Welcome to the World of Distributed Messaging Queue

How to choose the number of partitions in a Kafka cluster?

- Topic partition is the unit of parallelism in Kafka
- On both the producer and the broker side, writes to different partitions can be done fully in parallel. So expensive operations such as compression can utilize more hardware resources.
- On the consumer side, Kafka always gives a single partition's data to one consumer thread. Thus, the degree of parallelism in the consumer (within a consumer group) is bounded by the number of partitions being consumed.
- Therefore, in general, the more partitions there are in a Kafka cluster, the higher the throughput one can achieve

Number of partitions in a Kafka cluster

- A rough formula for picking the number of partitions is based on throughput. You measure the throughout that you can achieve on a single partition for production (call it *p*) and consumption (call it *c*). Let's say your target throughput is *t*. Then you need to have at least *max(t/p, t/c)*partitions.
- The Per-Partition through put depends on
  - batching size
  - compression codec
  - type of acknowledgement
  - replication factor
- The consumer throughput is often application dependent since it corresponds to how fast the consumer logic can process each message. So, you really need to measure it

More Partitions Requires More Open File Handles

- Each partition maps to a directory in the file system in the broker. Within that log directory, there will be two files (one for the index and another for the actual data) per log segment.
- Currently, in Kafka, each broker opens a file handle of both the index and the data file of every log segment. So, the more partitions, the higher that one needs to configure the open file handle limit in the underlying operating system.
  - ulimit -n 10000
- We have seen production Kafka clusters running with more than 30 thousand open file handles per broker.

More Partitions May Increase Unavailability

- Kafka supports intra-cluster replication, which provides higher availability and durability.
- A partition can have multiple replicas, each stored on a different broker.
- Internally, Kafka manages all those replicas automatically and makes sure that they are kept in sync.

More Partitions May Increase Unavailability

- Both the producer and the consumer requests to a partition are served on the leader replica.
- When a broker fails, partitions with a leader on that broker become temporarily unavailable. Kafka will automatically move the leader of those unavailable partitions to some other replicas to continue serving the client requests. This process is done by one of the Kafka brokers designated as the controller. It involves reading and writing some metadata for each affected partition in ZooKeeper. Currently, operations to ZooKeeper are done serially in the controller.
- In the common case when a broker is shut down cleanly, the controller will proactively move the leaders off the shutting down broker one at a time. The moving of a single leader takes only a few milliseconds. So, from the clients perspective, there is only a small window of unavailability during a clean broker shutdown.
- However, when a broker is shut down uncleanly (e.g., kill -9), the observed unavailability could be proportional to the number of partitions. Suppose that a broker has a total of 2000 partitions, each with 2 replicas. Roughly, this broker will be the leader for about 1000 partitions. When this broker fails uncleanly, all those 1000 partitions become unavailable at exactly the same time.

More Partitions May Increase End-to-end Latency

- The end-to-end latency in Kafka is defined by the time from when a message is published by the producer to when the message is read by the consumer. Kafka only exposes a message to a consumer after it has been committed, i.e., when the message is replicated to all the in-sync replicas. So, the time to commit a message can be a significant portion of the end-to-end latency.
- By default, a Kafka broker only uses a single thread to replicate data from another broker, for all partitions that share replicas between the two brokers. Our experiments show that replicating 1000 partitions from one broker to another can add about 20 ms latency, which implies that the end-to-end latency is at least 20 ms. This can be too high for some real-time applications

More Partitions May Require More Memory In the Client

- One of the nice features of the new producer is that it allows users to set an upper bound on the amount of memory used for buffering incoming messages. Internally, the producer buffers messages per partition.
- Enough data has been accumulated or enough time has passed, the accumulated messages are removed from the buffer and sent to the broker.
- If one increases the number of partitions, message will be accumulated in more partitions in the producer. The aggregate amount of memory used may now exceed the configured memory limit. When this happens, the producer has to either block or drop any new message, neither of which is ideal. To prevent this from happening, one will need to reconfigure the producer with a larger memory size.
- In general, more partitions in a Kafka cluster leads to higher throughput. However, one does have to be aware of the potential impact of having too many partitions in total or per broker on things like availability and latency.

