Many Streams Lead to Kafka
Introduction
Founded by creators of Kafka

- @jaykreps, @nehanarkhede, @junrao
- Started at LinkedIn
- Widely used

We help you gather, transport, organize, and analyze all of your stream data

What we offer

- Confluent Platform
  - Kafka **plus** critical bug fixes not yet applied in Apache release
  - Kafka ecosystem projects
  - Enterprise support
- Training and Professional Services
Confluent training partner

Previously:

- Curriculum Developer and Instructor @ Cloudera
- Senior Software Engineer @ Intuit
- Lead Developer @ Dibbs

Covered, Conferences and Published In:

- GigaOM, ArsTecnica, Pragmatic Programmers, Strata, OSCON, Wall Street Journal, CNN, BBC, NPR

See Me On:

- @jessetanderson
- http://tiny.smokinghand.com/linkedin
- http://tiny.smokinghand.com/youtube
Your experience as a developer, analyst or administrator

Which language(s) you use

Experience with Publish/Subscribe, Client/Server, Hadoop, Big Data or NoSQL

Expectations from this class
Start at 9 AM

Breaks and lunch

- Morning break 10:30 AM

Downloads

- Exercise guide (required)
- Slides (recommended)
- VM (required)
Data Movement Concepts
There are many means of storage

Choosing the right one depends on the access pattern

Are you storing archival data?

Are you randomly accessing data?

Is data coming more as an event?
Many data pipelines start out simple

- There was a single place where all data resided
- There was a single ETL process

Many data pipelines grow over time

- New systems are added with data needs
- Each system has to do its own ETL

Each ETL and system starts to deviate

- Codebase
- Data
Data comes in and is copied or streamed to various places

Hadoop Cluster

RDBMS

Real-time Processing

Data is run through an ETL process on each system

Data Source

Data Source

Data Source
Importance Of Decoupling

- **Data Source**
  - Data comes in and is copied or streamed to various places

- **Hadoop Cluster**
  - Hadoop processing requires real-time data

- **RDBMS**
  - Real-time processing requires update RDBMS data

- **Real-time Processing**

- **BI Analytics**

  - Adding a fourth system adds another separate ETL process
A publish subscribe pattern is used to decouple systems

Data is published or sent to another system
- Called a publisher

Data is consumed by another system
- Called a subscriber or consumer

The publisher and consumer don't know anything about each other
- They follow a contract about the data format

Copyright © 2015 Confluent, Inc. All rights Reserved
1. Data is published to cluster

2. Real-time processing performs the ETL work and publishes the ETL'd data

3. Hadoop and RDBMS retrieve ETL'd data.

Publisher Subscriber

Data Source

Data Source

Data Source

Pub/Sub Cluster

Hadoop Cluster

RDBMS

Real-time Processing

BI Analytics
The ETL'd data acts a single source of truth. Data is only processed once and consumed many times.

Pub/Sub Cluster

Loose coupling makes it easy to add a new consumer

Hadoop Cluster

RDBMS

Real-time Processing

BI Analytics
Kafka System
Kafka is a distributed publish subscribe system

It uses a commit log to track changes

Kafka was originally created at LinkedIn

- Open sourced in 2011
- Graduated to a top-level Apache project in 2012

Many Big Data projects are open source implementations of closed source products

- Unlike Hadoop, HBase or Cassandra, Kafka actually isn’t a clone of an existing closed source product
The same codebase being used for years at LinkedIn answers the questions:

- Does it scale?
- Is it fast?
- Is it robust?
- Is it production ready?

Kafka supports the traditional publish/subscribe features

It has other features aimed at Big Data

- Kafka scales by partitioning the data
- Failovers are automated
- Data can be consumed in batch and real-time
We will now demonstrate how Kafka works with Legos

Concepts shown:

- Publish/Subscribe
- Topics
- Partitioning
- Commit Logs
- Log compaction
Producers publish or send the data to the Kafka cluster.

All Producer data is sent across the network to the Kafka cluster.

All data is sent as keys and values.

Publisher

Consumer

Consumer
The Consumer pulls data from the Kafka Cluster

Consumers receive or consume data from the Kafka cluster

All data is received as keys and values
Producers send all data on topics. A Producer may produce data to multiple topics.

