Apache Kafka

Apache Kafka is a distributed, scalable Message Broker. Used to achieve heterogeneous and Asynchronous Integration with high throughputs.

Software Systems without a message broker

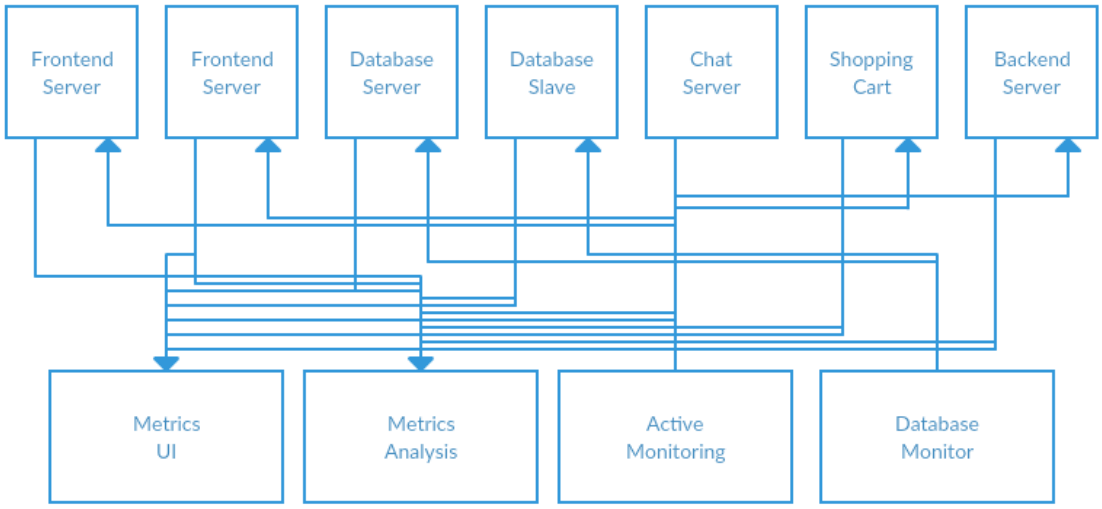


Figure 1-2. Many metrics publishers, using direct connections

Software Systems with a message broker

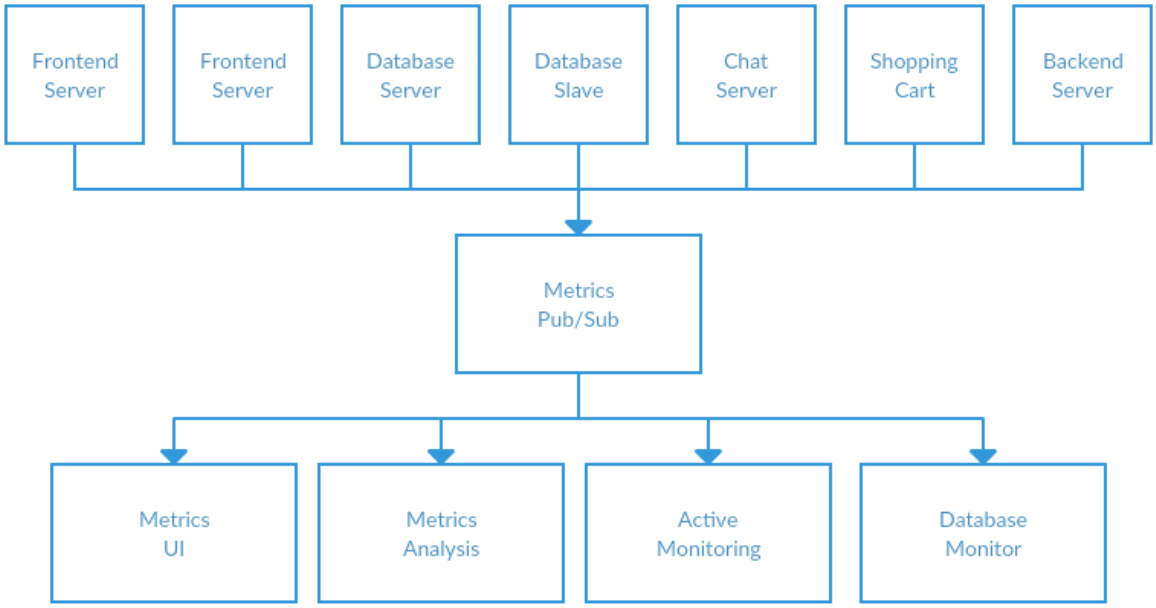
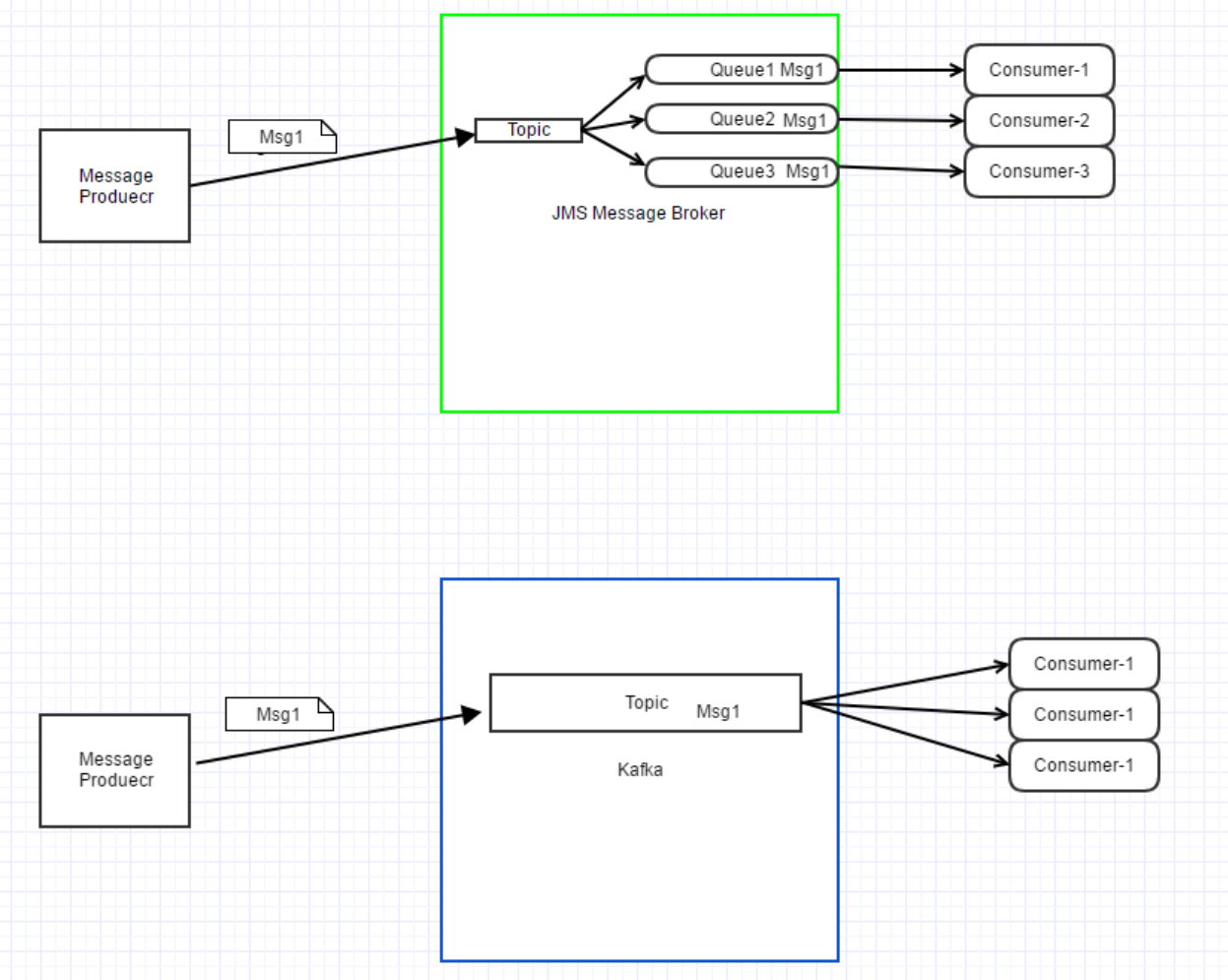


