Ad Metric Design Document

Author: Justin Yoon

Last Updated: 12/6/2021

Goal

The goal here is to create a data pipeline that provides ad metric data for a dashboard application that allows users to see how their ads are performing.

Due to the fact that the data is sourced via an external ingestion API, there will be no hard requirements on data freshness as of now.

- Max response time for a query: 10 seconds
- Availability: 99.999%

Tradeoffs

The pipeline will favor query speed over data freshness. This is for a few reasons.

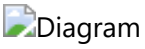
The first is that our data is sourced from an external API which we do not control. Without any SLA's between us and this API we cannot make any guarantees on data freshness as it is.

Second, the dashboard application that utilizes this data is only concerned with aggregated data at a hourly granularity at a minimum. This means there's less value on having fully up-to-date data and users should already expect that data for the current hour to not be complete anyways.

Assumptions Made

- There are 100M impressions and 1M click events per day.
- All ads, impressions, sessions, and users have their own unique ID's.
- Ad metric data is provided via an external ingestion API which consumes the ad data directly from the users' machines. They are then batched up as TSV files and stored in an S3 bucket which we have shared access to.
- The dashboard application will not require data aggregated at a granularity smaller than hourly in the future.

Component Diagram



S3

This is the S3 bucket that the ingestion API will write to.

- The bucket should only allow the ingestion API to write new files to the bucket.
- The bucket should give read-only access to the IAM role running on the EMR cluster.
- Encryption at rest should be configured since the data contains Personal Data.

- Intelligent Tiering should be configured so that older data files are moved into cold storage.
- Configure server access logging.
- Configure S3 bucket notifications so that "s3:ObjectCreated:*" events are sent to our SQS queue.

SQS

"s3:ObjectCreated:*" event notifications are sent to an SQS queue so that they can be processed by our Spark Job. A queue is utilized because it allows us to persist the event until we are done processing it (therefore guaranteeing we don't miss data for processing) and we can use long-polling to process multiple files at the same time.

- Queue should be a FIFO queue so that there are no duplicate messages.
- Content-based deduplication should be enabled.
- Encryption at rest should be configured.
- Configure a dead-letter queue for the queue to handle cases where the message processing repeatedly fails

EMR

The EMR cluster runs an Apache Spark job which polls the SQS queue for messages, reads the TSV files from S3, transforms the data, and then loads it into the Aurora Postgresql instance. EMR is used to make cluster management easier as well as for its built-in integrations with S3 (via EMRFS) and auto scaling capabilities.

- Auto scaling should be configured for the cluster.
- Apache Spark needs to be installed on the cluster.

Aurora Postgresql

The Aurora cluster is the final landing place of the ad metric data. The dashboard application will use SQL queries to query the Aurora database for its front end. Aurora was chosen for its performance, storage scalability, and ease of configuring replication and high availability. Aurora also allows you to provision an Aurora global database which allows for the creation of up to five read-only replica clusters in other AWS regions. This would increase the fault tolerance, as well as lower query latency in other regions, of the Aurora cluster.

Postgresql Schema

All tables are partitioned on ad_id.

ad_metrics

- ad_id int
- hour timestamp
- total_impressions int
- total_clicks int
- total_sessions int

users(unique constraint on all three columns)

- ad_id int

- hour timestamp
- user_id varchar

Spark Job

The Spark job would ideally be written in Scala since that is the language Spark is written in and the language with the most built-in support for the Spark engine.

Pseudocode (not representative of Scala syntax at all)

```
void main {
    while(true) {
      messages = sqsClient.getMessages(waitTimeSeconds = 20, maxNumberOfMessages =
10);
      if (!messages.isEmpty()) {
          dataset = spark.read.csv(messages);
          dataset = dataset.distinct(); // drop duplicates
          dataset = truncateTimestamp(dataset);
          adMetrics = dataset.groupBy("ad_id", "hour").agg(
            countDistinct(impression_id).as("total_impressions"),
            sum(when("event" === "click", 1).otherwise(0)).as("total_clicks"),
            countDistinct("ip").as("total_sessions)
          );
          users = dataset.select("ad_id", "hour", "user_id").distinct();
          execute(adMetrics, INSERT_INTO_AD_METRICS);
          execute(users, INSERT_INTO_USERS);
      }
    }
}

// Removes the minute and second portion of the timestamp field so that it is at
hourly granularity.
// Returned dataset has the same schema but timestamp is renamed to hour
DataSet truncateTimestamp(dataset);

// Executes the sql code using the dataset as parameters
void execute(dataset, sql);
```