Google Analytics like Backend System

Goal –
To build a reliable, scalable, high throughput, high availability, fault-tolerant, low end to end latency, flexible Google Analytics-like backend system.

Requirements –

1. To collect stats about the user's website activity and persist the data into a time series database that is capable of handling billions of write operations per day efficiently.
2. To present the customers with the metrics for efficient visualization and monitoring of their user's interaction data in near real-time.
3. To build the highly available system capable of handling millions of read/write/query requests.
4. To build a high throughput and scalable system which can store and convert the streaming data into some meaningful insight that can help our customers in their business decisions.
5. To store the historic data in case we need it in the future for some processing.

Options Explored –

Since billions of write events will be logged per day, therefore we need to have a highly durable, linearly scalable multi-tenant backend architecture. Hence, micro-services deployed on the docker container managed by the Kubernetes cluster for handling an incoming stream of events would be the best choice since they can horizontally scale and process the incoming stream of data in a well-organized manner. Moreover, we can use the paradigm of service mesh and can implement Istiod to manage and monitor the health, secure communication, etc. between microservices in a cluster.
In this manner, we can achieve the goal of having a highly available, resilient, and fault-tolerant distributed micro-services backend to handle billions of requests per day.

1. To handle high number of requests, we must have a load balancer that can distribute the load across the cluster of microservices with a specific endpoint. Following open-source options have been explored while choosing the right one pertaining to our use case.

    a. **HAProxy** – An open-source load-balancer with high availability providing both the Layer 4 and Layer 7 load balancing feature. It also has in-built statistics reporting.
    b. **Nginx** – Open-source version of Nginx provides the basic level of request routing along with the webserver features. However, the Nginx Plus edition provides a highly performant solution that can handle **millions** of requests **per second**.

The below benchmark gives us useful insight into the above load-balancers. Therefore, we can use multiple instances of HAProxy to manage the high volume of traffic per day.

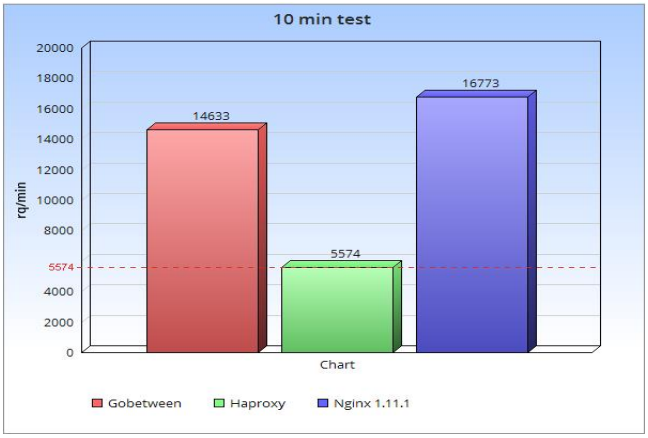

Figure 1 - Benchmarking of HAProxy with Nginx

2. Since exposing the microservices REST endpoints directly is not a good approach. Therefore, we need to have an API Gateway where we can keep our REST endpoints. In addition to the routing, authentication and authorization will also be handled in the Gateway itself. Kong, Netflix Zuul and Apigee gateways are explored as part of it. Zuul seems to be promising for us as it's developed in Java and has Spring Framework strong support.
   Since we are using Spring cloud for creating our micro-services and there is excellent support from Spring cloud for the Netflix stack, therefore Netflix Zuul along with the Netflix Eureka for service registry will be used to handle the requests.

3. We need a distributed, durable, elastically scalable mechanism for processing a stream of fast - data and which can easily fit into our entire microservices ecosystem. Apache Kafka seems to be the best open-source solution for stream processing needs. Being highly available, scalable, distributed, low-latency and fault-tolerant, we can rely on Kafka for building real-time streaming data pipelines.
   a. We can use topic partitioning and configure the replication factors to reduce the risk of losing ingested data in case any broker (server) goes down.
   b. In addition to this, we can leverage the offsets persisted in the Zookeeper in case there is a need to reprocess the same data.
   c. Also, at Uber, more than one trillion real-time messages are passed through Kafka. Hence, Kafka would be the best open source event streaming platform that we can implement.

