System Design – Web Analytics System

# Requirements –

The system needs to -

* + 1. Handle large volume data – Billions write events per day.
    2. Handle large read/query volume – Millions merchants want to get insights about their business. Read/query patterns are time-series related metrics.
    3. Provide metrics to customers with at most one hour delay.
    4. Run with minimum downtime.
    5. Have the ability to reprocess historical data in case of bugs in the processing logic.

# Estimations –

In order to collect webpage tracking information and user behavior, tracking code needs to be integrated in website, which in turn, will send all this information to analytics engine in backend. There are limitless possibilities to capture data-points which can be used for analytics purposes. Data metrics collected can be classified in any of below form –

* Page information – URL, Title, Page load time, Page flow, etc.
* User information – Location, Language, New/repeat user, Frequency, Recency, Age, Gender, Interests etc.
* Technology – Mobile device, Browser, OS, Screen resolution, etc.

Assuming 2KB of tracking information is collected on each hit, we can estimate the following –

## Traffic Estimation –

Number of write events – 1B/Day

Number of reads – Depends upon the query which uses time filter to extract related metrics from DB. We can assume 1M merchants are going to request for data in a day.

## Storage Estimation –

Each hit writes 2KB of data.

Considering 1B writes/day,

Amount of writes per day = 1B \* 2KB ~ 2 PB/Day

Assuming data is kept max for 2 years, Total storage requirement –

2PB \* 365(Day) \* 2(years) = 1460 PB

## Bandwidth Estimation –

Total writes per day = 1B

Total writes per second = 1B/24(hours)/60(minutes)/60(seconds) = 11575 writes/sec

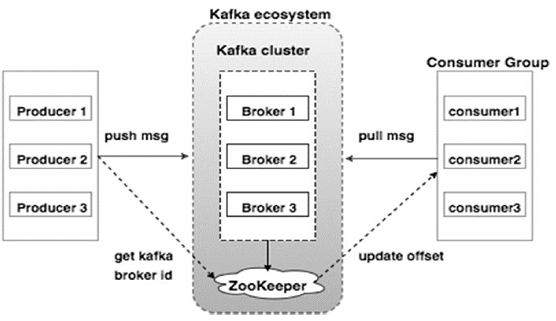
One request size is 2KB

Total amount of write traffic per second = 11575 \* 2KB = 22 MB/second

# Streaming Mechanism –

In order to address 22MB/Second write traffic in system, we need to introduce messaging system capable of moving high volume of data to DB. Apache kafka is a distributed messaging queue which can be put in place along with zookeeper to maintain availability in the system.

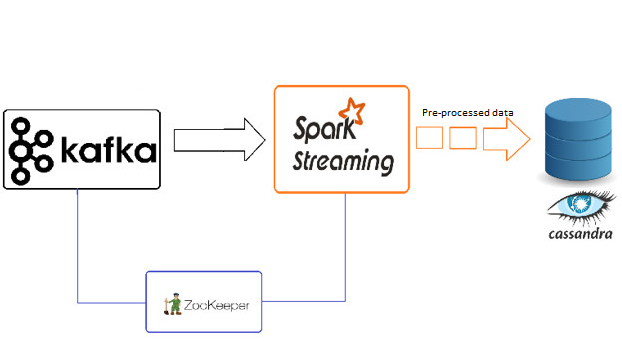
With kafka we can achieve 2M writes per second, which is good enough for our case.



# Data Aggregation –

In order to achieve high read performance, we should preprocess incoming data beforehand as per output reports requirements and store it in DB. This saves a lot of processing time and we can retrieve processed data directly from DB.

To achieve this we can use Spark RDD processing and define pipelines. After data is dumped into kafka queue, a copy of same data can be brought to spark for preprocessing and then processed data can be saved in DB. This will surely increase our DB size, but we will gain performance.



# DB Architecture –

Some of properties of data we are handling –

1. Writes exceeds reads by large margin.
2. Data should be available all the time.
3. Data should be consistent across all nodes.

Apache Cassandra can be used to support above requirements.

**Data Sharding** –

Considering number of users and number of data-points corresponding to each user, it makes more sense to store data over shards for faster access. Apache Cassandra supports sharding and can give better performance and scalability in real-time.

We can use *websiteID* for sharding purpose. A hash function can be applied on *websiteID* to retrieve node on which corresponding data is stored/to be stored.

# Microservice Architecture –

We will be addressing three different services in this system –

1. Insert Data to DB
2. Generate report for user.

Even though processing logic is very much straight forward, number of users, amount of traffic is too huge to handle in a monolithic architecture. For better availability and performance, we should be able to scale the systems in real-time. Also the services are pretty much independent of each other and can be developed/deployed in best manner, if microservice architecture is followed.

We can set up Kubernetes cluster to deploy microservices as containers with multiple instances. In order to balance load and distinguish services, we can use ngnix API gatway for better availability across nodes.

Microservices can be developed in SpringBoot and deployed in Kubernetes container.

# Caching –

As per 80:20 rule, we know 20% of daily reads are generating 80% of trafic. In order to serve this hot data, we can have a distributed cache server in place.

Redis is one of the widely used and most fitting technology for our case. We have 1M merchants requesting for data. If we assume that each merchant send 100 requests and average size of data retrieved be 2MB.

Then total data accessed in a day will be 1M \* 100 \* 2MB = 200 TB. We can setup a distributed cache server with multiple nodes to gain read performance.

# Load Balancing –

As discussed in microservice architecture, an API gateway itself can take care of load balancing between client and microservices. Cassandra has its own mechanism to distribute load across its nodes.

# High Level Design –

