Kafka

Daniel Hinojosa

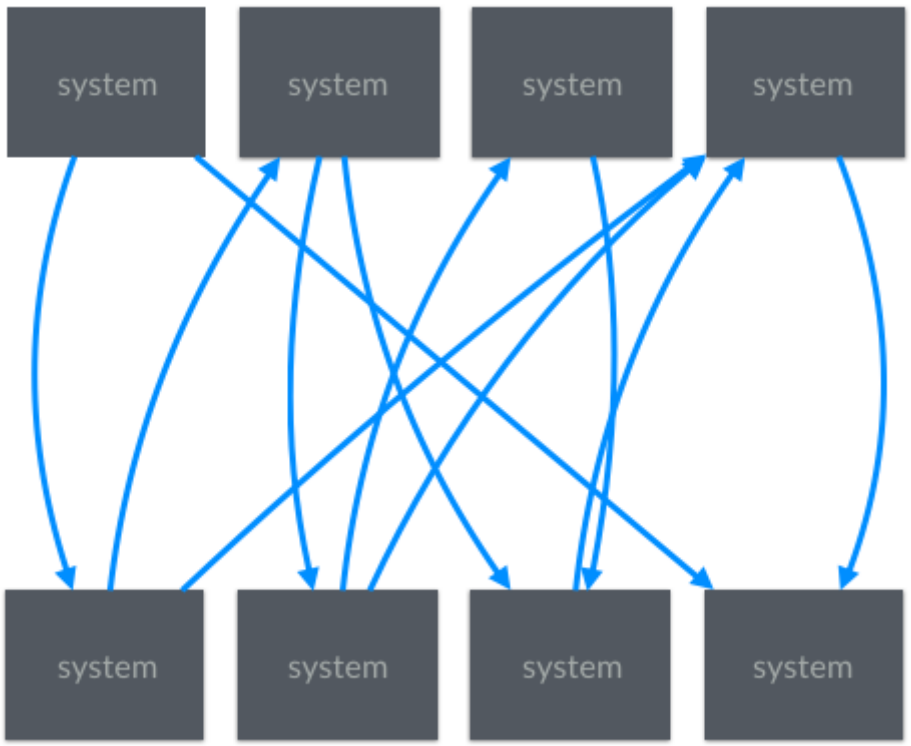
Introduction

About Kafka

- When you have multiple services and they all make connections to one another this can get tedious
- Known as the distributed "git log"
- Ability to replayed in a consistent manner
- Kafka is also stored durable, in order if key provided, and deterministic
- Data can also be distributed for resiliency
- Scalable and Elastic