Kafka Cluster

All data is sent and received with topics

Consumers receive all data on topics.
All data send from Producers is in the form of key/value pairs

All data received by Consumers is in the form of key/value pairs
Keys and values in Kafka can be Strings or byte arrays

Avro is a serialization format used extensively with Kafka and Big Data

- Avro can be serialized to byte arrays

Kafka uses a Schema Registry to keep track of Avro schemas

- Verifies that the correct schemas are being used
- Publisher and consumers need to use compatible versions of the schema
Kafka can be accessed pragmatically using various technologies and languages

The primary, real-time access is via the Java API.

The REST interface allows access from many languages and programs.

There are several interfaces for bringing data into Hadoop:

- Camus is a project for offloading Kafka data into Hadoop.
Commit Logs
Commit logs are a way to keep track of changes as they happen.

They are commonly used with databases to keep track of changes.

Kafka uses commit logs to keep track of changes for a topic.

Consumers can use the commit logs to retrieve previous data.
Producers don’t need to worry about overwriting another Producer’s data

Kafka cluster uses a commit log to order data from Producers

Commit log maintains order at high concurrency

Commit Log

Producer 1 K/V
Producer 3 K/V
Producer 2 K/V
A Consumer falling behind will not affect the Producers

Kafka Cluster

The Commit log maintains the order of the data. That allows the Consumer to read in the same order.

Commit Log

A Consumer will continue pulling data until it reaches the last entry

Consumer

Producer 1 KV
Producer 2 KV
Producer 3 KV
The Kafka Cluster maintains knowledge of where the Consumer stopped reading

Commit Log

A Consumer may go down due to maintenance or run as a batch program

Consumer

The Consumer will start at the last read K/V and continue until the latest K/V

Kafka Cluster

Last read K/V

Most current K/V
Kafka Architecture
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>A program that sends messages to Kafka</td>
</tr>
<tr>
<td>Topic</td>
<td>A way to group similar messages</td>
</tr>
<tr>
<td>Key/Value Pair</td>
<td>The form a message takes</td>
</tr>
<tr>
<td>Consumer</td>
<td>A program that pulls messages from Kafka</td>
</tr>
<tr>
<td>Consumer Group</td>
<td>A group of Consumers that allows for scaling and high availability</td>
</tr>
<tr>
<td>Offset</td>
<td>A logical identifier of a message within a partition</td>
</tr>
<tr>
<td>Broker</td>
<td>The daemon responsible for sending, receiving and saving data</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>A system for distributed coordination and service discovery</td>
</tr>
<tr>
<td>Partition</td>
<td>A smaller unit of a topic</td>
</tr>
<tr>
<td>Replica</td>
<td>Saving a partition’s data on more than one node for durability</td>
</tr>
</tbody>
</table>
Brokers make up the core of the Kafka Cluster

Each Broker is responsible for receiving, sending, and saving the log data.
Each Producer sends data on a topic

Each topic is further broken into partitions

Producer

Broker

Consumer

Producer

Broker

Consumer

Broker is responsible for a subset of partitions within a topic
Each partition has a physically separate commit log stored on the broker.

Broker

Commit Log
- Last read K/V
- Most current K/V

Commit Log
- Last read K/V
- Most current K/V
Each partition can be replicated in its entirety on another broker.
ZooKeeper is a system for distributed coordination and service discovery

- Is highly-available

ZooKeeper Features

- Distributed coordination
- Distributed queues
- Distributed locks
- Discovery service
- Leader election
- Watches can be placed
ZooKeeper Use

- **Producer**
- **Broker**
- **Consumer**

ZooKeeper is used extensively by Kafka for high availability.

Brokers use ZooKeeper for leader election.