Secure Sockets Layer (SSL)

- SSL is required to authenticate the client request before pushing/pulling the data from Kafka broker.
  - listeners=PLAINTEXT://10.191.209.10:9092,SSL://10.191.209.10:9093
  - security.inter.broker.protocol = SSL
  - ssl.keystore.location=/home/root/8x8/kafka_2.10-0.9.0.1/server.keystore.jks
  - ssl.keystore.password=india123
  - ssl.key.password=india123
  - ssl.truststore.location=/home/root/8x8/kafka_2.10-0.9.0.1/server.truststore.jks
  - ssl.truststore.password=india123
- Only supported in Kafka 0.9.x and above version

Generate SSL and Creating CA

- Generate SSL Key
  - keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genkey
- Creating your own CA
  - openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
  - keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
  - keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert

Signing the CA

- Signing the CA
  - keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-file
  - openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days {validity} -CAcreateserial -passin pass:{ca-password}
  - keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
  - keytool -keystore server.keystore.jks -alias localhost -import -file cert-signed

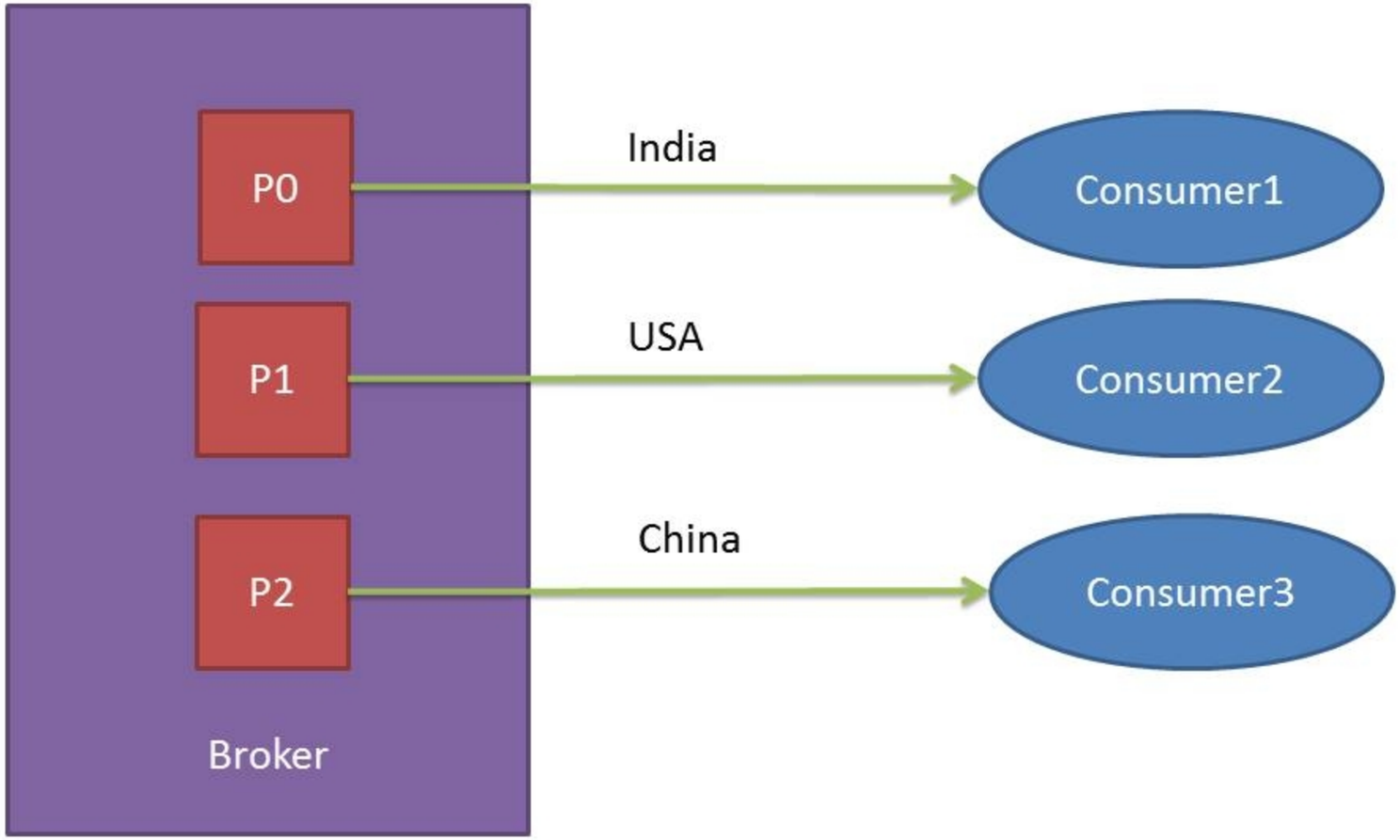
Configuration at Client end

- Producer and Conumer
  - security.protocol=SSL
  - ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks
  - ssl.truststore.password=test1234
  - ssl.keystore.location=/var/private/ssl/kafka.client.keystore.jks
  - ssl.keystore.password=test1234
  - ssl.key.password=test1234

