APACHE KAFKA Tutorials/Hands-On

Kafka is a distributed , partitioned , replicated commit log service.

Fast -> Think Big Data!
-> Handle hundreds of MBs of reads & writes per second from many clients
->Designed for real time activity streams

Distributed & highly Scalable
->Cluster-Centric Design
->Grow Elastically & transparent

Kafka was created to serve as a centralized online data pipelining system:
----> Kafka decouples the data pipelines.

Kafka Use-Cases

Messaging
Website Activity Tracking
Metrics
Log Aggregation
Stream Processing

Main Kafka concepts

Topics

-> Feeds of messages are organized into topics
->Category or feed name to which messages are published
->Each topic consists of partitioned log
  ->Partitions allow log to scale horizontally
->Partitions are ordered,immutable sequence of messages that are continually appended
->Each message in partitions assigned sequential id-number (“the offset”)
  ->Uniquely identifies message within partition
->Messages retained for a configurable period of time.
->Messages consist of fixed-size header and opaque payload

Brokers

->Kafka runs on a cluster of servers

->Kafka Cluster comprised of one or more servers-called “brokers”

->Each Kafka broker stores one or more partitions

->Can spread a single topic’s partitions across multiple brokers

->Kafka Brokers are stateless

->Consumers have to keep track of what offset position they would like to read

Replication

->Each Partition gets replicated actress a number of servers
->Each partition has one “leader” server & zero or more “followers”
  ->Leader handles all read & write requests for the partition.
->If a leader fails,a follower elected new leader.
  ->Leader handles all read & write requests for the partition
->Followers passively replicate the leader

->If leader fails, a follower elected new leader

->Leaders keep track of In Sync Replicas - ”ISR”

->Message “committed” when all ISR for partition have applied it to there log
    ->Only committed messages are given to consumers
    ->Producers have option of whether to wait for commit (latency vs durability)

Producers

->Processes that publish message to a kafka topic
->Producer decides which message goes to which partition
  ->Round-robin for simple load balancer or custom defined.
->Producers write to single leader
  ->Provides load balancing-different broker can service each write.
->Kafka comes with a command-line producer client or you can write your own.

Consumers

->Processes that subscribe to topics & process the feed of published messages.
->Consumers read from topics
->Kafka comes with a command line consumer client or can write your own.
->Consumers belong to a consumer group
  ->Each message published to a topic is delivered to one consumer instance within each subscribing consumer group.
  ->Consumer instances in a consumer group can be in separate processes or separate machines.
  ->All consumer instances in the same group have load balanced across them.

->Consumers instances in different groups that subscribe to the same topic each get a copy of the messages.

Kafka requires Zookeeper

ZOOKEEPER

->Kafka requires Zookeeper to do things like

: Cluster membership
: Electing a controller
: Topic configuration(which topics)

V0.8 -> Older “high-level” consumers used ZooKeeper (tracking offsets…)

V0.9+ -> Only broker uses Zookeeper
-> New consumers using instead of Zookeeper

Kafka Guarantees

->Messages sent by a producer to a particular topic partition will be appended in the order they are sent.

->Consumer instances sees messages in the

order they are stored in the log

->Committed messages will not be lost as long as at least one in sync replica alive , at all times.

Kafka Cluster Configurations

Single node, single broker

Single node , multiple brokers

Multiple nodes, multiple brokers

LAB1

Ambari-> iopbeta42.locadomain:8080/

#cd /usr/iop/current/kafka-broker/
#ls
#bin/kafka-topics --list --zookeeper localhost:2181

Kafka-Command-Line

Zookeeper scripts:

START

zookeeper-server-start.sh

STOP

zookeeper-server-stop.sh

Tool used when doing migration to update acl of znodes

zookeeper-security-migrate.sh

Runs Zookeeper Shell

#bin/zookeeper-shell.sh localhost:2181

Kafka server start & stop

->Starts the Kafka server("broker")