Brokers use ZooKeeper for failure detection.
Kafka API
Kafka's Java API is the only first class citizen

The Java API abstracts out the communication and connections

- This allows you to focus on writing the code

Data can be passed as \texttt{Strings} or byte arrays

Programs need to configure where ZooKeeper and Brokers are running
REST is a way of using HTTP to perform actions

The REST calls are translated to Kafka calls

Allows many non-Java languages to access Kafka
Other ecosystem Kafka clients exist

- These projects are not currently supported by Confluent

They often implement Kafka's line protocol

Your mileage may vary using these projects

- Project may not implement the latest protocol

These projects give native language bindings for Kafka
The Java API uses the `KafkaProducer` object

- Uses a `ProducerRecord` for the key and value

The REST API uses a `POST` to the `topics` URL

- Uses a base64 encoded JSON call for the key and value
The package names will be inconsistent

```java
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

Properties props = new Properties();
// Configure brokers to connect to
props.put("bootstrap.servers", "broker1:9092");
// Configure serializer classes
props.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

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

// Create ProducerRecord and send it
String key = "mykey";
String value = "myvalue";
ProducerRecord<String, String> record = new
    ProducerRecord<String, String>("my_topic", key, value);
producer.send(record);

producer.close();
```
#!/usr/bin/python

import requests
import base64
import json

url = "http://kafkarest1:8082/topics/my_topic"

headers = {
    "Content-Type": "application/vnd.kafka.binary.v1+json"
    }

# Output messages
payload = {"records":
    [{
        "key":base64.b64encode("firstkey"),
        "value":base64.b64encode("firstvalue")
    }]
    }

# Send the message
r = requests.post(url, data=json.dumps(payload), headers=headers)

if r.status_code != 200:
    print "Status Code: " + str(r.status_code)
print r.text
The Java API uses the `ConsumerConnector` object

- Uses a `KafkaStream` and `ConsumerIterator` for the key and value

The REST API uses a `GET` to the `topics` URL

- Gets a base64 encoded JSON for the key and value
- The keys and values must be base64 decoded
The package names will be inconsistent

```java
import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
import kafka.message.MessageAndMetadata;

String topic = "my_topic";
Properties props = new Properties();
// Configure ZooKeeper location
props.put("zookeeper.connect", "zookeeper1");
// Configure consumer group
props.put("group.id", "group1");

// Use the configuration to create the ConsumerConnector
ConsumerConfig consumerConfig = new ConsumerConfig(props);

// Create ConsumerConnector with createJavaConsumerConnector
ConsumerConnector consumerConnector = Consumer
    .createJavaConsumerConnector(consumerConfig);
```
// Create a map of topics we are interested in with the number of
// streams (usually threads) to service the topic
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put(topic, 1);

// Get the list of streams and configure it to use Strings
Map<String, List<KafkaStream<String, String>>> consumerMap =
    consumerConnector
        .createMessageStreams(topicCountMap, new StringDecoder(null),
                             new StringDecoder(null));

// Get the stream for the topic we want to consume
KafkaStream<String, String> stream = consumerMap.get(topic).get(0);

// Iterate through all of the messages in the stream
ConsumerIterator<String, String> it = stream.iterator();

// Note this should done with threads as this is a blocking call
while (it.hasNext()) {
    MessageAndMetadata<String, String> messageAndMetadata = it.next();

    String key = messageAndMetadata.key();
    String value = messageAndMetadata.message();

    // Do something with message
}
```python
#!/usr/bin/python

import requests
import base64
import json
import sys

# Base URL for interacting with REST server
baseurl = "http://kafkaest1:8082/consumers/group1"

# Create the consumer instance
print "Creating consumer instance"

payload = {
    "format": "binary"
}

headers = {
    "Content-Type": "application/vnd.kafka.v1+json"
}

r = requests.post(baseurl, data=json.dumps(payload), headers=headers)

if r.status_code != 200:
    print "Status Code: " + str(r.status_code)
    print r.text
    sys.exit("Error thrown while creating consumer")

# Base URI is used to identify the consumer instance
base_uri = r.json()("base_uri")
```
# Get the message(s) from the consumer
headers = {
    "Accept": "application/vnd.kafka.binary.v1+json"
}

# Request messages for the instance on the topic
r = requests.get(base_uri + " /topics/my_topic", headers=headers, timeout=20)

if r.status_code != 200:
    print "Status Code: " + str(r.status_code)
    print r.text
    sys.exit("Error thrown while getting message")

# Output all messages
for message in r.json():
    if message["key"] is not None:
        print "Message Key:" + base64.b64decode(message["key"])

        print "Message Value:" + base64.b64decode(message["value"])

# Delete the consumer now that we've sent the messages
headers = {
    "Accept": "application/vnd.kafka.v1+json"
}

r = requests.delete(base_uri, headers=headers)

if r.status_code != 204:
    print "Status Code: " + str(r.status_code)
    print r.text
Data is sent to the Broker responsible for handling that data