Figure 1-3. A metrics publish/subscribe system

Traditional JMS Message Broker Vs Kafka:



Kafka	**JMS**
High Through Out	Relatively Low through Out
Relatively High Latency	Low latency
Distributed	Non Distributed
Horizontally Scalable	Vertically Scalable
Messages are available even after consumption	Messages are deleted once consumed from message broker

Messages in Kafka are categorized into topics. The closest analogy for a topic is a database table, or a folder in a filesystem. Topics are additionally broken down into a number of partitions. Going back to the “commit log” description, a partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. Note that as a topic generally has multiple partitions, there is no guarantee of time-ordering of messages across the entire topic, just within a single partition.

Kafka Internals:

Kafka uses Apache Zookeeper to maintain the list of brokers that are currently members of the cluster. Every broker has a unique identifier that is either set in the broker configuration file or automatically generated. Every time a broker process starts, it registers itself with its id in Zookeeper by creating an ephemeral node.

The Controller:

The controller is one of the Kafka brokers that in addition to the usual broker functionality is also responsible for the task of electing partition leaders (we’ll discuss partition leaders and what they do in the next section). The first broker that starts in the cluster becomes the controller by creating an ephemeral node in Zookeeper, /controller. When other brokers start, they also try to create this node, but receive a “node already exists” exception and “realize” that the controller node already exists and that the cluster already has a controller.

The brokers create a Zookeeper watch on the controller node, so they get notified on changes to this node. This way we guarantee the cluster will only have one controller at a time. When the controller broker is stopped or loses connectivity to Zookeeper, the ephemeral node will disappear. Other brokers in the cluster will be notified through the Zookeeper watch that the controller is gone and will attempt to create the controller node in Zookeeper themselves. The first node to create the new controller in Zookeeper is the new controller, while the other nodes will receive “node already exists” exception and re-create the watch on the new controller node. Each time a controller is elected, it receives a new, higher, controller epoch number through a Zookeeper conditional increment operation. The brokers know the current controller epoch and if they receive a message from a controller with older number, they know to ignore it.

When the controller notices that a broker left the cluster (by watching the relevant Zookeeper path), it knows that all the partitions that had a leader on that broker will need a new leader. It goes over all the partitions that need a new leader, determine who the new leader should be (simply the next replica in the replica list of that partition) and sends a request to all the brokers that contain either the new leaders or the existing followers for those partitions. The request contains information on who is the new leader and who are the followers for the partitions. The new leaders now know that they need to start serving producer and consumer requests from clients, while the followers now know that they need to start replicating messages from the new leader.When the controller notices a broker joined the cluster, it uses the broker ID to check if there are replicas that exist on this broker. If there are, the

controller notifies both new and existing brokers of the change, and the replicas on the new broker start replicating messages from the existing leaders.To summarize, Kafka uses Zookeeper's ephemeral node feature to elect a controller and to notify the controller when nodes join and leave the cluster. The controller is responsible for electing leaders among the partitions and replicas whenever it notices nodes join and leave the cluster. The controller uses epoch number to prevent "split brain" scenario where two nodes believe each is the current controller.

Replicas:

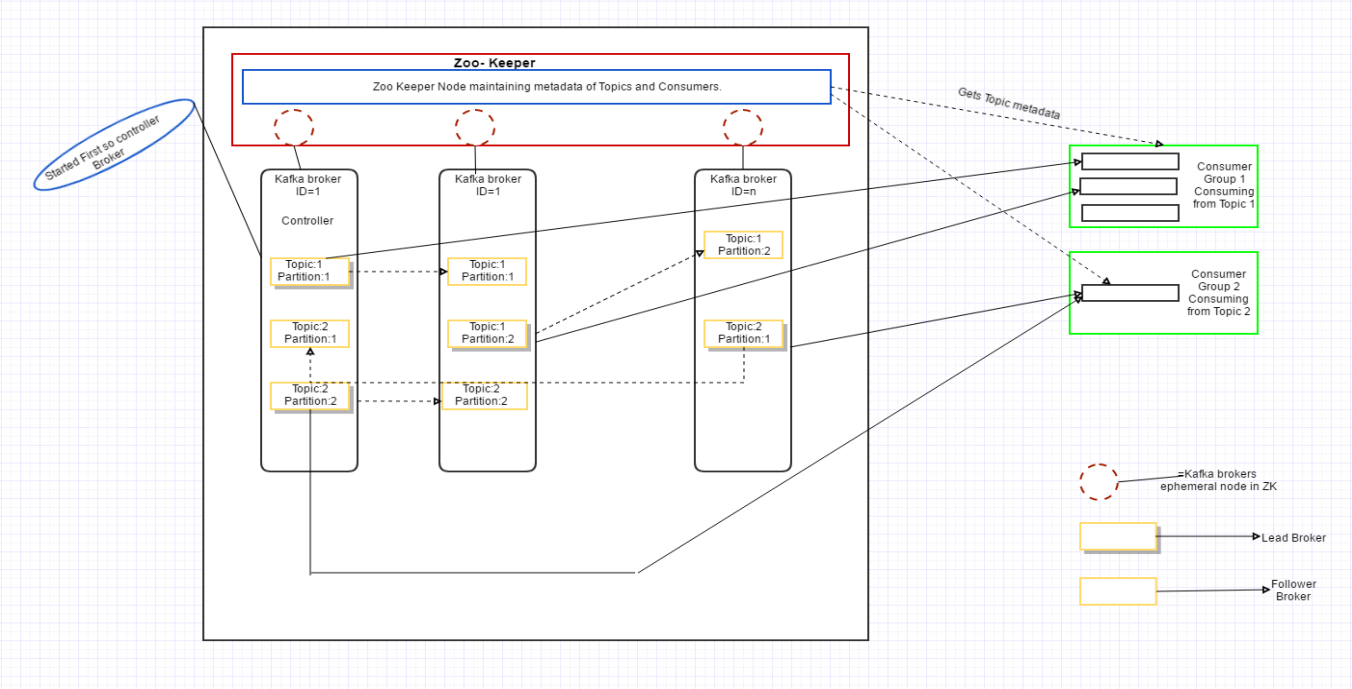
Each topic is partitioned, and each partition can have multiple replicas. Those replicas are stored on brokers and each broker typically stores hundreds or even thousands of replicas, belonging to different topics and partitions. There are two types of replicas

Leader replica - Each partition has a single replica designated as the leader. All produce and consume requests go through the leader, in order to guarantee consistency.

Follower replica - All replicas for a partition that are not leaders are called followers. Followers don't serve client requests, their only job is to replicate messages from the leader and stay up to date with the most recent messages the leader has. In the event a leader replica for a partition crashes, one of the follower replicas will be promoted to become the new leader for the partition.

In order to stay in sync with the leader, the replicas send the leader Fetch requests, the exact same type of requests that consumers send in order to consume messages. In response to those requests, the leader sends the messages to the replicas. Those Fetch requests contain the offset of the message that the replica wants to receive next, and they will always be in order.

The inverse, replicas that are consistently asking for the latest messages, are called "in sync replicas". Only in-sync replicas are eligible to be elected as partition leaders in case the existing leader fails.

