Kafka at Pinterest

- we use kafka to transport data to our data warehouse, including
- critical events like
- impressions
- clicks
- we also use kafka to transport visibility metrics for our internal services (for monitoring and alerting)
- on the real-time streaming side, kafka is used to power many streaming applications, such as
- fresh content indexing
- recommendation,
- spam detection and filtering,
- real-time advertiser budget computation and so on.
- with >2000 brokers running on AWS

Pinterest Kafka setup:

- currently, we have kafka in 3 regions of AWS:
- most are in us-east-1
- have a smaller footprint in us-east-2 & us-west-1.
- use mirrormaker to transport data among these 3 regions.
- we spread brokers among multiple clusters for topic level isolation.
- with that, one cluster failure only affects a limited number of topics.
- max limit of each cluster to 200 brokers.

// A **Spread Placement Group** is a feature offered by cloud providers (like AWS)

- to ensure that individual instances (virtual machines) are placed on distinct physical hardware within a data center or across Availability Zones (AZs).
- This helps reduce the risk of simultaneous failures affecting multiple instances.
-

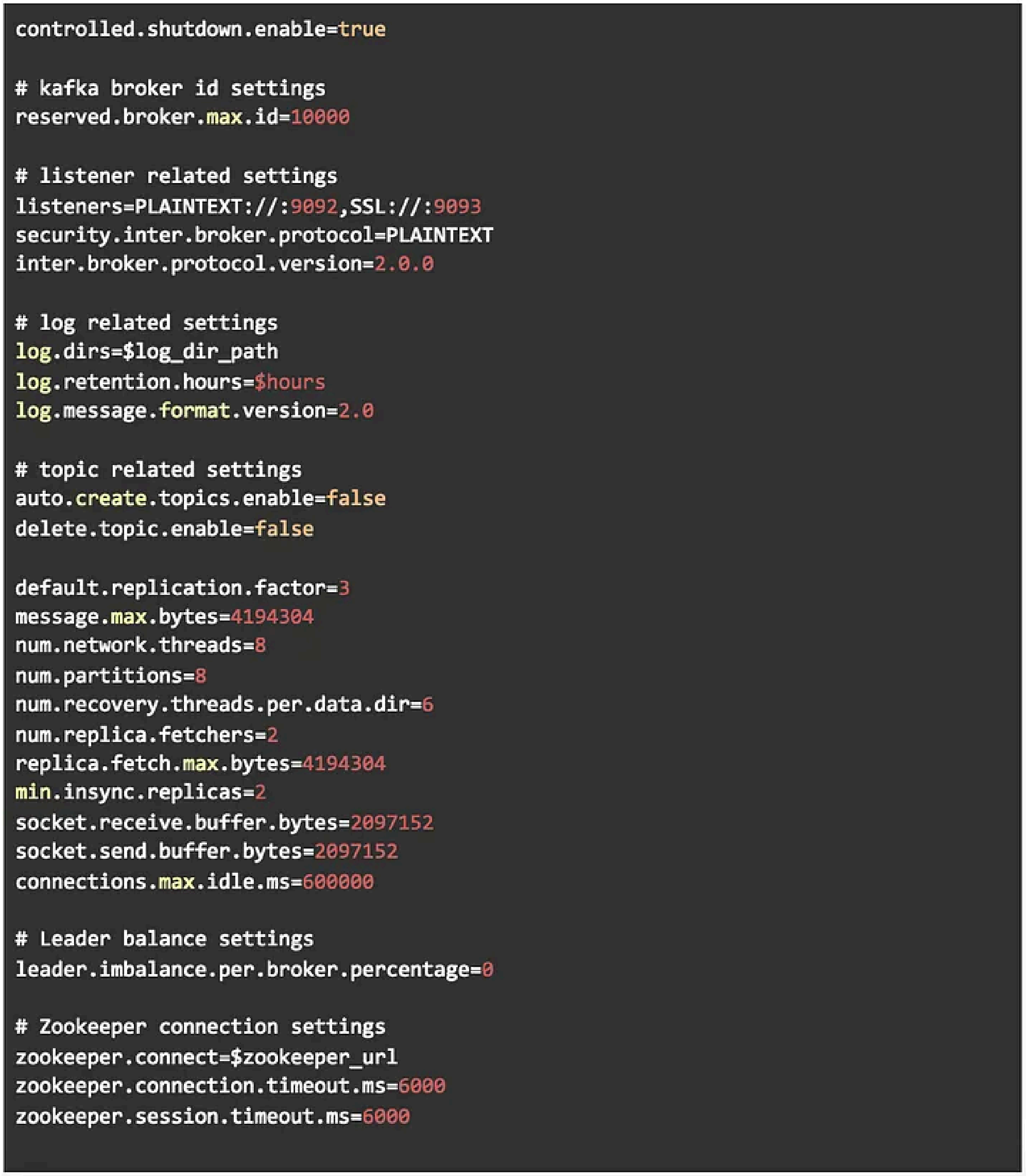
Feature	Spread Placement Group	Cluster Placement Group	Partition Placement Group
Instance Distribution	Spread across distinct hardware	Close together on the same hardware	Instances grouped into logical partitions

