## **Google Analytics Design**

[**Requirement**](#Requirement)

* [Functional Requirement](#FunctionalRequirement)
* [Non- Functional Requirement](#NonFunctionalRequirement)

[**Assumptions**](#Assumptions)

[**System requirement**](#SystemRequirement)

* [Storage estimates](#StorageEstimates)
* [Bandwidth estimates](#BandwidthEstimates)

[**System API**](#SystemsAPI)

[**High level solution**](#HighLevelSolution)

* [Architecture](#Architecture)
* [Write flow](#WriteFlow)
* [Read flow](#ReadFlow)
* [Few optimizations](#Optimizations)

[**Database schema**](#DatabaseSchema)

[**Sharding**](#Sharding)

[**Scalability/Performance**](#ScalabilityPerformance)

* [Handle large write volume: Billions of write events per day](#Solutiion1).
* [Handle large read/query volume: Millions of merchants wish to gain insight into their business. Read/Query patterns are time-series related metrics](#Solutiion2).
* [Provide metrics to customers with at most one hour delay](#Solutiion3).
* [Run with minimum downtime](#Solutiion4).
* [Have the ability to reprocess historical data in case of bugs in the processing logic](#Solutiion5).

**# Requirement**

Google analytics tracks and reports website traffic that helps us find out how our website is performing.

1. Functional Requirement:

1. User has to register his website to be analyzed by google analytics.
2. Users can query metrics like total users, total sessions, total new users, sessions per user, total pageview, pages per session, average session duration etc. for their websites.
3. Historical data can be reprocessed in case of bugs in processing logic.

2. Non- Functional Requirement:

1. Service should be highly scalable to handle large data and frequent queries.
2. System should be fault tolerant and highly available.
3. Low latency for query/read API.

**# Assumptions:**

1. Input data is collected by web client or from external system and sent to web server in the form of REST API.
2. For a website, only last 2 years of aggregated data and 5 years of raw data is kept in the system.

**# System requirement:**

Let’s assume we get 1 billion interactions with all the registered websites each day and size of one record is 1 KB.

1. Storage estimates: **Total storage requirement = ~2.3 PB**

* If we are keeping 5 years of raw data. Storage requirement for raw data will be 1 billion KB \* 365 days \* 5 years = 1.6 PB
* Since we are keeping 2 years of aggregated data, storage requirement for it would be 1 billion KB \* 365 days \*2 years = 696 TB (~ 700 TB - Assuming aggregated data will be same as size of raw data in worst case).

Total storage requirement = 1.6 PB + (~700 TB) = ~2.3 PB

2. Bandwidth estimates: **11 MB/sec**

Since we expect 1 billion KB data per day, total incoming data will be 11 MB/sec.

**# System API**

Following REST APIs are supported by the system

1. collectData ( websiteID, userID, sessionID , device type, browser type, operating system, traffic source, location, interaction type, interaction parameters, timestamp)

Gets the data from the client, processes and stores in the database.

Parameters:

* **websiteID -** The unique ID of website registered with google analytics.
* **userID -** Unique ID of user.
* **sessionID -** Unique client session ID. User can have multiple sessions.
* **device type -** Type of device from which website is visited.
* **browser type -** Type of browser from which website is visited.
* **operating system -** Type of operating system from which website is visited.
* **traffic source -** The path from where user reached the website.
* **location -** Geographical location of user.
* **interaction type -** Type of activity user performed on website. For example, page hit, click on link.
* **interaction parameters -** parameters related to interaction.
* **timestamp -** The system time when user performed the interaction.

2. readData (websiteID, start date, end date, dimension, metrics, filter, sort, samplingLevel, max\_results, output)

Reads the analytic data stored by the system. Each query requires websiteID, start date, end date and atleast one metrics.