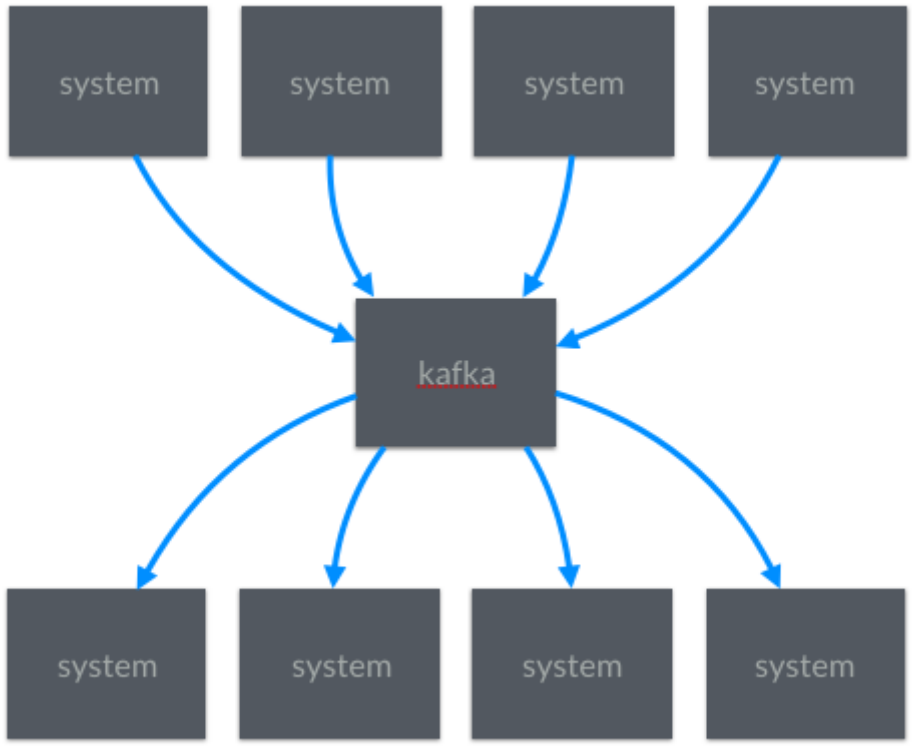
Complicated Architecture

`n * m` Connections

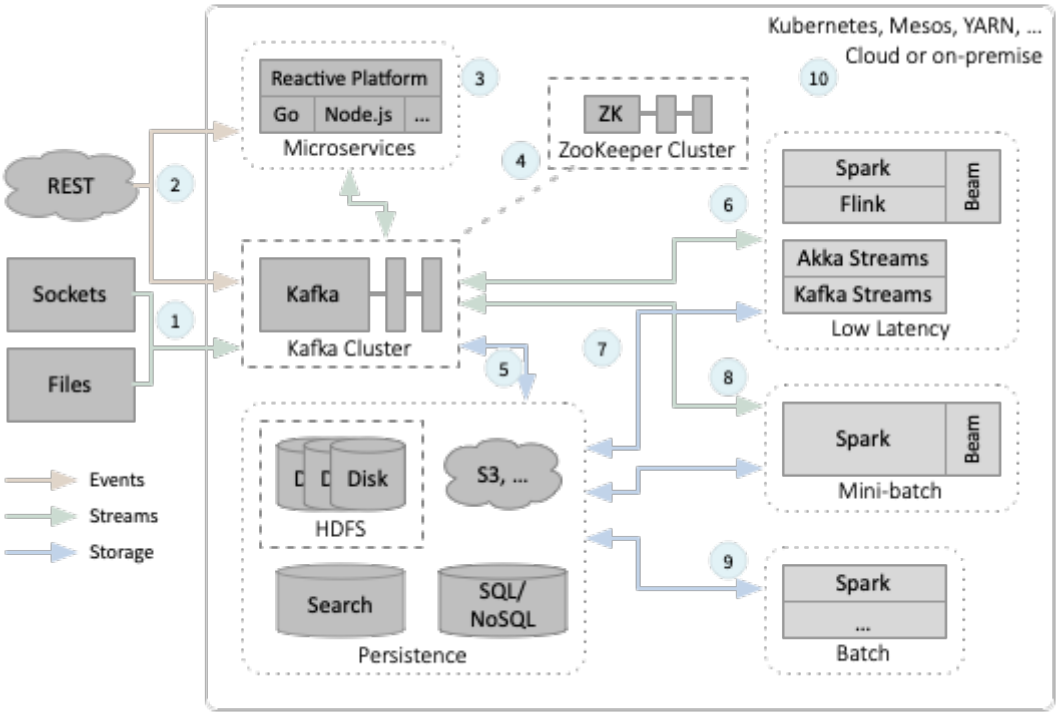


Kafka Backplane Architecutre

`n+m` Connections

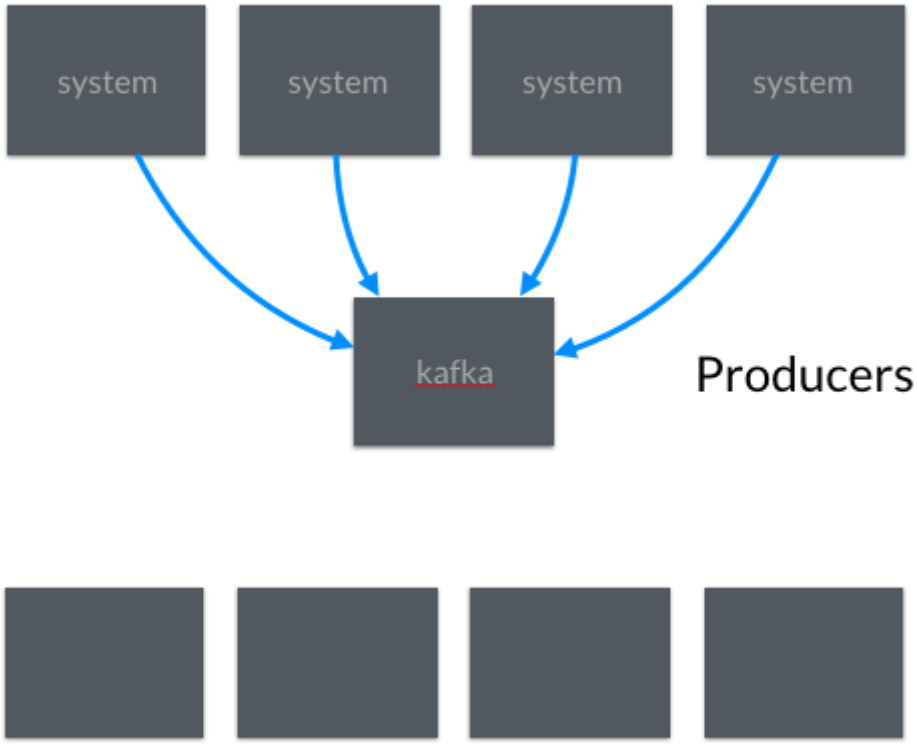


Streaming Architecture

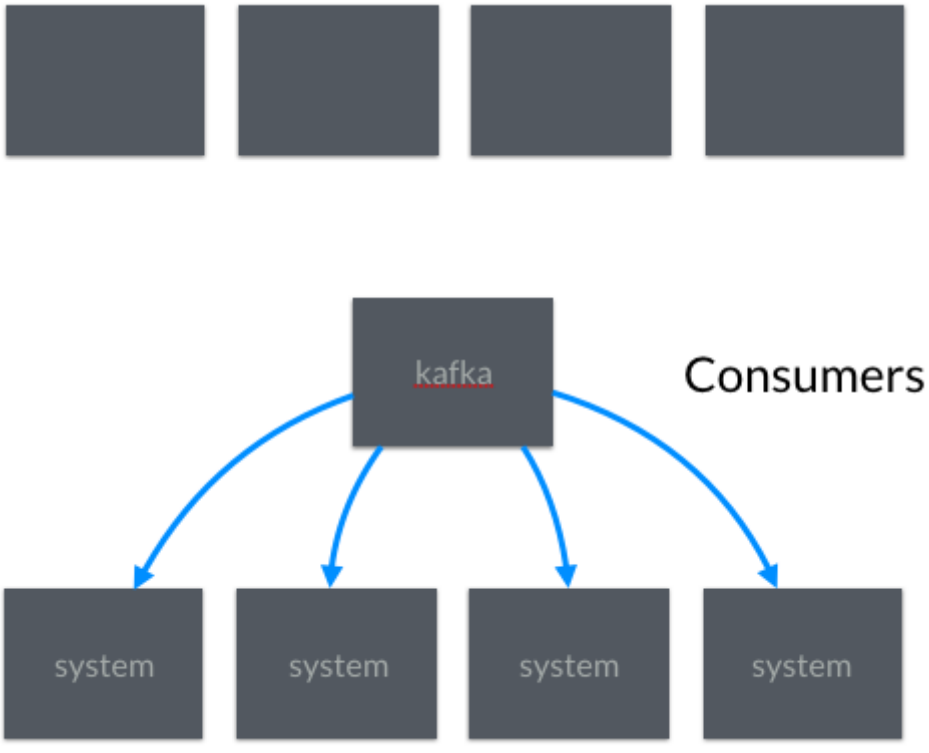


Credit: Dean Wampler

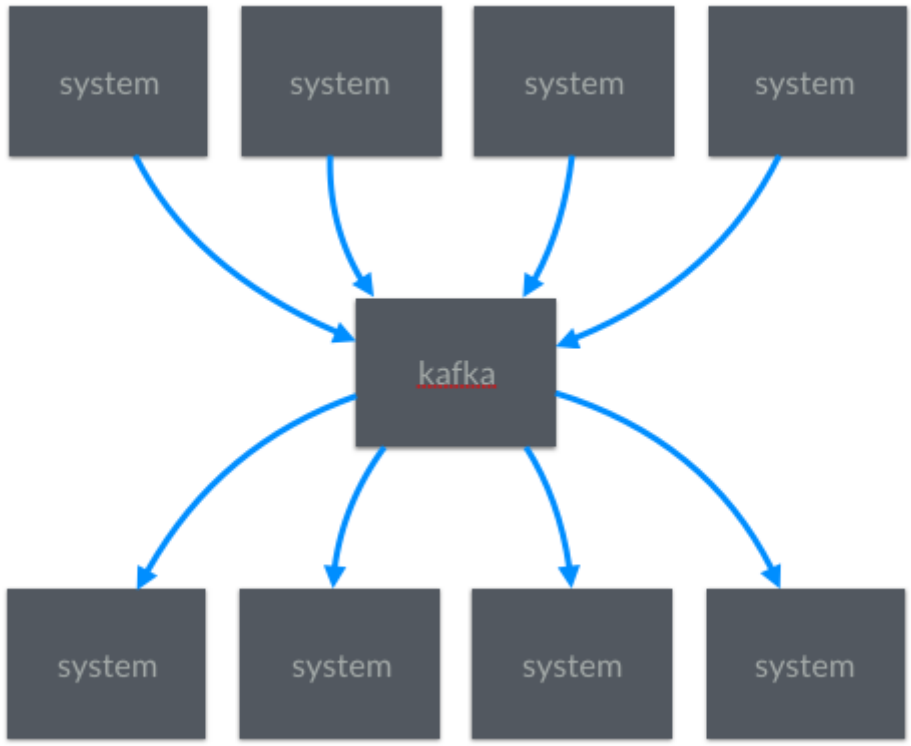
Kafka Producers



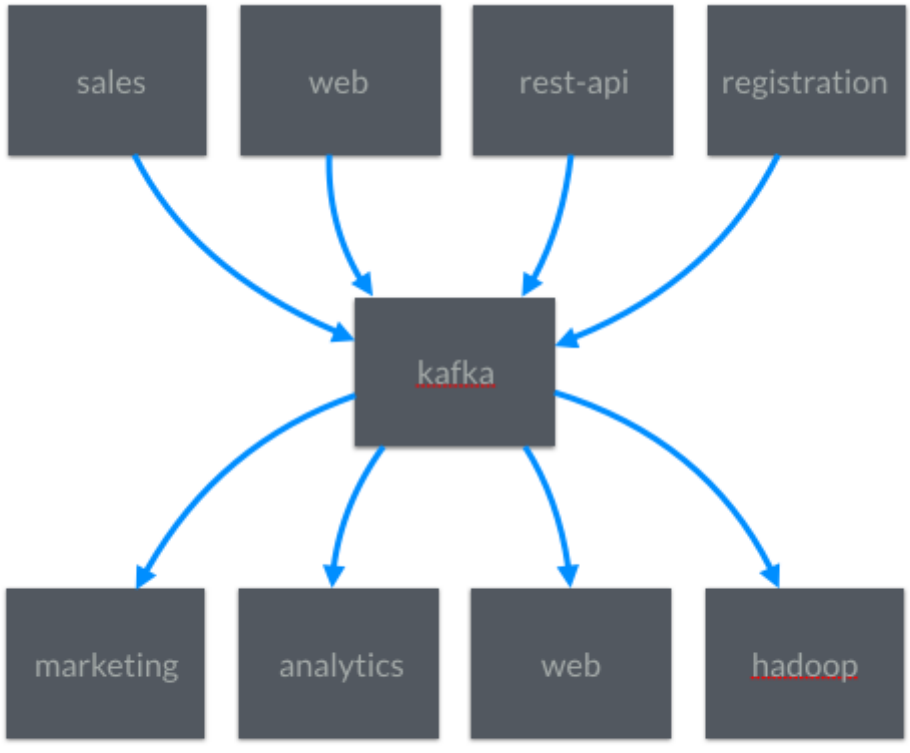
Kafka Consumers



Kafka Producers and Consumers

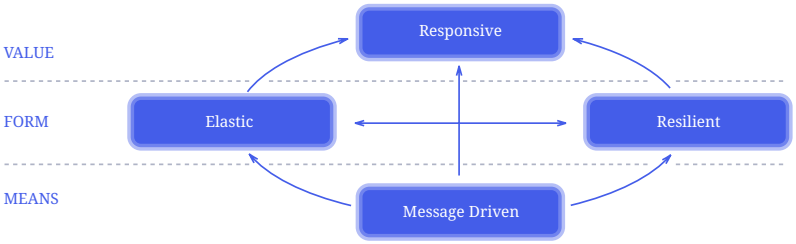


Kafka Producers and Consumers



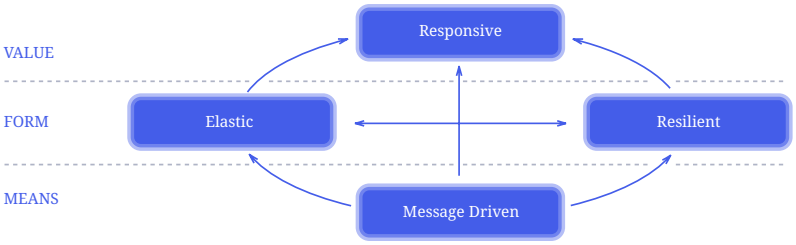
Oversimplied: A Producer can be a Consumer

Reactive Architecture



Reactive Manifesto (http://reactive-manifesto.org)

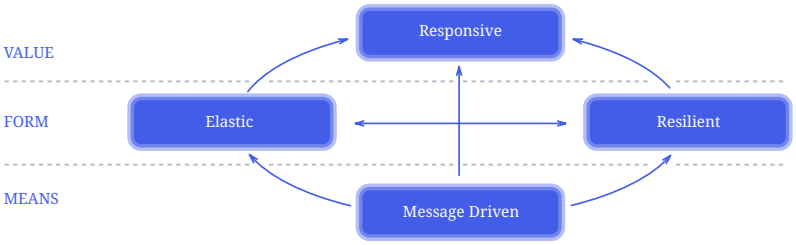
Responsive



- System responds in a timely manner if at all possible.
- Problems may be detected quickly and dealt with effectively.
- Responsive systems focus on providing rapid and consistent response times,
- Establishing reliable upper bounds so they deliver a consistent quality of service.
- Consistent behavior simplifies
  - Error handling
  - Builds end user confidence
  - Encourages further interaction.

Reactive Manifesto (http://reactive-manifesto.org)

Resilient

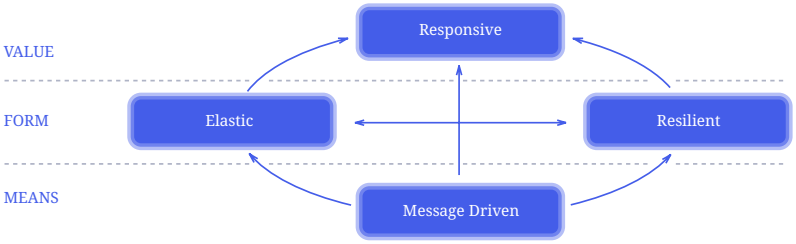


- The system stays responsive in the face of failure.
- Any system that is not resilient will be unresponsive after a failure.
- Achieved by replication, containment, isolation and delegation.

- Failures are contained within each component
- Isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole.
- Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary.
- The client of a component is not burdened with handling its failures.

Reactive Manifesto (http://reactive-manifesto.org)

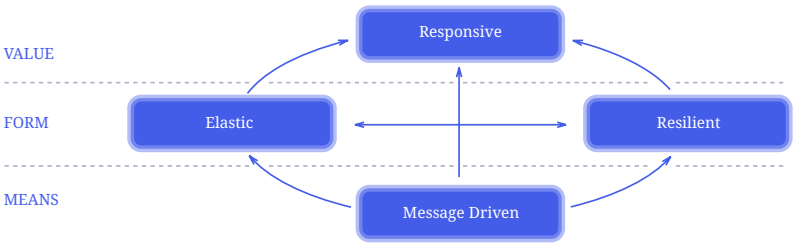
Elastic



- The system stays responsive under varying workload.
- React to changes in the input rate by increasing or decreasing the resources allocated to service these inputs.
- Implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them.
- Reactive Systems support predictive, as well as reactive, scaling algorithms by providing relevant live performance measures.
- They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

Reactive Manifesto (http://reactive-manifesto.org)

Message-Driven

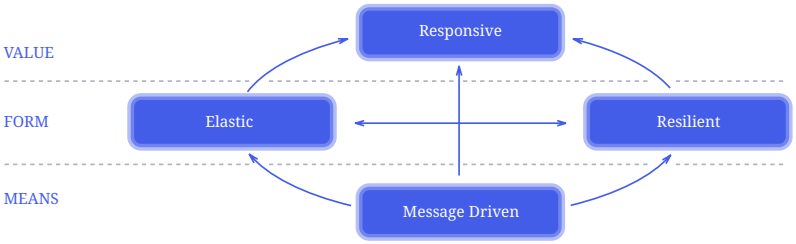


- Rely on asynchronous message-passing to establish a boundary between components that ensures:
  - Loose coupling
  - Isolation
  - Location Transparency.

- This boundary also provides the means to delegate failures as messages.

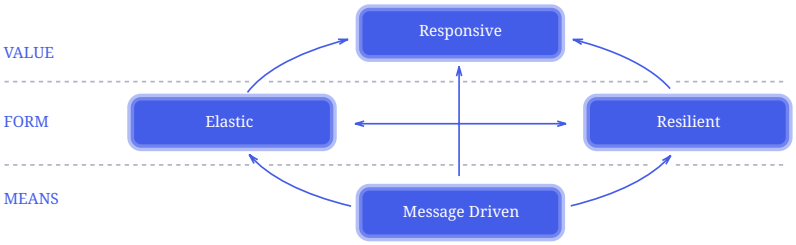
Reactive Manifesto (http://reactive-manifesto.org)

Message-Driven Continued



- Employing explicit message-passing enables:
  - Load management
  - Elasticity
  - Flow control

Message-Driven Flow Control



- Flow control can be used to:
  - Shape and monitor message queues
  - Apply back-pressure when necessary.
- Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host.
- Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

Reactive Manifesto (http://reactive-manifesto.org)

Architecture

The Kafka Players

- Zookeeper
- Messages
- Batches
- Topics
- Partitions
- Producers
- Consumers
- Brokers
- Clusters

Zookeeper

- Manages metadata of Kafka clusters
  - Metadata
  - Naming
  - Configuration
- Requires an odd number of servers (ensemble)

Kafka Messages

- A record of data
- Similar to:
  - Log Entry
  - Row of Data
  - One Metric
- Array of bytes

Kafka Messages

- Messages are composed of two sections of arrays of bytes
- One part of the message payload
- The other part is the message key

Kafka Batches

- Collection of messages
- Packaged and sent to the same topic and partition
- Avoids overhead of multiple messages
- Will often be compressed

Kafka Topics

- Analogous to a database table or folder
- Topics are divided by partitions
- Messages are written in a append manner
- Read in order from beginning to end.
- Subdivided into partitions

Kafka Partitions

- A Subdivision of a topic
- Holds messages in order
- Typically from the same key
- One Leader, Zero or More Replicas (depending on replication factor)

How Messages Find Partitions

- If a key is available, then it will go to a partition based on formula `murmur2(key) % num_partitions`

- If a key is not specified then it will be distributed on a round robin basis
- You can also provide your own `Partitioner` class to direct messages

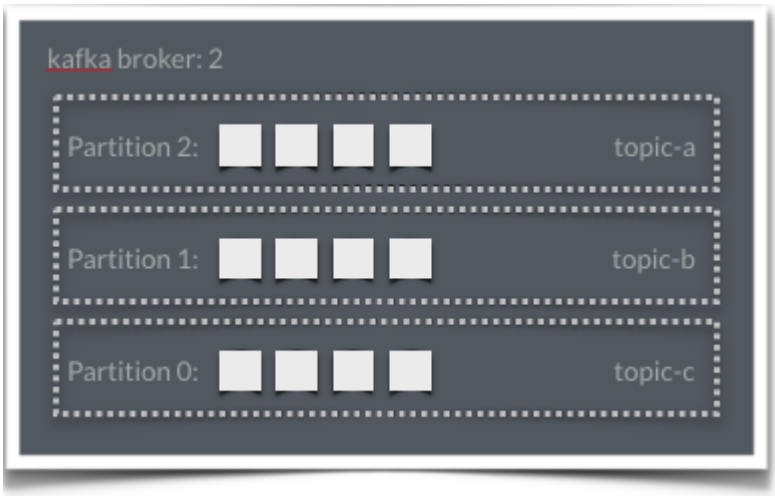
Kafka Brokers

- A single Kafka server is a *broker*
- Receives messages from producers
- Assigns Offsets
- Commits the messages to storage on disk
- One broker can handle thousands of partitions and millions of messages per second

Broker 0

Broker 1

Broker 2



Kafka Clusters

- Brokers are meant to be redundant
- One Kafka Broker will be nominated as the cluster *Controller* by all living members.
- A Controller is responsible for administrative operations:
  - Assigning Partitions
  - Monitoring Brokers

Rationale for Clustering

- Segregation of types of data
- Isolation for security requirements
- Multiple data centers (across different cloud regions)

Partition Commit Log

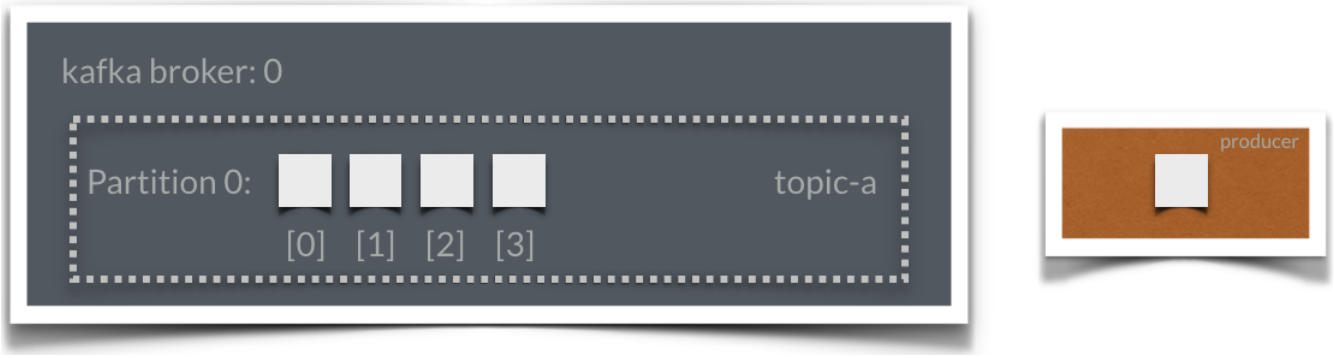
- Data is stored on disk
- This is called a commit log
- Partitions are replicated onto other machines for durability
- Messages are stored by offset

Kafka Retention

- Kafka can be configured to hold onto message for:
  - A determinate amount of time
  - Until a certain size has been reached

- After those limits are reached data is expired and removed
- ***A service can be taken offline for any moment of time and not lose messages***

Before Retention:



After Retention:

Data is Temporary

- By default data retention is 168 hours (1 week)
- This can be changed per broker
- This can be changed per topic

After Messages are Removed

- After retention time has expired messages are removed
- Offsets are maintained

Kafka Partition Leaders

- A partition
  - Is owned by *one* broker, and that broker is the *leader* of that partition
  - May be assigned to multiple clusters, which results in the partition being replicated.
- This will ensure redundancy

- *All consumers and producers on the partition must connect to the leader*

Leaders and Replicas

- Replicas are distributed on multiple brokers
- Producers and consumers communicate only to the leader
- Replicas that have all the information as the leader is called the ISR (In Sync Replica)

Broker 0 with Partitions

Broker 1 with Partitions

Broker 2 with Partitions

Kafka Producers and Consumers

- All clients can either be producers or consumers
- Kafka can handle multiple producers and consumers

Kafka Producers

- Create new messages and send to a *topic*
- Also known as *publishers* or *writers*
- Most of the time will not care about the partition
- If needed, producer can write to specific partition and apply the message so that it goes to the same partition each time

- They can use a custom partitioner

Kafka Consumers

- Read messages from a *topic*
- Also known as *subscribers* or *readers*
- Subscribes to one or more topics, reads the messages as they're produced
- It tracks the latest message it receives by an *offset*
  - Offset is an integer value that increases that is add when a message is added
  - Each message on a partition has a unique offset
- Through Kafka, the offset can be used to rewind to that location

Kafka Consumer Groups

- Consumer Groups are a consortium that makes sure that messages are only read by one consumer
- A consumer of a group
- Benefit: Consumers can horizontally scale to consume topics with large number of messages
- If a single consumer fails, the remainder of the group will *rebalance* to take over the missing member

Controller

- Partition Leader election is done by the *controller*
- A Kafka broker that is responsible for electing partition leaders
- The first broker will create a node in Zookeeper called `/controller`
- Other brokers that try to register as a controller will be told "node already exists"
- Other brokers will get notified when that changes
- When the broker fails `/controller` will be relinquished, and the first broker will be the *controller*
- The new controller will have *controller epoch* and used as a signature of it's leadership
- Any old *controller epochs* will be ignored by non-leaders

Why Kafka?