->Pass in a server properties file

kafka-server-start.sh

#bin/kafka-server-start.sh config/server.properties

kafka-server-stop.sh

Running Multiple brokers on a single machine

Generally would want to run brokers on separate machines

Each brokers gets own properties file

->
#cp config/server.properties config/server-1.properties

#cp config/server.properties config/server-2.properties

Kafka Topics

Two ways to create topics:
1.Enabling the auto.create.topics.enable property
    -> Topic created when broker receives first messages for non-existent topic
    ->Topic created based on num.partitions and default.replication.factor.settings

2.Using bin/kafka-topics.sh

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1

```
--partitions 1 --topic mytest
```

List & describe topics

```
#bin/kafka-topics.sh --list --zookeeper
localhost:2181


#bin/kafka-topics.sh --describe --zookeeper
localhost:2181 --topic the-replicated-topic
```

Modifying Topics

->can increase the no of topics in a partition

-> Pre-existing data in the topic will not be reshuffled

```
#bin/kafka-topics.sh --zookeeper
localhost:2181 --alter --topic my_topic
--partition 8
```

Add configs

```
#bin/kafka-topics.sh --zookeeper
localhost:2181 --alter--topic my_topic
--config retention.ms=142800000
```

Remove configs

#bin/kafka-topics.sh --zookeeper localhost:2181 --alter--topic myh_topic --deleteConfig retention.ms

Deleting Topics

To enable deletion first set delete.topic.enable=true

#bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic my_topic

Kafka Console Producer

Producer command line client

Start up the console producer.

#bin/kafka-console-producer.sh --broker-list localhost:9092 --topic mytopic

Kafka Console consumer

Consumer command line client

->Prints messages from topics to standard output

->Each message printed on a new line

->Script runs stopped

Start up the consumer command line client

```
#bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic mytopic
--from-beginning

Consumer groups & offset checking

#./kafka-consumer-groups.sh --zookeeper
localhost:2181 --describe --group
test-consumer-group  GROUP, TOPIC , PARTITION
, CURRENT OFFSET , LOG END OFFSET , LAG ,
OWNER
```

LAB2

```
#cd /usr/iop/current/kafka-broker/
ls

#bin/kafka-topics.sh --list --zookeeper
localhost:2181

#bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 1
--partitions 1 --topic test_topic

#bin/kafka-topics.sh --list --zookeeper
localhost:2181
```

```
#bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 1
--partitions 1 --config
max.message.bytes=64000 --topic
new_delete_this_topic


#bin/kafka-topics.sh --alter --zookeeper
localhost:2181 --topic delete_this_topic
-delete-config max.message.bytes

#bin/kafka-console-producer.sh --broker-list
abeta42.localdomain:6667 --topic test_topic
This is message 1
This is message 2

#cd /usr.iop/current/kafka-broker/

#bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic test_topic
--from-beginning

#cd /usr/iop/current/kafka-broker/

#bin /kafka-topics.sh --describe --zookeeper
localhost:2181 --topic tst_topic

#pwd -> /usr/iop/current/kafka-broker
```

#cd config

#cat consumer.properties

#bin/kafka-console-consumer.sh --zookeeper localhost:2181 -topic test_to_topic --from-beginning --consumer.config config/consumer.properties.

#bin/kafka-consumer-groups.sh --describe --zookeeper localhost:2181 --group test-consumer-group

KAFKA Producer Client Overview

->Kafka includes a new producer client
->Java API easy to work with.
->Variety of other source producers also available
   ->Python
   ->C++
   ->C#
->New Java producer replaces older Scala producer client (V0.8.2)
->Use the API to code your own producer logic

->Thread Safe producer sends data directly to

the broker that is the leader for the partition
   ->Maps messages to topic partition
   ->Sends produce requests to leader of that partition

->Client controls which partition it publishes messages to
   ->Hash partitioning if user specifies key to partition by
  ->Random load balancing if no key provided (round-robin)

->Batching is big driver of efficiency
  ->Producer will attempt to accumulate data in memory and to send out larger batched in a single request
  ->Batching is configurable.
  ->Trades off small amount of latency for better throughput