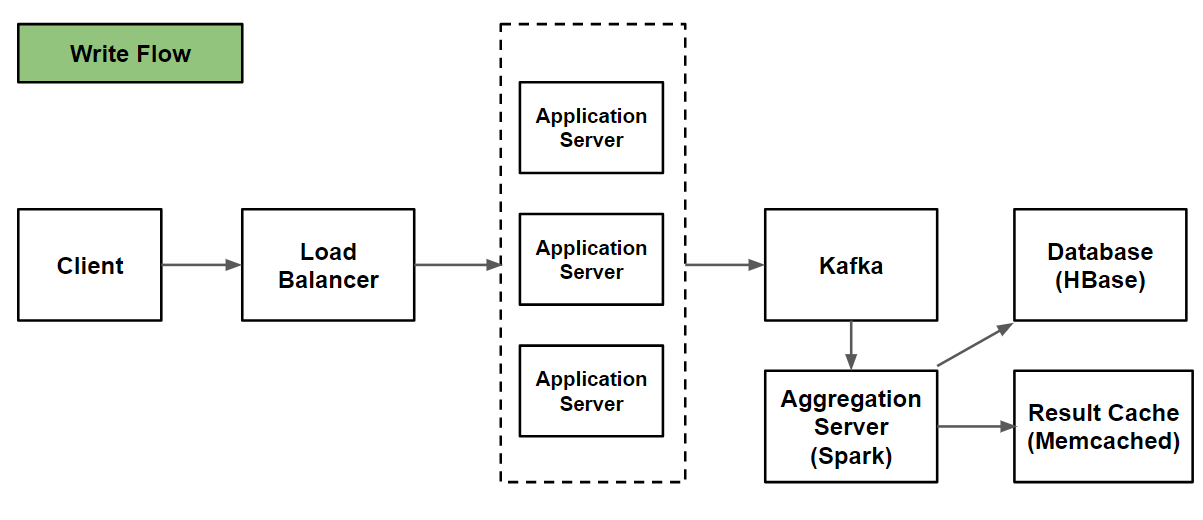
Parameters:

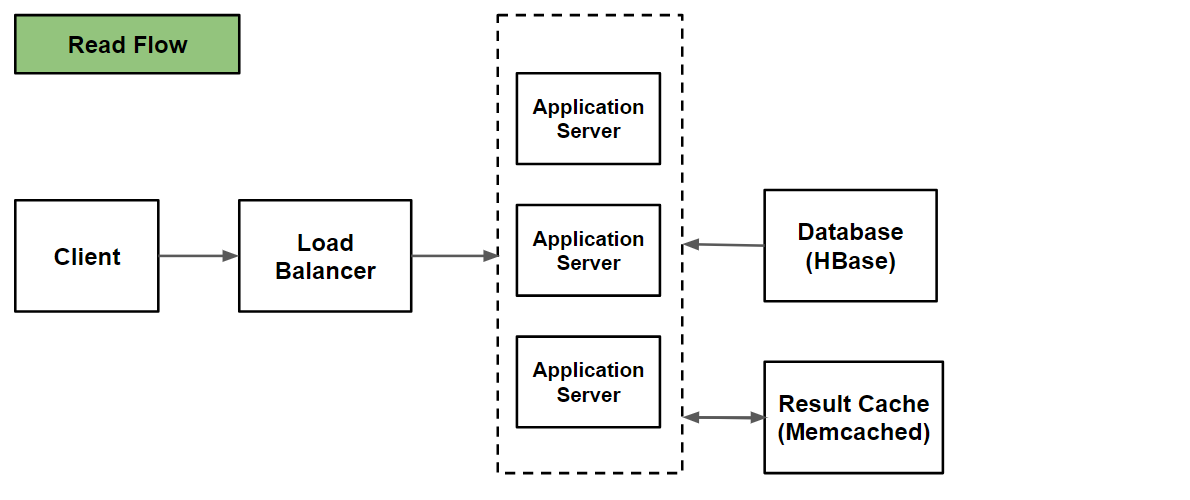
* **websiteID -** unique ID to identify website
* **start- date -** start date of data to be analyzed
* **end-date -** end date of data to be analyzed
* **dimension -** breaks down metrics by common criteria
* **metrics -** The aggregated statistics for user activity to your site, such as clicks or pageviews
* **sort -** A list of metrics and dimensions indicating the sorting order and sorting direction for the returned data
* **filter -** restricts the data returned from your request
* **samplingLevel -** Use this parameter to set the sampling level (i.e. the number of sessions used to calculate the result) for a reporting query
* **max-results -** Maximum number of rows to be included in this response
* **output -** output type of the Analytics data returned in the response

**# High level solution:**

* Data can be collected by either external system or by web client specific to a tracking ID that is added in each page of website. Request will be authorized based on website ID.
* Data collected has information about websiteID, userID, sessionID, user parameters like browser type, device type, source, location and interaction type and its parameters. Interaction with the website can be pageview, any click event or any transaction.
* Information is parsed and divided into dimensions and metrics
* Dimension – Characteristics of user, sessions. For example, user location, traffic source, page title.
* Metrics – quantitative measure of user action and session. For example, total users, pages per session.
* Information is processed and stored in database as raw data.
* Default metrics are also pre-aggregated and stored separately with dimensions so that they can be queried faster.
* Metrics that need to be calculated at runtime, are calculated from raw data. Dimension combination also calculated from raw data at runtime.

