronological Order when combining data from multiple CSV files. This is inherently handled by Spark when using union or unionAll.
2. Handle Overlapping Timestamps: Given the data is sorted within each file but may overlap across files, the Sessionization logic should correctly aggregate sessions even when timestamps from different files fall within the same session criteria (e.g., less than 30 minutes apart).

### Explanation of Databricks Widgets and Integration with Slack:

A screenshot of a computer

Description automatically generated

Integrating Databricks with Slack allows you to automate notifications and actions based on the results of your queries. This can be particularly useful for monitoring and alerting purposes.   
Databricks widgets allow you to add interactive input parameters to your notebooks and dashboards. This feature enhances the interactivity and usability of your notebooks, making it easier to run multiple queries with different parameters without modifying the code each time, in addition, widgets can be integrated into slack integrations with tools like Workato, that will enable anyone from the company to trigger these queries with a very simple way, and that way create self service for non technical workers. (Like in the project I shared on my last interview). The integration is made with Databricks API and slack API.

### Using the Databricks notebook:

* The Databricks widgets create a dropdown menu and text inputs for executing different queries. This allows users to select the type of query they want to run and provide the necessary parameters interactively (website URL, visitor ID).
* The notebook attached is also saving (Mock credentials) the final filtered DF that will be used for the queries as a research table on snowflake. So it can also be queried through the DB, and each time the notebook is updated with new CSVs, the new modified and filtered DF will be appended to the history of this data.

## Space and Time Complexity

* Space Complexity: The space complexity is primarily determined by the size of the input data (All CSV’s). The application loads the entire dataset into memory, so the space complexity is O(N), where N is the number of rows in the dataset (~145k in our project). Each record occupies space proportional to the number of attributes.
* Time Complexity: The time complexity for aggregations, loading, and processing the data is O(n log n) for each file due to the sorting operation. The overall time complexity can increase with the number of files and operations.

1. Num Sessions
   * Space Complexity: O(1) for each query execution since it operates on a subset of the DataFrame based on the site\_url.
   * Time Complexity: O(N) for filtering the DataFrame and O(1) for calculating the number of unique session IDs.
2. Median Session Length
   * Space Complexity: Similar to Num Sessions, O(1) for each query execution.
   * Time Complexity: O(N) for filtering and sorting the DataFrame, and O(N log N) for finding the median.
3. Num Unique Visited Sites
   * Space Complexity: O(1) for each query execution.
   * Time Complexity: O(N) for filtering the DataFrame and O(1) for calculating the number of unique visited sites.

## Testing

1. The application includes unit tests for individual functions (`test\_sessionizing\_app.py`) to verify the correctness of the query functions. The tests cover various scenarios and use predefined test cases to compare the actual results with the expected results.
2. Integration tests to verify that the entire flow works as expected, from reading CSV files to generating insights and visualizations.
3. Manual tests of the functions and comparing to expected results.