Comparing Kafka

As opposed to other Pub/Sub frameworks why Kafka?

- Multiple Producers
- Multiple Consumers
- Consistent Format
- Consuming Streams does not interfere with one another
- Consumers can choose to operate as part of a group and share a stream, while the *group* only processes once.

Scalability

- Allows any amount of data to be processed flexibly
- Start with a single broker proof of concept
- From there expand
- Expansion can be done without bringing down the any of the core system
- Resilient: Cluster of multiple brokers can handle a failure and still continue servicing clients.
- Can add higher replication factors

High Performance

- Producers, consumers, and brokers are scaled out to handle large messaging streams with ease
- Provides sub-second message latency from producing a message to having it available to consumers

Use Cases

- Activity Tracking
- Page Views
- Click Tracking
- Advertising Patronage
- Feed Artificial Intelligence
- Internet of Things

Messaging

- It can be used for standard messaging
- No decoration or formatting required, all bytes
- Can aggregate messaging

Metrics and Logging

- Multiple Producers of the same type from varying applications
- Multiple Consumers to monitor and alert of issue with the information
- Possibly filtered through Elasticsearch and Hadoop for long term analysis

Commit Logging

- Track and audit database changes
- Stream the list of updates
- Audit authors to ensure no unauthorized access
- Ability to do an autopsy on crashes

Stream Processing

- Ability to take short term data and analyze the contnts.
- Count metrics and partition
- Transform data for other uses.

Lab: Architecture

Lab: Architecture

Step 1: Ensure that you have Java 8 installed

Step 2: Download confluent open source from …, and unzip the download to your home directory

Step 3: Edit the *./etc/server.properties* file inside of *confluent-5.0.0*, change instances of `your.host.name` to `localhost`

Step 3: Go to the `confluent-5.0.0` directory and run:

```
$ ./bin/confluent start
```

or

```
> .\bin\confluent start
```

Step 4: Ensure that all service are running with the following messages:

```
Starting zookeeper
zookeeper is [UP]
Starting kafka
kafka is [UP]
Starting schema-registry
schema-registry is [UP]
Starting kafka-rest
kafka-rest is [UP]
Starting connect
connect is [UP]
Starting ksql-server
ksql-server is [UP]
Starting control-center
control-center is [UP]
```

Command Line Utilities

Starting Kafka

```
$ /usr/local/kafka/bin/kafka-server-start.sh -daemon \
     /usr/local/kafka/config/server.properties
```

Stopping Kafka

```
$ /usr/local/kafka/bin/kafka-server-stop.sh
```

Verifying Running Instances

```
$ /usr/local/zookeeper/bin/zkCli.sh -server <zookeeper>:2181
ls /brokers
```

Creating a Topic

```
/usr/bin/kafka-topics --create \
 --zookeeper zoo:2181 \
 --replication-factor 2 --partitions 4 \
 --topic <topic>
```

List all the topics

```
$ /usr/bin/kafka-topics --zookeeper <zookeeper>:2181 --list
```

Sending Messages to a Kafka Topic

```
$ /usr/local/kafka/bin/kafka-console-producer.sh \
  --broker-list <kafka-broker>:9092 \
  --topic test
```

Reading Back the Messages From a Topic

```
$ /usr/local/kafka/bin/kafka-console-consumer.sh \
  --bootstrap-server <kafka-broker>:9092 \
  --topic <topic> --from-beginning
```

Viewing How Partitions are Distributed

```
$ /usr/local/kafka/bin/kafka-topics.sh \
    --describe --topic <topic-name> \
    --zookeeper <zookeeper>:2181
```

kafkacat

- https://github.com/edenhill/kafkacat
- kafkacat is a generic non-JVM producer and consumer for Apache Kafka >=0.8, think of it as a netcat for Kafka.
- In *producer mode*, kafkacat reads messages from stdin, delimited with a configurable delimiter (`-D`, defaults to newline), and produces them to the provided Kafka cluster (`-b`), topic (`-t`) and partition (`-p`).
- In *consumer mode*, kafkacat reads messages from a topic and partition and prints them to stdout using the configured message delimiter

Installing kafkacat

- On recent enough Debian systems:

```
apt-get install kafkacat
```

- And on Mac OS X with homebrew installed:

```
brew install kafkacat
```

kafkacat Examples

- Subscribing to *topic1* and *topic2*

```
$ kafkacat -b mybroker -G mygroup topic1 topic2
```

- Read messages from stdin, produce to 'syslog' topic with snappy compression

```
$ tail -f /var/log/syslog | kafkacat -b mybroker -t syslog -z snappy
```

- Read messages from Kafka 'syslog' topic, print to stdout

```
$ kafkacat -b mybroker -t syslog
```

More `kafkacat` Examples

- Produce messages from file (one file is one message)

```
$ kafkacat -P -b mybroker -t filedrop -p 0 myfile1.bin /etc/motd
thirdfile.tgz
```

- Consume from all partitions from 'syslog' topic

```
$ kafkacat -C -b mybroker -t syslog
```

- Read the last 2000 messages from 'syslog' topic, then exit

```
$ kafkacat -C -b mybroker -t syslog -p 0 -o -2000 -e
```

Even More `kafkacat` Examples

- Output consumed messages in JSON envelope:

```
$ kafkacat -b mybroker -t syslog -J
```

- Output consumed messages according to format string:

```
$ kafkacat -b mybroker -t syslog -f 'Topic %t[%p], offset: %o, key: %k,
payload: %S bytes: %s\n'
```

- Read the last 100 messages from topic 'syslog' with `librdkafka` configuration parameter `broker.version.fallback` set to `0.8.2.1` :

```
$ kafkacat -C -b mybroker -X broker.version.fallback=0.8.2.1 -t syslog
-p 0 -o -100 -e
```

Lab: Command-Line

Lab: Command-Line

Step 1: Go to `confluent-5.0.0` and run `kafka-console-producer`

```
./bin/kafka-console-producer \
--broker-list localhost:9092 \
--topic first_topic
```

This will create a topic of 1 replica and 1 partition

Step 2: You will be provided a prompt. Using these prompts, enter a word and hit **ENTER**, add some more words and hit **ENTER**

Step 3: In another terminal window or tab, run `kafka-console-consumer`

```
./bin/kafka-console-consumer \
--bootstrap-server localhost:9092 \
--topic first_topic --from-beginning
```

Step 4: Ensure that the messages that you entered in the first terminal show up in the second

Step 5: Enter some more words in the producer, and view the messages in the terminal with the consumer

Step 6: Go to the documentation for `kafka-console-producer` and `kafka-console-consumer` and research how you can produce a key and value on the command line, and read read that key and value on the consumer.

Step 7: Shut down `kafka-console-producer` and `kafka-console-consumer` using **CTRL+C**

Clients

Including the Kafka Library in Maven

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.10.2.0</version>
</dependency>
```

Producers

Producers

- Clients are available for a variety of languages
- They all use `librdkafka` C-based library to connect
  - Python
  - C/C++
  - C#
- Java is the "native" language of Kafka

Creating a `Producer`

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
  "localhost:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);
```

- Typically in production `ProducerConfig.BOOTSTRAP_SERVERS_CONFIG` requires two connections
- Key Serializer is in charge of converting key objects to bytes
- Value Serializer is in charge of converting value objects to bytes

Prepackaged Serializers

- `ByteArraySerializer`
- `StringSerializer`
- `IntegerSerializer`

Custom Serializers

- All Serializers implement `org.apache.kafka.common.serialization.Serializer`
- Will serialize the key object to a byte array

Creating a `KafkaProducer`

- A `KafkaProducer` is thread-safe

- Establish a key and value type

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

Creating a `ProducerRecord`

- The record of what we are going to send, and must meet the generic types of the `KafkaProducer`

No Key Message

```
ProducerRecord<String, String> producerRecord =
  new ProducerRecord<>("scaled-cities", "Bismarck, ND");
```

Message with a Key

```
ProducerRecord<String, String> producerRecord =
  new ProducerRecord<>("scaled-cities", "ND", "Bismarck, ND");
```

Alternative `ProducerRecord` with Timestamp

- There are different `ProducerRecord` signatures for the class, one that can set up the timestamp for event.
- This would require that the topic be updated `message.timestamp.type` to use `CreateTime` instead of `LogAppendTime`

Message with a Key, Partition, and Timestamp

```
ProducerRecord<String, String> producerRecord =
  new ProducerRecord<>("scaled-cities", 3,
    new java.util.Date().getTime(), "ND", "Bismarck, ND");
```

Partition should be `0` based. In the above example, `3` means the 4th partition

Message with a Key and Timestamp

```
ProducerRecord<String, String> producerRecord =
  new ProducerRecord<>("scaled-cities", null,
     new java.util.Date().getTime(), "ND", "Bismarck, ND");
```

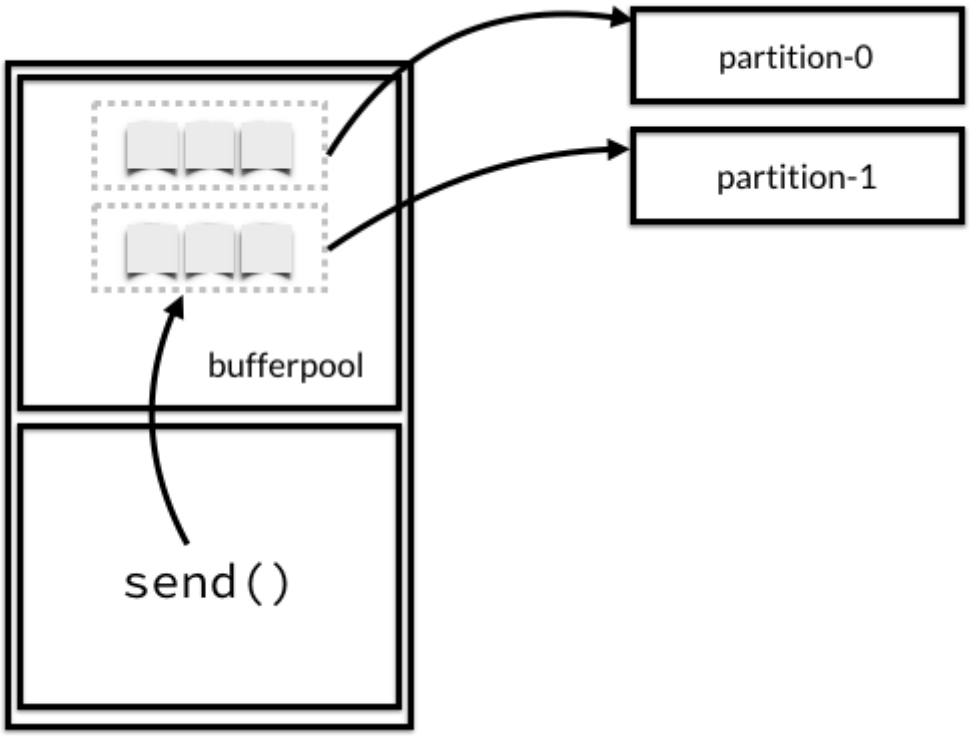
`null` for partition means that it will go to a partition based on key

Sending a Message

- Production of a message is synchronous
- `send` does not automatically send to the brokers
- It gets processed inside of a buffer pool

```
producer.send(producerRecord);
```

Bufferpool and batching



- When messages are sent, they are added to the "bufferpool"
- Given some properties they can be batched together
- Each batch goes to a destination partition
- Each batch can optionally be compressed

Listening for message processing

```
Future<RecordMetadata> future = producer.send(producerRecord);
try {
    RecordMetadata metadata = future.get();
    System.out.format("offset: %d\n", metadata.offset());
    System.out.format("partition: %d\n", metadata.partition());
    System.out.format("timestamp: %d\n", metadata.timestamp());
    System.out.format("topic: %s\n", metadata.topic());
    System.out.format("toString: %s\n", metadata.toString());
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

`future.get()` blocks, consider either `Callback`, `CompleteableFuture`, or a streaming library like RXJava, Akka Streams to process a future nicely.

Listening for message processing with a `Callback`

- A `Callback` is "lambdaizable" `interface`
- If `metadata` is null, then an error occurred
- If `exception` is null, then an record send was successful

```
producer.send(producerRecord, new Callback() {
  @Override
  public void onCompletion(RecordMetadata metadata,
    Exception exception) {
    if (metadata != null) {
        System.out.format("offset: %d\n",
          metadata.offset());
        System.out.format("partition: %d\n",
          metadata.partition());
        System.out.format("timestamp: %d\n",
          metadata.timestamp());
        System.out.format("topic: %s\n",
          metadata.topic());
        System.out.format("toString: %s\n",
          metadata.toString());
    } else {
        exception.printStackTrace();
    }
  }
});
```

Converting the `Callback` to a Lambda

```
producer.send(producerRecord, (metadata, exception) -> {
  if (metadata != null) {
      System.out.format("offset: %d\n", metadata.offset());
      System.out.format("partition: %d\n", metadata.partition());
      System.out.format("timestamp: %d\n", metadata.timestamp());
      System.out.format("topic: %s\n", metadata.topic());
      System.out.format("toString: %s\n", metadata.toString());
  } else {
      exception.printStackTrace();
  }
});
```

Obtaining the `producerRecord` via a Closure

- Inside a function we can make a reference to an object outside of a lambda
- This is called a *closure*
- The reference will need to be `final` or *effectively final*
- *effectively final* means a variable that no longer changes