Producer configuration Settings

Variety of producer configuration settings
->bootstrap.servers
  ->List of host/port pairs used for establishing initial connection to kafka cluster

->client.id

->id string passed to the server when making requests

->Makes tracking source of requests easier for logging vs using ip/port

->acks

->Number of acknowledgments producer requires leader to have received before considering a request complete - Controls durability of sent records

- acks=0 : producer will not wait for any acknowledgement from the server at all
- acks=1 : leader writes record to local log but responds without awaiting full acknowledgement from all followers
- acks=all : leader waits for full set of in-sync replicas (ISR) to acknowledge record

->retries

->Value >0 causes to resend any record whose send fails with a potentially transient error

->Allowing retrieves will potentially change the ordering of records

->batch.size

->Producer attempts to collect messages into batches.
->Increases throughput

->buffer.memory
->Limit total memory used to store each batch

->linger.ms
->Have producer delay sending to give more time for batches to be filled

->compression.type
->Compression type for all data generated by the producer
->Vaild values: none (default), gzip , snappy, or lz4.
->Compression is of full batches of data

->key.serializer and value.serializer
->How to turn key & value objects user provides with ProducerRecord into bytes.

Example of Configuration code:

```
Properties props = new Properties();
```

```
props.put(“bootstrap.servers”,”localhost:9092
”);
props.put(“acks”,”all”);

props.put(“retries”,0);

props.put(“batch.size”,16384);

props.put(“linger.ms”,1);

props.put(“buffer.memory”,33554432);

props.put(“key.serializer”,”org.apache.kafka.
common.serialization.StringSerializer”);

props.put(“value.serializer”,”org.apache.kafk
a.common.serialization.StringSerializer”);
```

KafkaProducer Java Class Overview:

->Class KafkaProducer<K,V>
->Kafka Client that publishes records to the Kafka Cluster
->Producer is thread safe.
->Producer consist of :
   ->Pool of buffer space that holds records that haven’t yet been transmitted to server.

->Background I/O thread responsible for turning records into requests and transmitting them to cluster

Methods:

close();
close(long timeout,java.util.concurrent.TimeUnit timeUnit);
flush();
metrics();
partitionsFor(java.lang.Stringtopic)
send(ProducerRecord<K,V>record)
send(ProducerRecord<K,V>record , Callback callback)

KafkaProducer constructor:

->KafkaProducer(java.util.Map<java.lang.String,java.lang.Object> configs)

-> A producer is instantiated by providing a set of key-value pairs as configuration.

->KafkaProducer(java.util.Map<java.lang.String,java.lang.Object> configs, Serializer<K> keySerializer , Serializer<V>

valueSerializer)

-> A producer is instantiated by providing a set of key-value pairs as configuration.a key & a value serializer

->KafkaProducer(java.util.Properties properties)

-> A producer is instantiated by providing a set of key-value pairs as configuration.

->KafkaProducer(jav.util.Properties properties , Serialzer<K> keySerializer , Serializer<V> valueSerializer)

-> A producer is instantiated by providing a set of key-value pairs as configuration , a key & a value Serializer.

Example KafkaProducer constructor code:

```
Properties props = new Properties();
props.put(“bootstrap.servers”,”localhost:9092
”);
props.put(“acks”,”all”);
props.put(“key.serializer”,”org.kafka.common.
serialization.StringSeriliazer”);
props.put(“value.serailizer”,”org.apache.kafk
```

```
a.common.seriliazation.StringSeriliazer”);


KafkaProducer<String, String> producer
 = new KafkaProducer<String,String(props)>;
```

KafkaProducer send method:

-> public
java.util.concurrent.Future<RecordMetadata> send
(ProducerRecord<K,V> record , Callback callback)

-> All writes are asynchronous by default.
    ->Method returns immediately once record has been stored in a buffer of records waiting to be sent.
    ->Allows sending many records in parallel without blocking to wait for a response after each one.

-> Send() API returns a Future for RecordMetadata which can be polled to get result of the send

    ->Invokes the provided callback when the send has been acknowledged.
    ->Result of send is RecordMetadata specifying partition & offset.