Feature	Spread Placement Group	Cluster Placement Group	Partition Placement Group
Fault Tolerance	High (isolated hardware per instance)	Low (all instances close together)	Medium (fault isolation within partitions)
Use Case	Small, critical workloads	Low-latency, high-throughput applications	Large-scale distributed systems
AZ Support	Can span multiple AZs	Must be in a single AZ	Can span multiple AZs

- AWS Spread Placement Groups limit running instances per availability zone per group to 7.
- Because of this limit, we cannot leverage spread placement groups to guarantee that replicas are allocated to different physical hosts in the same availability zone.
- Instead, we spread the brokers in each Kafka cluster among three availability zones,
- and ensure that replicas of each topic partition are spread among the availability zones to withstand up to two broker failures per cluster.

Kafka Cluster auto-healing:

- broker-failures
- we built DoctorKafka, it performs partition reassignment during broker failure.
- when underlying physical machines were degraded, brokers ran into unexpected bad states.
- producers and consumers from dependent services may still try to talk to failed broker, resulting in issues in dependent service.
- replacing failed brokers quickly is important for guaranteeing kafka service quality.
- we improved DoctorKafka with a broker replacement feature that allows to replace failed brokers automatically using provided scripts.
- replacing too many brokers in a short period of time can cause data loss, as clusters only store 3 replicas of data.
- to address this issue, we built a rate limiting feature in DoctorKafka that allows it to replace only 1 broker for a cluster in a period of time.
- AWS ec2 api allows users to replace instances while keeping hostnames and IP addresses unchanged, which enables us to minimize the impact of broker replacement on dependent services.



```
controlled.shutdown.enable=true

# kafka broker id settings
reserved.broker.max.id=10000

# listener related settings
listeners=PLAINTEXT://:9092,SSL://:9093
security.inter.broker.protocol=PLAINTEXT
inter.broker.protocol.version=2.0.0

# log related settings
log.dirs=$log_dir_path
log.retention.hours=$hours
log.message.format.version=2.0

# topic related settings
auto.create.topics.enable=false
delete.topic.enable=false

default.replication.factor=3
message.max.bytes=4194304
num.network.threads=8
num.partitions=8
num.recovery.threads.per.data.dir=6
num.replica.fetchers=2
replica.fetch.max.bytes=4194304
min.insync.replicas=2
socket.receive.buffer.bytes=2097152
socket.send.buffer.bytes=2097152
connections.max.idle.ms=600000

# Leader balance settings
leader.imbalance.per.broker.percentage=0

# Zookeeper connection settings
zookeeper.connect=$zookeeper_url
zookeeper.connection.timeout.ms=6000
zookeeper.session.timeout.ms=6000
```