```
producer.send(producerRecord, (metadata, exception) -> {
  if (metadata != null) {
      System.out.println(producerRecord.value());
      System.out.format("offset: %d\n", metadata.offset());
      System.out.format("partition: %d\n", metadata.partition());
      System.out.format("timestamp: %d\n", metadata.timestamp());
      System.out.format("topic: %s\n", metadata.topic());
      System.out.format("toString: %s\n", metadata.toString());
  } else {
      exception.printStackTrace();
  }
});
```

`acks`

- You can stipulate different level of *acknowledgments* when sending a message

Table 1. Various `acks`

acks	Description
`0`	No acknowledgment, assume all is well
`1`	At least one replica will `Producer` will receive a success response, error if unsuccessful, up to client to hand

acks	Description
`all`	All replicas must acknowledge. Higher latency, Safest

`acks` Inside the Producer

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "kaf0:9092,kaf1:9092,kaf2:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
properties.put(ProducerConfig.ACKS_CONFIG, "all");
...
```

`retries`

- Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error
- Can be couple with `RETRY_BACKOFF_MS_CONFIG` which is time between retries

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        IntegerSerializer.class);
properties.put(ProducerConfig.ACKS_CONFIG,
        "all");
properties.put(ProducerConfig.RETRIES_CONFIG,
        "10");
properties.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG,
        "400");
...
```

Lab: Writing a Producer

Lab: Writing a Producer

Step 1: Clone kafka-client repository from https://github.com/dhinojosa/kafka-client with:

```
git clone https://github.com/dhinojosa/kafka-client
```

or

```
git clone git@github.com:dhinojosa/kafka-client.git`
```

Step 2: Run the following commands to ensure that you have these folders available other wise the build will fail:

```
$ mkdir -p src/main/avro
$ mkdir -p src/main/java
$ mkdir -p src/test/java
$ mkdir -p src/main/resources
$ mkdir -p src/test/resources
```

For Windows, remove the `-p`

Lab: Writing a Producer (Continued)

Step 3: Run the following command with Maven: `mvn clean compile` to ensure that dependencies are downloaded

Step 4: Create a topic called `my_orders` in Apache Kafka, with a replication factor of `1` and `3` partitions

Step 5: Create a producer called `MyProducer` inside of a package `com.xyzcorp`

Step 6: Create a `public static void main(String[] args)` application, and create a array of states in the United States, using the following `String`

```
String stateString =
       "AK,AL,AZ,AR,CA,CO,CT,DE,FL,GA," +
       "HI,ID,IL,IN,IA,KS,KY,LA,ME,MD," +
       "MA,MI,MN,MS,MO,MT,NE,NV,NH,NJ," +
       "NM,NY,NC,ND,OH,OK,OR,PA,RI,SC," +
       "SD,TN,TX,UT,VT,VA,WA,WV,WI,WY,";
```

Lab: Writing a Producer (Continued)

Step 7: In the `main` method, create a loop of your choosing and `send` a message with the state as the key, and a random value from 10 to 100000 for the cost of the total order using `java.util.Random.nextInt`. Using a `Future<RecordMetadata>` or `Callback` and print the metadata either using `System.out.println` or `System.out.format`, you can use a closure to enclose the key and value and print the content.

Step 8: Have a `Thread.sleep` with a random amount of time from 5 to 30 seconds using the same `java.util.Random` instance to represent some time between messages to mimic the real world

Step 9: Run in your IDE, or run in Maven using `mvn exec:java -Dexec.mainClass="com.xyzcorp.MyProducer"`

Lab: Writing a Producer (Continued)

Step 10: Use `kafka-console-consumer` to consume message from `my_orders`

```
./bin/kafka-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic my_orders --from-beginning \
  --key-deserializer
org.apache.kafka.common.serialization.StringDeserializer \
  --value-deserializer
org.apache.kafka.common.serialization.IntegerDeserializer \
  --property print.key=true \
  --property key.separator=,
```

Step 11: Stop the `kafka-console-consumer` instance and stop `MyProducer` with an interrupt

Consumers

Establishing Properties for Consumers

```
Properties properties = new Properties();
properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
  "kaf0:9092, kaf1:9092, kaf2:9092");
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  "org.apache.kafka.common.serialization.StringDeserializer");
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  "org.apache.kafka.common.serialization.StringDeserializer");
```

- `GROUP_ID_CONFIG` establishes the group identifier used by more than one consumer
- Consumers are binary equivalent

Creating the Consumer

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

- First `String` is the type of the *key*
- Sencond `String` is the type of the *value*

Subscribing to Topics

```
consumer.subscribe(
    Collections.singletonList("scaled-cities"));
```

```
consumer.subscribe(
    Arrays.asList("stock-news", "stock-trade"));
```

Iterating Messages

```
while (true) {
    ConsumerRecords<String, String> records =
            consumer.poll(500);
    for (ConsumerRecord<String, String> record : records) {
        System.out.format("offset: %d\n", record.offset());
        System.out.format("partition: %d\n", record.partition());
        System.out.format("timestamp: %d\n", record.timestamp());
        System.out.format("timeStampType: %s\n",
          record.timestampType());
        System.out.format("topic: %s\n", record.topic());
        System.out.format("key: %s\n", record.key());
        System.out.format("value: %s\n", record.value());
    }
}
```

- `consumer.poll`
  - Is a timeout, if there are no messages it will return an empty `ConsumerRecords`
  - Sends the `offset` automatically if 5 seconds have lapsed
  - Lets the group controller know that the consumer is still alive
- `ConsumerRecords` is an instance of `Iterable` and can be applied to a for loop

Closing Resources Appropriately

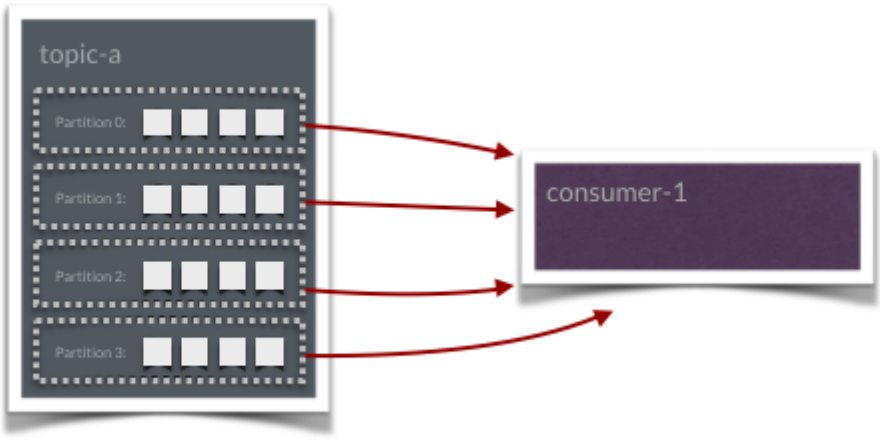
- Place the following *before* the `while` loop

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    consumer.close();
}))
```

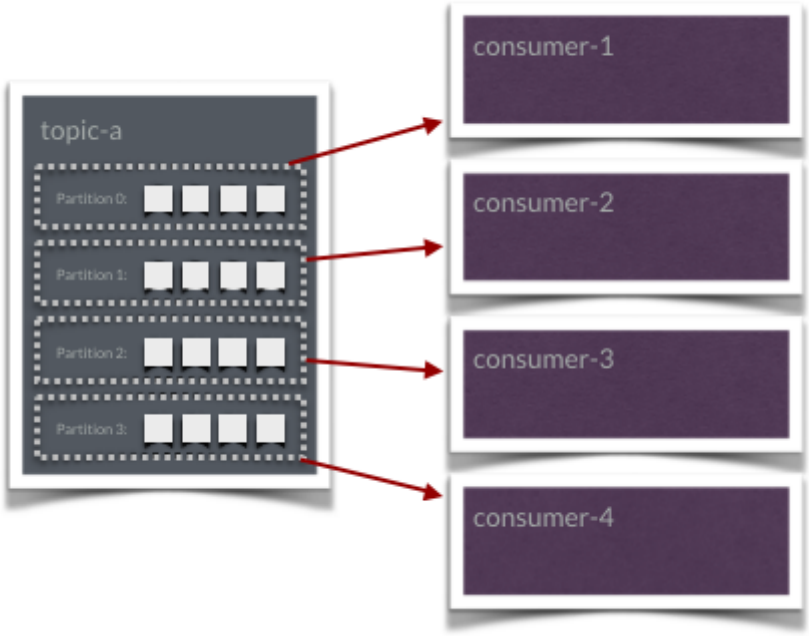
- Using a method reference:

```
Runtime.getRuntime().addShutdownHook(new Thread(consumer::close));
```

Why a consumer group?



Distribution in a group



- What binds all the consumers, is the `group.id`
- Consumers should be on separate:
  - Threads
  - Machines

What is our first valid offset?

- Offsets are stored in the `__consumer_offsets` topic
- Managed by `group_id` and partition
- When no offset is valid:

- Because you are starting a consumer `group` for the first time with `auto.offset.reset`
- Options:
  - `earliest`
  - `latest`
  - `none` (Handle the errors yourself)

Committing Offsets

- Offsets are committed automatically by default, every 5 seconds (5000 milliseconds)

Using `kafka-consumer-groups` Status

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
 --describe --group my-group

TOPIC    PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG  CONSUMER-ID
my-topic 0          2               4               2    consumer-1-029a
my-topic 1          2               3               1    consumer-1-029a
my-topic 2          2               3               1    consumer-2-42c1
```

Reading from the Beginning

```
consumer.poll(0);
consumer.seekToBeginning(consumer.assignment());

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(5);
    for (ConsumerRecord<String, String> record : records) {
        System.out.format("offset: %d\n", record.offset());
        System.out.format("partition: %d\n", record.partition());
        System.out.format("timestamp: %d\n", record.timestamp());
        System.out.format("timeStampType: %s\n",
          record.timestampType());
        System.out.format("topic: %s\n", record.topic());
        System.out.format("key: %s\n", record.key());
        System.out.format("value: %s\n", record.value());
    }
}
```

Reading from the End

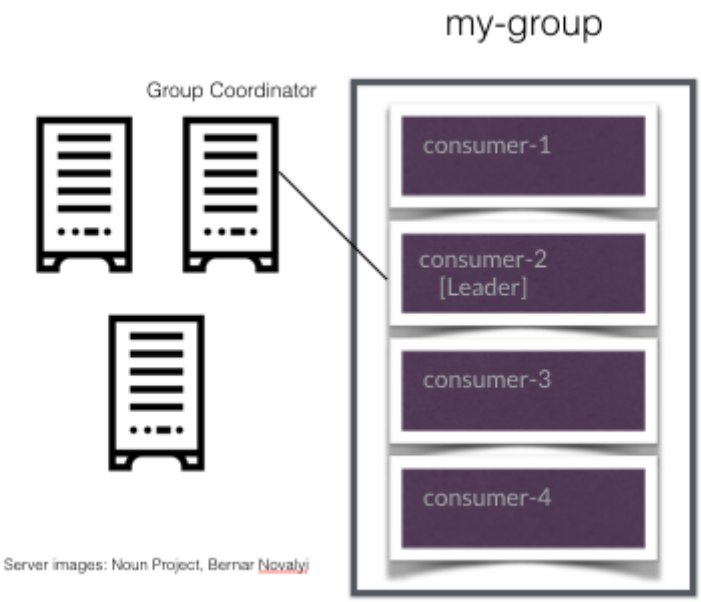
```
consumer.poll(0);
consumer.seekToEnd(consumer.assignment());

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(5);
    for (ConsumerRecord<String, String> record : records) {
        System.out.format("offset: %d\n", record.offset());
        System.out.format("partition: %d\n", record.partition());
        System.out.format("timestamp: %d\n", record.timestamp());
        System.out.format("timeStampType: %s\n",
          record.timestampType());
        System.out.format("topic: %s\n", record.topic());
        System.out.format("key: %s\n", record.key());
        System.out.format("value: %s\n", record.value());
    }
}
```

Reading from Wherever You Please

- `getTopicPartitionFromExternalSource` is used to look up the next viable partition

```
consumer.poll(0);
for (TopicPartition topicPartition: consumer.assignment()) {
    consumer.seek(topicPartition,
       getTopicPartitionFromExternalSource(topicPartition));
}
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(5);
    for (ConsumerRecord<String, String> record : records) {
        System.out.format("offset: %d\n", record.offset());
        System.out.format("partition: %d\n", record.partition());
        System.out.format("timestamp: %d\n", record.timestamp());
        System.out.format("timeStampType: %s\n",
          record.timestampType());
        System.out.format("topic: %s\n", record.topic());
        System.out.format("key: %s\n", record.key());
        System.out.format("value: %s\n", record.value());
    }
}
```

Group Coordinator and Consumer Group Leader



Group Coordinator

- The way consumers maintain membership in a consumer group by sending heartbeats to a Kafka broker designated as the group coordinator
- Broker can be different for different consumer groups
- As long as the consumer is sending heartbeats at regular intervals, it is assumed to be alive.
- Heartbeats are sent:
  - When the consumer polls (i.e., retrieves records)
  - When it commits records it has consumed.

Group Leader * First consumer is the *leader* * Responsible for assigning partitions

Knowing when Partitions are Revoked or Assigned

- A `ConsumerRebalanceListener` establishes two methods
  - `onPartitionsRevoked` when a partition is revoked from the `Consumer`
  - `onPartitionsAssigned` when a partition is assigned to the `Consumer`

```
consumer.subscribe(Collections.singletonList("scaled-cities"),
  new ConsumerRebalanceListener() {
      @Override
      public void onPartitionsRevoked
         (Collection<TopicPartition> partitions) {
            System.out.println("Partitions " + partitions + " revoked");
      }

      @Override
      public void onPartitionsAssigned
         (Collection<TopicPartition> partitions) {
            System.out.println("Partitions " + partitions + "
assigned");
      }
  });
```

Manual Commits

- Exercise more control over the time at which offsets are committed
- Eliminate the possibility of missing messages and to
- Reduce the number of messages duplicated during rebalancing.
- The consumer API has the option of committing the current offset at a point that makes sense to the application developer
- Currently auto commits are done every 5 seconds

Turning off Auto Commits

```
Properties properties = new Properties();
properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kaf0:9092,
kaf1:9092");
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "testGroup4");
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
```

Synchronous Commit

- Blocks
- Commit the latest offset returned by `poll()`

- Return once the offset is committed
- Throwing an exception if commit fails for some reason.

Asynchronous Commit

- Non-Blocking
- Higher throughput
- Move on

Using Synchronous and Asynchronous Example

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(5);
        for (ConsumerRecord<String, String> record : records) {
          ...
        }
        consumer.commitAsync(); ①
    }
  } catch (WakeupException e) {
      // ignore for shutdown
  } finally {
      consumer.commitSync(); ②
      consumer.close();
      System.out.println("Closed consumer and we are done");
  }
}
```

① `commitAsync` done on another thread, increase throughput

② `commitSync` blocks right before going offline

Lab: Writing a Consumer

Lab: Writing a Consumer

Step 1: Create a consumer called `MyConsumer` inside of a package `com.xyzcorp`

Step 2: Create a `public static void main(String[] args)` in `MyConsumer` and create a single `Consumer` that listens to `my_orders` topic. Make your `group.id`, `my_group`

Step 3: Ensure that you listen to **CTRL+C** and shut down appropriately.

