

Apache Kafka - The must know Pub-Sub

Daniel Hinojosa

About Me...

Daniel Hinojosa
Programmer, Consultant, Trainer

Testing in Scala (Book)
Beginning Scala Programming (Video)
Scala Beyond the Basics (Video)

Contact:

dhinojosa@evolutionnext.com

<https://mastodon.social/@dhinojosa>

<https://www.linkedin.com/in/dhevolutionnext/>



Agenda

Agenda

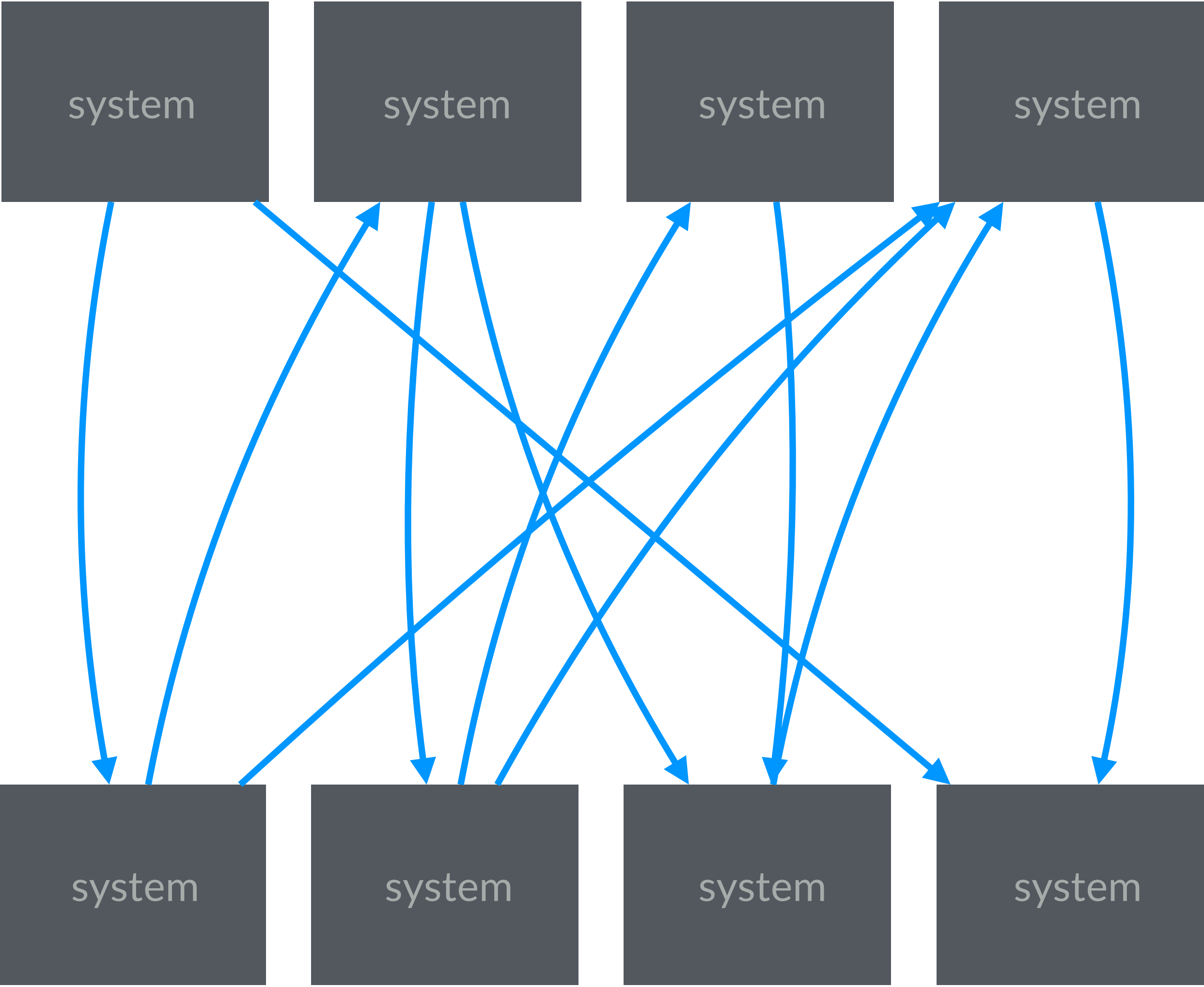
★ Understand Kafka

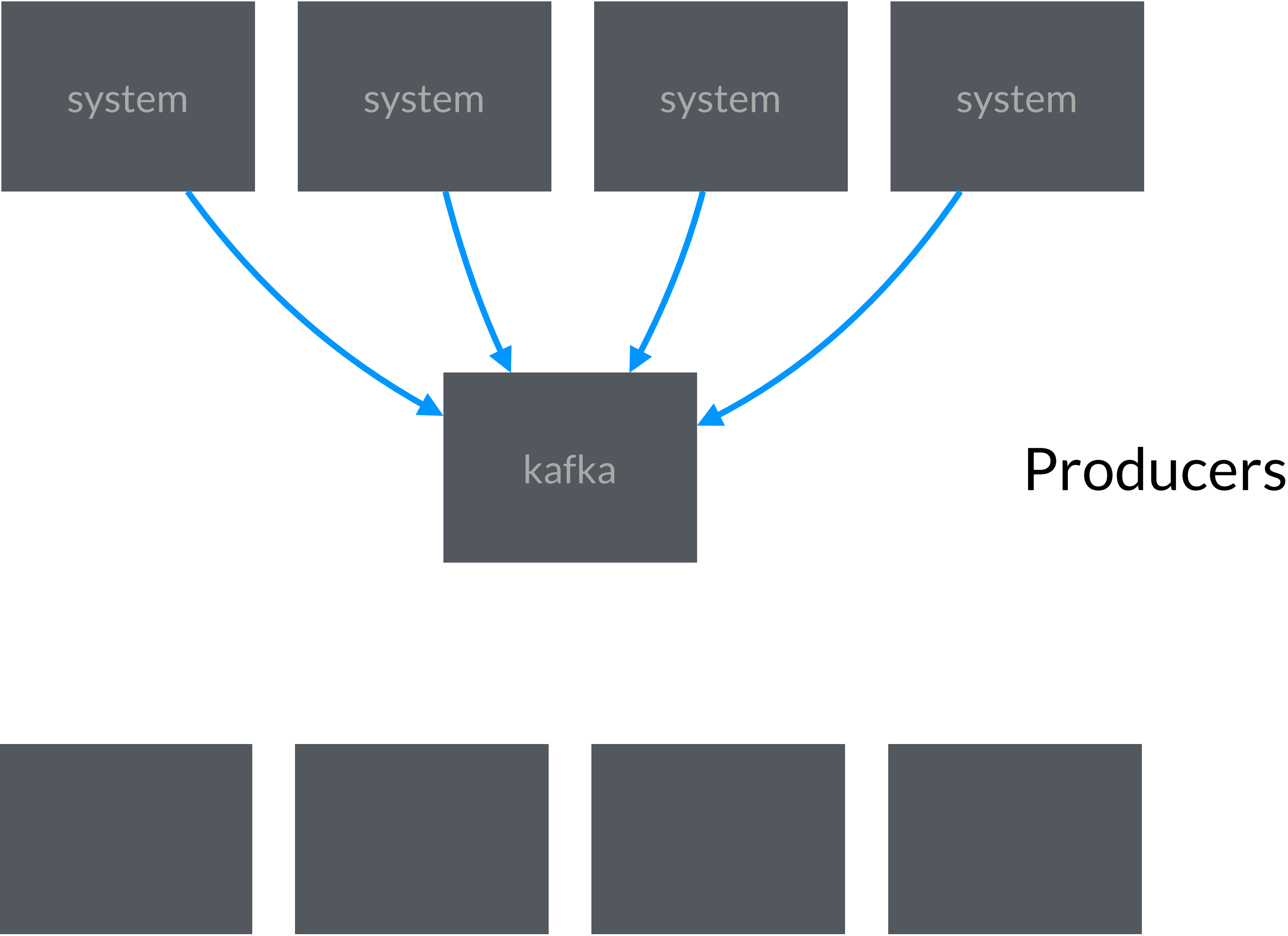
★ Understand Producer