Choosing the Broker is based on the key

- When there is no key, data is sent round robin

Programs can choose the partition to send to:

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

// Get the list of partitions to decide which one to send to
List<PartitionInfo> partitionsList =
    producer.partitionsFor("my_topic");

String key = "mykey";
String value = "myvalue";
// Add the partition number when creating the ProducerRecord
ProducerRecord<String, String> record = new 
ProducerRecord<String, String>(
    "my_topic", 0, key, value);
```
Topics are automatically created when a message is sent to them

The topic is created with all defaults

- Data is only stored on one Broker

To create a topic with configuration:

```bash
$ kafka-topics --zookeeper zk_host:port/chroot --create \
--topic my_topic_name --partitions 20 --replication-factor 3
```
Message Ordering

The broker determines the ordering of messages from different clients.

- **Producer**: Messages from the same Producer thread to the same partition will be delivered to the consumer in send order.

- **Consumer**: Messages from different partitions are not delivered in send order.

- Systems that need global ordering will need to include timestamp or other information.
ProducerRecord<String, String> record = new
    ProducerRecord<String, String>(
        "my_topic", key, value);

producer.send(record, new CustomCallback(record));

class CustomCallback implements Callback {
    ProducerRecord<String, String> record;

    public CustomCallback(ProducerRecord<String, String> record) {
        super();
        this.record = record;
    }

    @Override
    public void onCompletion(RecordMetadata metadata, Exception exception) {
        if (exception != null) {
            // There was an error sending the ProducerRecord
            exception.printStackTrace();

            // Handle exception in some way
            System.out.println("Error sending message:" + record.value());
        } else {
            // The send was successful
            System.out.println("Record offset:" + metadata.offset());
        }
    }
}
In this exercise you will use the Kafka command line utilities

Time: 15 minutes
In this exercise you will write a Hello World program with the Kafka API

Time: 15 minutes
Advanced Consumers
Kafka is more than a simple Publish/Subscribe system

The Kafka Cluster maintains knowledge of where the Consumer stopped reading

Commit Log
- Last read K/V
- Most current K/V

A Consumer may go down due to maintenance or run as a batch program

Consumer

The Consumer will start at the last read K/V and continue until the latest K/V
The API allows advanced Consumers to interact with the commit log

Offsets are committed via the API

- Offsets are committed automatically by default
- Offsets can be programmatically committed

Reading previous log data requires configuration changes in the API
Properties props = new Properties();
// Configure ZooKeeper location
props.put("zookeeper.connect", "zookeeper1");
// Configure consumer group
props.put("group.id", String.valueOf(new Random().nextInt()));
// Always start from beginning
props.put("auto.offset.reset", "smallest");

ConsumerIterator<String, String> it = stream.iterator();

// Note this should done with threads as this is a blocking call
while (it.hasNext()) {
    // Messages will arrive from the beginning of the log
}
payload = {
    "format": "binary",
    "auto.offset.reset": "smallest"
}

headers = {
    "Content-Type": "application/vnd.kafka.v1+json"
}

r = requests.post(baseurl, data=json.dumps(payload), headers=headers)

if r.status_code != 200:
    print("Status Code: " + str(r.status_code))
    print(r.text)
    sys.exit("Error thrown while creating consumer")

# Output all messages
for message in r.json():
    # Messages start from the beginning
By default, Consumers read from the end of the log

To specify, this configuration set `auto.offset.reset` to `largest`
Properties props = new Properties();
// Configure ZooKeeper location
props.put("zookeeper.connect", "zookeeper1");
// Configure consumer group
props.put("group.id", "manualcommitgroup");
// Always start from beginning
props.put("auto.offset.reset", "smallest");
// Manually/Programmatically commit offset
props.put("auto.commit.enable", "false");
// Configure to use Kafka for commits
props.put("offsets.storage", "kafka");
props.put("dual.commit.enabled", "false");

// Note this should done with threads as this is a blocking call