Step 4: Ensure that we start listening to the very beginning if there isn't a valid offset.

Step 5: Run `MyProducer` from the previous lab

Step 6: Run `MyConsumer` in either your IDE or in Maven using `mvn exec:java -Dexec.mainClass="com.xyzcorp.MyConsumer"`, view the results.

Step 7: Stop `MyConsumer` and start it again, why didn't the previous information show? Stop the consumer.

Step 8: When calling `subscribe`, implement a `ConsumerRebalanceListener` that will print out when a partition is assigned to a consumer.

Step 9: Run the same application three times. Ask yourself and discuss why you're running 3 separate instances.

Step 10: Stop a single consumer, show the assignments and revocations.

Step 11: Stop all consumers and producers

Avro

About Avro

- Created by Doug Cutting; Creator of Hadoop
- Serialization is defined by schema
- Schemas are JSON Based
- Codegen available at command line
- Codegen available by Maven, Gradle, SBT Plugin
- Supports the following languages:
  - C, C++, Java, Perl, Python, Ruby, PHP
- Named after WWI, WWII Aircraft Company, A.V. Roe

Types of Avro Serialization

- Generic
  - Develop code that reflects your schema
- Specific
  - Code generate your class from a schema, an `.avsc` file
  - This is the most common form
- Reflection
  - Auto-create schema from an existing class

Avro Primitive Types

Table 2. Avro and Corresponding Java Types

Avro Type	Java Type
`null`	`null`
`double`	`double`
`float`	`float`
`int`	`int`

Avro Type	Java Type
`long`	`long`
`bool`	`bool`
`string`	Unicode `CharSequence`
`bytes`	Sequence of 8-bit unsigned bytes

Avro Complex Types

```
{
  "namespace": "com.xyzcorp", ①
  "type": "record", ②
  "doc" : "An music album",
  "name": "Album", ③
  "fields": [
    {
      "name": "name",
      "type": "string"
    },
    {
      "name": "yearReleased",
      "type": [ ④
        "int",
        "null"
      ]
    }
  ]
}
```

① `namespace` is what `package` per language

② `record` is a complex type with multiple fields

③ `name` what `class` will this represent

④ A union type, can either be `int` or `null`

Avro Field Options

Avro Field	Description
`name`	Name of the Field
`doc`	Documentation

Avro Field	Description
`type`	Type of the Field
`default`	Default Value
`order`	What order does this impact record
`aliases`	Any other names

Avro Array

```
{"type": "array", "items": "string"}
```

Avro Map

- All keys in an Avro map are `string`

```
{"type": "map", "values": "long"}
```

Avro Enum

```
{ "type" : "enum",
  "name" : "rainbowColors",
  "doc" : "Colors of the Rainbow",
  "symbols" : ["RED", "ORANGE", "YELLOW",
               "GREEN", "BLUE", "INDIGO",
               "VIOLET"]}
```

Specification for Avro

https://avro.apache.org/docs/1.8.2/spec.html

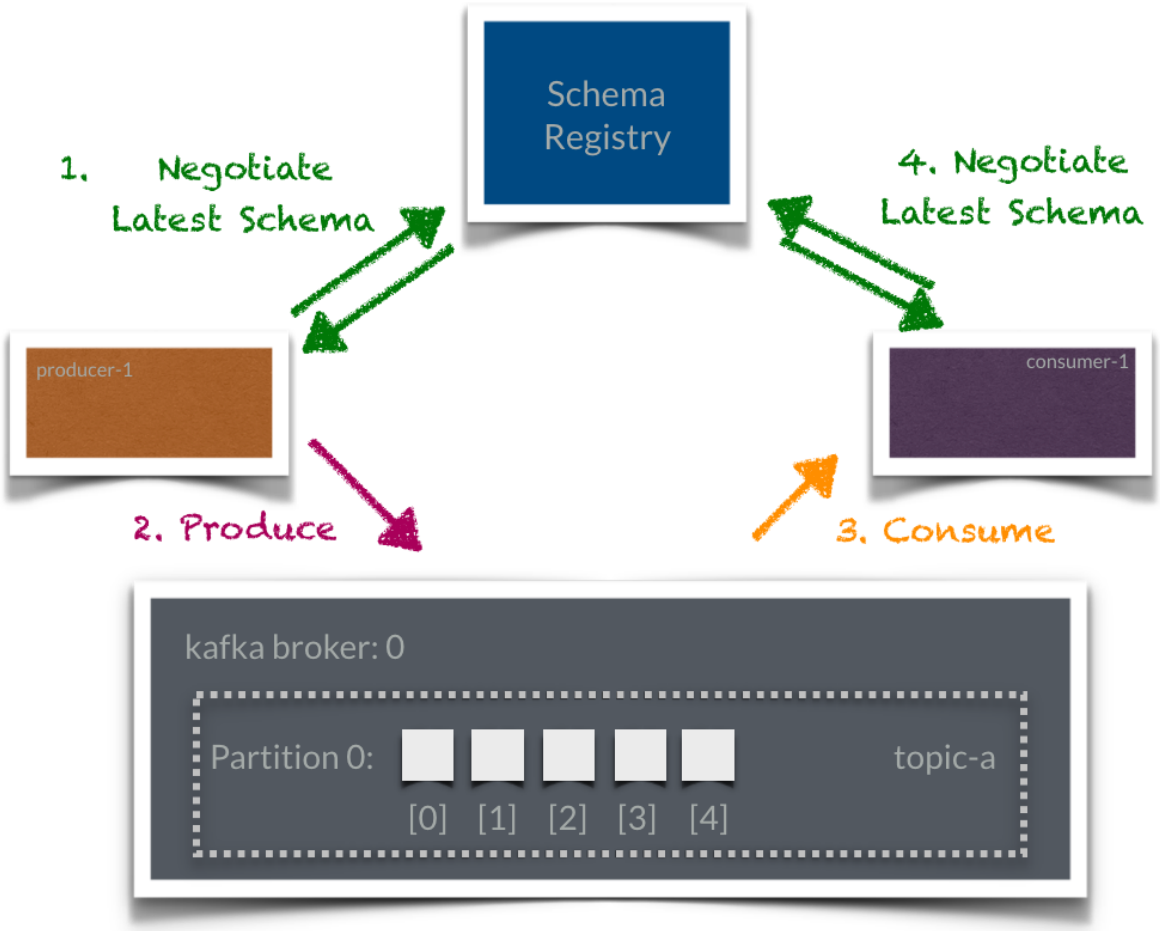
Avro Tools

```
java -jar avro-tools-1.8.2.jar compile schema <schema file>
<destination>
```

Available Avro Build Tools

Schema Registry

- Service that contraints and typechecks messages before sending
- Stores schemas in registry for and assigns a global ID
- Ensures *backward* and *forward* compatibility
- Does so through Serializers and Deserializers
- Automatically done through Java Kafka Client API



Different Message Format

- The first byte is a zero byte
- The second byte is the global identifier of the Schema that registered in the Schema Registry

Schema Registry API

- REST based API used for:
  - Registration of Schemas
  - Updating of Schemas
  - Preferred format of `application/vnd.schemaregistry.v1+json`

Registering a New Schema

```
POST /subjects/my_topic-key/versions HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json,
        application/vnd.schemaregistry+json,
        application/json

{
  "schema":
    "{
       \"type\": \"record\",
       \"name\": \"test\",
       \"fields\":
         [
           {
             \"type\": \"string\",
             \"name\": \"field1\"
           },
           {
             \"type\": \"int\",
             \"name\": \"field2\"
           }
         ]
     }"
}
```

"Subjects" from the Client API will have the form "topic-key" and "topic-value"

Getting the Response for the new Schema

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{"id":1}
```

`id` is the global identifier

More API Calls Available

https://docs.confluent.io/current/schema-registry/docs/api.html

Retrieving the Schema by Global ID

```
GET /schemas/ids/1 HTTP/1.1
Host: schemaregistry.example.com
Accept: application/vnd.schemaregistry.v1+json,
        application/vnd.schemaregistry+json,
        application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "schema": "{\"type\": \"string\"}"
}
```

Schema Registry Producer

```
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        io.confluent.kafka.serializers.KafkaAvroSerializer.class);
properties.setProperty("schema.registry.url", "http://localhost:8081");
```

- What makes schema registry work is the Serializer
- You must add `schema.registry.url` and specify the location of the registry

Sending a custom object from Kafka

```
Album album = new Album("Purple Rain", "Prince", 1984,
   Arrays.asList("Purple Rain", "Let's go crazy"));

ProducerRecord<String, Album> producerRecord =
    new ProducerRecord<>("music_albums", "Prince", album);

producer.send(producerRecord);
```

Schema Registry Consumer

- What makes schema registry work is the Deserializer
- You must add `schema.registry.url` and specify the location of the registry

- You must add `specific.avro.reader` and specify that you used specific avro mode

```
Properties properties = new Properties();
properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        io.confluent.kafka.serializers.KafkaAvroDeserializer.class);
properties.setProperty("schema.registry.url", "http://localhost:8081");
properties.setProperty("specific.avro.reader", "true");
```

Receiving a Custom Object from Kafka

```
while (true) {
  ConsumerRecords<String, Artist> records =
          consumer.poll(Duration.of(500, ChronoUnit.MILLIS));
  for (ConsumerRecord<String, Artist> record : records) {
      System.out.format("offset: %d\n", record.offset());
      System.out.format("partition: %d\n", record.partition());
      System.out.format("timestamp: %d\n", record.timestamp());
      System.out.format("timeStampType: %s\n",
              record.timestampType());
      System.out.format("topic: %s\n", record.topic());
      System.out.format("key: %s\n", record.key());
      System.out.format("value: %s\n", record.value());
  }
}
```

Lab: Using Avro

Lab: Using Avro

Step 1: In the `src/main/avro` folder create a schema for `Order` that includes the following fields:

Field Name	Type
Total	Integer
Shipping	Enumeration ("next_day", "2_day", "ground")
State	String
Discount	Float
Gender	Union Type: `null` Enumeration ("male", "female")

Step 2: Ensure that you generate a class from the avro by invoking `mvn generate-sources`. Verify that the classes are created by looking in the *target/generated-sources* folder.

Step 3: Create a topic called `my_avro_orders` in Apache Kafka, with a replication factor of `1` and `3` partitions

Step 4: Create a producer called `MyAvroProducer` inside the package of `com.xyzcorp`

Step 5: Create a `public static void main(String[] args)` application, and create a array of states in the United States, using the following `String`

```
String stateString =
       "AK,AL,AZ,AR,CA,CO,CT,DE,FL,GA," +
       "HI,ID,IL,IN,IA,KS,KY,LA,ME,MD," +
       "MA,MI,MN,MS,MO,MT,NE,NV,NH,NJ," +
       "NM,NY,NC,ND,OH,OK,OR,PA,RI,SC," +
       "SD,TN,TX,UT,VT,VA,WA,WV,WI,WY,";
```

Lab: Avro (Continued)

Step 6: In the `main` method, create a loop of your choosing like you did in the Producer lab, only this time, instantiate an `Order` and with a random value from `java.util.Random` create a total from 10 to 100000, a shipping selection, a state for field and key, a discount from 0 to .15 and a gender for `male` or `female`.

Step 7: Have a `Thread.sleep` with a random amount of time from 5 to 30 seconds using the same `java.util.Random` instance to represent some time between messages to mimic the real world

Step 8: Send information using a Schema Registry at `localhost:8081/schemaregistry` in your properties

Step 9: Ensure that the information is in the `my_avro_orders` topic using `kafka-avro-console-consumer`

Step 10: Copy the consumer, and put it into group `my_avro_group`. Make sure you are listening to `my_avro_orders`

Streams

Stream Processing

- A data stream is an abstraction representing an unbounded dataset
- Unbounded means infinite and ever growing
- The dataset is unbounded because over time, new records keep arriving

Kafka Streams

- Stream Processor for building applications and microservices
- Is an abstraction over a `Consumer`-`Producer` pattern
  - "As we consumer we will produce"
- Not a framework, but a library. Opposed to other streaming solutions
- Distribution
  - Uses `application.id` much like the consumer `group.id`
  - Small footprint, just `main` methods

Establishing the Stream

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG,
  "my_stream_app"); 1
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
  "localhost:9092"); 2
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
  Serdes.String().getClass().getName()); 3
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
  Serdes.String().getClass().getName());
```

- `APPLICATION_ID_CONFIG`
  - Identifier for the Stream app
  - Used to coordinate and name local stores
- `SERDE`
  - Combination of `Serializer` and `Deserializer`

Establishing the Kafka Builder

```
StreamsBuilder builder = new StreamsBuilder();
```

- Provide the high-level Kafka Streams DSL
- Specify a Kafka Streams topology
- Uses the notion of the builder pattern to establish a Topology

Use the `StreamBuilder` to attach to a topic

- Once `StreamBuilder` is established you can apply the Kafka Streams DSL
- The KafkaStreams DSL is nothing more than function programming
  - `map`
  - `filter`
  - `flatMap`
  - `groupBy`

Topology can also be branched

- Once the `StreamBuilder` is established
  - One branch can be processed a certain way to another topic
  - Another branch can diverge to perform other actions and flow to another topic

```
KStream<String, String> streamAfterCleanup = builder.map((k, v) -> ..);

streamAfterCleanup.filter(...).map(...).to(...);
streamAfterCleanup.flatMap(...).filter(...).groupBy().count();
```

Kafka Streams DSL: `map`

Given a Stream of the following:

```
(1, "Hello"), (2, "Zoom"), (3, "Fold")
```

Applying `map`:

```
stream.map((key, value) -> new KeyValue<>(key + 1, value + "!"));
```