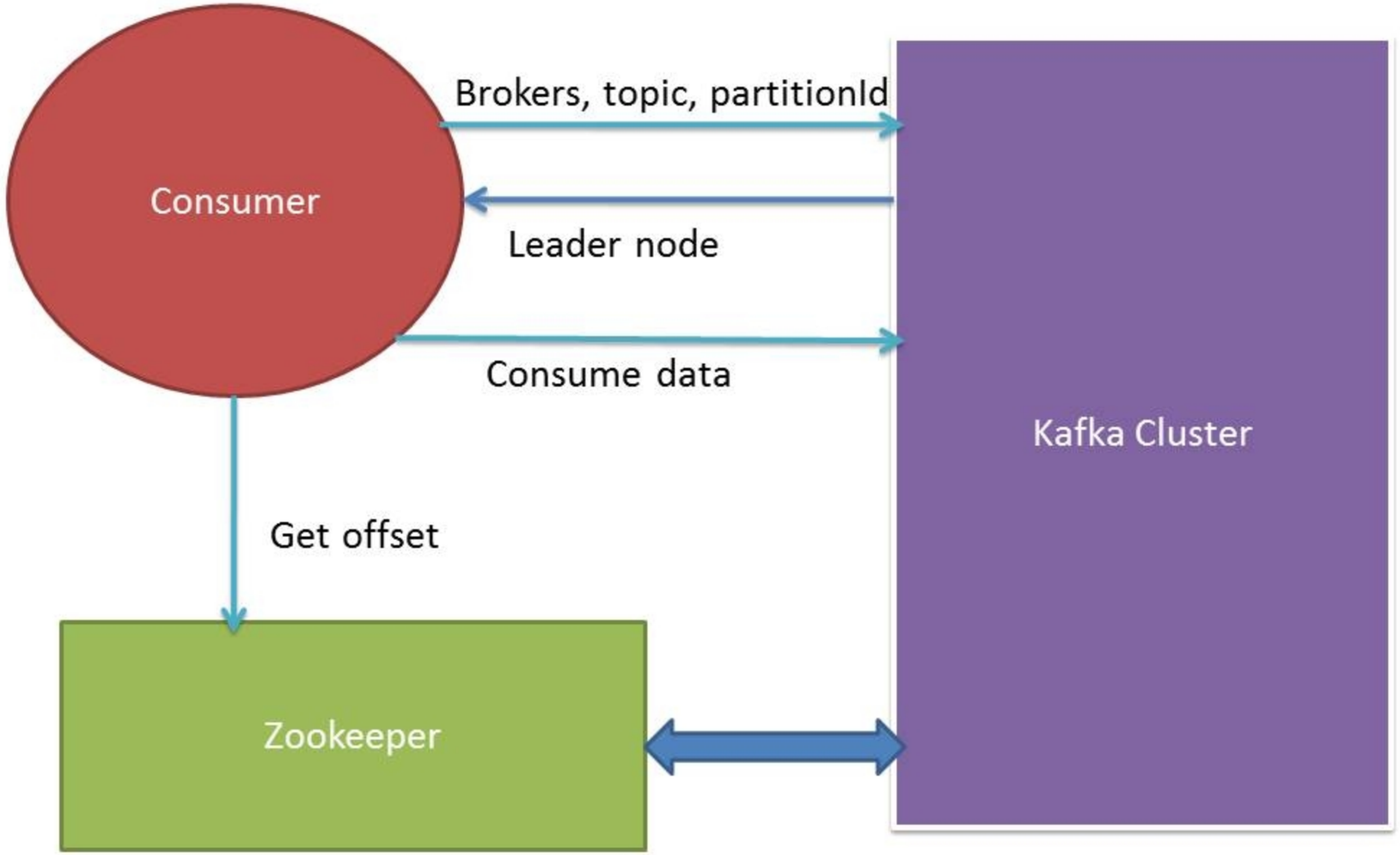
Kafka low level consumer

- We can read a message multiple times.
- We can control the message reading mechanism
- We need to identify the active Broker and also find out which Broker is the leader for your topic and partition.
- Build the input request that define what data we want to read.
- Fetch the data from partition.
- Identify and recover from leader changes

Control Message Reading Mechanism



Low level consumer Hands-On



1. Consumer send the request to find the leader partition of a broker (request contains the broker list, topicName, partitionId)
2. Return the leader node
3. Get the offset from which we need to start the data read
4. Start data consuming
5. Re-elect the leader, if leader node goes down

Data Retention

- The Kafka cluster retains all published messages whether or not they have been consumed for a configurable period of time.
- For example if the log retention is set to two days, then for the two days after a message is published it is available for consumption, after which it will be discarded to free up space.
- Kafka's performance is effectively constant with respect to data size so retaining lots of data is not a problem.