->Invoking get() on future, blocks until associated request completes

->Then return metadata for record or throws any exception that occurred while sending the record.
->For synchronouswrites , use get() immediately to simulate simple blocking call

ProducerRecord and RecordMetadata

Class ProducerRecord

ProducerRecord(java.util.String topic , java.lang.Integer partition, K key , V value)

->A key/value pair(message) to be sent to kafka.
->Consists of topic name to which record is being sent , optional partition number , and optional key ,and the message value.
->If valid partition number is specified - that partition is used when sending a record.
->If no partition specified but key is present , partition chosen using hash of key
->If neither key nor partition present a partition assigned in round-robin fashion.

Class RecordMetadata

->The metadata for a record that has been acknowledged by the server

->Send() API returns a Future for this RecordMetaData

->Can use this to get offset , topic , partition

Asynchronous KafkaProducer send example:

```
ProducerRecord<byte[],byte[]> myRecord = new
ProducerRecord<byte[],byte[]>("mytopic",key,value);


producer.send(myRecord , new Callback()
 {
   public void onCompletion(RecordMetadata metadata
,Exception e)
          {
      if(e!=null)
        e.printStackTrace();
        System.out.pritln("Offset of sent record:"
+metadata.offset());
          }

});
```

Kafka Consumer Client Overview

->Kafka includes a new consumer client that consumes records from the Kafka cluster.

->Java API easy to work with:
   Class kafkacConsumer<k,V>

->Variety of other source producers also available
   ->Python
   ->C++
   ->C#

->Use the API to code your own consumer logic.

->Transparently handles failure of servers in kafka cluster

->Interacts with the server to allow groups of consumers to load balance consumption using consumer groups.

->Maintains TCP connections to the necessary brokers to fetch data.

->You can let consumer automatically handle much of the fine-grained details (like prior “high-level” consumer) OR you can manage lower-level features (like prior “simple” consumer)

General consumer application flow

->Create consumer configuration & use to create new consumer instance.

->Subscribe to topics/partitions

->Poll the topic(s) from within a loop to fetch messages

->Do something useful with the messages consumed

->Commit offsets(automatically or manually)

->Close the consumer.

Group coordination

->Consumers use group coordination protocol built into Kafka (prior version used zookeeper)

->For each group, one of the brokers is selected as the group coordinator.

->Coordinator responsible for managing the state of the group.

->Act of reassigning partitions is called rebalancing the group.

->Rebalancing is process where group of consumer instances within a consumer group, Coordinate to own a mutually exclusive set og partitions of topics that the group has subscribed to.

->Coordinator monitors the heartbeat of group members to ensure they are alive.

Manual vs Automatic Offset committing

->Consumer can automatically commit offsets, or you can manually commit

->Config **enable.auto.commit**

->enablr.auto.commit=true:
Offsets are committed automatically with frequency controlled by auto.commit.interval.msconfig

->enable.auto.commit=false :
Manually offset commits

->Committing offsets manually allows you to perform processing logic and only commit once processing is completed.

->For Example: batch a quantity of records & then submit to a database in a single transaction. Commit after the transaction is complete.

->Provides “at-least-once delivery” guarantees

->If auto-committing in this scenario,could commit after messages consumed but before processing logic completes resulting in lost messages and at-most-once delivery guarantee.

->Manually commit offsets with commitSync() and commitAsync() methods

Consumer configuration settings

Variety of consumers configuration settings

->bootstrap.servers

->List of host/port pairs used for establishing initial connection to kafka cluster

->client.id

->id string passed to the server when making requests

->group.id

->Unique string that identifies consumer groups that consumer belongs to

->key.deserializer and value.deserializer

->How to deserialize message keys and values

->fetch.min.bytes

->Minimum amount of data the server should return for a fetch request

->1 byte by default - means fetch request answered as soon as single byte of data is available

->enable.auto.commit

->Set to true for automaticcommits

->auto.commit.interval.ms

->The frequency in milliseconds that the consumer offsets are auto-committed to kafka if enable.auto.commit is set to true.

