CSP554
BIG DATA TECHNOLOGIES

Module 09

High Velocity Data Processing

Apache Kafka

High Level View

What is High Velocity Data Processing?

- AKA Streaming or Stream Processing
- So far we have focused on batch processing of data file
  - Sitting in HDFS, queried with Hive, transformed with Pig & Spark
  - Using Sqoop as an ingestion tool… sitting in databases
- But how does new data get into Hadoop at the time it is generated?
  - New log entries from your web servers
  - New sensor data from you IoT systems
  - New transactions form oyur PoS systems
- Stream processing lets you accept this data from external systems into your Hadoop cluster in real time
  - And you can even process it in real time on its arrival

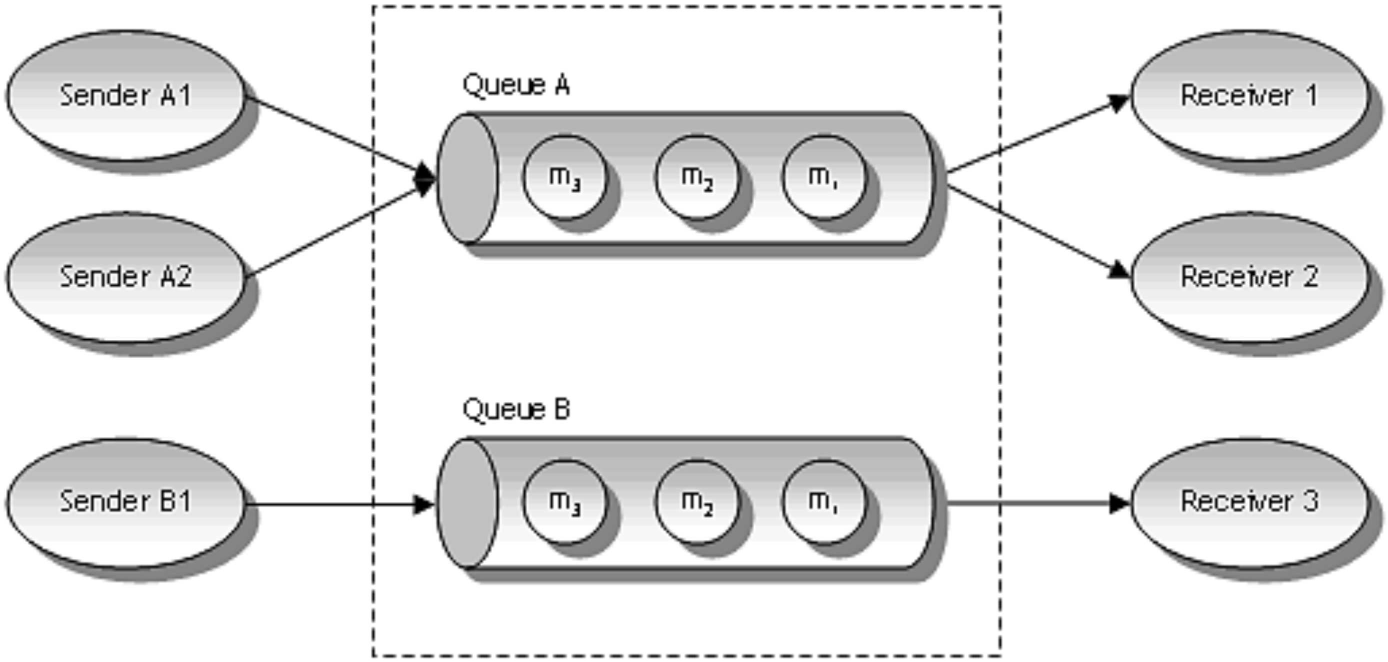
Two Aspects of Stream Processing

- Accepting data from different sources into Hadoop
  - Apache Kafka
- Processing the data ASAP when it gets into there
  - Spark Structured Streaming

Key Concepts

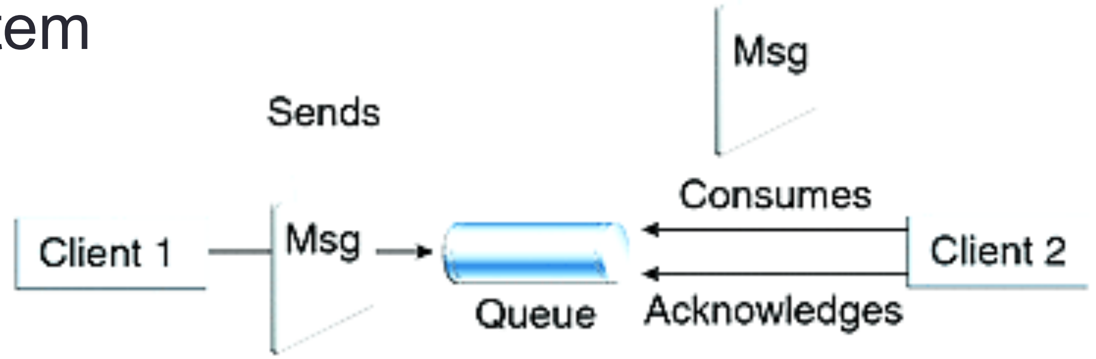
What is a Messaging System?

- Responsible for transferring data from one application to another
- So the applications can focus on data and not worry about how to share it
- Distributed messaging is based on the concept of reliable message queuing

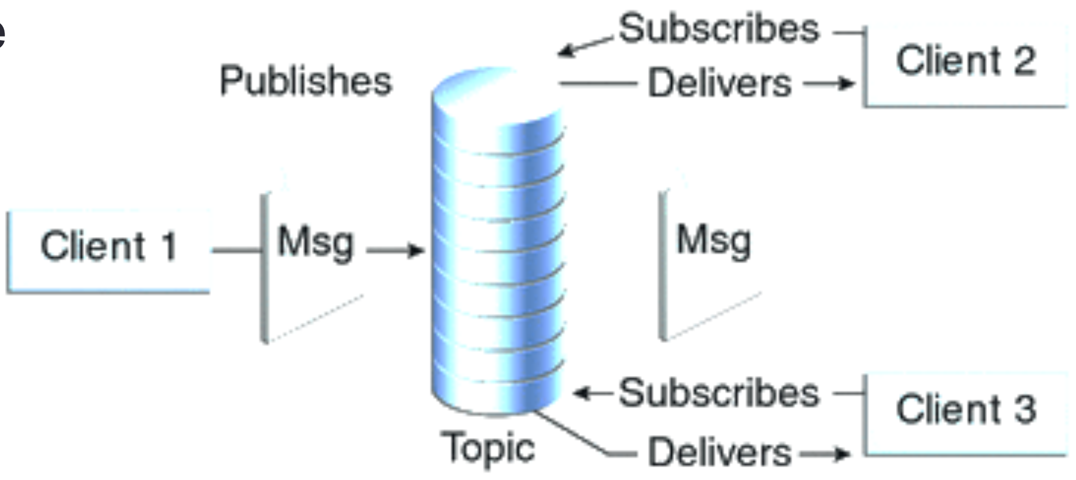


What is a Messaging System?

- Two types of messaging patterns are possible in a messaging system
  - Point to point

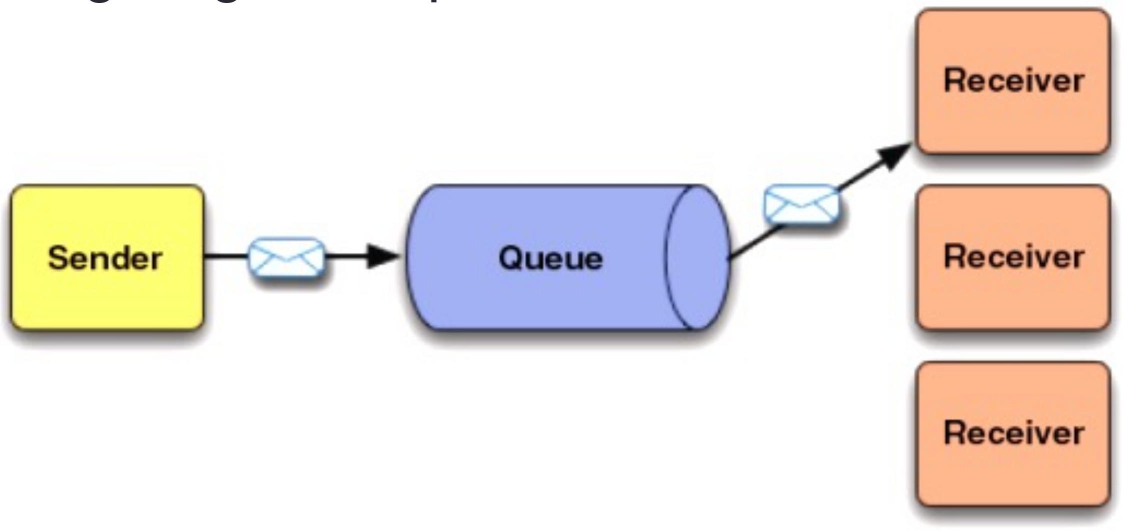


  - Publish-subscribe



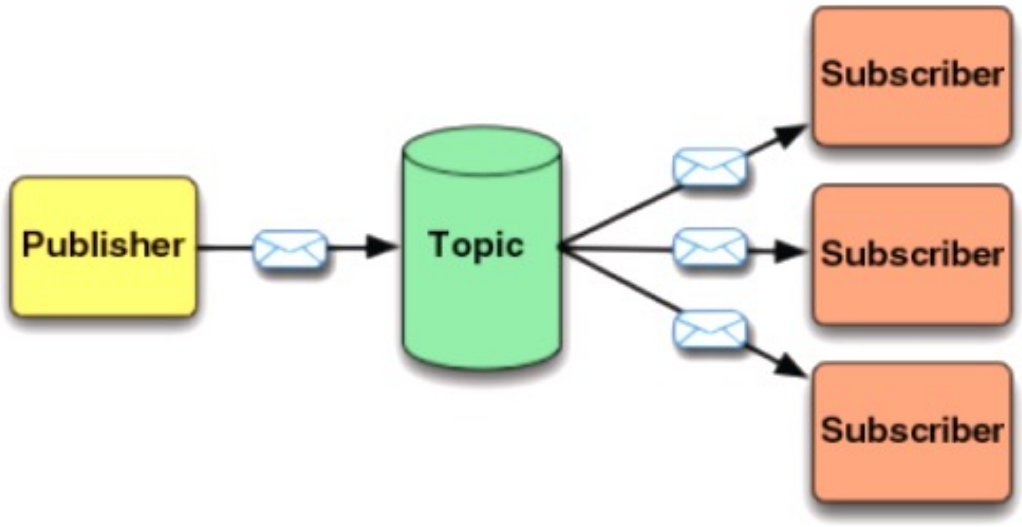
Point to Point Messaging System

- In a point-to-point system, messages are persisted in a queue
- One or more consumers can consume messages
- But each message can be consumed by one consumer only
- Once a consumer reads a message it disappears from that queue
- The following diagram depicts the structure



