Big Data course project

Sydorenko Valeriia, Omelkina Daria

Overview:

The system, which was created for this course project, reads and processes Wikipedia page creation stream data and provides two types of API categories to users. This project involves data storage and modeling, and batch and ad hoc processing.

Link to GitHub repository: [Big Data project: WIKI stream]

Step 1: Requirements and Goals of the System

Our system should meet the following requirements:
Functional Requirements:

1. Users should be able to choose a question from two sets of APIs.
2. Users should be able to provide parameters of their choice for some of the questions.
3. Users should be able to retrieve data for the question of their choice at any time and independently of other users.

Non-Functional Requirements:

1. The system should be available. This is required because, if either the web application or data storage will become unavailable none of the functional requirements of the system will be met.
2. Requests should take minimal time to be performed.
3. The system should be immune to errors that might occur due to damaged request data or wrong/malicious user-provided parameters.
4. The system should be accessible through REST API.

Step 2: Back-of-the-envelope estimation

- Scale is expected from the system: the system should be able to handle get requests to the Wikipedia stream, meaning that not only it should effectively retrieve the data, but also quickly process it and send it further to the database. Because of this, we used an already available python Requests framework, because it is expected to be more optimized than any manually created request frameworks that we might create in a short amount of time.
- Storage needs: In order to work with large amounts of data (5 tables for category B API and 4 for category A, each handling some kind of data from Wikipedia stream messages),

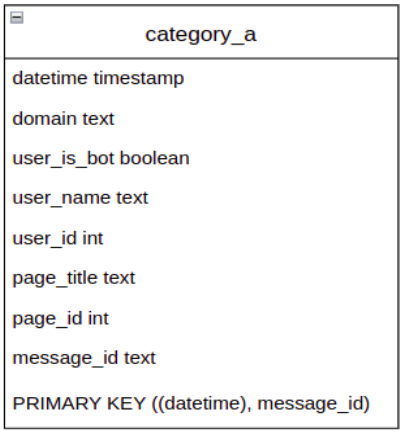
we chose a NoSQL database — Cassandra, which is considered better for such data amounts than a relational database (like Postgres).

- We expect traffic to be quite busy on the communication with the stream side (as described above) and less busy on the web application side. Yet, for the web application, we also use a ready and reliable Flask framework, which will allow us to handle multiple users at the same time.

Step 3: Defining the data model

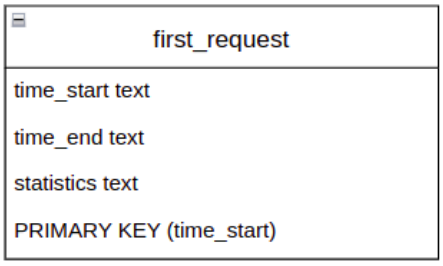
The data model for Category A requests:

category_a: this is the table where data is stored right after retrieving from the stream. It is later used to retrieve data for Spark processing.
The fields are:



The primary key consists of partition key datetime and clustering key message_id. The datetime field was chosen as the partition key because the data is retrieved by hours for Spark processing. The message_id is the clustering key in order to guarantee the uniqueness of the record.

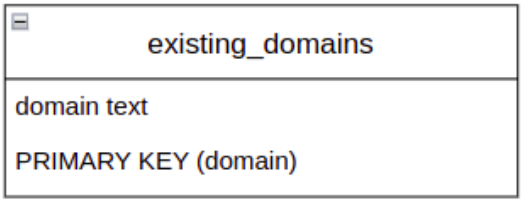
first_request, second_request, third_request: statistics for each of the request is stored in the corresponding table. The structure of the tables is the same. The fields are:



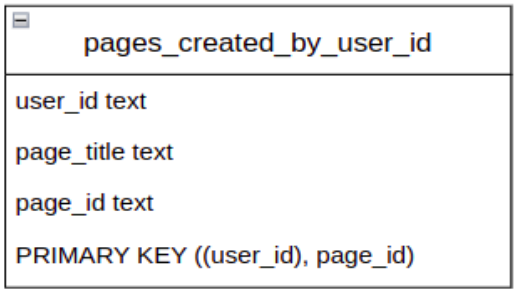
The time_start was chosen as the primary key because we retrieve statistics for the request by a certain hour.