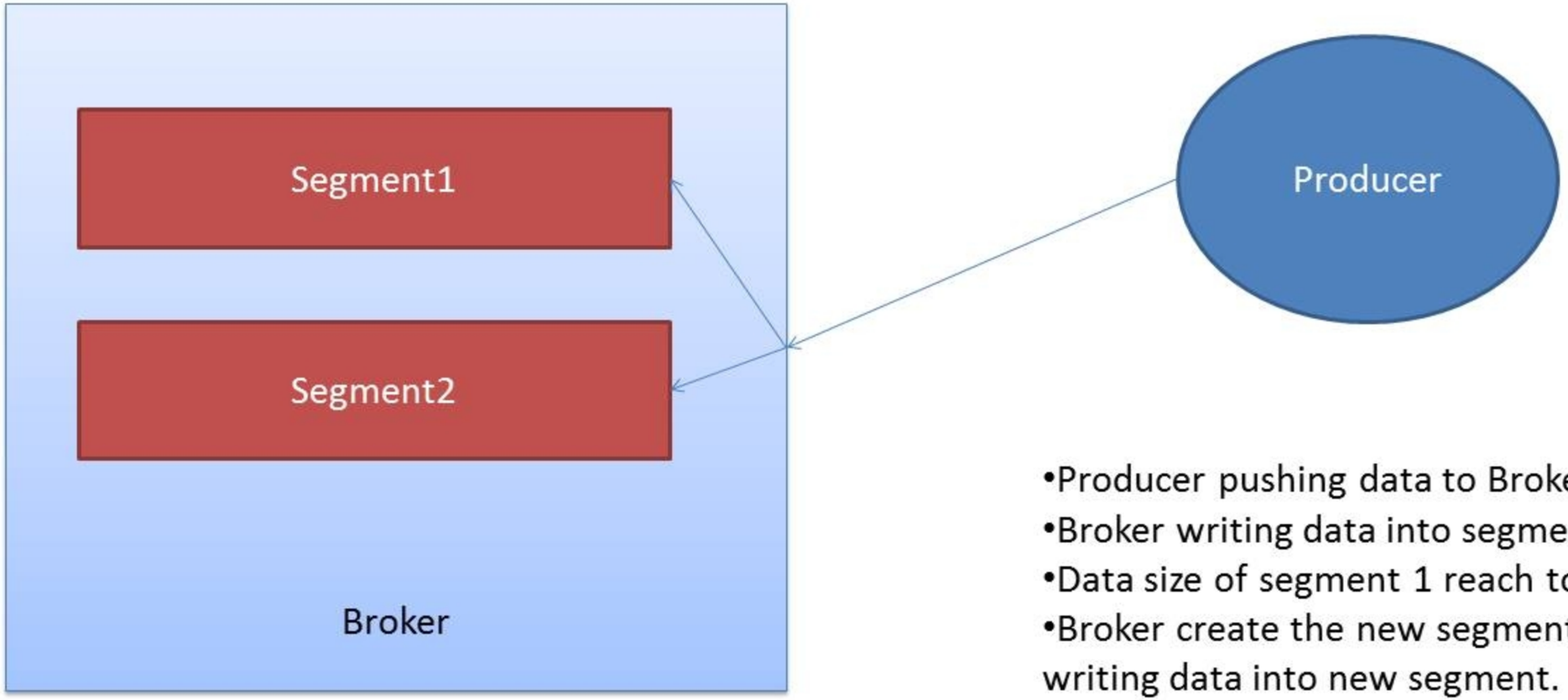
Data Retention

- The property **log.retention.{minutes,hours}** define the amount of time to keep a log segment before it is deleted, i.e. the default data retention window for all topics. The default value of this property is 7 days.
- The property **log.retention.bytes** define the amount of data to retain in the log for each topic-partitions. Note that this is the limit per-partition so multiply by the number of partitions to get the total data retained for the topic. The default value of this property is -1.
- Also note that if both **log.retention.hours** and **log.retention.bytes** are both set we delete a gsegment when either limit is exceeded.
- We can overwrite this property by setting **retention.bytes** and **retention.ms** properties at the time of topic creation.
- The propety **log.retention.check.interval.ms** define the period with which we check whether any log segment is eligible for deletion to meet the retention policies. The default value of this property is 5 minutes.

Log Segment

- The property **log.segment.bytes** define the log for a topic partition is stored as a directory of segment files. This setting controls the size to which a segment file will grow before a new segment is rolled over in the log. The default value is 1GB.
- The property **log.roll.hours** define the setting will force Kafka to roll a new log segment even if the log.segment.bytes size has not been reached. The defaulr value is 168 hours.

Log Segment