Publish-Subscribe Messaging System

- In the publish-subscribe system, messages are persisted in one or more topics (think file folders or feeds)
- Unlike point-to-point system, consumers can subscribe to one or more topics and consume all the messages in that topic
- Here message producers are called publishers and message consumers are called subscribers
- The following diagram depicts the structure

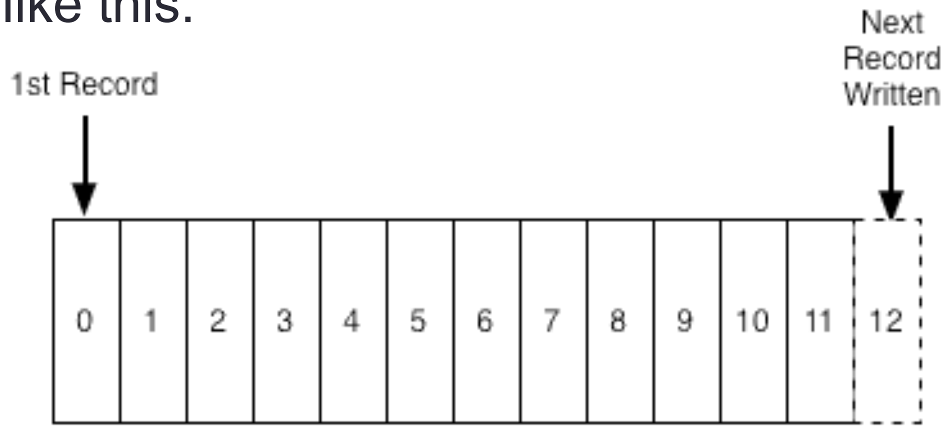


Publish-Subscribe Messaging System

- One example of this model is Twitter
- A publisher established an account (a topic)
- Multiple subscriber then indicate interest in (subscribe) receiving tweets from that account
- The publisher then enters (publish) one or more tweets (message)
- Each tweet (message) is then made available to every subscriber of that account (topic)

What Is a Log?

- A log is perhaps the simplest possible storage abstraction
- It is an append-only, totally-ordered sequence of records ordered by time. It looks like this:



- Records are appended to the end of the log, and reads proceed left-to-right
- Each entry is assigned a unique sequential log entry number
- The ordering of records defines a notion of "time"
- Since entries to the left are defined to be older then entries to the right
- Log entry number can be thought of as the "timestamp" of the entry

What Is a Log?

- The ordering of records defines a notion of "time" since entries to the left are defined to be older then entries to the right. The log entry number can be thought of as the "timestamp" of the entry.

Logs in Databases (Commit Logs)

- The most crucial structure for database recovery operations
- Store all of the changes made to the database as they occur
- Only reliable source of information about current state of the database
- Used to recover consistent state of a database, upon failure

Detailed View

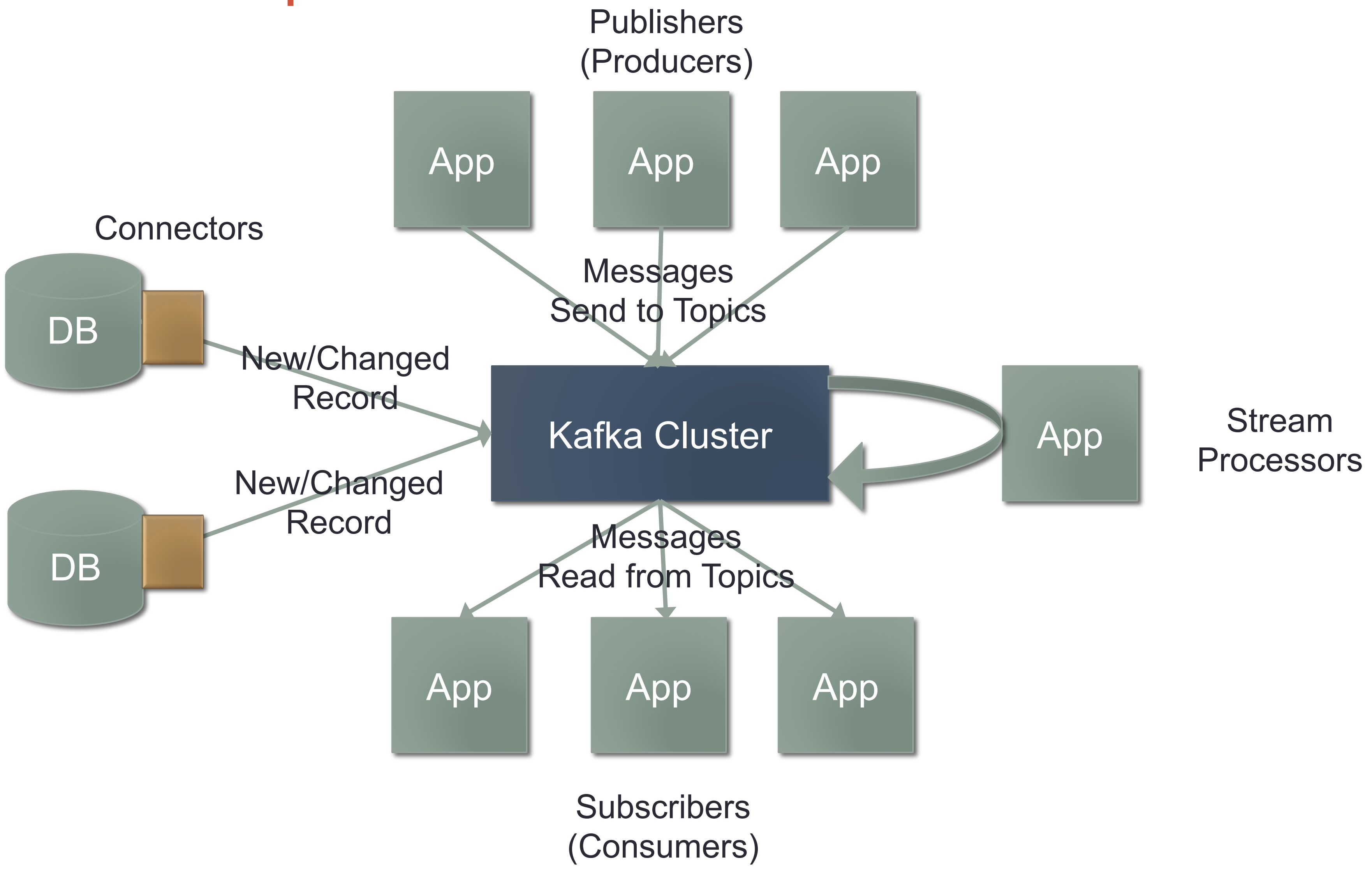
Accepting Data into Hadoop in Real Time Using Kafka

- Kafka is a general purpose publish and subscribe messaging system
- Kafka servers/nodes accept and store incoming messages from data sources
- Data sources are referred to as data publishers
- Kafka stores messages into groupings as topics
- Message consumers, known as subscribers, register interest with Kafka about one or more topics
- When data is stored in a topic it is made available to that topic's subscribers

Accepting Data into Hadoop in Real Time Using Kafka

- Kafka systems may be used by apps executing on Hadoop
- But Kafka can also be used generally as a very high reliability and high performance message processing system

Kafka Architecture
Initial Perspective



Apache Kafka

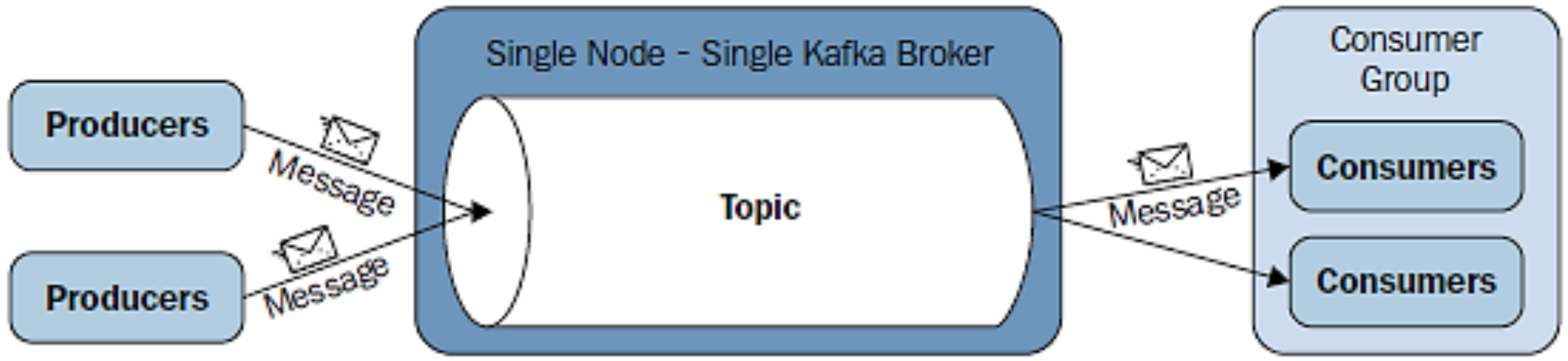
- A distributed publish and subscribe messaging system rethought as a distributed commit log
  - More on “commit log” and “publish subscribe” in a bit
- Developed at LinkedIn
- Provides a solution to handling the ingestion of high volume and high velocity data into Hadoop clusters

Apache Kafka

- Apache Kafka is fast becoming the preferred messaging infrastructure for dealing with contemporary, data-centric workloads such as Internet of Things, gaming and online advertising
- The ability to ingest data at a lightening speed makes it an ideal choice for building complex data processing pipelines
- Kafka is designed from the ground up to deal with transporting millions of high velocity events generated in rapid succession
- It guarantees low latency and “at-least-once”, delivery of messages to consumers
- Kafka also supports retention of data for offline consumers
- Which means that the data can be processed either in real-time or in offline/batch mode