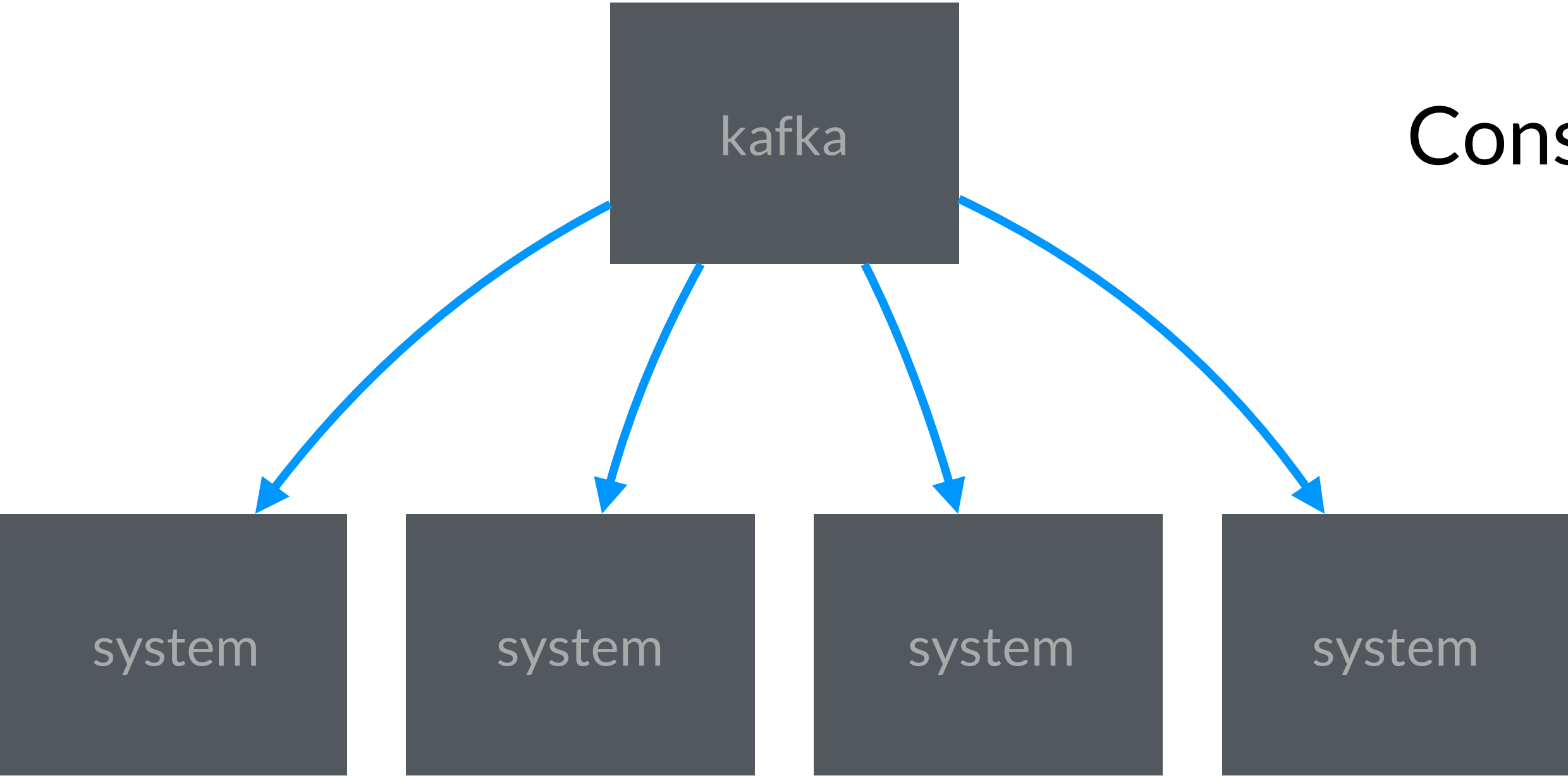
★ Understand Consumer

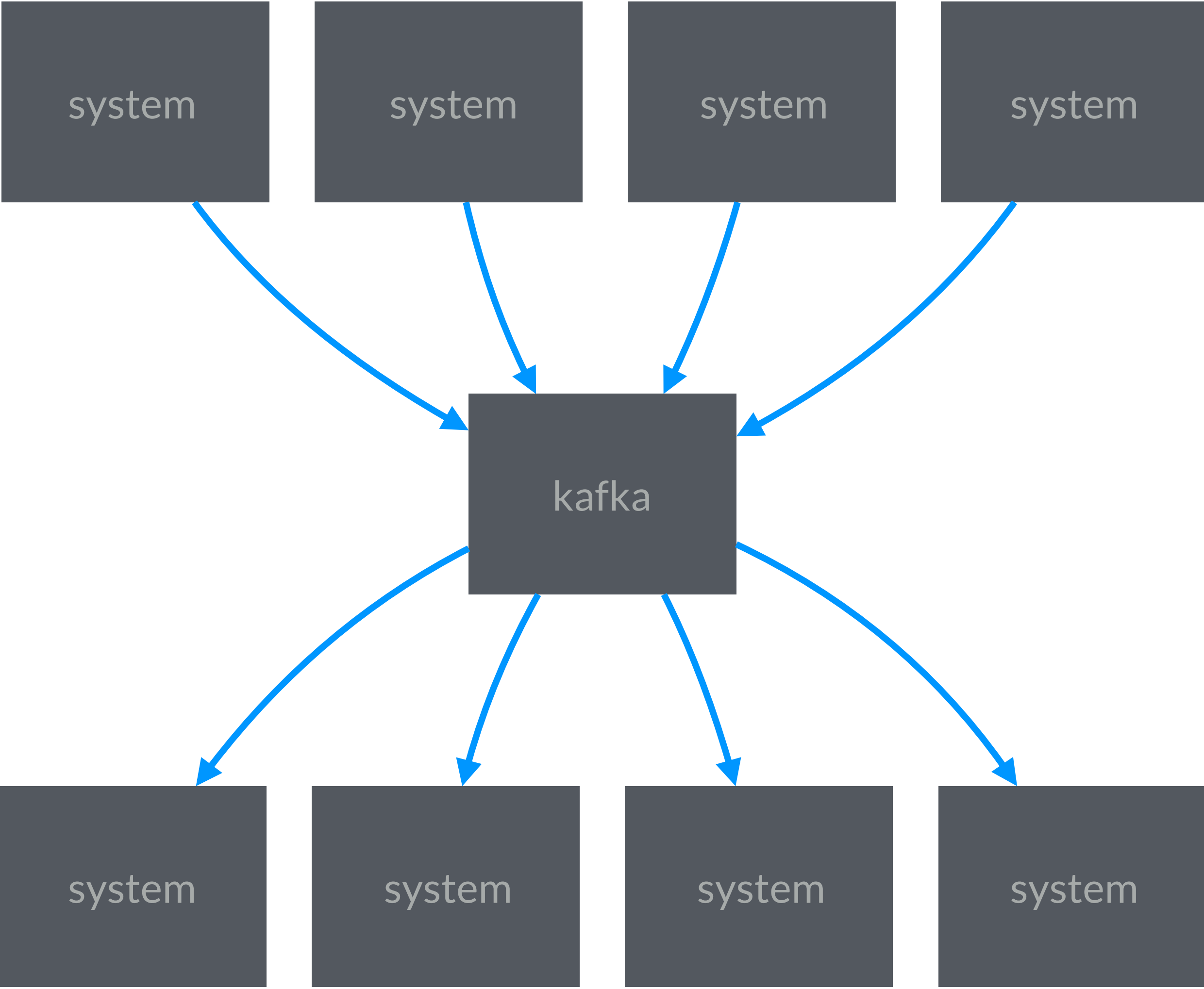
★ Understand Rebalancing