- Producer pushing data to Broker.
- Broker writing data into segment 1.
- Data size of segment 1 reach to 1 GB.
- Broker create the new segment and start writing data into new segment.

Log Segment and Data Retention

- Kafka doesn’t delete single message but delete all the records belong to one segment in one go.
- It only mark the data deleted (soft delete).
  - Data inserted before this offset is marked as deleted.
- Why it does not delete the single record?
  - Deleting a single record from a file is very performance incentive task

Assignment

- Create a three nodes kafka cluster.
- Let's assume the file contains data of four countries india, usa, uk and chine.
  - Log file must contains following four fields
  - Tweet text, country, time, username
- Read the data from multiple files, convert the record into Map<String,Object> and pushed into Kafka using Sync producer.
- Write a Map encoder/decoder to convert the Map to bytes and bytes to Map.
- Run the producer on machine other then brokers.
- Create a topic having four partitions and replication factor 3.
- Create a partition class to push data of India on partition 0, data of USA on partition 1 and so on.
- Consume the data from Kafka and store all the data of India on 1 file, data of USA on other file and so on.
- Run the consumer on machine other then brokers

Apache Storm

- ✓ High Distributed real time **computation** system
- ✓ Horizontally scalable
- ✓ Fault Tolerance
- ✓ Can easily be used with any programming language
- ✓ Guaranteed message processing