The data model for Category B requests:

existing_domains: this is the table for storing and retrieving all existing domains for which pages were created.
The structure is:

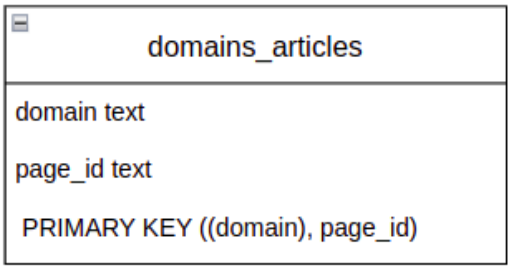


pages_created_by_user_id: this is the table for storing and retrieving all the pages created by the specified user.
The structure is:



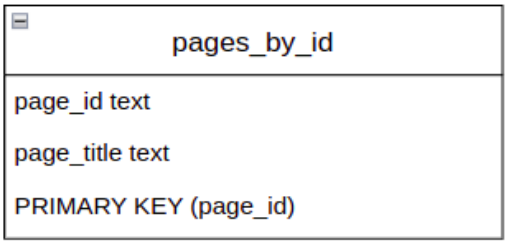
The primary key consists of partition key user_id and clustering key page_id. The user_id was chosen as the partition key because the data is retrieved for the specified user (‘where user_id =

<certain user_id>' in select statements). The page_id is clustering key in order to guarantee the uniqueness of the record.
Domains_articles: this is the table for retrieving the number of articles created for a specified domain.
The structure is:

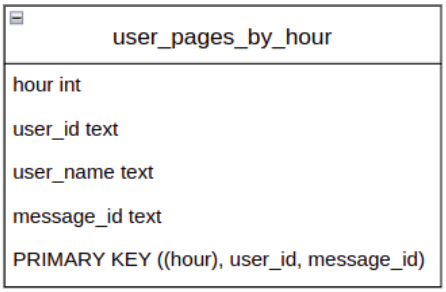


The primary key consists of partition key domain and clustering key page_id. The domain was chosen as the partition key because the data is retrieved and grouped by the specified domain. The page_id is a clustering key in order to guarantee the uniqueness of the record.

Pages_by_id: this is the table for retrieving the page with the specified page_id.
The structure is:



user_pages_by_hour: this is the table for retrieving the id, name, and the number of created pages of all the users who created at least one page in a specified time range.
The structure is:



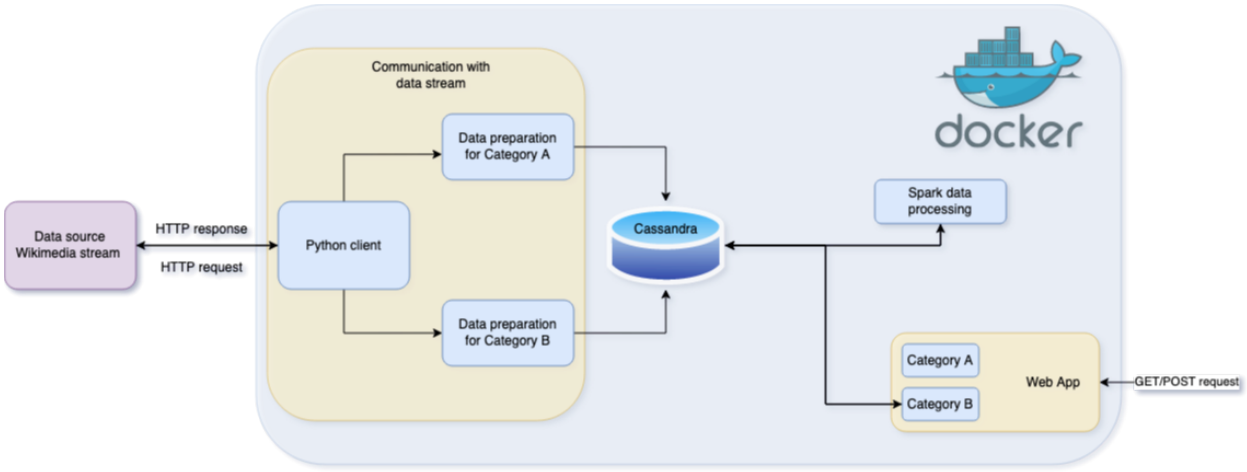
The primary key consists of partition key hour and composite clustering key user_id and message_id. The hour was chosen as the partition key because the data is retrieved for a time period chosen by the user and grouped by the user. The message_id is a clustering key in order to guarantee the uniqueness of the record.

Step 4: High-level design

There are three main parts to our system design:

1. A module that is responsible for communication with the Wikipedia stream (our data source)
2. Our database: Cassandra
3. Module for processing category A data using Spark
4. Web application for communication with the user through predefined API of both categories

All of the components are designed to be installed in the same docker network, but running in different containers. The whole system is run and managed through **Docker Compose**, which we also used for adding health checks and preparing Cassandra's keyspace for work.