Apache Kafka

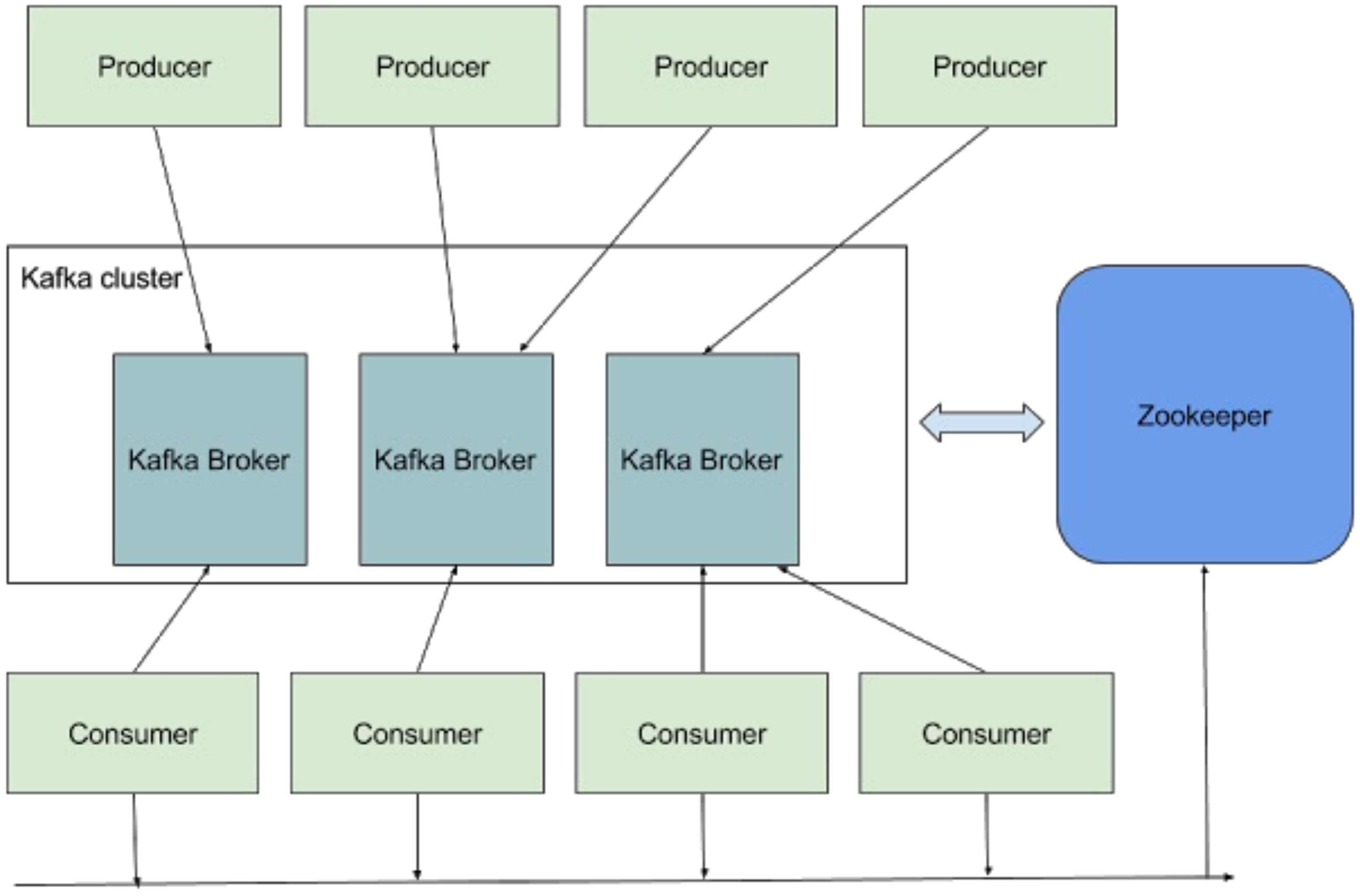
- A producer publishes messages to a Kafka topic
- Topics represent a logical collection of messages
- Consumers subscribe to the Kafka topics to get the messages
- Each consumer is a process or thread and these are organized into consumer groups



Why Kafka?

Feature	Description
High Throughput	Support for up to millions of messages/s with relatively modest cluster computing hardware
Scalability	Horizontally scalable with no down time
Replication	Messages can be replicated across a cluster which provides load balancing and failure support
Durability	Provides support for persistence of messages even across network, disk and server failures
Integration	Kafka can be used with real time streaming systems like Spark Streaming and Storm

Kafka Architectural Landscape



Messages and Batches

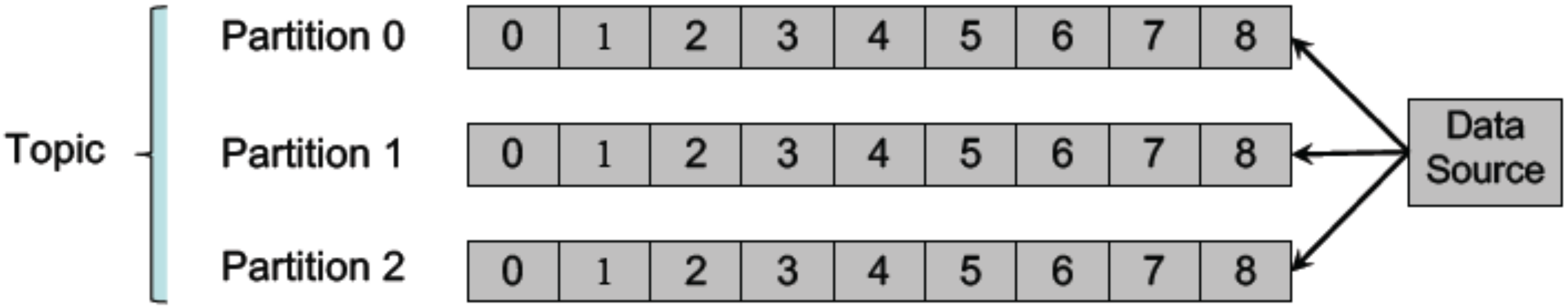
- The unit of data within Kafka is called a *message*
- If you are approaching Kafka from a database background, you can think of this as similar to a *row* or a *record*
- A message is simply an array of bytes, as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka
- Messages can have an optional bit of metadata which is referred to as a key
- The key is also a byte array, and as with the message, has no specific meaning to Kafka
- Keys are used when messages are to be written to partitions in a more controlled manner.

Messages and Batches

- For efficiency, messages are written into Kafka in batches
- A *batch* is just a collection of messages, all of which are being produced to the same topic and partition
- An individual round trip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this
- This, of course, presents a tradeoff between latency and throughput…
- The larger the batches, the more messages that can be handled per unit of time…
- But the longer it takes an individual message to propagate
- Batches are also typically compressed, which provides for more efficient data transfer and storage at the cost of some processing power

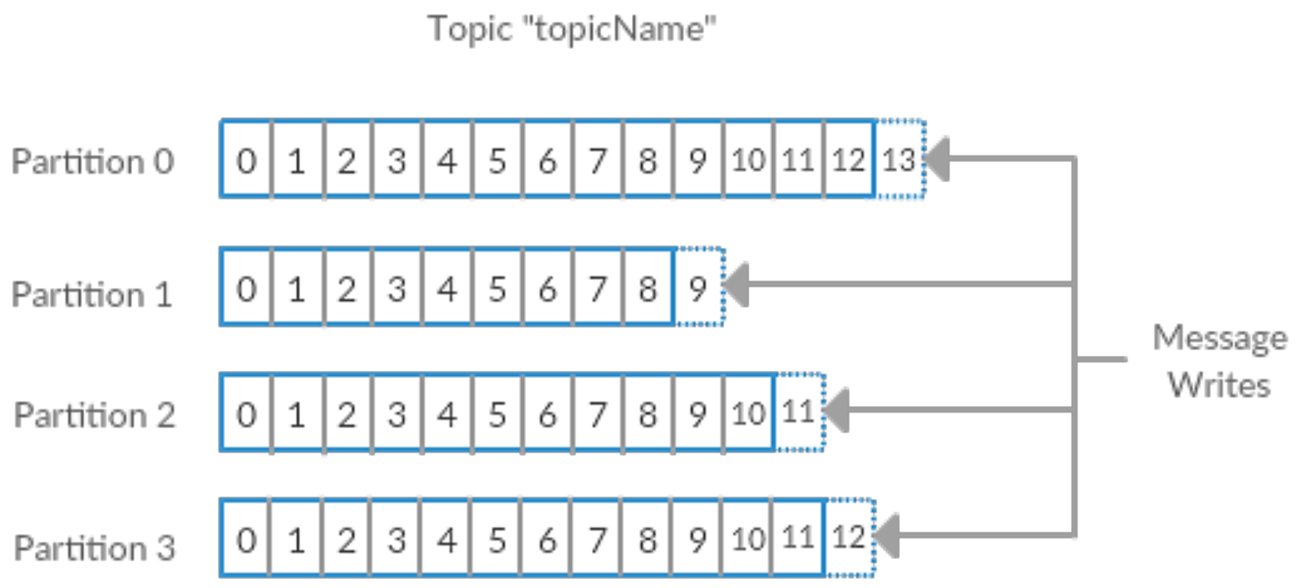
Topics and Logs

- Messages are organized into topics, and each topic is split into partitions
- Each partition is an immutable, time-sequenced log of messages on disk
- Note that time ordering is guaranteed within, but not across, partitions



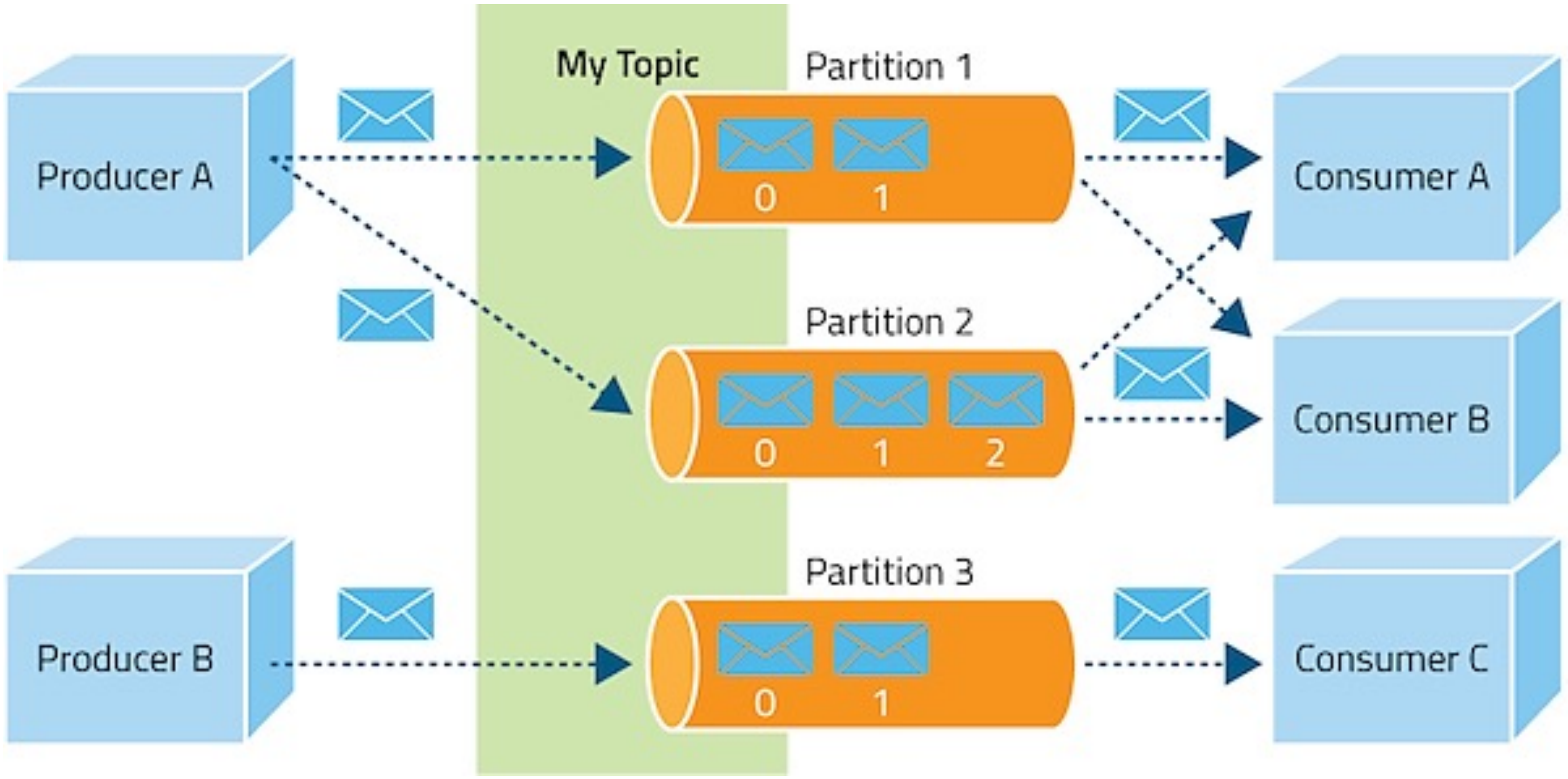
Topics and Logs

- Here we show a topic with 4 partitions, with writes being appended to the end of each one
- Partitions are also the way Kafka provides redundancy and scalability
- Each partition can be hosted on a different server
- This means that a single topic can be scaled horizontally across multiple servers…
- To provide for performance far beyond the ability of a single server



Topics and Logs
Architectural Detail

- .



