GOOGLE ANALYTICS LIKE BACKEND SYSTEM

Abstract

This document represents the architectural design of an analytics system like Google Analytics capable of gathering the data from interactions of users with application and reporting it in a customizable way to help merchants to monitor and improve their business

Akhilesh Gupta

INDEX

1. Overview

This document is prepared to design a Google Analytic like Backend System a high-level solution design for the backend system. Use of any open source tools is allowed.

2. Background

Google Analytics is a web analytics service offered by Google that tracks and reports website traffic. It is used to track the website activity of the users such as session duration, pages per session, bounce rate, etc. along with the information on the source of the traffic. Its approach is to show high-level, dashboard-type data for the casual user, and more in-depth data further into the report set. Google Analytics analysis can identify poorly performing pages with techniques such as funnel visualization, where visitors came from (referrers), how long they stayed on the website and their geographical position.

3. Goals:

Build a system capable of:

- Handle large write volume: Billions write events per day.
- Handle large read/query volume: Millions of merchants want to get an insight into their business. Read/Query patterns are time-series related metrics.
- Provide metrics to customers with at most one-hour delay.
- Run with minimum downtime.
- Have the ability to reprocess historical data in case of bugs in the processing logic.

4. Non-goals:

Building the following points are part of the system but can't be realized unless more clarity/information is provided:

- Tracking of user interactions and their respective channels.
- UI Design of the dashboard and the parameters to visualize the data.

5. Proposed Design:

5.1 Assumptions: Since there many specifics of requirement missing, we will assume the following things to assist us in understanding the design. The assumptions are as follows:

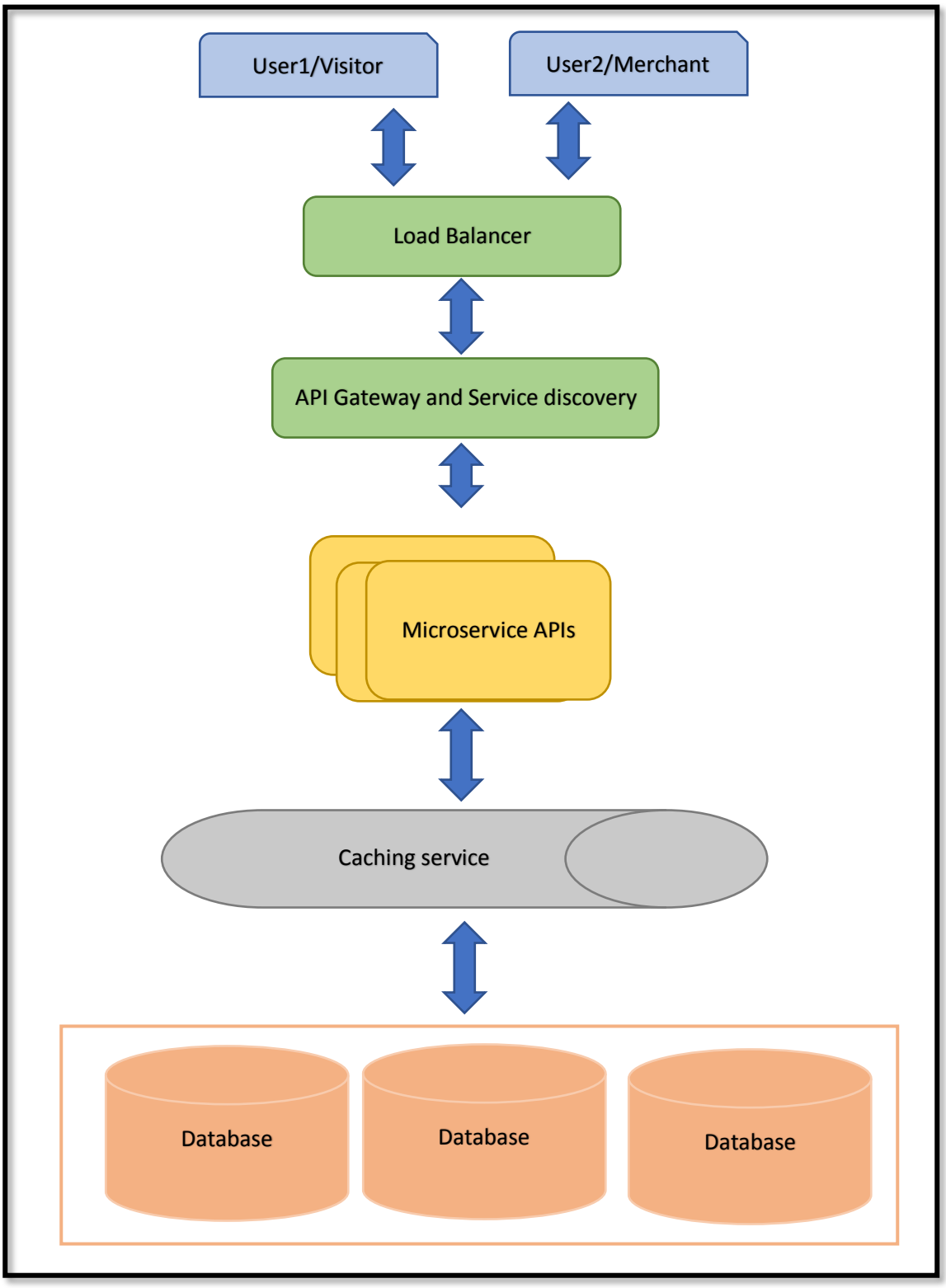
- Use of distributed system architecture to manage the load of processing high volumes of data and reliability.
- Reporting UI contains a dashboard to provide users the information based on different parameter and metrics stored in distributed databases.
- The data model for the system will contain only the details about sessions, browser, and location as part of data.