Step 5: Detailed design

Let's dive deeper into some of the main parts of the system:

1. Communication with Wikipedia stream:
    a. Part of the module is dedicated to solely obtaining Wikipedia stream messages (we use GET requests with the "stream" parameter for this task), a minimum preprocessing (full message is actually retrieved in few parts: "id" and "data", so we wait until we got both of them before sending the message for further

processing) and redirection of the message into Class, dedicated to the communication with the database.

b. Class, which is responsible for communication with data storage, connects to Cassandra and performs two types of operations: message processing, and sending data:
    i. Processing allows for creating a dictionary, which only contains data relevant to the available API calls: timestamp (date-time), day, hour, domain, user_is_bot, user_name, user_id, page_title, page_id, message_id.
    ii. Afterward, data is divided between 6 different tables and is sent into the Cassandra database. 5 of these tables will be used for direct selects in order to answer requests from category B, the sixth table will be used as a data source in the Spark processing module and will produce 3 more tables for category A requests.

2. Data storage (Cassandra):
   The data model, which we used for the Cassandra is described in one of the sections above. We launch and prepare Cassandra using Docker Compose in order to be able to check if the Cassandra is ready for communication with other containers (using health check). The Cassandra is used for storing: tables with data for category B queries, 1 table with full data for Spark processing, and 3 tables with results of the Spark processing. Four containers communicate with the Cassandra container:
    a. “cassandra-load-keyspace” which is meant just for the creation of keyspace and tables in Cassandra and then retires.
    b. “Communicator” uses python client in order to access the keyspace in Cassandra and inserts prepared data to appropriate tables using query execution.
    c. “Spark” also uses the python client to retrieve data from the table and after processing inserts data into the other 3 tables.
    d. “Web-app” also uses the python client and for each API request from the user performs the appropriate SELECT query and retrieves data from the Cassandra.

3. Module for processing category A data using Spark
    a. The module contains the class SparkProcessor that is responsible for retrieving data from Cassandra, its processing, and saving results to Cassandra.
    b. At first the connection to Cassandra is established with the connect_to_db() method. Then, at the beginning of every hour the update() method is called, where separate methods for each report type are called. Inside each of the methods, the data for the last 6 hours (excluding last hour) is retrieved and new reports are computed using Spark. The result statistics for each hour are saved as a string to the corresponding table in Cassandra with starting hour (for which the report was computed) as the primary key.

4. Application for communication with the user:
    a. The application connects to the Cassandra database using the same python client as the other containers.
    b. The application provides two sets of APIs to the user:
        i. Category A: focused on pre-computed queries, uses only Cassandra.
        ii. Category B: focuses on ad-hoc queries, and uses Spark for data processing, but results are still saved in the Cassandra.
    c. The application also provides a front-end — web interface for easier communication with users, which allows easily choose a question from any API set and provide parameters. After we retrieve data from Html forms we process it and perform appropriate SELECT queries to the Cassandra database. Retrieved results are prepared for easier visual representation and are then used as parameters for an Html page with results.
    d. In order to serve this application, we use Flask. Considering that it is run inside of a docker container we just map the default 5000 port into the local machine to access the website.
    e. For page retrieval, we use GET requests, for making API calls we use POST requests with the parameters (if any) provided by the user.

Step 6: Identifying and resolving bottlenecks

Possible bottlenecks:

1. Considering that we have (almost) microservice architecture monitoring of the system and keeping the separate parts in check can present a challenge. To resolve this problem we put the whole system into the Docker compose and in there we manage containers by performing health checks and choosing which container depends on which and with what conditions. For example, we load keyspace only when Cassandra is already up and healthy, and we run the web app only when loading keyspace was successfully finished and retired.
2. One of the most prominent bottlenecks is the only one module for communication with the data stream, which is not asynchronous in our case and might be a little slow for retrieval.
3. Another important bottleneck is the Web application, which is used as the API getaway. It is problematic because we might have a lot of users connecting to the web app and sending a lot of requests per time unit. This problem can be partially solved by scaling the web application. The scale can be performed using Docker compose, but it presents challenges for mapping the application port into the local machine. If the port was only exposed to other docker containers, there would be no such problem and scaling would work well.

4. In order to solve any problems that might arise from using simple data storage (like plain text), we use a better database (Cassandra) which already has a lot of improvements by design.
5. Another problem, which we did not yet solve is malicious user queries, which are now not filtered in any way.