Topics and Logs

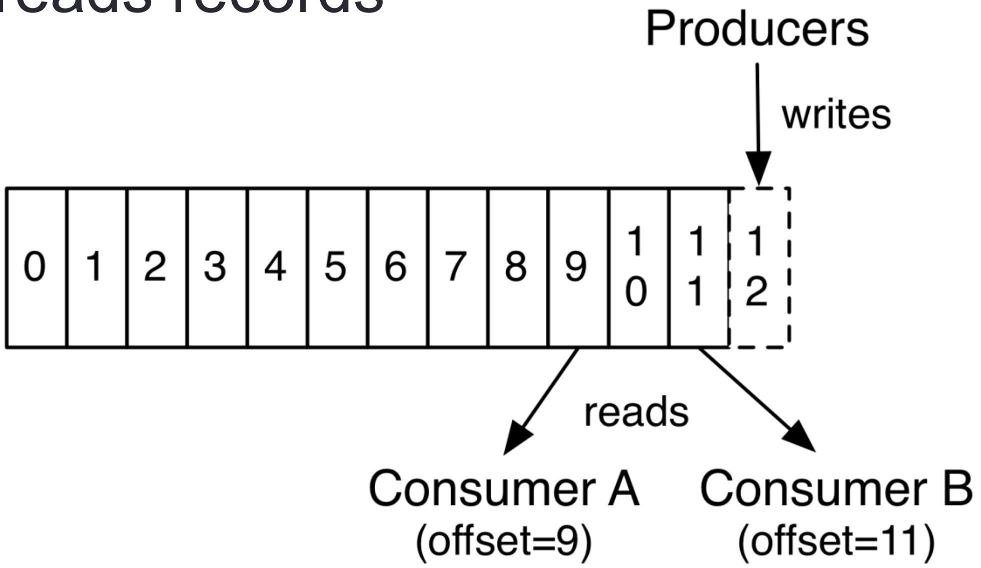
- Each partition is an ordered, immutable sequence of records that is continually appended to—a structured commit log
- The records in the partitions are each assigned a sequential id number called the *offset* which uniquely identifies each record within the partition
- The cluster retains all published records, whether or not they have been consumed using a configurable retention period
- For example, if the retention policy is set to two days…
- Then for the two days after a record is published, it is available for consumption after which it will be dropped to free up space
- Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem

Topics and Logs

- The partitions in the log serve several purposes…
- First, they allow the log to scale beyond a size that will fit on a single server
- Each individual partition must fit on the server that hosts it
- But a topic may have many partitions (distributed across multiple servers)
- This allows a topic to handle an arbitrary amount of data
- Second partitions act as the unit of parallelism—more on that in a bit

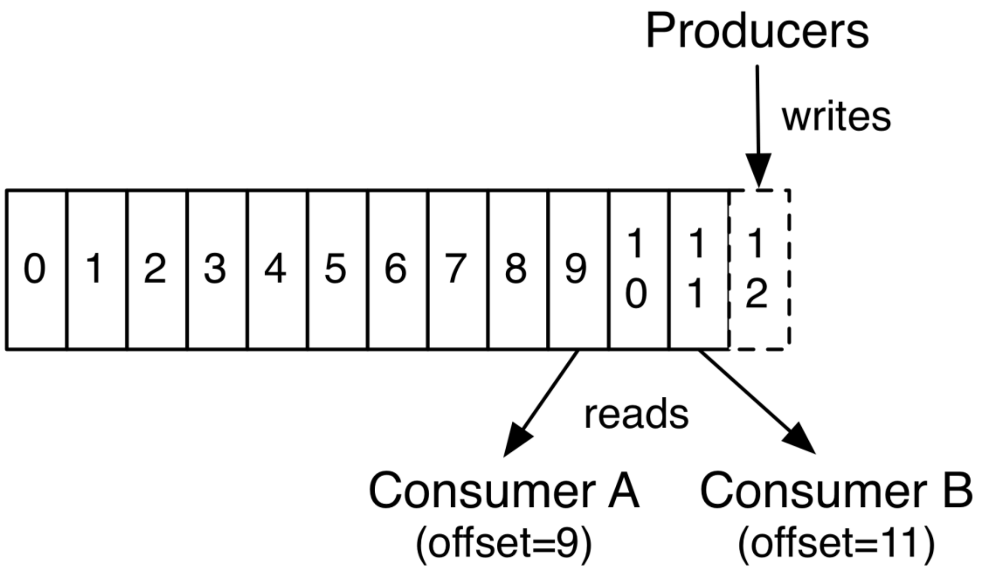
Topics and Logs

- The only metadata retained on a per-consumer basis is the offset or position of that consumer in the log
- This offset is controlled by the consumer and saved in a special Kafka topic
- Normally a consumer will advance its offset linearly to the next as it reads records



Topics and Logs

- But, in fact, since the position is controlled by the consumer it can consume records in any order it likes
- For example a consumer can reset to an older offset to reprocess data from the past
- Or the consumer can skip ahead to the most recent record and start consuming from "now"



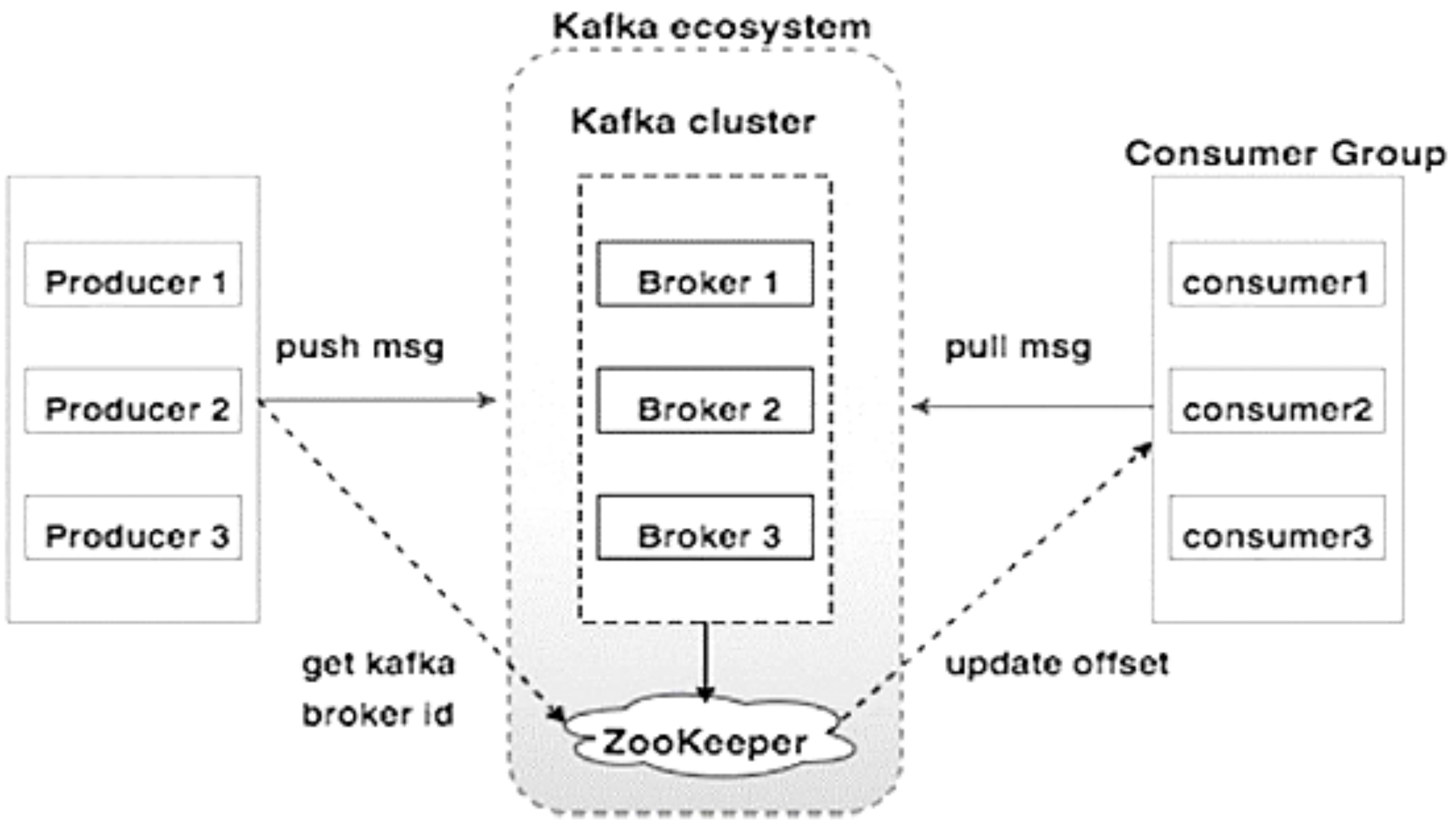
Topics and Logs

- So Kafka does not need complex logic to provide the consumers with their next message
- This means that Kafka consumers are very cheap…
- They can come and go without much impact on the cluster or on other consumers

Brokers and Clusters

- Each Kafka instance belonging to a cluster is called a broker
- The brokers primary responsibilities are…
  - To receive messages from producers
  - To assign offsets to these messages
  - To commit these messages to persistent storage
- Each broker can easily handle thousands of partitions and millions of messages per second.
- The partitions in a topic may be distributed across multiple brokers
- This redundancy ensures high availability of messages

Brokers and Clusters
Architectural Detail



Brokers and Clusters

- Within a cluster of brokers, one will always function as the cluster *controller*
- This controller is elected automatically from the live broker members of the cluster
- The controller is responsible for administrative operations, including assigning partitions to brokers and monitoring for broker failures

