PAYTM LABS - WEBSITE ANALYTICS DESIGN

by Steve Kotsopoulos, version 1.0, May 10, 2019

This is a high-level design for a Google Analytics like Backend System for PayTM customers, using open source components.

Requirements

Our solution must support the following requirements:

1. Handle large write volume: Billions of write events per day.
2. Handle large read/query volume: Millions of merchants wish to gain insight into 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.

Features we need to support

Our backend will track and report website traffic statistics, such as:

- Session duration
- Pages per session
- Bounce rate
- Referrers
- Geographical information
- Source (client) information
- Visitor segmentation

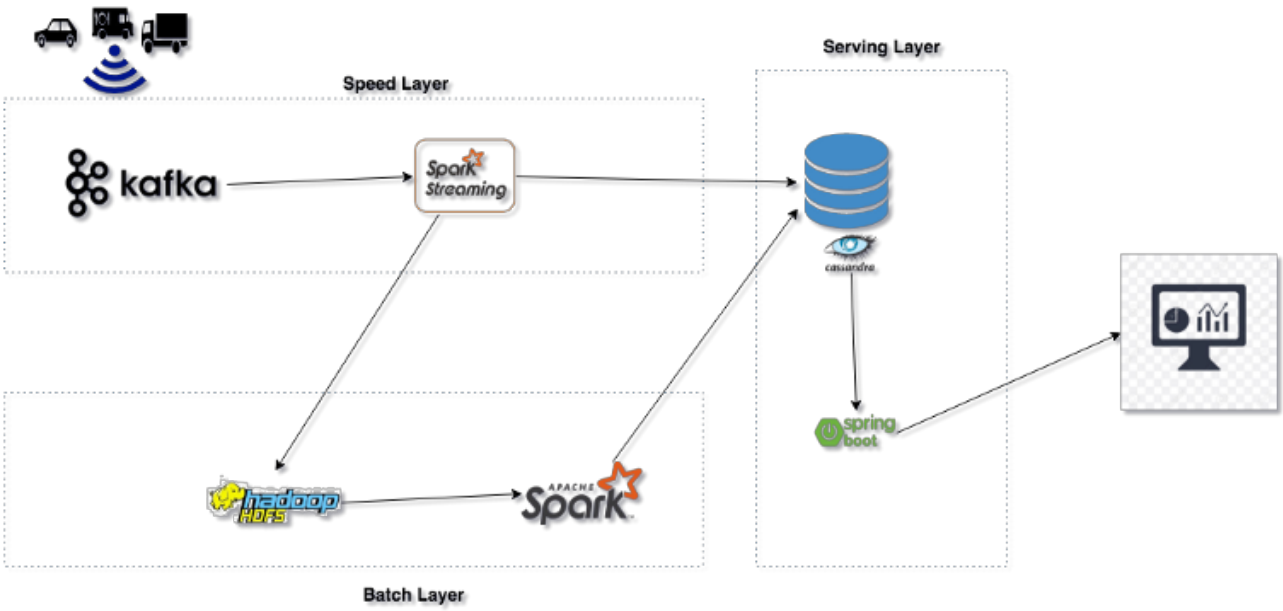
All components of the architecture will need to run with minimal downtime and high throughput, so must support clustering for fault tolerance and horizontal scalability. Exact sizing of each cluster is not specified here, with the expectation that developers will work with the devops team to predict what is needed to support production loads.

Architecture

Our solution follows the Lambda architecture. Lambda is a well-established architecture pattern for handling massive quantities of data.

- Our speed layer is based on Netty, Akka, Apache Kafka and Spark Streaming.
- Our batch layer is based on Apache Hadoop, HBase, Spark and OpenTSB.
- Our serving layer uses MySQL, OpenTSB, Spring and Spring Boot.
- Configuration and tuning parameters for all layers would be stored in MySQL.
- All services would be deployed to the same version of Linux, to ensure ease of monitoring, maintenance and a unified approach to security. All servers and services would push metrics via Collectd/InfluxDB/Grafana to assist in OS and application tuning & monitoring. Production servers would be monitored using Nagios and remote logins would be over ssh via a secure jumpbox.

The below diagram closely illustrates our design – it is from the “Lambda architecture – how to build a big data pipeline” article listed in the References at the end of this document.



Speed Layer

Each customer website would be assigned a universally unique identifier UUID.