Resulting in:

```
(2, "Hello!"), (3, "Zoom!"), (4, "Fold!")`
```

Kafka Streams DSL: `filter`

Given a Stream of the following:

```
(1, "Hello"), (2, "Zoom"), (3, "Fold"), (4, "Past")
```

Applying `filter`:

```
stream.filter((key, value) -> key % 2 == 0);
```

Resulting in:

```
(2, "Zoom"), (4, "Past")
```

Kafka Streams DSL: `flatMap`

Given a Stream of the following:

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

Applying `flatMap`:

```
stream.flatMap((key, value) -> Arrays.asList(
    new KeyValue(key, value),
    new KeyValue(key * 100, value + " Hundred")))
```

Resulting in:

```
(1, "One"), (100, "One Hundred"), (2, "Two"),
(200, "Two Hundred"), (3, "Three"),
(300, "Three Hundred"), (4, "Four"),
(400, "Four Hundred")
```

Kafka Streams DSL: `peek`

`peek` peers into the Stream and view what goes through

```
stream1.peek((key, value) ->
  System.out.printf("key: %s, value: %d", key, value));
```

Taking your results and putting them into a Topic

From a `KStream`, you can send data into a topic

With `to`:

```
kstream.to("end_topic");
```

With `through`:

```
kstream.through("end_topic");
```

Kafka Streams DSL: `groupBy`

Given a Stream of the following:

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

Applying `groupBy`:

```
stream.groupBy((key, value) -> key % 2 == 0 ? "Even", "Odd")
```

Resulting in: `KGroupedStream` object

- `groupBy` creates a `KGroupedStream` object
- You have to decide what to do with this group
  - `count`
  - `aggregate`
  - `reduce`
  - `windowBy`
- Once you call any of the above, you have a `KTable`

`KTable`

- Represents a changelog, where the oldest records are not important
- Ideal for counters, and state
- `KTable` values are stored at each application's *ephemeral storage*
- Most ephemeral storage is backed by Rocks DB
- All streaming is then backed up as a topic for resilience

Kafka Streams DSL: `groupByKey`

Given a Stream of the following:

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

Applying `groupByKey`:

```
stream.groupByKey()
```

- Resulting in `KGroupedStream` object
- Is synonymous with the following:

```
stream.groupBy((key, value) -> key)
```

Kafka Streams DSL: `count`

- Aggregates the `count` of what has been grouped
- Records with `null` key or `value` are ignored
- Returns `KTable` that represents the latest rolling count
- Will maintain the count in ephemeral storage, (like RocksDB)

Given a Stream of the following

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

```
stream.groupBy((key, value) -> key % 2 == 0 ? "Even" : "Odd").count();
```

Will represent in the RocksDB ephemeral storage:

```
("Even", 2)
("Odd", 2)
```

Kafka Streams DSL: `reduce`

- Aggregates a calculation based on all values
- Selects the first value as the running total

Given a stream of the following:

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

```
stream.groupBy((key, value) -> key % 2 == 0 ? "Even" : "Odd")
      .reduce((value1, value2) -> value1 + "," + value2);
```

Will represent in the RocksDB ephemeral storage:

```
("Even", "Two,Four")
("Odd", "One,Three")
```

Kafka Streams DSL: `aggregate`

- Aggregates a calculation based on all values
- Provides the opportunity to add a seed
- Provides the opportunity to include the key as part of the aggregation

Given a stream of the following:

```
(1, "One"), (2, "Two"), (3, "Three"), (4, "Four")
```

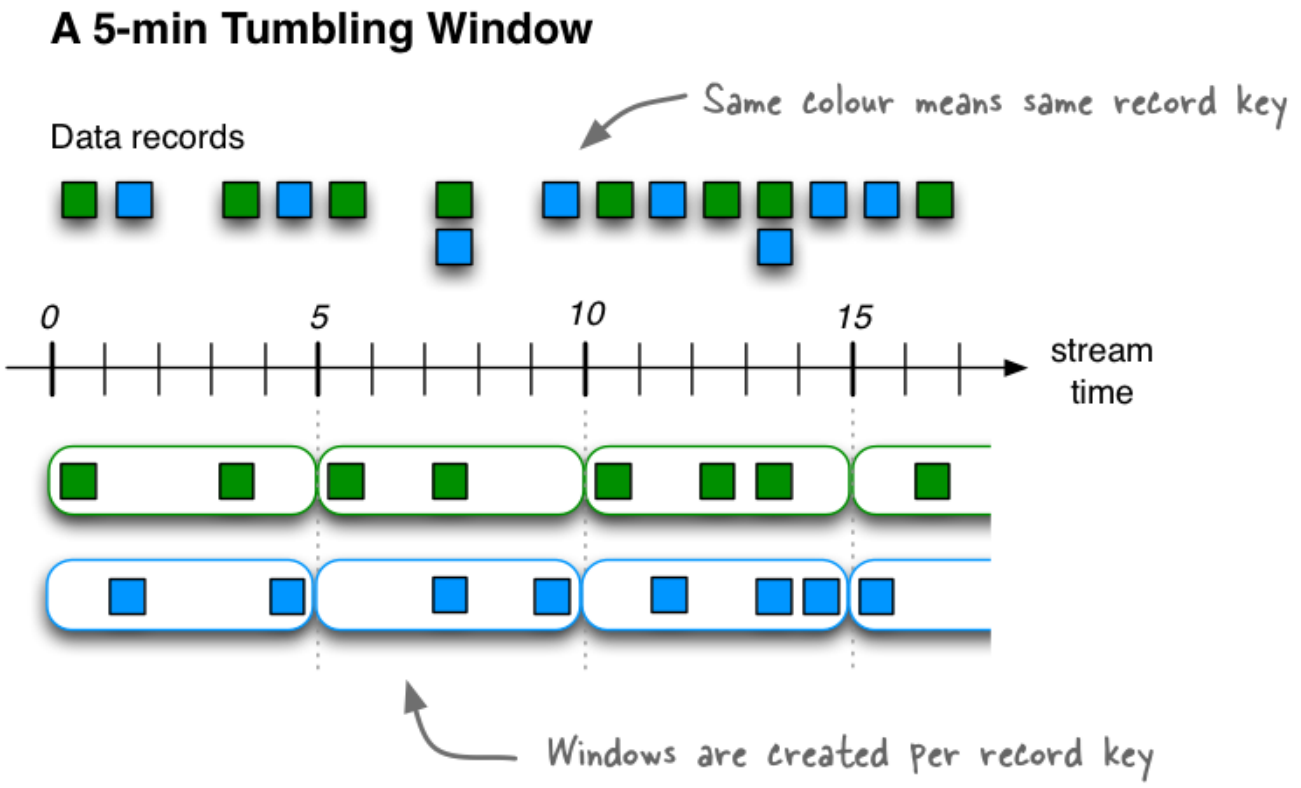
```
groupedStream.aggregate(() -> "0:Zero,",
     (nextKey, nextValue, aggString) ->
        aggString + nextKey + ":" + nextValue ",");
```

Will represent in the RocksDB ephemeral storage:

```
("Even", "0:Zero,2:Even,4:Four")
("Odd", "0:Zero,1:Odd,3:Three")
```

Windowing

Tumbling Windows



Kafka Streams DSL : Tumbling Windows

A tumbling time window with a size of 5 minutes (and, by definition, an implicit advance interval of 5 minutes).

```
import java.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.TimeWindows;

long windowSizeMs = TimeUnit.MINUTES.toMillis(5); // 5 * 60 * 1000L
TimeWindows.of(windowSizeMs);
```

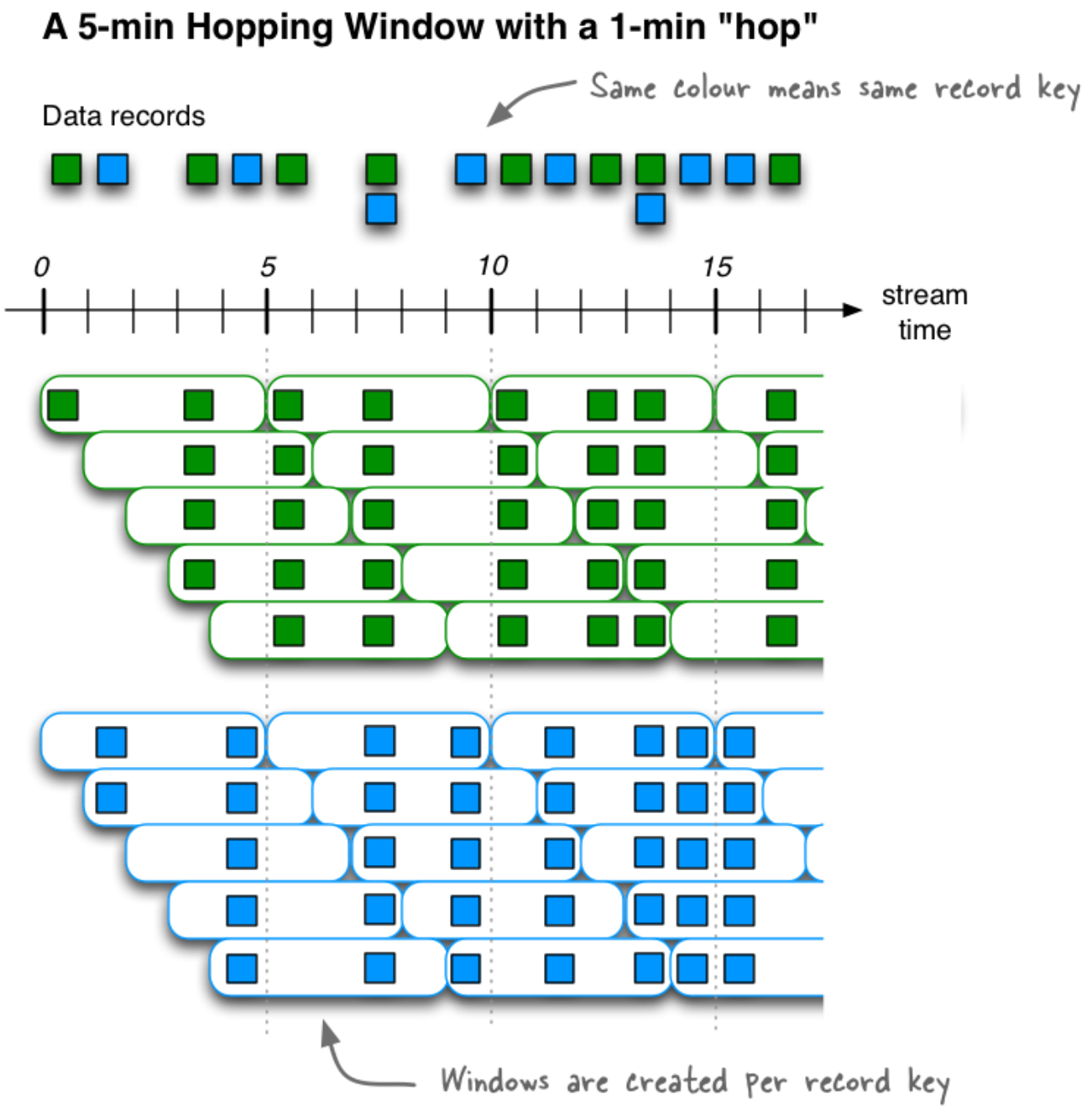
The above is equivalent to the following code:

```
TimeWindows.of(windowSizeMs).advanceBy(windowSizeMs);
```

Plugging into a stream:

```
streams.groupBy(TimeWindows.of(windowSizeMs));
```

Hopping Window



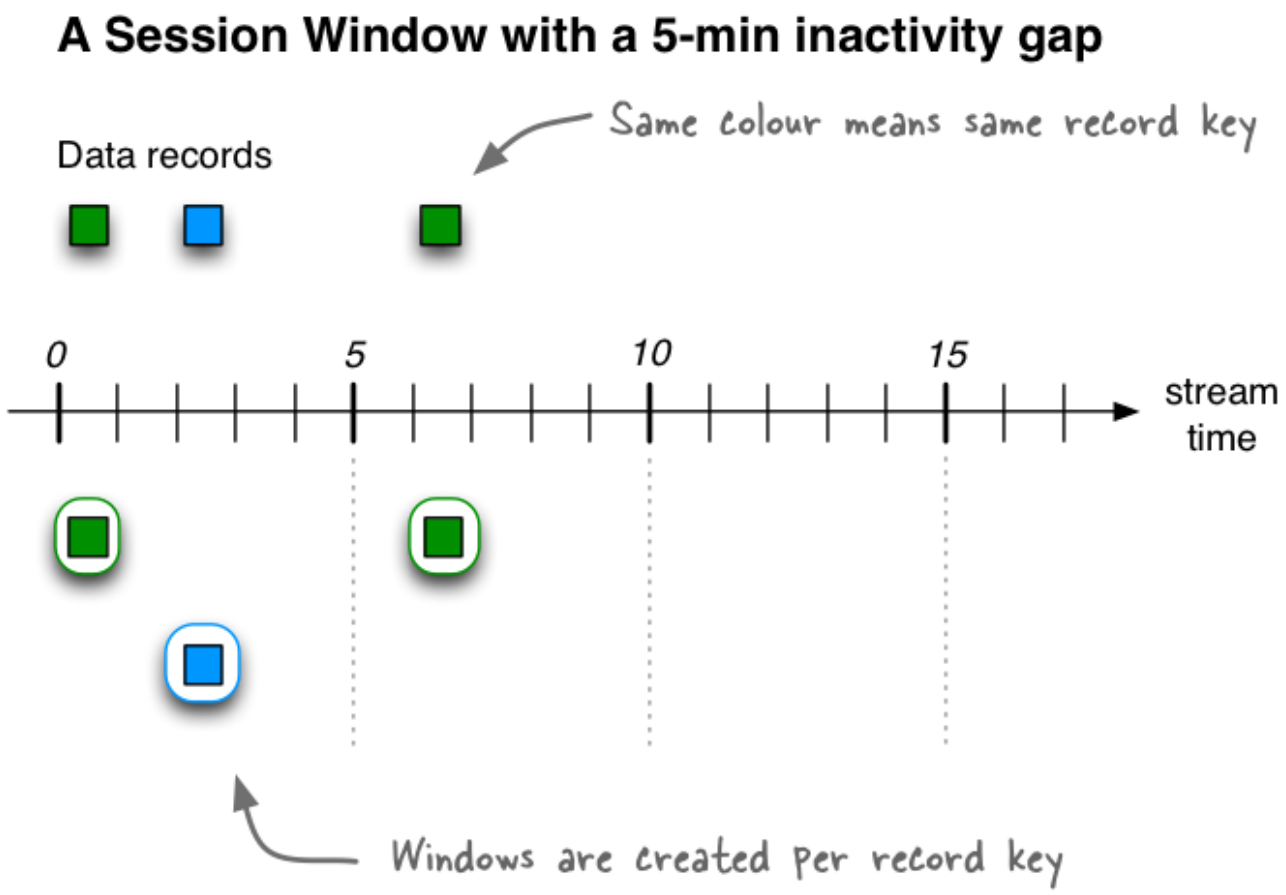
Kafka Streams DSL: Hopping Windows

- A hopping time window with a size of 5 minutes and an advance interval of 1 minute.
- The window's name — the string parameter — is used to e.g. name the backing state store.