5.2 System Architecture: The system architecture can be divided into the following components:

- Load Balancer
- API gateway
- Microservices API

- Distributed database
- Caching service

Following is the flow of interactions between different levels of the system. The user interacts with the API through gateway and load balance which manages the request load. Microservices process these request and fetch/commit data from DB accordingly.



The details of each component are provided below:

1. **Load Balancer:** As the system needs to serve hundreds of thousands, maybe millions, of concurrent requests from users or clients and return the corresponding response to each one of them. To cost-effectively scale to meet these high volumes, best practice is to add more servers. A load balancer will come in front of these servers and route client requests across all servers capable of fulfilling those requests to maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. There are many open source load balancers available. Below are top 3 which can be used as per requirement and compatibility:
    - **Seesaw:** It is a reliable Linux-based virtual load balancer server to provide necessary load distribution in the same network. It is an open source system developed and used also by Google.
    - **LoadMaster:** it is load balancer developed by Kemp Inc. It can either be implemented in the system or used after deploying in the cloud. It is used by some of the big brands like Apple, Sony, JP Morgan, Audi, Hyundai, etc. It provided features like firewall, intrusion prevention, tunneling, etc.
    - **HAProxy:** It is one of the popular systems in the market to provide high-availability, proxy, TCP/HTTP load-balancing. HAProxy is used by some of the reputed brands in the world, like Airbnb, GitHub, Imgur, Reddit, etc.

2. **API Gateway and Service discovery:** Since the system is using microservices it is important to expose a single consolidated endpoint rather than many having many service APIs representing the same task. It is a widely used practice to use API gateway when using microservice architecture. Since the request is a single entity and many services can be used to fetch the required response, service discovery in the service mesh will be implemented. Some API gateways currently used are:
    - **Spring Boot API Gateway:** Spring framework provides a library for building API gateway on top of the microservices which can also be designed using Spring.
    - **Kong API Gateway:** Kong is a scalable, open source API gateway for developers giving them a centralized management layer on top of microservices and APIs that power the system. Kong allows developers to manage authentication, data encryption, logging, rate limiting, etc.