4. Since we want to present the customers with the near real-time dashboard analytics, therefore, we need a processing engine that can process the unbounded stream of data coming from Kafka and perform some in-memory operations in a distributed manner. In addition to this, since we are interested in the time at which the event occurred rather than when it is being observed, therefore we need an engine that can efficiently handle the event time.

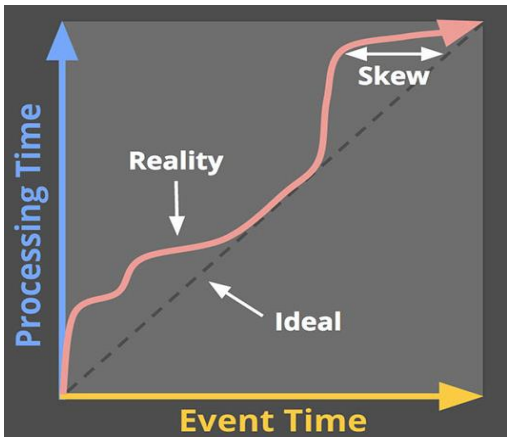

Figure 2- The X-axis represents event time completeness in the system, i.e. the time X in event time up to which all data with event times less than X have been observed. The Y-axis represents the progress of processing time, i.e. normal clock time as observed

Since we are consuming data from multiple partitioned topics, hence we need to have a framework that supports the state management to persist the state locally on each node and is highly performant. Post aggregating the streamed data, we need to save the data in the database as well. Following open-source frameworks were explored as part of the processing engine-

a. **Apache Spark** – It is based on the micro-batching framework wherein the records ingested are batched together using the concept of RDD's and then are sent up to the worker nodes for producing the final output after doing some transformations.

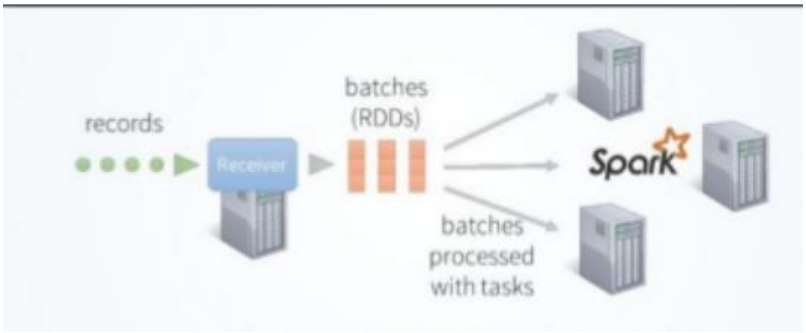

Figure 3 - Records processed in batches with short tasks

The Micro-batching framework doesn't seem to align with the low-latency requirements of the streaming analytics use case we have at hand. Therefore, we can move to the native streaming frameworks defined below.

b. **Apache Storm** – This is the oldest open-source streaming engine capable of handling only **simple event-based** use cases. In addition to this, this framework does not provide us the state management feature which is crucial in our use case.

c. **Apache Flink** – The underlying architecture supported by Flink is similar to that of Apache Spark. The difference is that the former is a true streaming engine treating batch as a special case of streaming with the bounded data. Also, it comes with more advanced features like event processing, watermarks, etc.

Figure 4 - Apache Flink basic architecture using streams and transformations

It is quite evident from the below benchmarking that Flink is approximately two times faster than the former. In addition to that, configuring Spark cluster is relatively harder as compared to its counterpart.