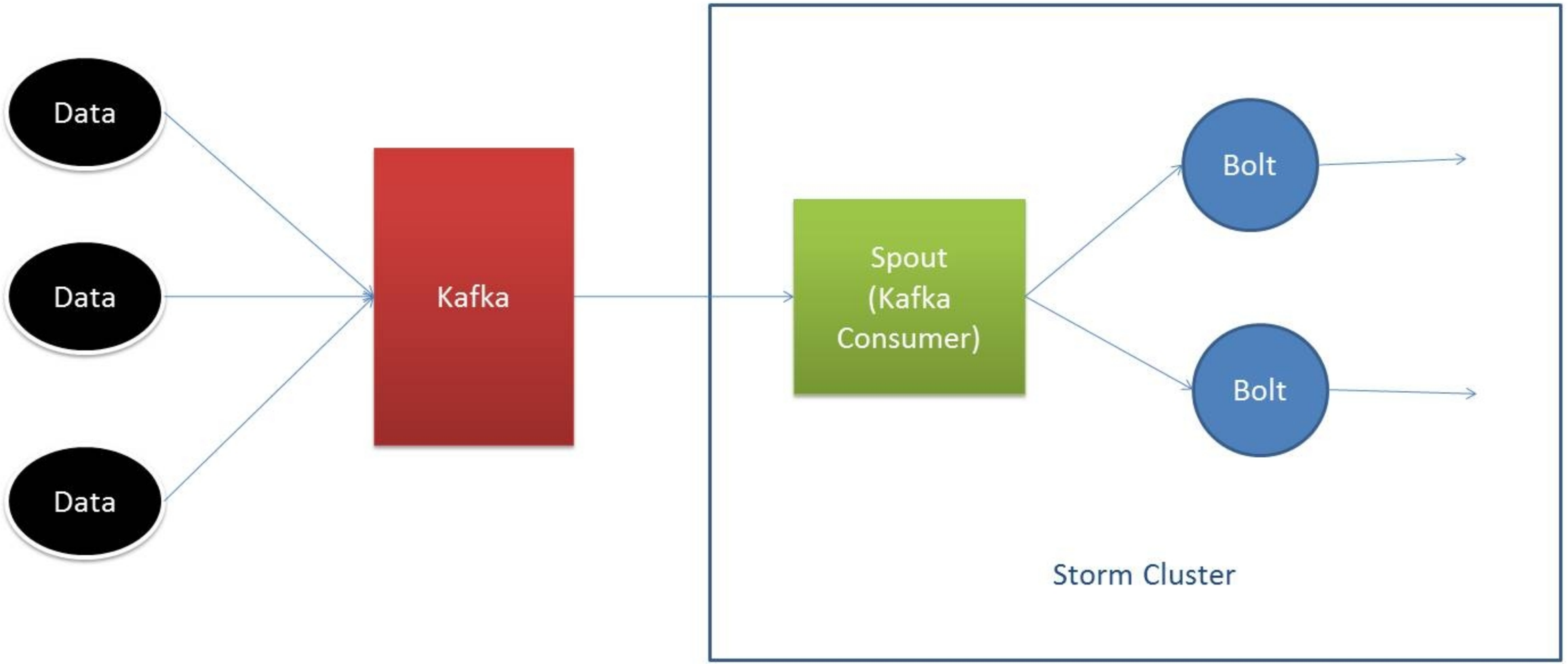
Apache Storm

- Apache 2.0 license
- Written in closure and API are exposed in Java
- Master/Slave architecture
- Rich community
- Easy to operate:
  - Storm is much easy to deploy and manage.
- Fast:
  - Storm Cluster can process **billion of records** per second

Kafka with Storm

- Consider, we have a real time app handling high volume data.
- Storm Spout doesn’t buffer/Queue the data.
- We would require external buffer/Queue for storing that data.
- Kafka is best choice for Queuing high volume data.
- Storm will read the data from Kafka and applies some required manipulation.

Kafka with Storm



DataFlair Web Services Pvt Ltd

+91-8451097879
info@data-flair.com
http://data-flair.com