while (it.hasNext()) {
    // Do something with message

    if (shouldCommit()) {
        // Commit the offsets
        consumerConnector.commitOffsets();
    }
}
The commit will use the offset of the last message sent in the JSON response
Kafka has another Consumer called the `SimpleConsumer`.

This consumer allows more advanced ways of consuming a topic:

- Allows you to read a message multiple times
- Consume only certain partitions
- Manage transactions to ensure a message is processed only once

You will need to handle more exceptions and offsets.
Avro and Kafka
Properties props = new Properties();
// Configure brokers to connect to
props.put("bootstrap.servers", "broker1:9092");
// Configure serializer classes
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
          io.confluent.kafka.serializers.KafkaAvroSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
          io.confluent.kafka.serializers.KafkaAvroSerializer.class);
// Configure schema repository server
props.put("schema.registry.url", "http://schemaregistry1:8081");

// Create the producer expecting Avro objects
KafkaProducer<Object, Object> avroProducer = new
   KafkaProducer<Object, Object>(
      props);

// Create the Avro objects for the key and value
CardSuit suit = new CardSuit("spades");
SimpleCard card = new SimpleCard("spades", "ace");

// Create the ProducerRecord with the Avro objects and send them
ProducerRecord<Object, Object> record = new
   ProducerRecord<Object, Object>(
      "my_avro_topic", suit, card);

avroProducer.send(record);
# Read in the Avro files
key_schema = open("my_key.avsc", 'rU').read()
value_schema = open("my_value.avsc", 'rU').read()

producer_url = "http://kafkarest1:8082/topics/my_avro_topic"

headers = {
    "Content-Type": "application/vnd.kafka.avro.v1+json"
}

payload = {
    "key_schema": key_schema,
    "value_schema": value_schema,
    "records":
    [{
        "key": {"suit": "spades"},
        "value": {"suit": "spades", "card": "ace"}
    }]
}

# Send the message
r = requests.post(producer_url, data=json.dumps(payload), headers=headers)

if r.status_code != 200:
    print "Status Code: " + str(r.status_code)
    print r.text
// Initialize and configure the Kafka Avro objects
VerifiableProperties vProps = new VerifiableProperties(props);
KafkaAvroDecoder keyDecoder = new KafkaAvroDecoder(vProps);
KafkaAvroDecoder valueDecoder = new KafkaAvroDecoder(vProps);

Map<String, List<KafkaStream<Object, Object>>> consumerMap =
    consumerConnector
        .createMessageStreams(topicCountMap, keyDecoder, valueDecoder);

KafkaStream<Object, Object> stream = consumerMap.get(topic).get(0);
ConsumerIterator<Object, Object> it = stream.iterator();

while (it.hasNext()) {
    MessageAndMetadata<Object, Object> messageAndMetadata = it.next();

    // Kafka only gives back GenericRecords. This code converts the
    // GenericRecord to a SpecificRecord
    CardSuit suit = (CardSuit) SpecificData.get().deepCopy(
        CardSuit.SCHEMA$, messageAndMetadata.key());
    SimpleCard card = (SimpleCard) SpecificData.get().deepCopy(
        SimpleCard.SCHEMA$, messageAndMetadata.message());

    // Do something with the Avro objects
}
# Get the message(s) from the consumer
headers = {
    "Accept": "application/vnd.kafka.avro.v1+json"
}

# Request messages for the instance on the topic
r = requests.get(base_uri + "\topics/my_avro_topic", headers=headers, timeout=20)

if r.status_code != 200:
    print "Status Code: " + str(r.status_code)
    print r.text
    sys.exit("Error thrown while getting message")

# Output all messages
for message in r.json():
    keysuit = message["key"]["suit"]

    valuesuit = message["value"]["suit"]
    valuecard = message["value"]["card"]

    # Do something with the data
Maintaining the same Avro schema across Producers and Consumers is important

- Avro schemas need a central place for storage

The Schema Registry allows Producers and Consumers to coordinate schema

- Allows newer schemas to be created and registered

The Schema Registry can be used to enforce Avro compatibility rules

- Updated Avro schemas will pass or fail based on the rules
Submitting the Avro schemas with each Producer request may be inefficient

A schema may be submitted once and use a schema ID after that

- The IDs are returned in the JSON reply
- The schema IDs are returned in the JSON reply as `key_schema_id` and `value_schema_id`
Conclusion
Confluent at
www.confluent.io
@nehanarkhede
@jessetanderson