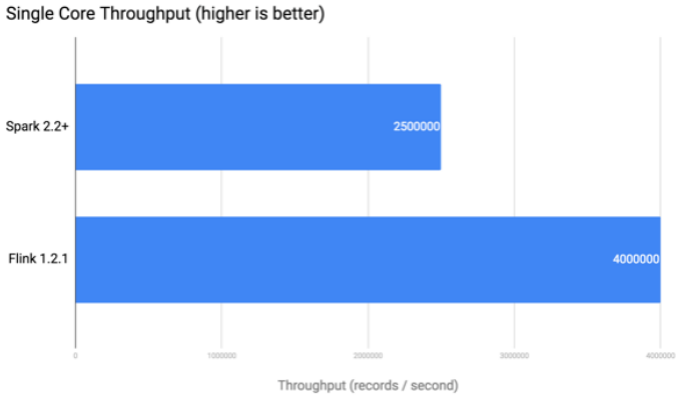

Figure 5 - Single core benchmarking of Flink and Spark

5. We need a platform to manage the data efficiently between elements of computation. Below
Redis – It's primarily used as an in-memory key-value store used for data caching. It is efficient in use-cases which require faster read performance. However, since we need efficient write operations, therefore Apache Ignite would be the best solution to go within our case.
Apache Ignite – With Apache Ignite, we can easily pass computational results between loosely coupled computation modules instead of writing the intermittent results to disk or any persistent storage. Apache Ignite gives us the ability to process large incoming finite as well as never-ending volumes of data in a scalable and fault-tolerant way in the cluster. The rate at which data can be injected into Ignite is very high and easily exceeds millions of events per second on a moderately sized cluster. We can process stream data/ SQL queries in a parallel, durable, and consistent manner on all the nodes concurrently. Apart from this, to make the cluster fault-tolerant, we need to increase the data redundancy level. The number of redundant copies can be configured to make the data available on other cluster nodes as well in case one of the nodes becomes unavailable. Some of the proven use-cases for apache ignite can be referred to here.
We can use Apache Ignite's data streaming APIs that can inject a continuous stream of data in the cluster in a scalable and fault-tolerant way. Apache Ignite automatically takes care of partitioning and distributing the data between the nodes in a cluster. It enables the data to be processed concurrently in a co-located fashion as described in the below figure -

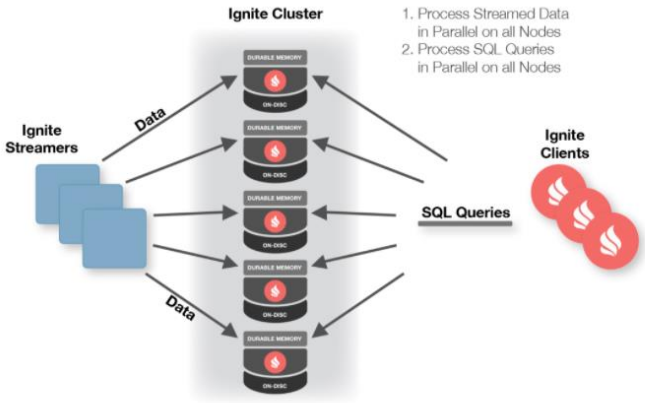

Figure 6 -Apache Ignite cluster processing a large number of unbounded data streams

6. To avoid losing the data, we need to persist the data in the database. Because of the high influx of event-based disparate data, a significant amount of customization and configuration will be required to store time-series data into relational databases. Therefore, NoSQL would be the best suited in our case as it can store the data in JSON format, and operations such as aggregation would be way faster as compared to SQL database. Since we must show that is associated with certain timestamps, therefore we need a Time-Series Database to store large volumes of timestamped data in such a way that insertion and retrieval of data are much faster.
   TSDB's solutions differ in features such as expandability, performance Below options were considered while choosing the database specific to our use case -
   a. **Influx DB** – Influx DB is a very popular time-series database capable of tracking billions of metrics per day. It has its ecosystem encompassing of four below components
      I. Telegraf – It is the data collector
      II. InfluxDB – Main component of the whole tick stack.
      III. Chronograf – This is used for the visualization and interaction with other components.
      IV. Kapacitor – provides the tool for real-time data processing, monitoring, or alerting.