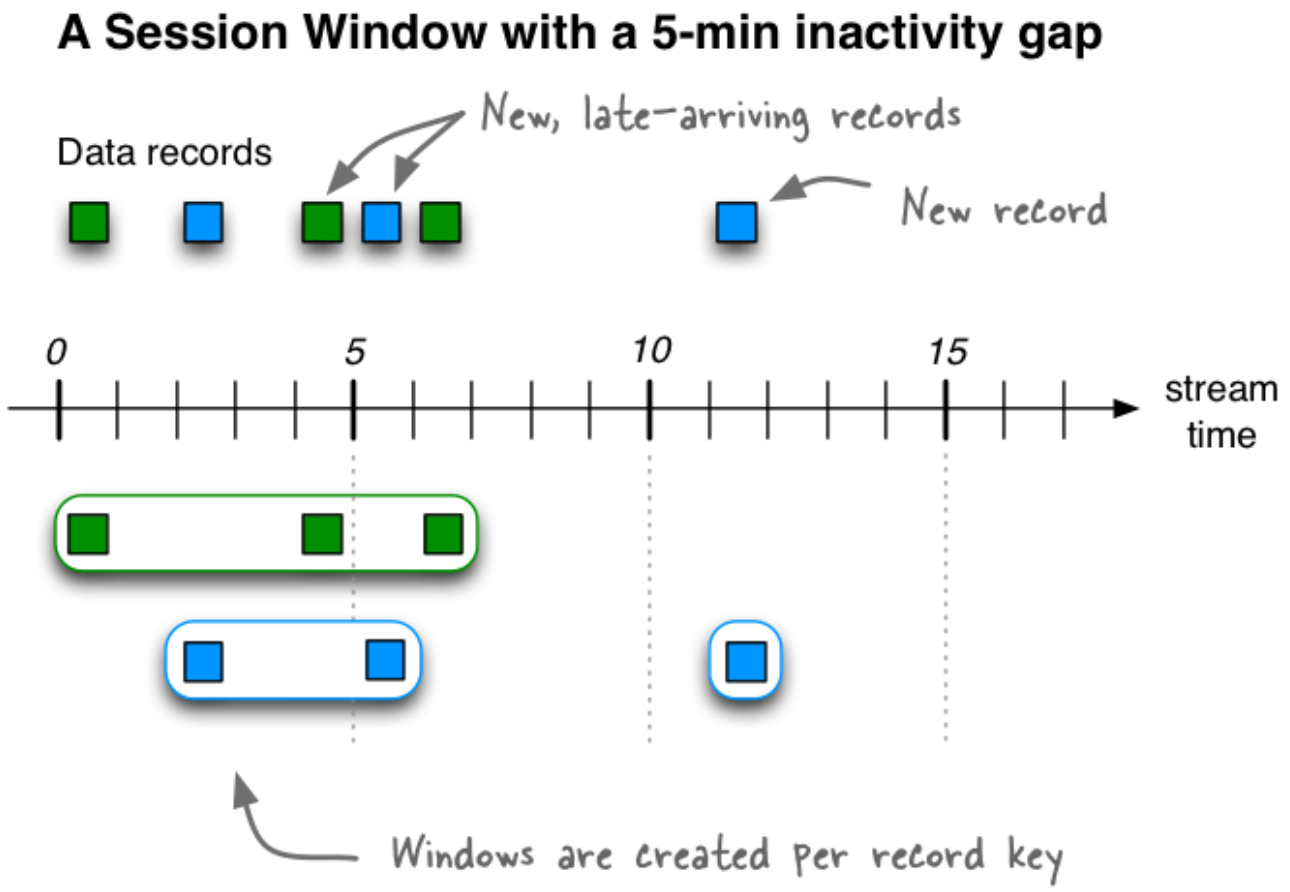
```
import java.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.TimeWindows;

long windowSizeMs = TimeUnit.MINUTES.toMillis(5); // 5 * 60 * 1000L
long advanceMs =    TimeUnit.MINUTES.toMillis(1); // 1 * 60 * 1000L
TimeWindows.of(windowSizeMs).advanceBy(advanceMs);
```

Source: https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html

Session Window



Session Window With More Data



Kafka Streams DSL:

- A session window with an inactivity gap of 5 minutes.

```
import java.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.SessionWindows;

SessionWindows.with(TimeUnit.MINUTES.toMillis(5));
```

Joining Operations and Windowing

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN
KStream-to-KStream	Windowed	Supported	Supported	Supported
KTable-to-KTable	Non-windowed	Supported	Supported	Supported
KStream-to-KTable	Non-windowed	Supported	Supported	Not Supported
KStream-to-GlobalKTable	Non-windowed	Supported	Supported	Not Supported
KTable-to-GlobalKTable	N/A	Not Supported	Not Supported	Not Supported

Kafka Streams Joining

- Given the same paritions for two separate topic you can do a process of co-partitioning
- Consider one topic for `stock_trades` and one for `stock_news`
- Both topics have the same number of partitions
- Therefore keys will distributed across equally in all partitions
- Also therefore, one consumer, or stream application will receive the both the same news and trade price for `IBM` for example

Kafka Streams DSL: `join` between Streams

```
import java.util.concurrent.TimeUnit;
KStream<String, Long> left = ...;
KStream<String, Double> right = ...;

KStream<String, String> joined = left.join(right,
    (leftValue, rightValue) ->
      "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),
    Joined.with(
      Serdes.String(),
      Serdes.Long(),
      Serdes.Double())
);
```

Kafka Streams DSL: `leftJoin` between Streams

```
KStream<String, Long> left = ...;
KStream<String, Double> right = ...;

KStream<String, String> joined = left.leftJoin(right,
    (leftValue, rightValue) ->
      "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),
    Joined.with(
      Serdes.String(),
      Serdes.Long(),
      Serdes.Double())
  );
```

For each input record on the left side that does not have any match on the right side, the

ValueJoiner will be called with `ValueJoiner#apply(leftRecord.value, null)`

Kafka Streams DSL: `outerJoin` between Streams

```
import java.util.concurrent.TimeUnit;
KStream<String, Long> left = ...;
KStream<String, Double> right = ...;

KStream<String, String> joined = left.outerJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ",
       right=" + rightValue,
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),
    Joined.with(
      Serdes.String(),
      Serdes.Long(),
      Serdes.Double())
  );
```

For each input record on one side that does not have any match on the other side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null)` or `ValueJoiner#apply(null, rightRecord.value)`

Stream to Stream and Join Reference

Timestamp	Left (KStream)	Right (KStream)	(INNER) JOIN	LEFT JOIN	OUTER JOIN
1	null	-	-	-	-
2	-	null	-	-	-
3	A	-	-	[A, null]	[A, null]
4	-	a	[A, a]	[A, a]	[A, a]
5	B	-	[B, a]	[B, a]	[B, a]
6	-	b	[A, b], [B, b]	[A, b], [B, b]	[A, b], [B, b]
7	null	-	-	-	-
8	-	null	-	-	-
9	C	-	[C, a], [C, b]	[C, a], [C, b]	[C, a], [C, b]
10	-	c	[A, c], [B, c], [C, c]	[A, c], [B, c], [C, c]	[A, c], [B, c], [C, c]
11	-	null	-	-	-
12	null	-	-	-	-
13	-	null	-	-	-
14	-	d	[A, d], [B, d], [C, d]	[A, d], [B, d], [C, d]	[A, d], [B, d], [C, d]

Timestamp	Left (KStream)	Right (KStream)	(INNER) JOIN	LEFT JOIN	OUTER JOIN
15	D	-	[D, a], [D, b], [D, c], [D, d]	[D, a], [D, b], [D, c], [D, d]	[D, a], [D, b], [D, c], [D, d]

Kafka Table Joining

- Tables are change logs, synonymous with a database table
- No Windowing Required since they are stateful
- Support for inner, left, and outer joins
- The topics must be co-partitioned

Kafka Streams DSL: `join` between KTables

```
KTable<String, Long> left = ...;
KTable<String, Double> right = ...;

KTable<String, String> joined = left.join(right,
    (leftValue, rightValue) -> "left=" + leftValue +
      ", right=" + rightValue
  );
```

Kafka Streams DSL: `leftJoin` between KTables

```
KTable<String, Long> left = ...;
KTable<String, Double> right = ...;

KTable<String, String> joined = left.leftJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue +
      ", right=" + rightValue
);
```

For each input record on the left side that does not have any match on the right side, The `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null);`

Kafka Streams DSL: `outerJoin` between KTables

```
KTable<String, Long> left = ...;
KTable<String, Double> right = ...;

KTable<String, String> joined = left.outerJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue +
      ", right=" + rightValue
);
```

For each input record on one side that does not have any match on the other side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null)` or `ValueJoiner#apply(null, rightRecord.value)`

Table to Table and Join Reference

Timestamp	Left (KTable)	Right (KTable)	(INNER) JOIN	LEFT JOIN	OUTER JOIN
1	null	-	-	-	-
2	-	null	-	-	-
3	A	-	-	[A, null]	[A, null]
4	-	a	[A, a]	[A, a]	[A, a]
5	B	-	[B, a]	[B, a]	[B, a]
6	-	b	[B, b]	[B, b]	[B, b]
7	null	-	null	null	[null, b]
8	-	null	-	-	null
9	C	-	-	[C, null]	[C, null]
10	-	c	[C, c]	[C, c]	[C, c]
11	-	null	null	[C, null]	[C, null]
12	null	-	-	null	null
13	-	null	-	-	-
14	-	d	-	-	[null, d]
15	D	-	[D, d]	[D, d]	[D, d]

Stream-Table Joining

- Tables are change logs, synonymous with a database table
- Streams are flows of data with no storage
- No Windowing Required since one side is stateful
- Support for inner, and left
- The topics must be co-partitioned
- Allow you to perform table lookups against a KTable (changelog stream) upon receiving a new record from the KStream
- There is only a stream to table join relationship

Kafka Streams DSL: `join` between Stream and Table

```
KStream<String, Long> left = ...;
KTable<String, Double> right = ...;

KStream<String, String> joined = left.join(right,
    (leftValue, rightValue) -> "left=" + leftValue +
       ", right=" + rightValue,
    Joined.keySerde(Serdes.String())
      .withValueSerde(Serdes.Long())
  );
```

Kafka Streams DSL: `leftJoin` between Stream and Table

```
KStream<String, Long> left = ...;
KTable<String, Double> right = ...;

// Java 8+ example, using lambda expressions
KStream<String, String> joined = left.leftJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ",
      right=" + rightValue,
    Joined.keySerde(Serdes.String())
      .withValueSerde(Serdes.Long())
  );
```

For each input record on the left side that does not have any match on the right side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null);`

Stream to Table Join Reference

Timestamp	Left (KStream)	Right (KTable)	(INNER) JOIN	LEFT JOIN	1
null	-	-	-	2	-
null	-	-	3	A	-
-	[A, null]	4	-	a	-
-	5	B	-	[B, a]	[B, a]
6	-	b	-	-	7
null	-	-	-	8	-
null	-	-	9	C	-
-	[C, null]	10	-	c	-
-	11	-	null	-	-

12	null	-	-	-	13
-	null	-	-	14	-
d	-	-	15	D	-

Converting a `KTable` to `KStream`

Once established as a `KTable`, it is required to convert to a `KStream` before placing it into a topic

With `to`:

```
ktable.toStream().to("end_topic");
```

With `through`:

```
ktable.toStream().to("end_topic");
```

Lab: Writing a Streams Application

Lab: Writing a Streams Application

Step 1: Create a Streams application with the `application.id` named 'my_streams_app` in a class named `MyStreams` in the `com.xyzcorp` package.

Step 2: Have the `MyStreams` application attach to the `my_orders` topic and takes all information that has the key for your home state, or if you are out of the United States, choose a state that either you visited or would like to visit and put into another topic called `priority_state_orders`

Step 3: Turn on `MyProducer` and run `kafka_console_consumer` that listens to `priority_state_orders`. You may also want to shorten the sleep time so that more orders come in. It may be awhile before your state shows up.

```
./bin/kafka-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic priority_state_orders --from-beginning \
  --key-deserializer
org.apache.kafka.common.serialization.StringDeserializer \
  --value-deserializer
org.apache.kafka.common.serialization.IntegerDeserializer \
  --property print.key=true \
  --property key.separator=,
```

Step 4: Bonus, extend your application to aggregate the total count of all orders per state using the same topology using a `KTable`. Convert the table to a `KStream` and send it to `state_orders_count`

Step 6: Run `kafka_console_consumer` that listens to `state_orders_count` and ensure that you are listening to the total.