->session.timeout.ms

->Timeout used to detect failures of a consumer

->Kafka group coordinator expects to receive a heartbeat from consumers within this time period.

->heartbeat.interval.ms

->Expected time between heartbeats to the consumer coordinator

->Heartbeats used to ensure consumers' sessions stay active & to facilitate rebalancing when new consumers join or leave the group.

Configuration CODE example

```
Properties props = new Properties();
props.put(“bootstrap.servers”,”localhost:9092
”);
props.put(“group.id”,”testgroup”);
props.put(“enable.auto.commit”,”true”);
props.put(“auto.commit.interval.ms”,”1000”);
props.put(“session.timeout.ms”,”30000”);
```

```
props.put(“key.deserializer”,”org.apache.kafk
a.common.serialization.StringDeserailzer”);

props.put(“value.deserializer”,”org.apache.ka
fka.common.serilization.StringDeserializer”);
```

KafkaConsumer Java Class Overview:

KafkaConsumer constructor

**->KafkaConsumer(java.util.Map<java.lang.Strin
g, java.lang.Object> configs)**
->Consumer instantiated by providing set of key-value pairs as configuration

**->KafkaConsumer(java.util.Map<java.lang.Strin
g, java.lang.Object> configs, Deserailzer<K>
keyDeserializer , Deserializer<V>
valueDeserializer)**
->Consumer instantiated by providing set of key-value pairs as configuration ,a ConsumerRebalanceListener implementation , a key & value Deserializer

**->KafkaConsumer(java.util.properties
properties)**
->Consumer instantiated by providing object as configuration

->KafkaConsumer(java.util.Properties properties, Deserializer<K> keyDeserializer , Deserializer)

->Consumer instantiated by providing Properties as objects as configuration and a ConsumerRebalanceListener implementation ,a key & value Deserializer.

Subscribing to Topics

->Subscribe to the given list of topics to get dynamically assigned partitions
->Can subscribe with a list of topics or a regex pattern.
->Kafka assigns fair share of partitions from these topics to consumers
->Topics subscriptions are not incremental
->List replaces the current assignment (if there is one)

```
KafkaConsumer<String,String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList(“topic1”,”topic2”));
```

Manual partition assignment

->can alternatively subscribe to specific partitions.
->Consumer will just get the partitions it subscribes to
->If consumer instance fails no attempt is made to rebalance partitions

```
TopicPartition partition0 = new TopicPartition(“mytopic”,0);

TopicPartition partition1 = new TopicPartition(“mytopic”,1);

consumer.assign(Arrays.asList(partition0,partition1));
```

Polling
----

public ConsumerRecords<K,V> poll(long timeout)
 -> Use poll to fetch messages that consumers is subscribed to
 ->timeout

->Time in millis ,spent waiting in poll if data is not available

->If 0 , returns immediately with any records that are available now

->Returns map of records since last fetch for subscribed list of topics and partitions

->Meant to be run in an event loop

```
try{
    while(running)
            {
        ConsumerRecords<String,String> records
= consumer.poll(100);
    //Process the records…
  }
}
```

Close the Consumer

---

public void close()

->Use to close the consumer

->Waits indefinitely for any needed cleanup

->If auto-commit enabled calling close() will commit current offsets’

->Always call close()

  ->Cleans up any sockets in use
  ->Ensures consumer can alert coordinator about its departure from the group

```
try{
    while(running){
   ConsumerRecords <String,String> records = consumer.poll(100);
 }
}finally{
    consumer.close();
 }
```

Storing offsets outside of kafka & controlling consumer’s position

->Can choose to store offsets outside of kafka

->Allows for fully atomic consumption and “exactly-one” semantics
   ->Stronger than “at-least-once” semantics that come with kafka offset commit functionality

->For Ex. could store offsets along with processed data at same time in a database transaction
  ->Either transaction succeeds and offset updated or transaction fails and neither offset nor processed data is stored
 ->seek(TopicPartition partition ,long offset)
 ->seekToBeginning(TopicPartition .. partitions)
 ->seekToEnd(TopicPartition ...partitions)