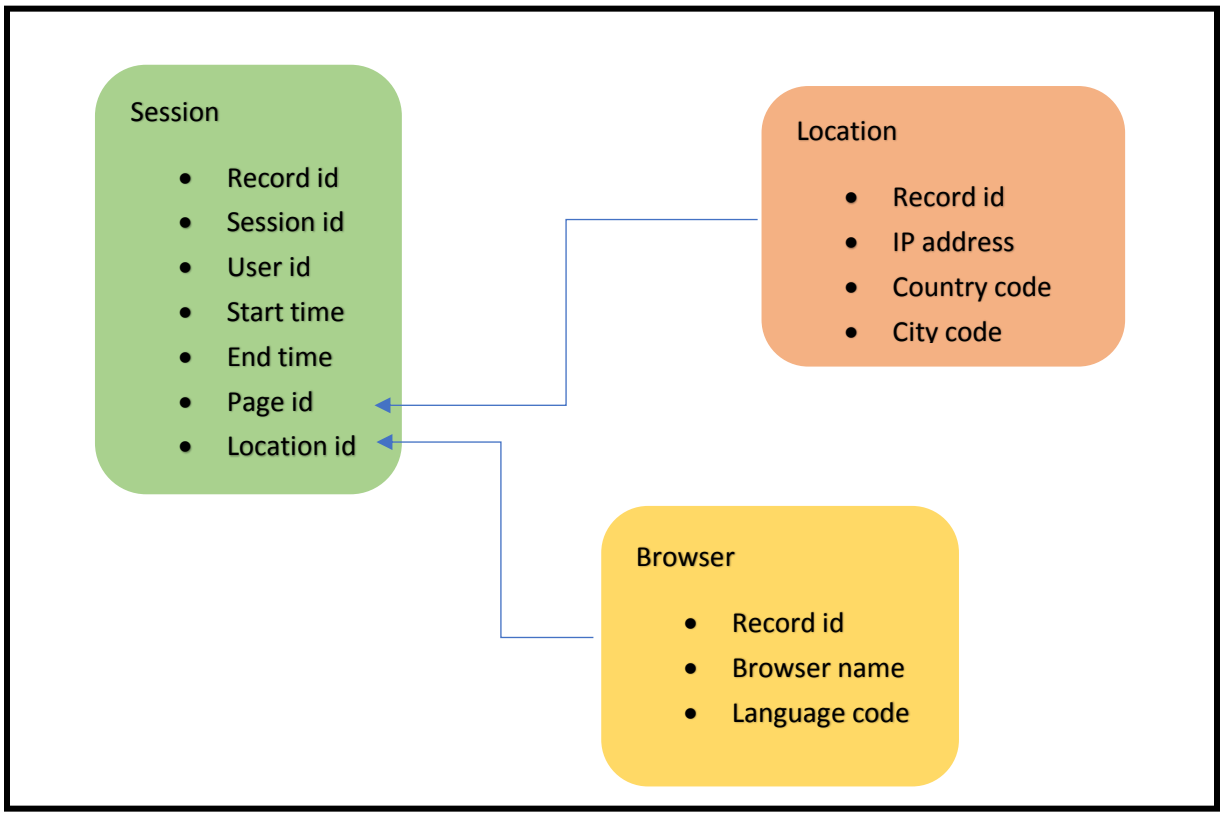
3. **Microservices:** Using microservice architecture allows for rapid, reliable and highly maintainable application development since it breaks downs the complex logic into small services running independently and communicating. Microservices API for different services like user detail capture, report data fetch can be developed independently and invoked as required. Following language can be used to develop microservices:
    - **Python:** Python has great code readability, strong support for integration with other technologies, high programming productivity.
    - **Java:** Java provides lots of programming resources and libraries, lots of developers, ability to move easily from one system to another. Spring boot framework which is a

popular application framework for Java platform can also be used to develop microservice APIs.

4. **Caching service:** Since we are using a distributed database architecture handling a large volume of data, it'll be very efficient if we can cache a large amount of data in RAM rather than moving them around in HDD which has a high efficiency and latency cost. Following are some options for available to implement this:
    - **Apache Ignite:** It is a distributed memory-centric database and caching platform. It is used to achieve true in-memory performance at scale and avoid data movement from a data source to applications.
    - **Ehcache:** It is an open-source, standards-based cache that boosts performance, offloads user database, and simplifies scalability. It's the most widely-used Java-based cache because it's robust, proven, full-featured, integrates with other popular libraries and framework.

5. **Database:** We need a highly scalable and distributed database able to manage a high volume of read/writes with high availability and scalability. Following are the options available:
    - **Bigtable:** Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. It has low latency and highly scalable.
    - **Apache Cassandra:** Apache Cassandra is a free and open-source, distributed, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

5.3 Data Model: As per the assumptions made in point 5.2 above, the logical structure of data can be represented as below:



The above data model is an abstract representation. With more details can be added or it can be modified with more information about the requirement.

6. Limitations/Drawbacks:

Even though we decide to design the system with the best possible systems out there, there are still going to be some drawbacks of implementing them. Let's look at the following points to mention some.

- Distributed architecture requires coordination and maintenance, so implementing it can be a little costly.
- Open source software/systems are constantly changing, incorporating every change depending on the requirement may also require a change in the implemented system at various level.
- We cannot interface any system with any other system. We still must consider the compatibility of each component used.
- The scalability of the system cannot be determined unless a concrete implementation is done. Even though we consider the optimum parameters, the actual parameters may differ.
- There are still more specifications required to design the system more in detail. The assumptions made a virtual conceptualization possible but in reality, this system may not be possible.

7. Summary:

Overall the designed system can handle a large volume of data and process client request with any much latency. A downtown may be required for maintenance in case system upgrade but not if any server becomes unavailable.