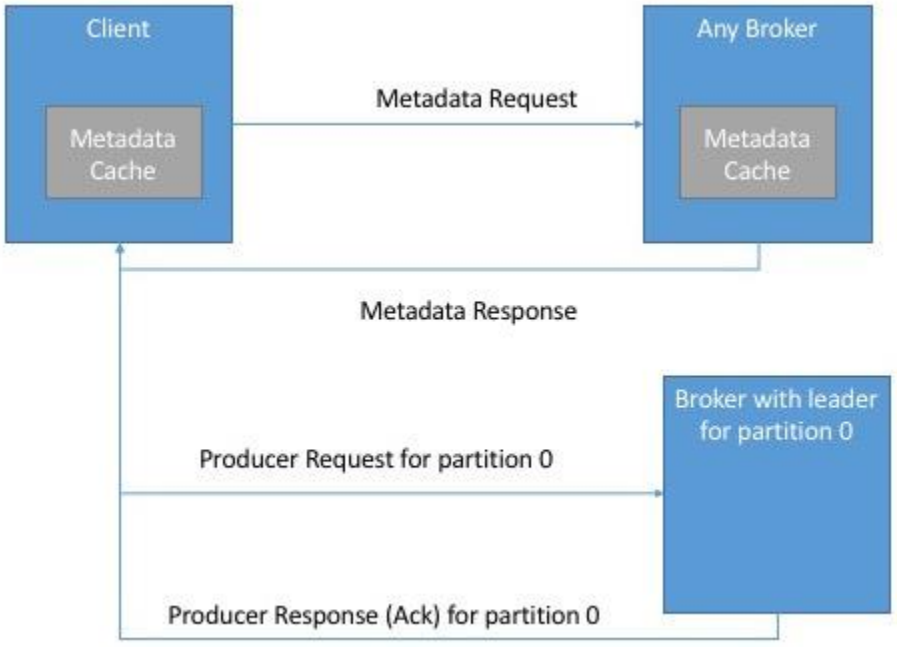


How do the clients know where to send the requests?

Both produce requests and fetch requests have to be sent to the leader replica of a partition. If a broker receives a produce request for a specific partition and the leader for this partition is on a different broker, the client that sent the produce request will get an error response with the error “Not a Leader for Partition”. The same error will occur if a fetch request for a specific partition arrives at a broker that does not have the leader for that partition. It is the responsibility of Kafka’s clients to always send produce and fetch requests to the broker that contains the leader for the relevant partition for the request.

Kafka clients use another request type called metadata request. The request includes a list of topics the client is interested in. The server response specifies which partitions exist in the topics, who are the replicas for each partition and which replica is the leader. Metadata request can be sent to any broker since all brokers have a metadata cache that contains this information. Clients typically cache this information and use it to direct produce and fetch requests to the correct broker for each partition. They also need to occasionally refresh this information (refresh intervals are controlled by the + metadata.max.age.ms+ configuration parameter) by sending another metadata request, so they will know if the topic metadata changed - for example if a new broker was added and some replicas were

moved to a new broker. In addition, if a client receives “Not a Leader” error to one of its requests, it will refresh its metadata before trying to send the request again, since the error indicates that the client is using outdated information and is sending requests to the wrong broker.



\

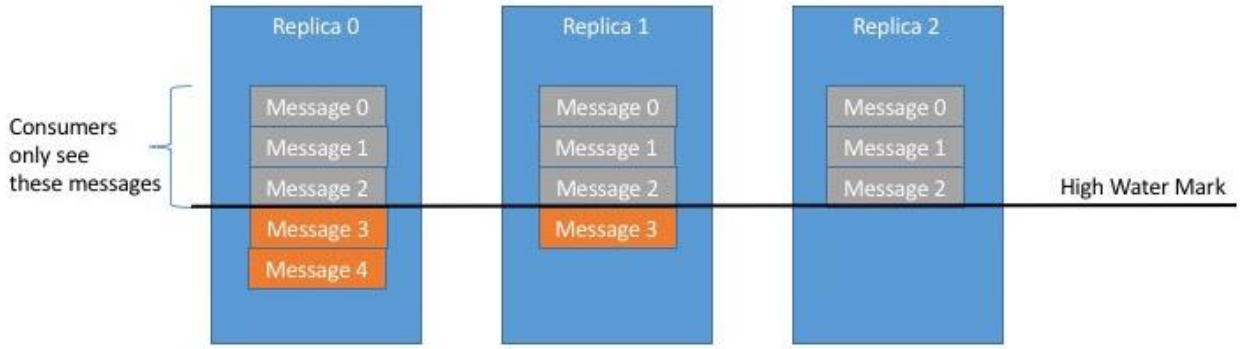
Fetch Requests:

Brokers process fetch requests in a way that is very similar to the way produce requests are handled. The client sends a request, asking the broker to send messages from a list of topics, partitions and offsets. Something like “Please send me messages starting at offset 53 in partition 0 of topic Test and messages starting at offset 64 in partition 3 of topic Test”. Clients also specify a limit to how much data the broker can return for each partition. The limit is important because clients need to allocate memory that will hold the response sent back from the broker. Without this limit, brokers could send back replies large enough to cause clients to run out of memory.