      InfluxDB is more advanced in this regard and can work with even nanosecond timestamps.
      Since we are dealing with billions of write operations per day, we need a clustered environment to avoid a single point of failure. However, open-source versions do not support clustering, we need to go for the enterprise edition to get that feature.

   b. **Prometheus** – This can store the time-series data; however, it is primarily used for monitoring purposes. Prometheus can write data with millisecond resolution timestamps. For instance, Uber, Slack uses this database for monitoring the metrics.

   c. **OpenTSDB** – "OpenTSDB is a distributed, scalable Time Series Database (TSDB) written on top of HBase which is a management system used for handling big table storage elegantly and efficiently. OpenTSDB was written to address a common need: store, index, and serve metrics collected from computer systems (network gear, operating systems, applications) at a large scale, and make this data easily accessible and graphable." This means that it is easy to scale OpenTSDB horizontally. In addition to

this, we can store hundreds of billions of data rows over distributed instances of TSD servers.

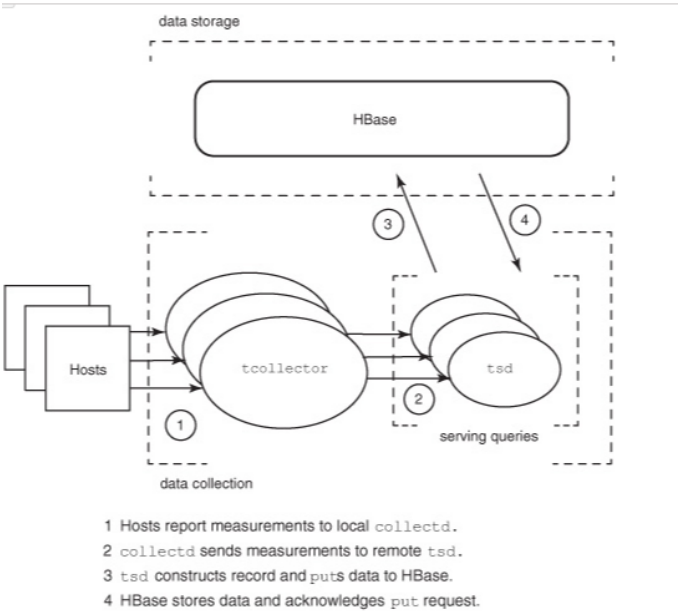

Figure 7 - OpenTSDB architecture: separation of concerns. The three areas of concern are data collection, data storage, and serving queries

This data can then be presented on the user's dashboard using the REST endpoints provided in the OpenTSDB. Spring-boot microservices hosted in the docker container can fetch the data in various formats determined by the serializer selected and can present the metrics to the user's dashboard. The APIs can be referred to here.