Architecture:





* **Client:** Component that sends data to the system and queries stored data for analysis.
* **Load Balancer:** Distributes incoming requests equally among backend servers either in simple round robin fashion or more intelligently by analyzing the traffic on each server. If a server is dead, LB will stop sending any traffic to it.
* **Application server:** provides services to store and query data in/from the system.
* **Kafka messaging queue:** Stores streams of incoming interactions in fault tolerant and durable way.
* **Aggregation server:** Aggregates interactions and precompute metrics based on the dimensions using spark.
* **Result cache:** Used to store result of a query so that later on if same request occurs, it can be served from cache. Distributed memory caching system like memcached can be used for high performance.
* **Database:** NoSQL database like HBase can be used to handle large volume of data and real-time querying.

Write flow:

1. Client sends data to application server through load balancer.
2. Application server parses the request and identifies dimensions and metrics and publishes records to kafka messaging queue.
3. Aggregator server consumes records from message queue and compute aggregations.
4. Raw data and aggregated data is then stored in database and result cache is updated if needed.

Read flow:

1. Client sends read request to application server through load balancer.
2. If same request is found in result cache, its result is fetched from cache and returned.
3. If not found in cache, data is read from database.
4. If queried metrics is not pre-aggregated in DB, it is calculated from raw data.
5. Result is also stored in result cache.

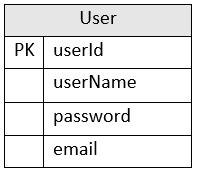
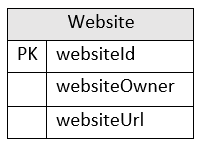
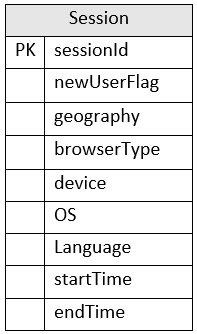
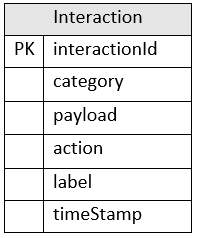
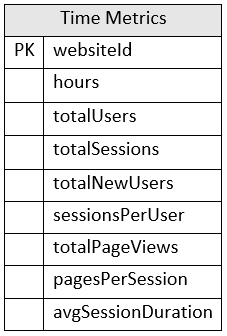
Few optimizations:

Following optimizations can be applied to the system depending on the type of request.

1. **Materialization -** If a metric is queried many times and it is not pre-aggregated, system can start pre-aggregating it in future.
2. **Sampling -** Some metrics can be calculated from sample data instead of whole data. At runtime, request can be served from smaller group of data.

**# Database schema:**

We need to store data about website, user, sessions, interactions and aggregated data. We create below schemas to store metadata for various entities.

We can store the above schema in a distributed key-value NoSQL store to achieve scalability.

* We will store **relationship between website and users to know users corresponding to a website** using columnar datastore like **HBase**. The key would be website ID and value will be list of UserIDs.
* We will have similar relationship schema for other tables. **UserSession schema would have key as userId and value as list of sessionId.**
* **SessionInteraction schema would have key as sessionId and value as list of interactionId.**

**# Sharding:**

* We can append websiteId with userId, sessionId and interactionId and partition the table based on userId, sessionId and interactionId respectively.
* Since primary key for these tables have websiteId appended, we can write partitioning scheme such as all the records corresponding to one website are stored together.
* We can subpartition interaction table based on timestamp.
* Time Metrics table will be partitioned on websiteId.

**# Scalability/Performance:**

**1. Handle large write volume: Billions of write events per day**

This is done by using kafka for real time streaming, spark for faster aggregations.

**2. Handle large read/query volume: Millions of merchants wish to gain insight into their business. Read/Query patterns are time-series related metrics.**

Cluster is used for handling large read requests. Load balancer distributes the traffic between application servers based on load.

Default metrics are pre-aggregated, other metrics are also calculated on the fly with NoSQL key/value pair database like HBase that makes real time querying faster. Results are cached using distributed memory cache. Query pattern can be analyzed to pre-aggregate any high queried metrics which is not yet pre-aggregated.

**3. Provide metrics to customers with at most one hour delay.**

Default metrics are pre-aggregated, other metrics are also calculated on the fly with NoSQL key/value pair database like HBase that makes real time querying faster. Results are cached using distributed memory cache. Query pattern can be analyzed to pre-aggregate any high queried metrics which is not yet pre-aggregated.

**4. Run with minimum downtime.**

Since we are using cluster of application servers, service will be highly available.

If a few servers die down, the system will be still available and serving. Data processed by server will not be lost because of highly reliable kafka message queueing system.

**5. Have the ability to reprocess historical data in case of bugs in the processing logic.**

Along with aggregated data, we keep raw data also in the database. Any metric can be reprocessed using raw data.