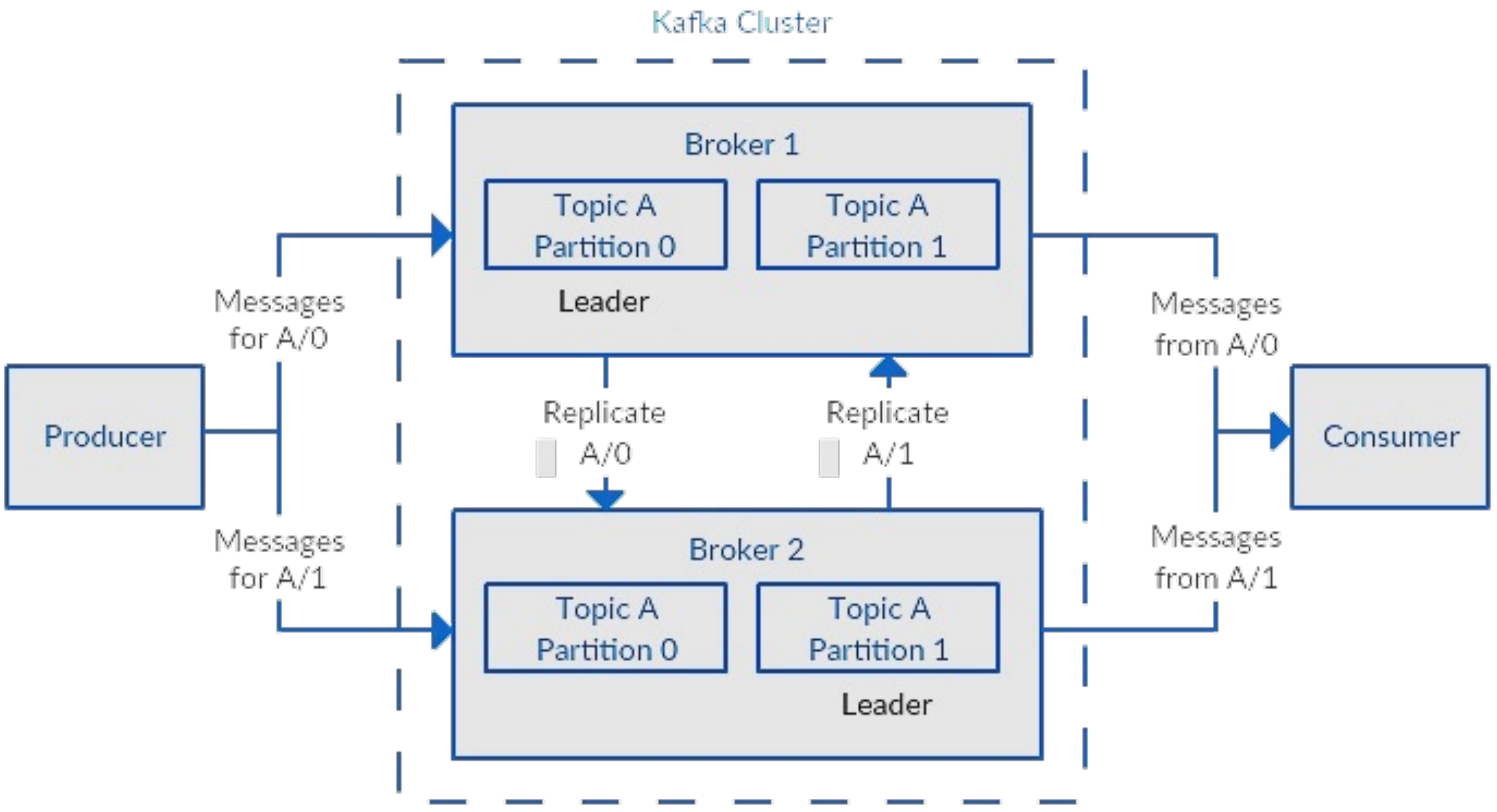
Partitions

- The partitions of each topic are distributed over servers in the Kafka cluster…
- With each server handling data and requests for a share of the partitions
- Each partition is replicated across a configurable number of servers for fault tolerance
- While sending data, producers don’t mention the partition but consumers are aware of the available partitions
- Kafka may use the message key to automatically group similar messages into a partition
- This scheme enables Kafka to dynamically scale the messaging infrastructure

Partitions

- Each partition has one server which acts as the "leader" and zero or more servers which act as "followers“
- The leader handles all read and write requests for the partition while the followers passively replicate the leader
- If the leader fails, one of the followers will automatically be promoted become to the new leader
- Each server acts as a leader for some of its partitions and a follower for others
- This allow I/O load to be well balanced within the cluster

Partitions


Kafka Cluster
Broker 1
Topic A Partition 0
Topic A Partition 1
Leader
Messages for A/0
Producer
Messages for A/1
Replicate A/0
Replicate A/1
Broker 2
Topic A Partition 0
Topic A Partition 1
Leader
Messages from A/0
Consumer
Messages from A/1

Partitions

- In replication, each partition of a message has *n* replicas and can afford *n-1* failures to guarantee message delivery
- Out of the *n* replicas, one replica acts as the lead replica for the rest of the replicas
- ZooKeeper keeps the information about the lead replica and the current in-sync follower replica
- The lead replica maintains the list of all in-sync follower replicas

The Role of ZooKeeper

- Kafka uses Apache ZooKeeper as a distributed configuration store
- It forms the backbone of Kafka cluster that continuously monitors the health of the brokers
- When new brokers get added to the cluster, ZooKeeper will start utilizing it by creating topics and partitions on it
- Initial versions of Kafka used ZooKeeper for storing the partition and offset information for each consumer
- Starting from Kafka version 0.10, that information has moved to an internal Kafka topic

Producers

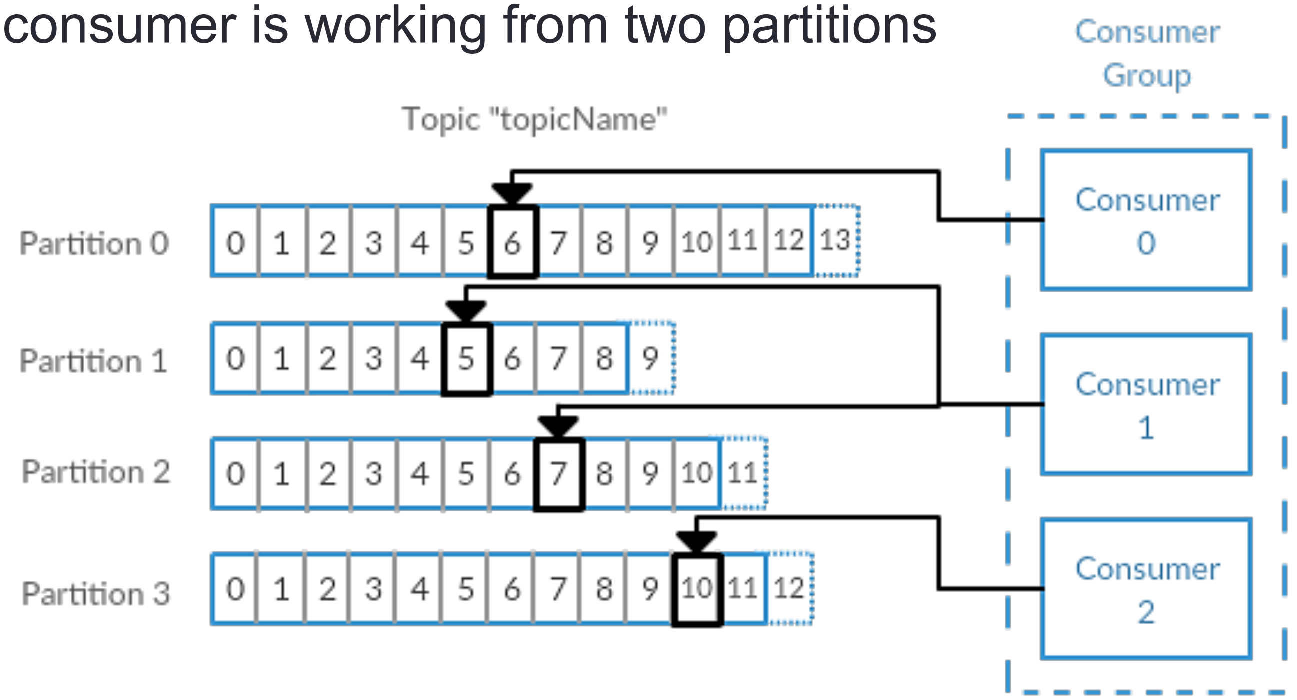
- Producers publish data to the topics of their choice
- The producer is responsible for choosing which record to assign to which partition within the topic
- This can be done in a round-robin fashion simply to balance load
- Or it can be done according to some partition function (say based on some key in the record)

Consumers

- Consumers label themselves with a *consumer group* name…
- And each record published to a topic is delivered to one consumer instance within each subscribing consumer group
- If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances
- If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes

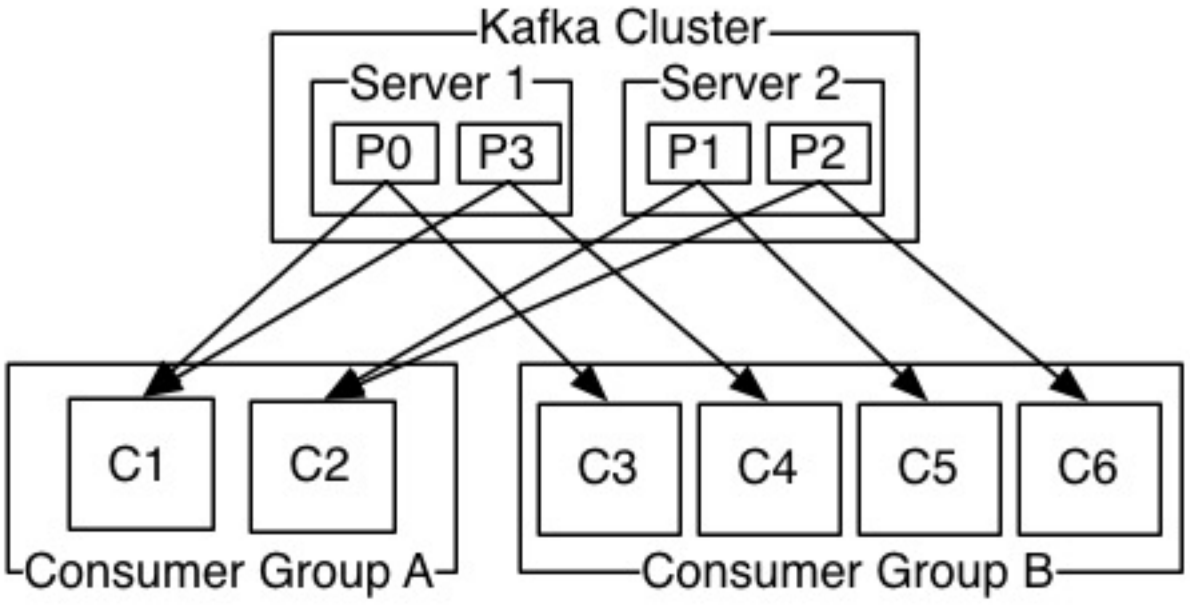
Consumers

- Consumers are clients that consume records from a cluster
- Consumers work together as part of a *consumer group*
- This is one or more consumers that work together to consume a topic
- The group assures each partition is only consumed by one member
- Below there are three consumers in a single group consuming a topic
- Two of the consumers are working from one partition each, while the third consumer is working from two partitions