For Example
,could use seek to re-consume messages if transaction failed.

javac -d MyKafkaConsumer.java ->
java MyKafkaConsumer

#pwd

```
#bin/kafka-console-producer.sh --broker-list
abeta42.localdomain:6667 --topic test_topic
This is a test 1

#try with manual offsets
```

Kafka Connect & Spark streaming

Kafka Connect

->Adopting Kafka for data integration required significant development skills

->Developing a kafka connector required building on the client API’s

->New(V0.9)Kafka Connect is framework for large scale , real-time stream data integration using kafka.

->Simplifies adoption of connectors for stream data integration

->Makes building and managing stream data integration.

->Encourages development of rich ecosystems of open source connectors

->Deploy Kafka connectors that work well with each other and can be monitored, deployed , and administered in consistent manner

  ->Focusses on the ETL
  ->Kafka must always be on one side of the equation - source or sink.

->Abstracts away common problems every connector to Kafka faces:
  ->Schema management , fault tolerance , partitioning , offset management & delivery semantics & monitoring

Kafka connect features

->A common framework for kafka connectors

->Distributed & standalone modes

->REST interface

->Automatic offset management

->Streaming/batch integration

->Leveraging Kafka's existing capabilities , Kafka Connect great for bridging streaming and batch data systems

Connectors

->Can develop your own connectors or use an already available one

->Some nice pre-built connectors out there

->HDFS Connector

->Exports Data from Kafka topics to HDFS files in variety of formats

->Periodically polls data from Kafka and writes them to HDFS

->Data from each Kafka topic is partitioned and divided into chunks

->Each chunk of data is represented as an HDFS Filewith topic, Kafka partition, start and end offsets of these data chunks in the filename.

JDBC Connector

->Import data from any relational database with JDBC driver into Kafka Topics.

->Data is loaded by periodically executing a SQL Query and creating an output record for each row in the result set.

->The database is monitored for new or deleted tables and adapts automatically.

->FileStream Connector

->Variety of others being developed
  ->MongoDB,Cassandra

Connectors , Tasks and Workers

->Connector instance
   ->Logical job responsible for managing copying of data between Kafka & other systems.
  ->Connector either source or sink connector

->Each Connector can instantiate a set of tasks that actually copy the data.
  ->Ability to break a single job to multiple tasks allows for parallelism.

->All Kafka Connect sources and sinks map to partitioned streams of records

->A Kafka Connect cluster consists of set of Worker Process

->ex-if streams represents a database , each stream partition would represent table in the DB

->Every record in partitioned stream consists a key, a value , and associated offset

->Offsets are tracked by kafka Connect and mark position of every record in the stream partition.

->Similar in concept to kafka offsets , however can be different formats

->For Example offsets for records coming from database might be timestamp column.

->A Kafka Connect cluster consists of set of Worker processes

->Containers that execute Connectors and Tasks.

->Auto coordinate with each other to distribute work and provide scalability and fault tolerance

->Connect has standalone and distributed workers.

Worker Standalone vs Distributed modes

Standalone

->Simplest mode

->Single process responsible for executing all connectors and tasks

->Requires minimal configuration -passwd in via command line

bin/connect-standalone.sh worker.properties

Connector1.properties[connector2.properties.]

Distributed

->Provides scalability and automatic fault tolerance

->Have many worker process share a group.id

->Workers will automatically coordinate to schedule execution of connectors and tasks across all available workers

->Interaction with distributed-mode cluster using REST API

->Example: GET /connectors - returns a list of active connectors

Kafka Connect StandAlone Example

#bin/connect-standaone.sh config/connect-standaolne.properties config/connect-file-source.properties

Kafka & Spark

->Many use cases entail Kafka feeding streaming data into spark streaming

->Newer Spark Streaming/Kafka Direct API (introduced Spark 3.1)

Kafka **=>** Spark Streaming

=>HDFS/Databases/DAshboards