Final Design –

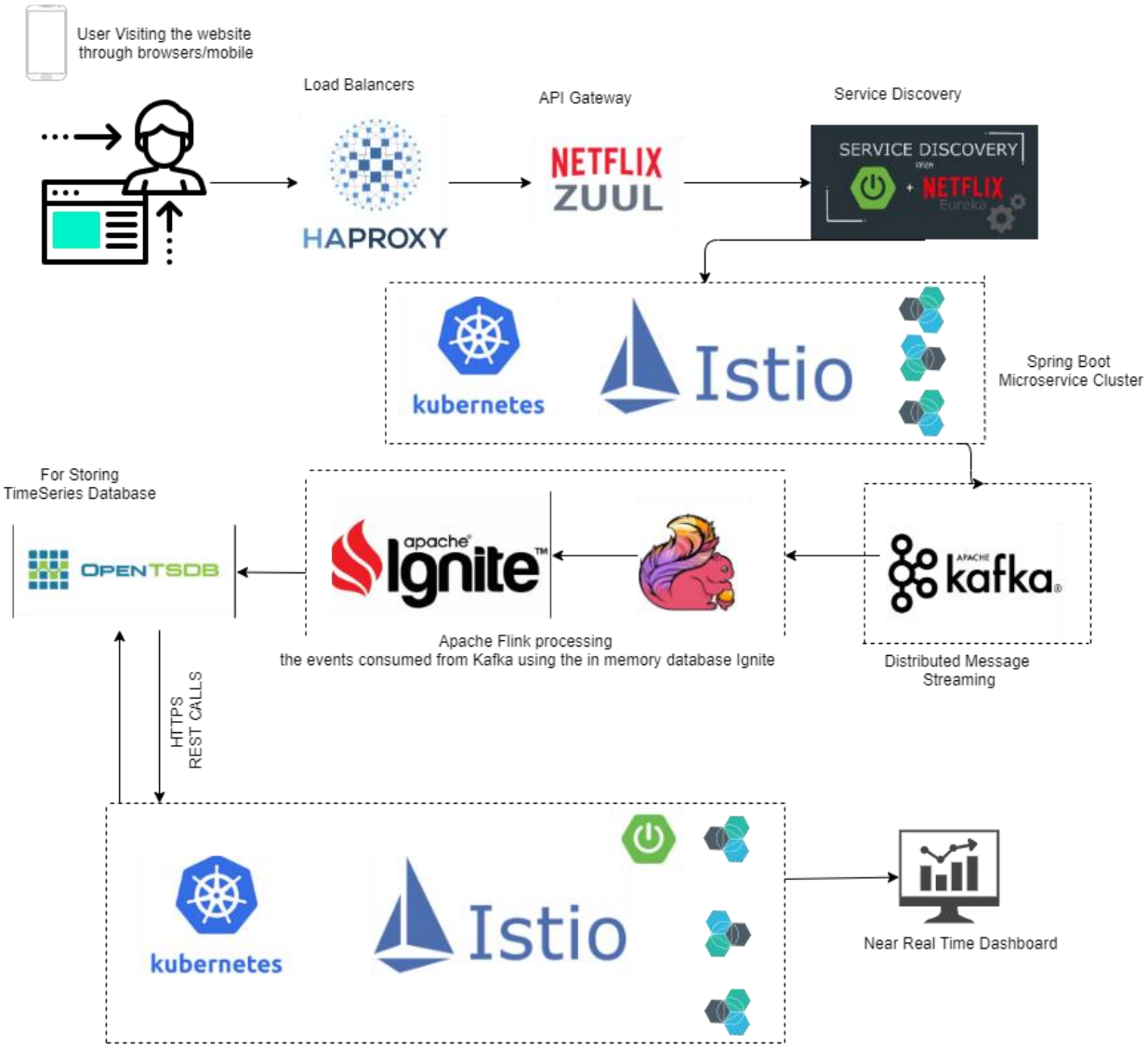

Figure 8 - High level system design

References –

1. https://dl.acm.org/doi/pdf/10.1145/2723372.2742788
2. https://www.gridgain.com/technology/apache-ignite/ignite-vs-redis#:~:text=Apache%C2%AE%20Ignite%E2%84%A2%20is,often%20promoted%20as%20a%20database.&text=Apache%20Ignite%20and%20Redis%20both,and%20cache%20data%20in%20memory.
3. https://geekflare.com/open-source-load-balancer/
4. https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b
5. https://eng.uber.com/athenax/
6. https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/
7. https://www.oreilly.com/radar/the-world-beyond-batch-streaming-102/

8. https://www.grin.com/document/299975
9. https://engineering.opsgenie.com/comparing-api-gateway-performances-nginx-vs-zuul-vs-spring-cloud-gateway-vs-linkerd-b2cc59c65369