- The following is Pinterest Kafka java parameters.

```
-Xms8g -Xmx8g
-XX:NewSize=512m -XX:MaxNewSize=512m
-XX:MetaspaceSize=128m -XX:+UseG1GC -XX:MaxGCPauseMillis=25
-XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=25 -XX:MaxMetaspaceFreeRatio=75
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution
-Xloggc:/var/log/kafka/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=40
-XX:GCLogFileSize=50M
-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.rmi.port=9999
```

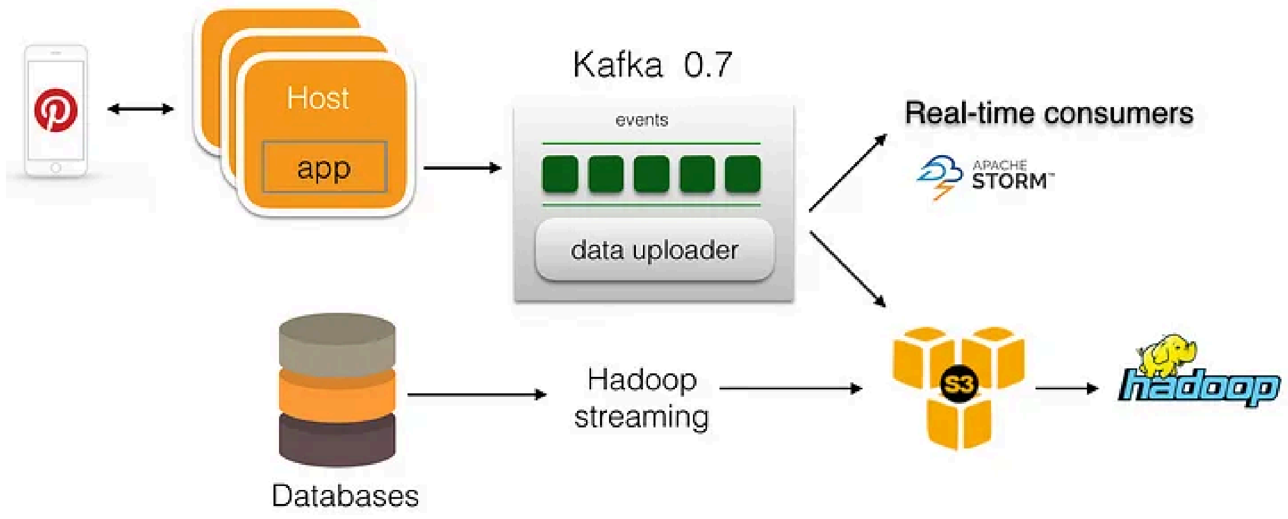
- We enable TLS access for Kafka at Pinterest.
- each KafkaChannel with a ssl connection costs ~122K memory,
- and Kafka may accumulate a large number of unclosed KafkaChannels due to frequent re-connection
- We use a 8GB heap size to minimize the risk of having Kafka run into long-pause GC.
- We used a 4GB heap size for Kafka process before enabling TLS.

At Pinterest, we use data to gain insights to make decisions for everything, and ultimately improve the whole experience for Pinners.