When the leader receives the request it first checks if the request is valid - does this offset even exist for this particular partition? If the client is asking for a message that is so old that it got deleted from the partition or an offset that does not exist yet, the broker will respond with an error.

It is also interesting to note that not all the data that exists on the leader of the partition is available for clients to read. Most clients can only read messages that were written to all in-sync replicas (follower

replicas, even though they are consumers, are exempt from this - otherwise replication would not work). We already discussed that the leader of the partition knows which messages were replicated to which replica, and until a message was written to all in-sync replicas, it will not be sent to consumers - attempts to fetch those messages will result in an empty response rather than an error. The reason for this behavior is that messages that were not replicated to enough replicas yet are considered “unsafe” - if the leader crashes and another replica takes its place, these messages will no longer exist in Kafka. If we allowed consumers to read these messages, this will lead to inconsistent behavior where some consumers read and processed a message that simply doesn’t exist if someone later looks for it. Instead we wait until all the in-sync replicas get the message and only then we allow consumers to read it. This behavior also means that if replication between brokers is slow for some reason, it will take longer for new messages to arrive to consumers (since we wait for the messages to replicate first). This delay is limited to replica.lag.time.max.ms - the amount of time a replica can be delayed in replicating new messages while still being considered “in-sync”.



Physical Storage:

The basic storage unit of Kafka is a partition replica. Partitions cannot be split between multiple brokers and not even between multiple disks on the same broker. So the size of a partition is limited by the space available on a single mount point (a mount point will consist of either a single disk, if JBOD configuration is used, or multiple disks if RAID is configured.

When configuring Kafka, the administrator defines a list of directories in which the partitions will be stored - this is the log.dirs parameter.

Because finding the messages that need purging in a large file and then deleting a portion of the file is both time consuming and error prone, we chose to instead split each partition into many segments. By default each segment contains either 1GB of data or a week of data, whichever is smaller. As a Kafka broker is writing to a partition, if the segment limit is reached, we close the file and start a new one. The segment we are currently writing to is called an active segment. The active segment is never deleted, so if you set log retention to only store a day of data, but each segment contains 5 days of data, you will really keep data for 5 days as we can't delete the data before the segment was closed.

File Format:

Inside the file, we store Kafka messages and their offsets. The format of the data on the disk is identical to the format of the messages that we send from the producer and later send to consumers. Using the same message format on disk and over the wire is what allows Kafka to use the zero-copy optimization when sending messages to consumers and also avoid de-compressing and re-compressing messages that the producer already compressed.

Administering Kafka:

It is permitted, but not recommended, to have topic names that start with two underscores. Topics of this form are considered internal topics for the cluster (such as the __consumer_offsets topic for consumer group offset storage).

It is not possible to reduce the number of partitions for a topic.

The brokers in the cluster must have been configured with the delete.topic.enable option set to true. If this option has been set to false, then the request to delete the topic will be ignored.

Kafka Commands Primer:

1. **Kafka Home Directory:** HDP VM Comes with Single node Kafka instance and Kafka will be installed in ***/usr/hdp/current/kafka-broker*** location.

cd /usr/hdp/current/kafka-broker

2. **Execute the below command to start Kafka Server:**
   bin/kafka-server-start.sh config/server.properties
   Broker Starts on port 6667 by default
3. **Use the below command to see the list of Topics:**
   bin/kafka-topics.sh --list --zookeeper localhost:2181
   By default Zookeeper runs on 2181 port on HDP VM.

4. **Command to create a Kafka Topic.**
   bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic demo_topic
5. **Command to publish messages to Kafka topic using the utilities provided with in Kafka.**
   bin/kafka-console-producer.sh --broker-list sandbox.hortonworks.com:6667 --topic demo_topic

   Use the below command to run a java code (Built for demo) to publish messages to Kafka topic.

   java -cp /root/kafka/kafka-1.0-jar-with-dependencies.jar kafka.SimpleProducer_VM -topic demo_topic -brokers sandbox.hortonworks.com:6667 -file /root/kafka/picked_order_messages/picked_order_messagebody.txt -messagingMode Sync –messagKay Picked_order

6. **Command to consume messages from Kafka Topic using Kafka utilities.**
   bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic demo_topic --from-beginning

   Use the below command to run a java code (Built for demo) to consume messages from Kafka topic.