Consumers

- Here is an example of a two server Kafka cluster hosting a four partition (P0-P3) topic with two consumer groups
- Consumer group A has two consumer instances and group B has four



Consumers

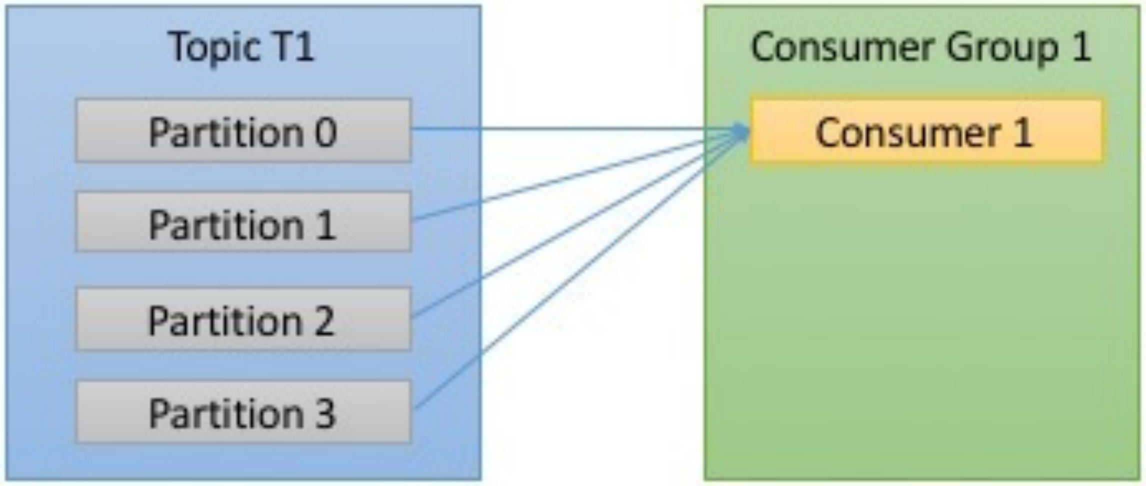
- Each group is composed of many consumer instances for scalability and fault tolerance
- This is nothing more than an extension of typical publish-subscribe semantics…
- But here the subscriber is a cluster of consumers instead of a single process

Consumer Groups

- Kafka consumers are typically part of a consumer group
- When multiple consumers are subscribed to a topic and belong to the same consumer group…
- Then each consumer in the group will receive messages from a different subset of the partitions in the topic

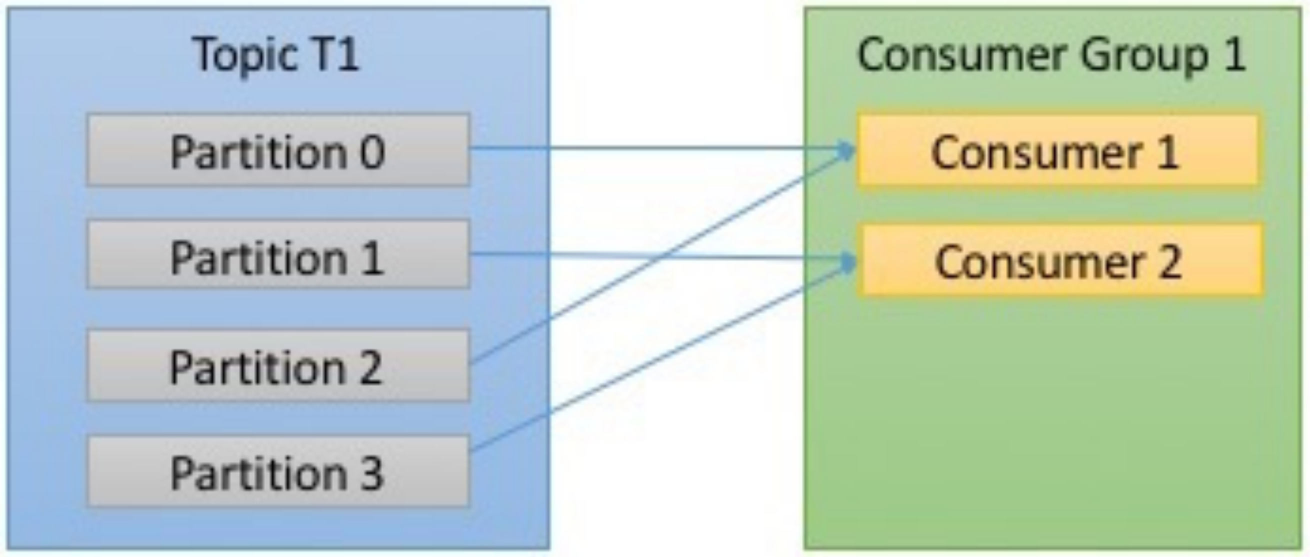
Consumer Groups

- Lets take topic *t1* with 4 partitions
- Now suppose we created a new consumer, *c1*, which is the only consumer in group *g1* and use it to subscribe to topic *t1*
- Consumer c1 will get all messages from all four of t1 partitions



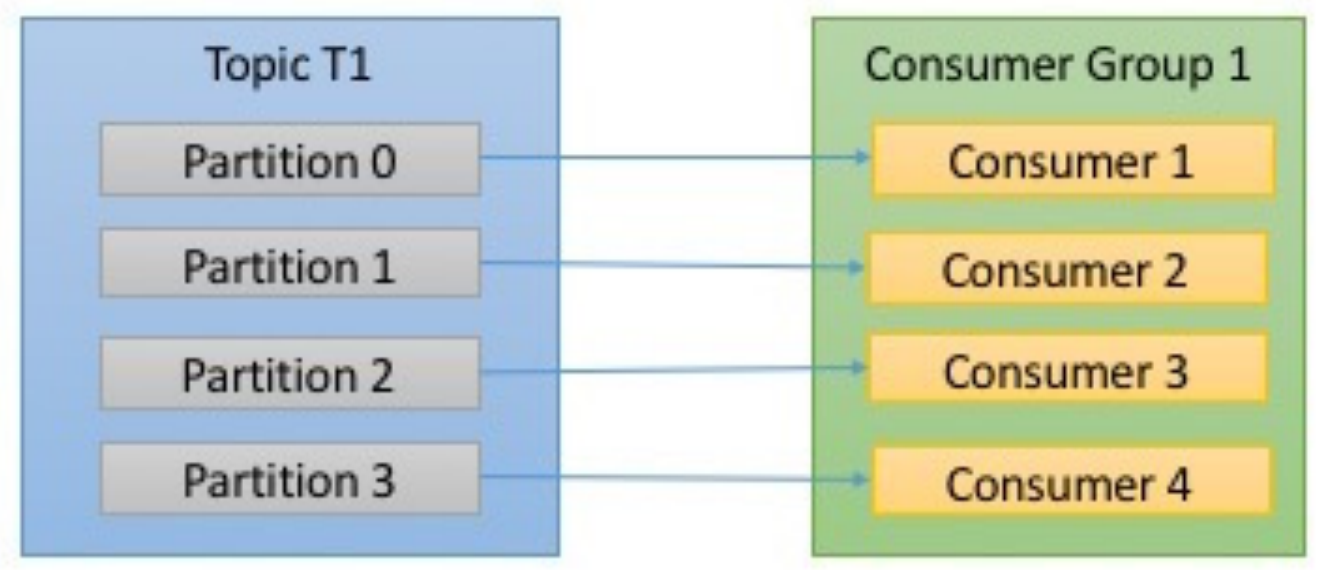
Consumer Groups

- If we add another consumer, *c2* to group *g1*, each consumer will only get messages from two partitions
- Perhaps messages from partition 0 and 2 go to *c1* and messages from partitions 1 and 3 go to consumer *c*



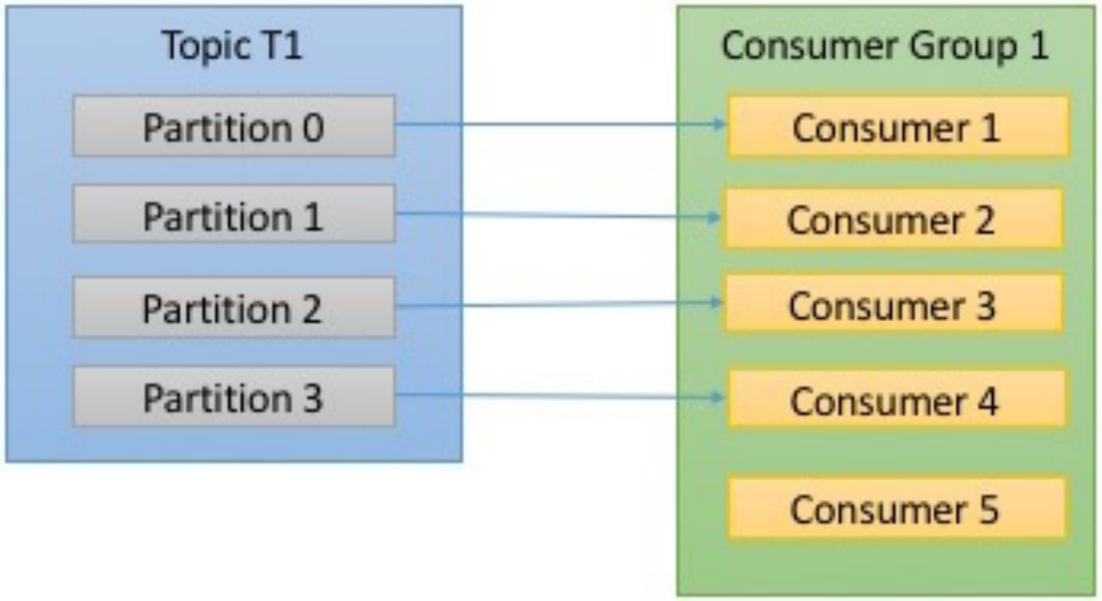
Consumer Groups

- If *g1* has 4 consumers, then each will read messages from a single partition



Consumer Groups

- If we add more consumers to a single group with a single topic than we have partitions…
- Then some of the consumers will be idle and get no messages at all.