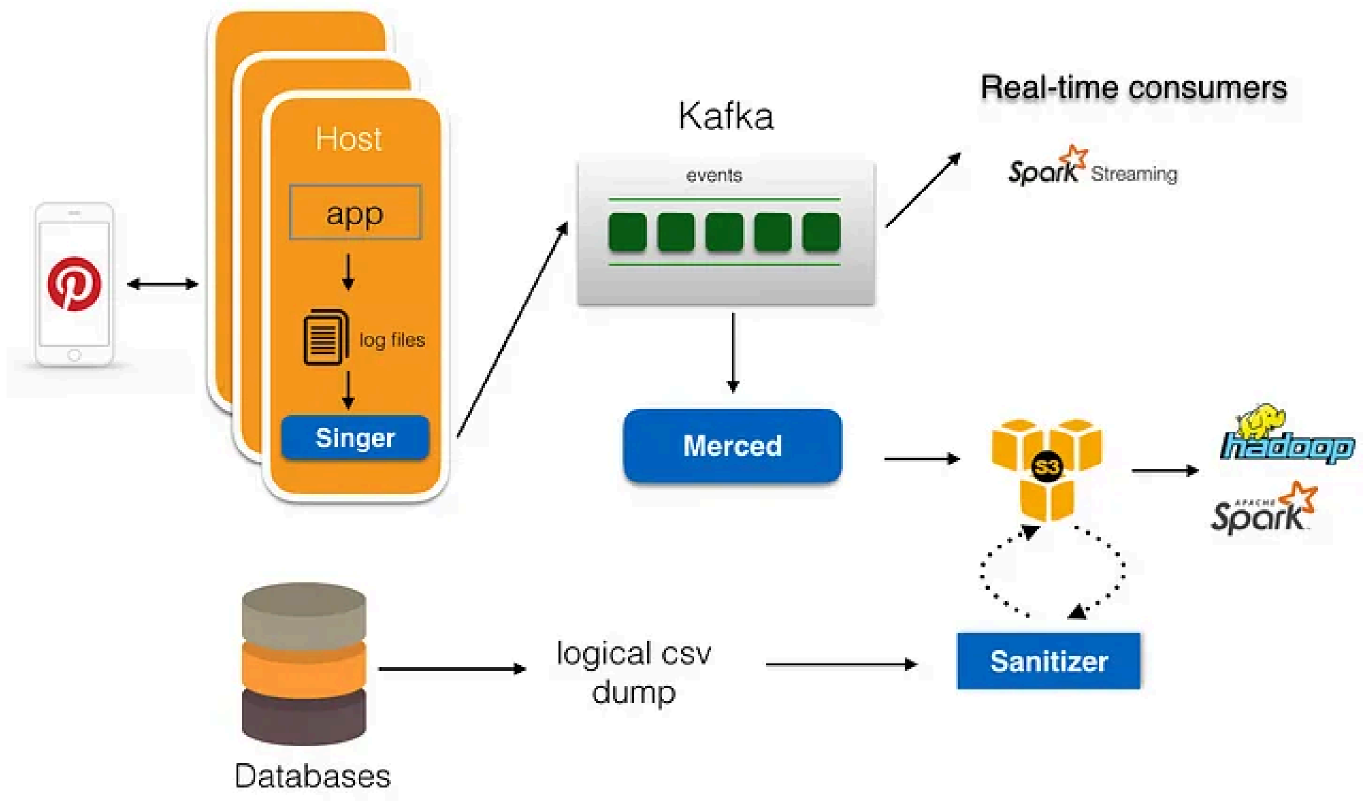
- everyday we log over 100 TB of data.
- to use data effectively, we first have to reliably ingest it and prepare it for downstream usage.

Data Ingestion Overview:

- all about collecting data from various sources and moving it to persistent storage.
- with following requirements:
  - Reliability: data collection & transportation should be with min data loss.
  - Performant: high throughput and low latency
  - efficient: minimum computation and storage resources usage
  - flexibility: should support various sources and data formats
- 2 primary categories of data sets:
- online service logs and DB dumps
- service logs are continuous log streams generated by services across 1000s of hosts.
- pipeline for collecting service logs is composed of 3 stages: log collections, log transportation and log persistence.

- DB dumps are logical DB backups and are generated hourly and daily.
- DB ingestion often includes DB dump generation and post processing.
- 

- in 2014, we used kafka as central msg transporter on online service side.
- app servers wrote log directly to kafka.
- A data uploader on each kafka broker uploaded kafka log files to s3.
- on DB dump side, we had a customized hadoop streaming job to pull data from DB and write the results to s3.
- later on, instead of using hadoop stream to pull data from DB, the DBs directly store logical dump to s3 and use sanitization framework to generate datasets needed for downstream processing.


Host
app
log files
Singer
Kafka
events
Real-time consumers
Spark Streaming
Merced
S3
hadoop
Apache Spark
Sanitizer
logical csv dump
Databases