   java -cp /root/kafka/kafka-1.0-jar-with-dependencies.jar kafka.BasicConsumerExample_VM -topic demo_topic -brokers sandbox.hortonworks.com:6667

7. **Command to increase number of Partitions for a Kafka Topic.**
   bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic demo_topic
   bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic demo_topic --partitions 2
   bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic demo_topic
8. **Command to sun the Spark Consumer.**

   spark-submit --class sparkStreaming.VM_Hive_Checkpoint /root/kafka/kafka-1.0-jar-with-dependencies.jar sandbox.hortonworks.com:6667 samplekafkaconsumer KVJ1 test.KVJ_Topic_Offsets

   sandbox.hortonworks.com:6667 is broker URL

samplekafkaconsumer : Consumer Group
KVJ1 : Topic To Consume From
test.KVJ_Topic_Offsets : MySql Table in which offsets are maintained.

For Demo:

1. **Create the below hive table:**

```
CREATE TABLE `tmp_picked_order`(
 `orderid` string,
 `recommendedfulfillmenttime` string,
 `tenantid` string,
 `lineitemid` string,
 `tpnb` string)
```

2. **Create the below Mysql Table (user name= root, password= blank)**
   create table test.KVJ_Topic_Offsets (insert_date_time text, topic text, partition text, fromoffset text, untilloffset text);
3. **Place the below files in** */root/kafka/picked_order_messages* **in VM***(picked_order_messagebody_set2.txt, picked_order_messagebody.txt)*


picked_order_messagebody.txt


picked_order_messagebody_set2.txt

Kafka Spark Streaming code artifacts.


pom.xml


SimpleProducer_VM.java


BasicConsumerExample_VM.java


VM_Hive_Checkpoint.scala

VM_Hive_Lowest_Offset.scala

Role of ZooKeeper in Kafka :

All metadata about topics, brokers, consumer offsets are managed in ZooKepper.

In the introduction to Apache Kafka we listed some of ZooKeeper use cases in Kafka. This time it's a good moment to describe them better. Generally, ZooKeeper stores a lot of shared information about consumers and brokers:

1. **Brokers**
   - *state* - ZK determines if broker is alive always when it regularly sends *heartbeats* requests. In additional, when broker is constraint to handle replication, it must be able to follow replication needs, ie. not have an important replication debth.
   - *quotas* - Kafka allows some clients (identified by client.id property) to have different producing and consuming quotas. This value is set in ZK under /config/clients path. This change can be made in *bin/kafka-configs.sh* script.
   - *replicas* - ZK keeps a set of in-sync replicas (ISR) for each topic. This set is synchronized. It means that every time when one node fails, the ISR is updated. If the failing node was previously selected leader, ZK will, based on currently live nodes, elect new leader.
   - *nodes and topics registry* - ZK stores nodes and topic registries. We can find there all available brokers and, more precisely, which topics are held by each broker. They're stored under /brokers/ids and /brokers/topics *zNodes*. Nodes and topic registries are ephemeral nodes which means that they're alive only when given broker keeps a connection (session) open to ZK instance. All this information is destroyed once session closes.

     The register is made automatically by the broker when it's started.
2. **Consumers**
   - *offsets* - in Kafka's 0.9.1 release, ZooKeeper is the default storage engine for consumer offsets. As announced in the documentation, the default storage mechanism will migrate further to special Kafka broker called Offset Manager. But by now, all information about how many messages were consumed by each consumer are stored in ZK.
   - *registry* - as in the case of brokers, consumers also have their own registry. And the same rules apply to it, ie. as ephemeral zNode, it's destroyed once consumer goes down and the register process is made automatically by consumer.
   - *partitions registry* - other registry related to consumers is about partitions ownership. Since each partition is consumed by exactly one consumer belonging to specific consumer group, Kafka must know the ownership relationship between partitions and consumers. And this information is stored in ZK under /consumers/${groupId}/owners/${topic} path as a zNode called by the pattern of ${brokerId}-${partitionId}. As previous consumer registry, this one is also ephemeral.

As you can see through that list, ZooKeeper is essentially employed to guarantee the synchronization between all participants in Kafka workflow - consumers and brokers.