Consumer Groups

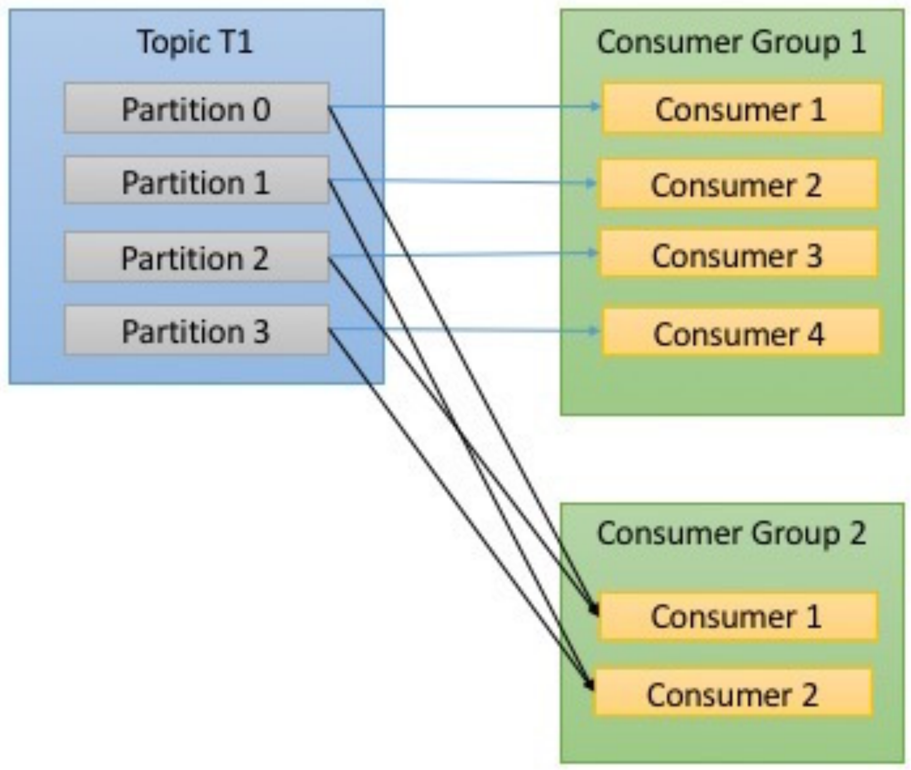
- The main way we scale consumption of data from a Kafka topic is by adding more consumers to a consumer group
- It is common for Kafka consumers to do high latency operations such as write to a database or to HDFS, or a time-consuming computation on the data
- In these cases, a single consumer can’t possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling
- This is a good reason to create topics with a large number of partitions - it allows adding more consumers when the load increases
- Keep in mind that there is no point in adding more consumers than you have partitions in a topic - some of the consumers will just be idle.

Consumer Groups

- It is very common to have multiple applications that need to read data from the same topic
- In fact, one of the main design goals in Kafka was to make the data produced to Kafka topics available for many use-cases throughout the organization
- In those cases, we want each application to get all of the messages, rather than just a subset
- To make sure an application gets all the messages in a topic…
- You make sure the application has its own consumer group

Consumer Groups

- If we add a new consumer group 2 it will get messages in topic t1 independently of what *group1* is doing



Guarantees

- At a high-level Kafka gives the following guarantees:
- Messages sent by a producer to a particular topic partition will be appended in the order they are sent
- That is, if a record M1 is sent by the same producer as a record M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log
- A consumer instance sees records in the order they are stored in the log
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log

Using Kafka

- There are many libraries allowing access to Kafka
- Most widely used is the standard Java language library
- But there is a Python library which we will explore
  - See https://github.com/dpkp/kafka-python

Using Kafka
Core APIs

- The Producer API allows an application to publish a stream of records to one or more Kafka topics.
  - Kafka.KafkaProducer(): A Kafka client that publishes records to the Kafka cluster
  - The producer is thread safe and sharing a single producer instance across threads will generally be faster than having multiple instances
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them
  - Kafka.KafkaConsumer(): Consume records from a Kafka cluster
  - Transparently handles the failure of servers in the Kafka cluster
  - Adapts as topic and partitions are created or migrate between brokers
  - Interacts with the assigned Kafka Group Coordinator node to allow multiple consumers to load balance consumption of topics

Using Kafka
Simple message producer and consumer

Producer

```
from kafka import KafkaProducer
from time import sleep


producer = KafkaProducer(
bootstrap_servers=['localhost:9092'])


producer.send('sample3', b'a_value')
sleep(5)


producer.close()
```

Consumer

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(
'sample3', auto_offset_reset='earliest',
bootstrap_servers=['localhost:9092'],
consumer_timeout_ms=1000)

for message in consumer:
    # value and key are raw bytes
    # decode if necessary!

print ("%s:%d:%d: key=%s value=%s" %
(message.topic, message.partition,
message.offset, message.key, message.value))

consumer.close()
```

```
from kafka import KafkaProducer
from time import sleep

producer =
KafkaProducer(bootstrap_servers
=['localhost:9092'])

producer.send('sample3',
b'a_value')
sleep(5)

producer.close()
```

Kafka Producer
Overview

- Here is an example of using a Kafka producer to send messages
- From an implementation perspective the producer has…
  - A pool of buffer space that holds records (messages) that haven't yet been transmitted to the server
  - A background I/O thread responsible for turning these records into requests and transmitting them to the Kafka cluster
- The example shows the principle phases of a Kafka producer's lifecycle
  1. Creating the producer
  2. Sending one or more messages to Kafka brokers
  3. Closing the producer to release any resources

Kafka Producer
Phase 1—Create and configure the producer (default)

- The following code snippet shows how to create a new KafkaProducer with a default configuration

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers=['localhost:9092’])
```

Kafka Producer
Phase 1—Create and configure the producer (default)

- A function encapsulating setting up a KafkaProducer including handling connection (and other) excpetions

```
def connect_kafka_producer():
    _producer = None
    try:
        _producer = KafkaProducer(bootstrap_servers=['localhost:9092'])
    except Exception as ex:
        print('Exception while connecting Kafka')
        print(str(ex))
    finally:
        return _producer
```

Kafka Producer

Phase 1—Setting up the operational configuration (custom)

- Set up to produce json format messages
  - { ‘name’ : value }

```
from json import dumps
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                         value_serializer=lambda x: dumps(x).encode('utf-8’))
```

- value_serializer=lambda x: dumps(x).encode(‘utf-8’)
  - function of how the data should be serialized before sending to the broker. Here, we convert the data to a json file and encode it to utf-8.

Kafka Producer

Phase 2—Sending one or more messages to brokers (simple)

```
producer.send('test', b’a_value’)
```

Topic name

Value, must be bytes

```
producer.send('test’, value=b’a_value’, key=b’a_key’)
```

Key, must be bytes

Kafka Producer

Phase 2—Sending one or more messages to brokers (simple)

```
from time import sleep

str_value = ‘a_string_value’
bytes_value=bytes(str_value, ‘utf-8’)
producer.send(‘test’, bytes_value)
sleep(5)

int_value = 5
bytes_value=bytes(int_value)
producer.send(‘test’, bytes_value)
sleep(5)
```

Convert a string value to bytes

Convert an int value to bytes

Kafka Producer

Phase 2—Sending one or more messages to brokers (advanced)

- Send json format messages

```
from time import sleep
from json import dumps
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                         value_serializer=lambda x: dumps(x).encode('utf-8’))

for e in range(1000):
    data = {'number' : e}
    producer.send('test', data)
    sleep(5)
```

Kafka Producer
Phase 2—Sending one or more messages to brokers (advanced)

```
# Asynchronous by default
future = producer.send('test', b'raw_bytes')

# Block for 'synchronous' sends
try:
    record_metadata = future.get(timeout=10)
except KafkaError:
    # Decide what to do if produce request failed...
    log.exception()
    pass

# Successful result returns assigned partition and offset
print (record_metadata.topic)
print (record_metadata.partition)
print (record_metadata.offset)
```

Kafka Producer

Phase 2—Sending one or more messages to brokers (advanced)

```
def publish_message(producer_instance, topic_name, key, value):
    try:
        key_bytes = bytes(key, encoding='utf-8')
        value_bytes = bytes(value, encoding='utf-8')
        producer_instance.send(topic_name, key=key_bytes,
value=value_bytes)
        producer_instance.flush()
        print('Message published successfully.')
    except Exception as ex:
        print('Exception in publishing message')
        print(str(ex))
```

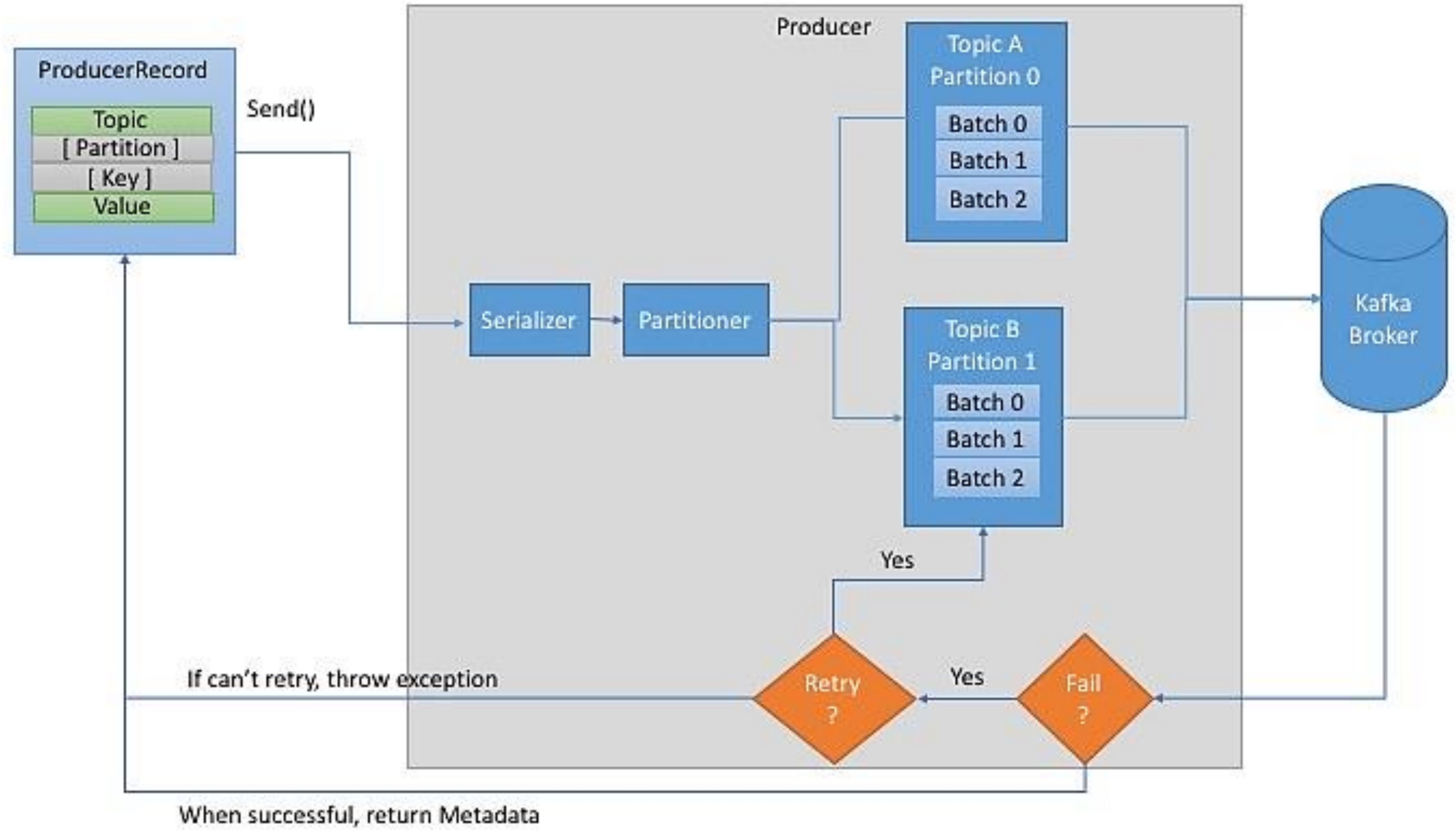
Kafka Producer

Phase 2—Sending one or more messages to brokers

- Next, the data is sent to a partitioner
- If we specified a partitioner as a KafkaProducer() parameter it returns the partition to which to send the message
- If we didn't, the default partitioner will choose a partition for us