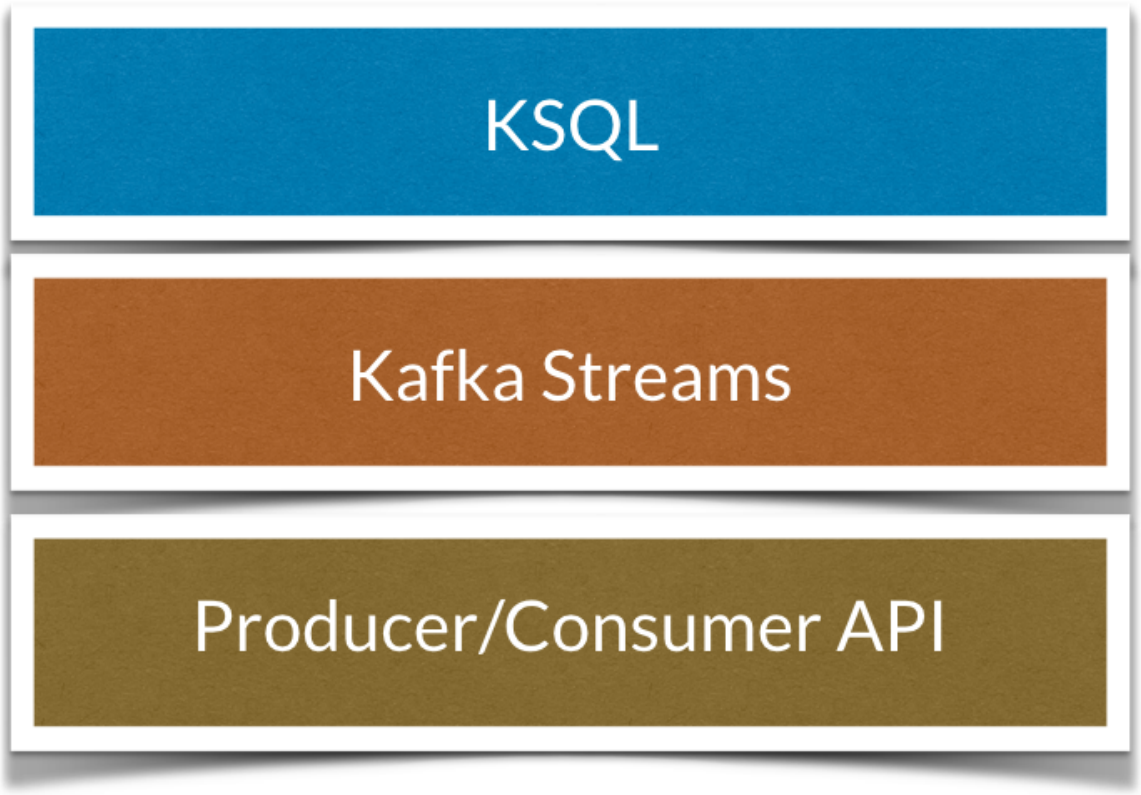
Step 7: Run 3 different instances of the KafkaStreams application.

KSQL

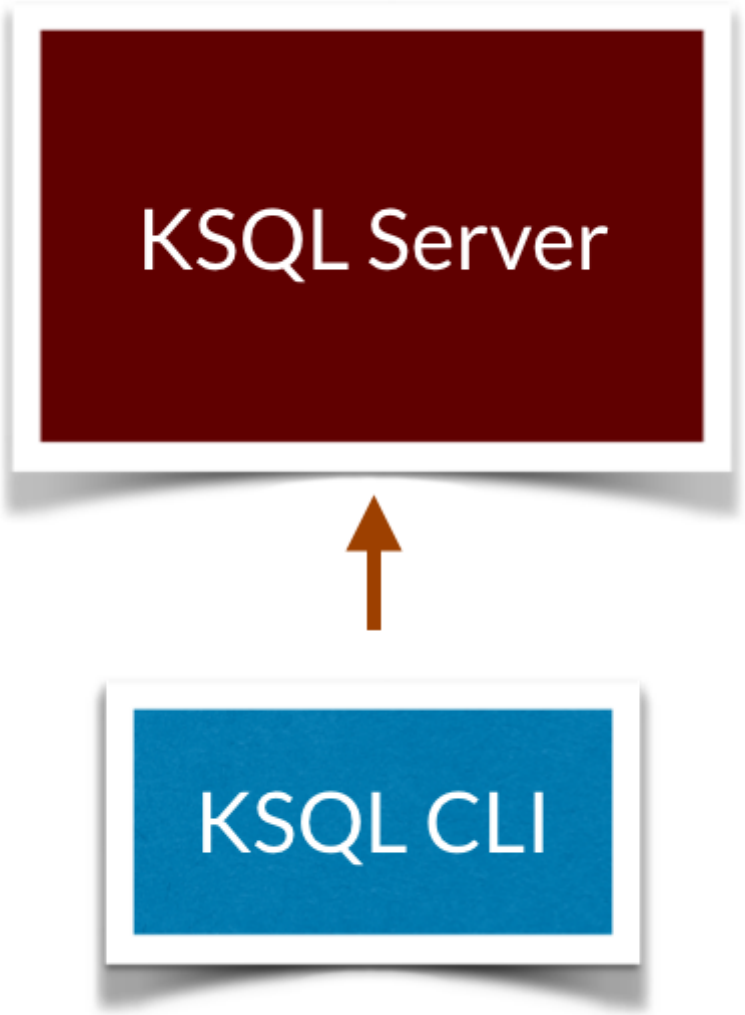
About KSQL

- Open source streaming SQL engine for Apache Kafka
- Provides SQL interface for stream processing on Kafka
- Scalable, elastic, fault-tolerant, and real-time.
- Supports a wide range of streaming operations, including:
  - Data filtering
  - Transformations
  - Aggregations
  - Joins
  - Windowing
  - Sessionization.

Layers of Abstractions



Infrastructure of KSQL



- KSQL Server communicates with the broker
- KSQL servers are run separately from the KSQL CLI client and Kafka brokers.
- You can deploy servers on remote machines, VMs, or containers
- KSQL CLI interacts with KSQL Servers via a REST API
- KSQL Server runs in two modes
  - CLI
  - Headless

Starting in CLI Mode

- Starting the CLI Client

```
/bin/ksql http://localhost:8088
```

Starting in Headless Mode

For production, run in headless mode:

```
/bin/ksql-start-server server_path/etc/ksql/ksql-server.properties \
  --queries-file /path/to/queries.sql
```

Varying ways to create a `STREAM`

- Use the `CREATE STREAM` statement to create a stream from a Kafka topic.
- Use the `CREATE STREAM AS SELECT` statement to create a query stream from an existing topic

Varying ways to create a `TABLE`

- Use the `CREATE TABLE` statement to create a table from a Kafka topic.
- Use the `CREATE STREAM AS SELECT` statement to create a table with results from an existing topic

Creating an initial stream from a topic

```
CREATE STREAM pageviews \
  (viewtime BIGINT, \
   userid VARCHAR, \
   pageid VARCHAR) \
  WITH (KAFKA_TOPIC='pageviews', \
        VALUE_FORMAT='DELIMITED');
```

- Stream and `KAFKA_TOPIC` will be the same name for the initial topic
- `VALUE_FORMAT` can be `DELIMITED`, `JSON`, or `AVRO`
- KSQL adds the implicit columns `ROWTIME` and `ROWKEY` to every stream and table
  - Represents the corresponding Kafka message timestamp and message key
  - The timestamp has milliseconds accuracy.

Creating an initial table (changelog) from a topic

```
CREATE TABLE users \
  (registertime BIGINT, \
   userid VARCHAR, \
   gender VARCHAR, \
   regionid VARCHAR) \
  WITH (KAFKA_TOPIC = 'users', \
        VALUE_FORMAT='JSON', \
        KEY = 'userid');
```

Ensuring that Stream is created

Call the following:

```
SHOW STREAMS;
```

And get the result:

```
Stream Name | Kafka Topic | Format
-----------------------------------------
PAGEVIEWS   | pageviews   | DELIMITED
-----------------------------------------
```

Supported Column Types

In an effort to bring KSQL as close to SQL than to functional programming, the column types should be familiar for SQL Developers and Administrators

- `BOOLEAN`
- `INTEGER`
- `BIGINT`
- `DOUBLE`
- `VARCHAR` (or `STRING`)
- `ARRAY<ArrayType>` (JSON and AVRO only. Index starts from 0)
- `MAP<VARCHAR, ValueType>` (JSON and AVRO only)
- `STRUCT<FieldName FieldType, …>` (JSON and AVRO only)

The STRUCT type requires you to specify a list of fields. For each field you must specify the field name (FieldName) and field type (FieldType).

Various Operators

- Scalar Functions

`ABS`, `ARRAYCONTAINS`, `CEIL`, `CONCAT`, `EXTRACTJSONFIELD`, `FLOOR`, `GEO_DISTANCE`, `IFNULL`, `LCASE`, `LEN`, `MASK`, `RANDOM`, `ROUND`, `STRINGTOTIMESTAMP`, `SUBSTRING`, `TIMESTAMPTOSTRING`, `TRIM`, `UCASE`

- Aggregate Functions (Requires Grouping)

`COUNT`, `MAX`, `MIN`, `SUM`, `TOPK`, `TOPKDISTINCT`, `WindowStart`, `WindowEnd`

Displaying information

Print the contents of Kafka topics to the KSQL CLI.

```
PRINT <topic>
```

Show topics available, can use `LIST`

```
SHOW <topic>
```

Show streams and tables created

```
SHOW | LIST STREAMS;
SHOW | LIST TABLES;
```

Show available functions

```
SHOW | LIST FUNCTIONS;
```

Show all running persistent queries

```
SHOW QUERIES;
```

Describing Streams or Tables

`DESCRIBE`: List the columns in a stream or table along with their data type and other attributes.

`DESCRIBE EXTENDED`: Display DESCRIBE information with additional runtime statistics, Kafka topic details, and the set of queries that populate the table or stream

```
DESCRIBE [EXTENDED] (stream_name|table_name);
```

Persistent Query

The following will be sent to a new topic and will run continuously after invoked

- What makes this a persistent query is the form `CREATE (TABLE|STREAM) AS SELECT`
- Once executed, it will continuously run, until terminated with `TERMINATE` command

```
CREATE TABLE users_female AS \
  SELECT userid, gender, regionid FROM users \
  WHERE gender='FEMALE';
```

Non Persistent Query

The following will not be sent to a topic and merely be shown in the CLI

```
SELECT * FROM pageviews
  WHERE ROWTIME >= 1510923225000
    AND ROWTIME <= 1510923228000;
```

A `LIMIT` can be used to limit the number of rows returned. Once the limit is reached the query will terminate.

Persistent vs. Non Persistent

Persistent	Non Persistent
Will store the results into a topic	Will not store results into topic
Use `TERMINATE` to stop query	Use **CTRL+C**
`CREATE STREAM AS SELECT …` or `CREATE TABLE AS SELECT`	`SELECT …`

Selecting a key from list of fields

```
CREATE STREAM pageviews \
  (viewtime BIGINT, \
   userid VARCHAR, \
   pageid VARCHAR) \
 WITH (KAFKA_TOPIC='pageviews', \
       VALUE_FORMAT='DELIMITED', \
       KEY='pageid');
```

Selecting customized fields for new stream

In the following:

- We are stream as a topic with JSON valued content
- We are creating a number of partitions of `5`
- `PARTITION BY` takes the field `userid` and makes it the key.

```
CREATE STREAM pageviews_transformed \
  WITH (TIMESTAMP='viewtime', \
        PARTITIONS=5, \
        VALUE_FORMAT='JSON') AS \
  SELECT viewtime, \
         userid, \
         pageid, \
         TIMESTAMPTOSTRING(viewtime, 'yyyy-MM-dd HH:mm:ss.SSS') AS
timestring \
  FROM pageviews \
  PARTITION BY userid;
```

Setting KSQL to read from the beginning

```
SET 'auto.offset.reset' = 'earliest';
```

Tumbling Window

```
SELECT item_id, SUM(quantity)
  FROM orders
  WINDOW TUMBLING (SIZE 20 SECONDS)
  GROUP BY item_id;
```

Hopping Window

```
SELECT item_id, SUM(quantity)
  FROM orders
  WINDOW HOPPING (SIZE 20 SECONDS, ADVANCE BY 5 SECONDS)
  GROUP BY item_id;
```

Session Windows

```
SELECT item_id, SUM(quantity)
  FROM orders
  WINDOW SESSION (20 SECONDS)
  GROUP BY item_id;
```

Lab: Using KSQL

Lab: Using KSQL

Step 1: Run a ksql terminal that will attach to the KSQL Server using

```
$ ksql http://localhost:8088
```

Step 2: In the KSQL CLI, create an initial stream from the `my_avro_orders` topic using the following

```
CREATE STREAM my_avro_orders \
(total BIGINT, shipping VARCHAR, state VARCHAR, discount DOUBLE, \
gender VARCHAR) WITH (kafka_topic='my_avro_orders',
value_format='AVRO');
```

Step 3: Ensure that your stream is available when you call `SHOW STREAMS:`

Step 4: In KSQL, add `SET 'auto.offset.reset'='earliest';` so that you can start processing all information.

Step 5: Give ksql for a spin with a non-persistent query using `SELECT * from my_avro_orders;`

Lab: Using KSQL (Continued)

Step 6: Using KSQL, create another non-persistent query that selects all females from `my_avro_orders` from the state of Vermont

Step 7: Using KSQL, create a non-persistent query that selects all next day shipping orders and groups by state.

Step 8: Using KSQL, after you ensure that the non-persistent query works, turn it into a persistent *table* query and persist the content into a topic called `next_day_shipping_by_state`

Step 9: On a new terminal, enter the following at the command line in the *confluent-5.0.0* directory

```
./bin/kafka-avro-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic next_day_shipping_by_state \
  --from-beginning \
  --key-deserializer
org.apache.kafka.common.serialization.StringDeserializer \
  --property print.key=true \
  --property key.separator=,
```

Step 10: `TERMINATE` the query, exit ksql

Step 11: Stop all producers and consumers that are currently running

Step 12: Shut down all system processes by executing from the `confluent-5.0.0` folder:

```
$ ./bin/confluent destroy
```

Glossary

Glossary

Latency - The time required to perform some action or to produce some result. Latency is measured in units of time; hours, minutes, seconds, nanoseconds or clock periods.

Throughput - The number of such actions executed or results produced per unit of time. This is measured in units of whatever is being produced, like messages

Ephermeral Drives or Storage - Extra storage add to handle larger loads. One example is the *Elastic Block Storage* on Amazon Web Services

Page Cache - Referred to as a disk cache. Unused memory holds cache of items for faster access and increases performance

Swap Space - Swap space in Linux is used when the amount of physical memory (RAM) is full. If the system needs more memory resources and the RAM is full, inactive pages in memory are moved to the swap space which is on the hard drive.

Dirty Pages - Pages in the page cache that have been updated in memory but still is yet to be updated in the cache

Replication Factor - Number of other servers that should hold your data in case of failure. Typically the it is

Failover - Method of protecting a computer system from failure

Glossary (Continued)

Multicasting — Send data across a computer network to several users at the same time.

Transient Error — Error that will be resolved soon

Heartbeat — Periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a computer system.