Sites would embed some javascript on every page, which will pass information about the http request along with the UUID to our data ingestion REST API which forms the front-end of our speed layer. The data ingestion REST API would be implemented Spring Boot. If we find that isn't performant enough, we can move to asynchronous frameworks such as Akka & Netty. The IP address of the client would be mapped to a Geographic location via the MaxMind GeoLite2 City database. The REST API would persist the raw data to a Kafka cluster. We choose Kafka because it supports high throughput, can store many days of raw data and we can rewind the offsets to re-process if we ever introduce a bug in our business logic.

The data ingestion REST API would store a persistent cookie in the client browser, so we can track user access during and across sessions. If the user has cookies turned off or clears them, new requests will appear to be unknown users (very little can be done to mitigate this).

The ingestion REST API would be deployed on multiple servers behind multiple NGINX http proxies. NGINX is highly performant, and would distribute load across the cluster. Our DNS would be configured so there are multiple IP address for the ingestion REST API hostname, each address mapping to a different NGINX proxy server. This provides no single point of failure for the incoming data, as losing data here would be unrecoverable.

We considered both MySQL and Postgres for storing account/client and configuration data. They are both open source and support clustering. We would start with MySQL as it is used elsewhere at PayTM. All applications using MySQL would use a common codebase leveraging Spring Entities, JPA Repositories and Services – so it would be easy to switch to a different database if we ever needed to.

The database schema would be managed with Flyway, ensuring consistency across environments and making it easy for developers to have their own local development database.

Batch Layer

A Spark streaming job would run every 30 minutes to consume raw data from Kafka and persist it to HBase.

For time series data, my initial choice would have been InfluxDB because I am familiar with it and have seen it used to track billions of metrics per day. The problem is that the open source version does not support clustering and we would like to avoid the cost of the enterprise license required for clustering. Instead we will use OpenTSB on top of HBase to support time series operations.

The business logic would calculate or obtain the metrics such as session duration, bounce rate, etc and store them to HBase for later consumption by the Serving Layer.

Serving Layer

A Spring boot backend would provide REST APIs to access the above databases, format reports and support a dashboard front-end. Spring boot was chosen because it makes it very easy to build production quality REST services with a minimal amount of code. This API would use the following technologies: Spring (Core, Data JPA, Web, Security, Boot Actuator), logback, thymeleaf, springfox swagger, flyway, dropwizard metrics, hibernate, Lombok, apache commons-lang3 and jackson.

If we find we need a separate reporting database, we could use Apache Druid, which is specifically designed for ad-hoc analytics.

This REST API layer would use the same NGINX and DNS solution described earlier for fault tolerance and scalability.

Summary

This high-level design outlines an approach to implementing a website analytics solution for our customers. It employs low-cost, open source, best of breed components that are already used elsewhere at PayTM. This design is a living document – comments and suggestions are welcome. Please send

your feedback to the author or attend the up-coming design review session so we can improve on this design.

References

- Lambda Architecture, how to build a big data pipeline - https://towardsdatascience.com/lambda-architecture-how-to-build-a-big-data-pipeline-part-1-8b56075e83fe
- Lambda Architecture - https://en.wikipedia.org/wiki/Lambda_architecture
- DNS Failover - https://ns1.com/resources/dns-failover-basic-concepts-and-limitations
- Akka - https://akka.io/
- Netty - https://en.wikipedia.org/wiki/Netty_(software)
- Kafka – https://kafka.apache.org/
- Spark - https://spark.apache.org/
- Hadoop - https://hadoop.apache.org/
- HBase - https://hbase.apache.org/
- OpenTSB - http://opentsdb.net/index.html
- Best Time Series Databases - https://medium.com/schkn/4-best-time-series-databases-to-watch-in-2019-ef1e89a72377
- MySQL - https://www.mysql.com/
- PostgreSQL vs. MySQL - http://www.postgresqltutorial.com/postgresql-vs-mysql/
- Why Uber moved to MySQL - https://eng.uber.com/mysql-migration/
- Spring Boot - https://spring.io/projects/spring-boot
- Flyway - https://flywaydb.org/
- Collectd - https://collectd.org/
- InfluxDB - https://www.influxdata.com/
- Grafana - https://grafana.com/
- NGINX - https://www.nginx.com/
- Nagios - https://www.nagios.org/