Kafka Producer

Phase 3—Sending one or more messages to brokers



Kafka Producer
Phase 3—Sending one or more messages to brokers

- Once a partition is selected, the producer knows which topic and partition the record will go to
- It then adds the record to a batch of records that will also be sent to the same topic and partition
- A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

Kafka Producer

Phase 3—Closing the producer to release any resources

- The producer has a pool of buffer space to holds records and a background thread for transmitting records
- Failure to close the producer after use will leak these resources

```
producer.close();
```

Kafka Consumer

Overview

- The example shows the four principle phases of a Kafka consumer's lifecycle
  1. Creating and configure the kafka consumer
  2. Retrieve sent messages from a topic partition
  3. Close the consumer to release any resource

Kafka Consumer
Phase 1—Creating the consumer (simple)

- The following code snippet shows how to create a new consumer with a specified configuration

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(
    'test',
     bootstrap_servers=['localhost:9092'],
     auto_offset_reset='earliest’,
     consumer_timeout_ms=10000,
     group_id='my-group’)
```

Kafka Consumer

Phase 1—Creating the consumer (simple)

- By providing auto_offset_reset='earliest' you are telling Kafka to return messages from the beginning.
  - When set to ‘latest’, the consumer starts reading at the end of the log. When set to ‘earliest’, the consumer starts reading at the latest committed offset.
- The parameter consumer_timeout_ms helps the consumer to disconnect after the certain period of time. Default block forever
- The parameter group_id allows you to name the group your consumer is in. Default: None
-

Kafka Consumer

Phase 1—Creating the consumer (advanced)

```
from kafka import KafkaConsumer
from json import loads

consumer = KafkaConsumer(
    'test',
     bootstrap_servers=['localhost:9092'],
     auto_offset_reset='earliest',
     enable_auto_commit=True,
     group_id='my-group',
     value_deserializer=lambda x: loads(x.decode('utf-8’)))
```

- The value_deserializer deserializes the data into a common json format, the inverse of what our value serializer was doing

Kafka Consumer

Phase 1—Creating the consumer

- Kafka uses the concept of *consumer groups* to allow a pool of processes to divide the work of consuming and processing records
- All consumer instances sharing the same group.id will be part of the same consumer group
- Each consumer in a group can set the list of topics it wants to subscribe to through one of the subscribe APIs
- Kafka will deliver each message in the subscribed topics to one process in each consumer group
- This is achieved by balancing topic partitions between members in the consumer group…
- So that each partition is assigned to exactly one consumer in the group
- So if there is a topic with four partitions, and a consumer group with two processes, each process would consume from two partitions

Kafka Consumer

Phase 2—Retrieve sent messages from a topic partition (simple)

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(
    'test',
     bootstrap_servers=['localhost:9092'],
     auto_offset_reset='earliest’,
     consumer_timeout_ms=10000,
     group_id='my-group’)

for message in consumer:
```

A KafkaConsumer is a Python Iterable

```
    # message value and key are raw bytes -- decode if necessary!
    # e.g., for unicode: `message.value.decode('utf-8')`
    print ("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,
message.offset, message.key, message.value))
```

Kafka Consumer

Phase 2—Retrieve sent messages from a topic partition (simple)

- Note, the consumer will wait for up to consumer_timeout_ms=10000 milliseconds for the next message
- If no message is received the for loop will terminate
- Without this timeout a consumer program will wait for the next message forever, or until you manually terminate it

Kafka Consumer

Phase 3—Closing the consumer to release any resources

```
consumer.close()
```

Kafka Consumer

Phase 2—Retrieve sent messages from a topic partition (advanced)

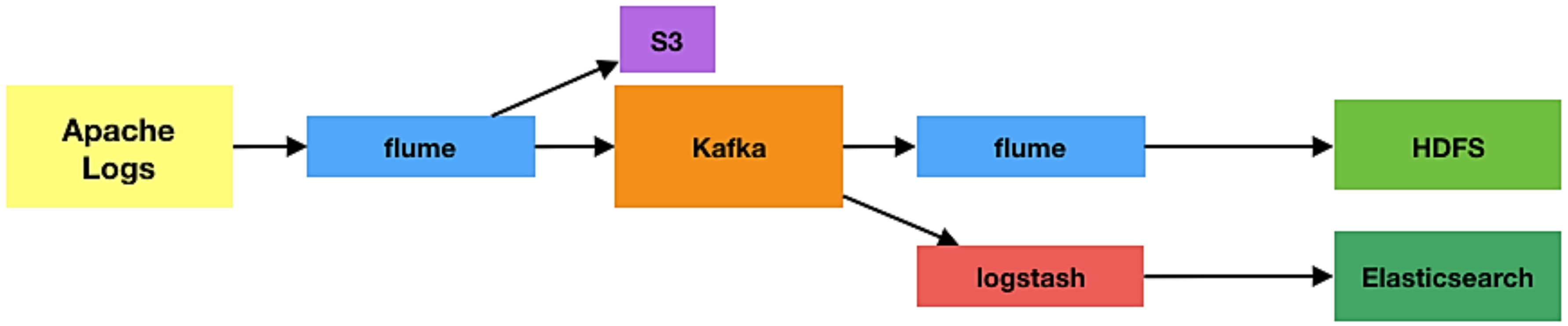
- Read json format messages

```
from kafka import KafkaConsumer
from json import loads

consumer = KafkaConsumer(
    'test',
     bootstrap_servers=['localhost:9092'],
     auto_offset_reset='earliest',
     enable_auto_commit=True,
     group_id='my-group',
     value_deserializer=lambda x: loads(x.decode('utf-8’)))

for msg in consumer:
       record = json.loads(msg.value)
       print (record)
```

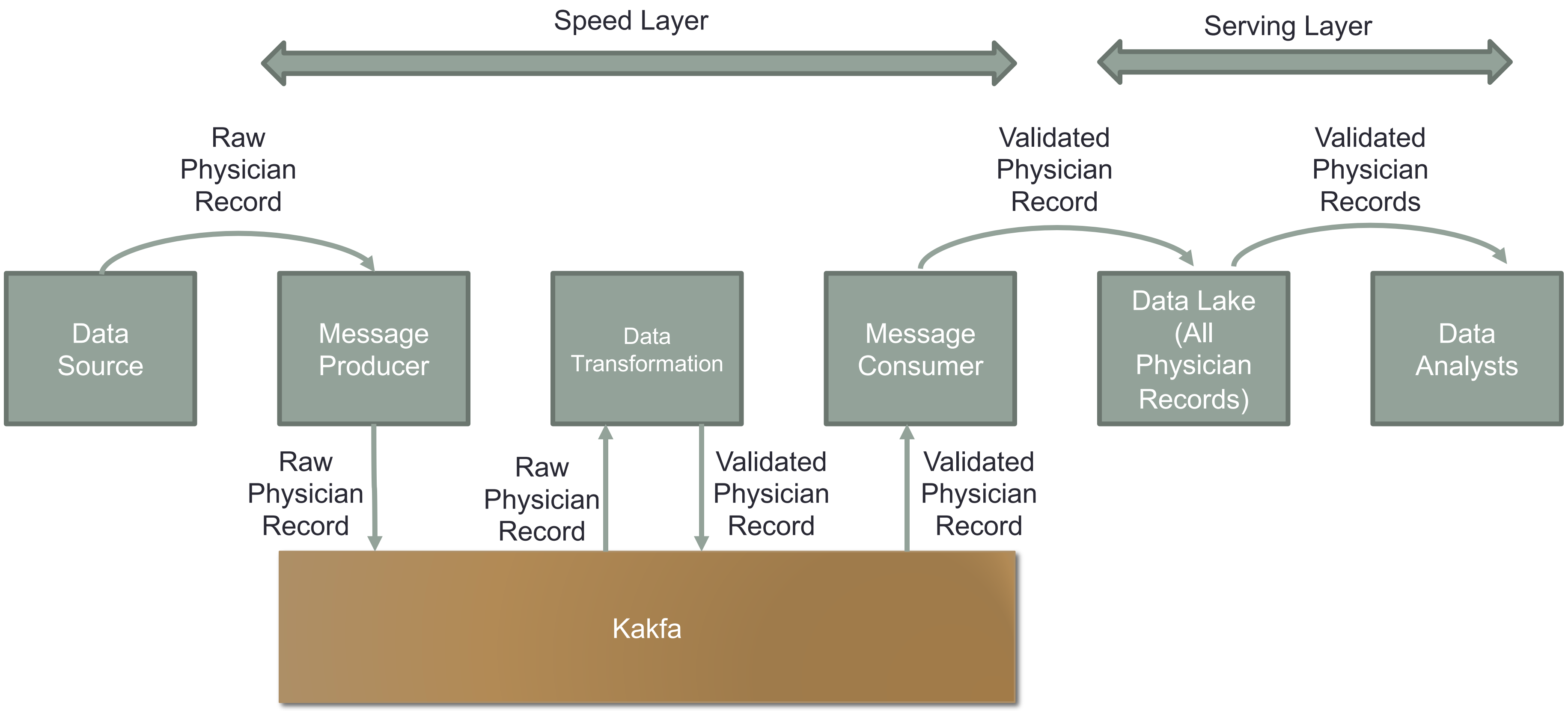
Deserialize each retreived message from bytes to a json object

Convert a json object into a Python dictionary

Kafka As Part of a Message Processing Pipeline



Kafka As Part of a Message Processing Pipeline Kappa Architecture


Speed Layer
Serving Layer
Raw Physician Record
Validated Physician Record
Validated Physician Records
Data Source
Message Producer
Data Transformation
Message Consumer
Data Lake (All Physician Records)
Data Analysts
Raw Physician Record
Raw Physician Record
Validated Physician Record
Validated Physician Record
Kakfa