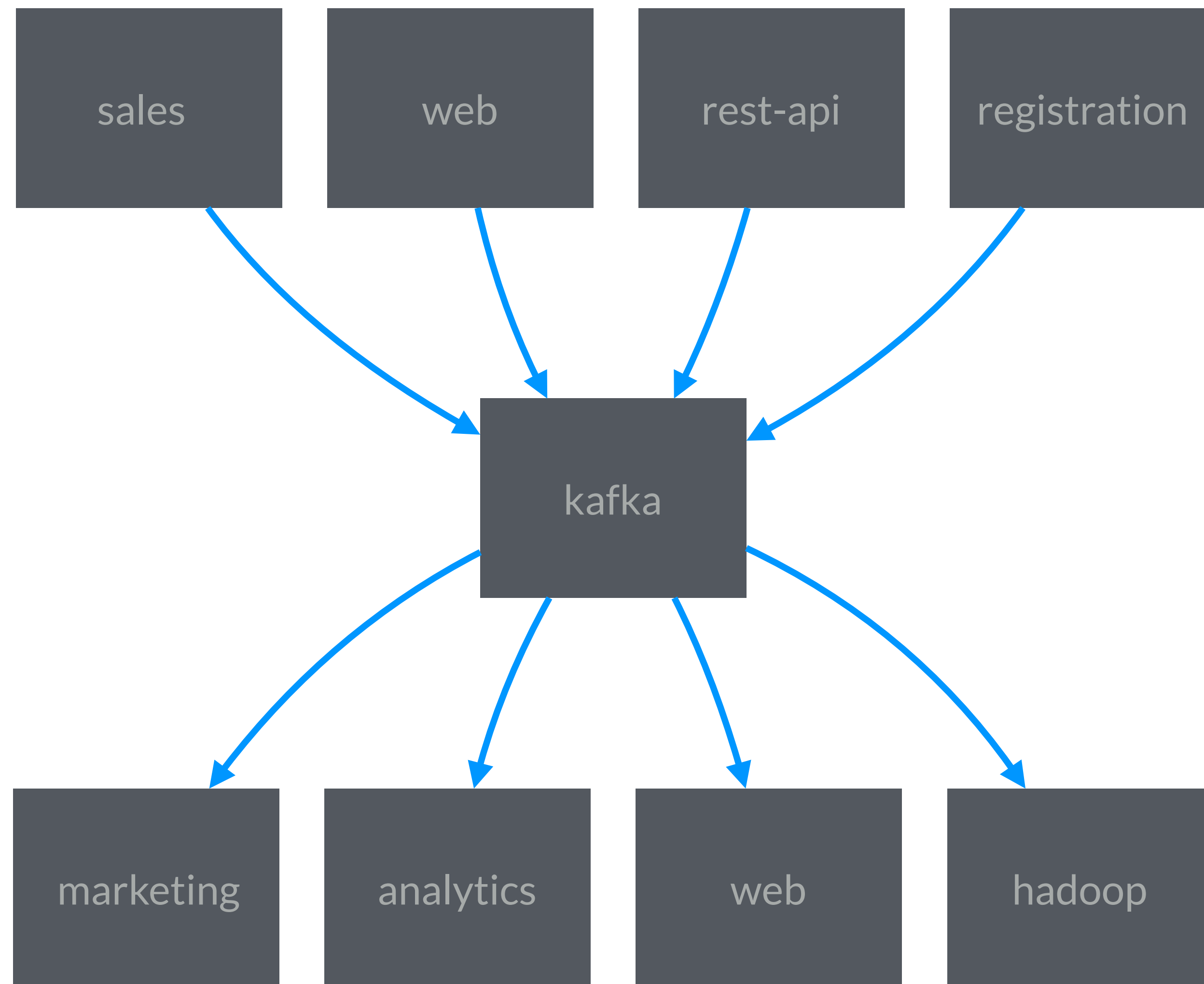
Kafka Introduction









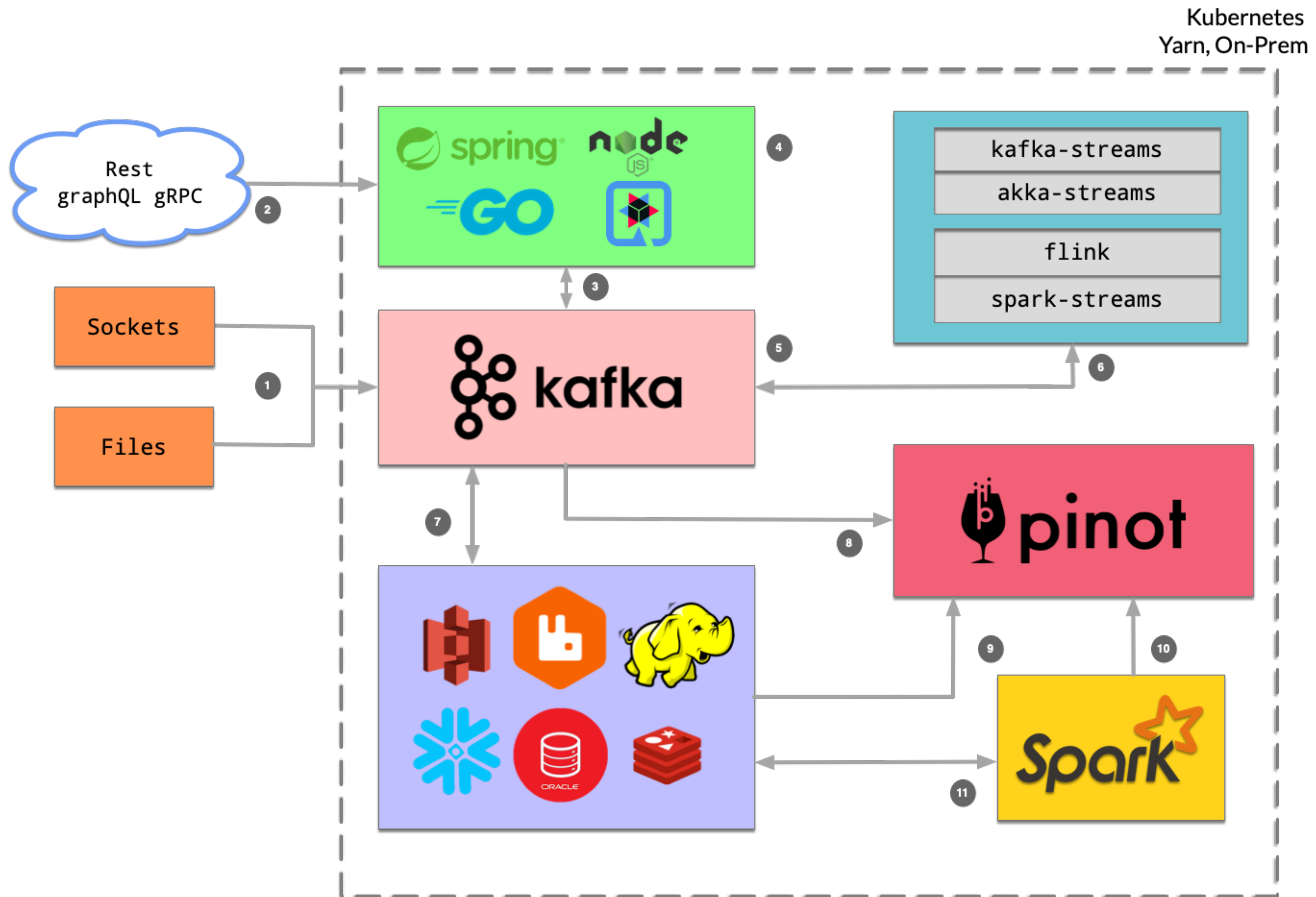


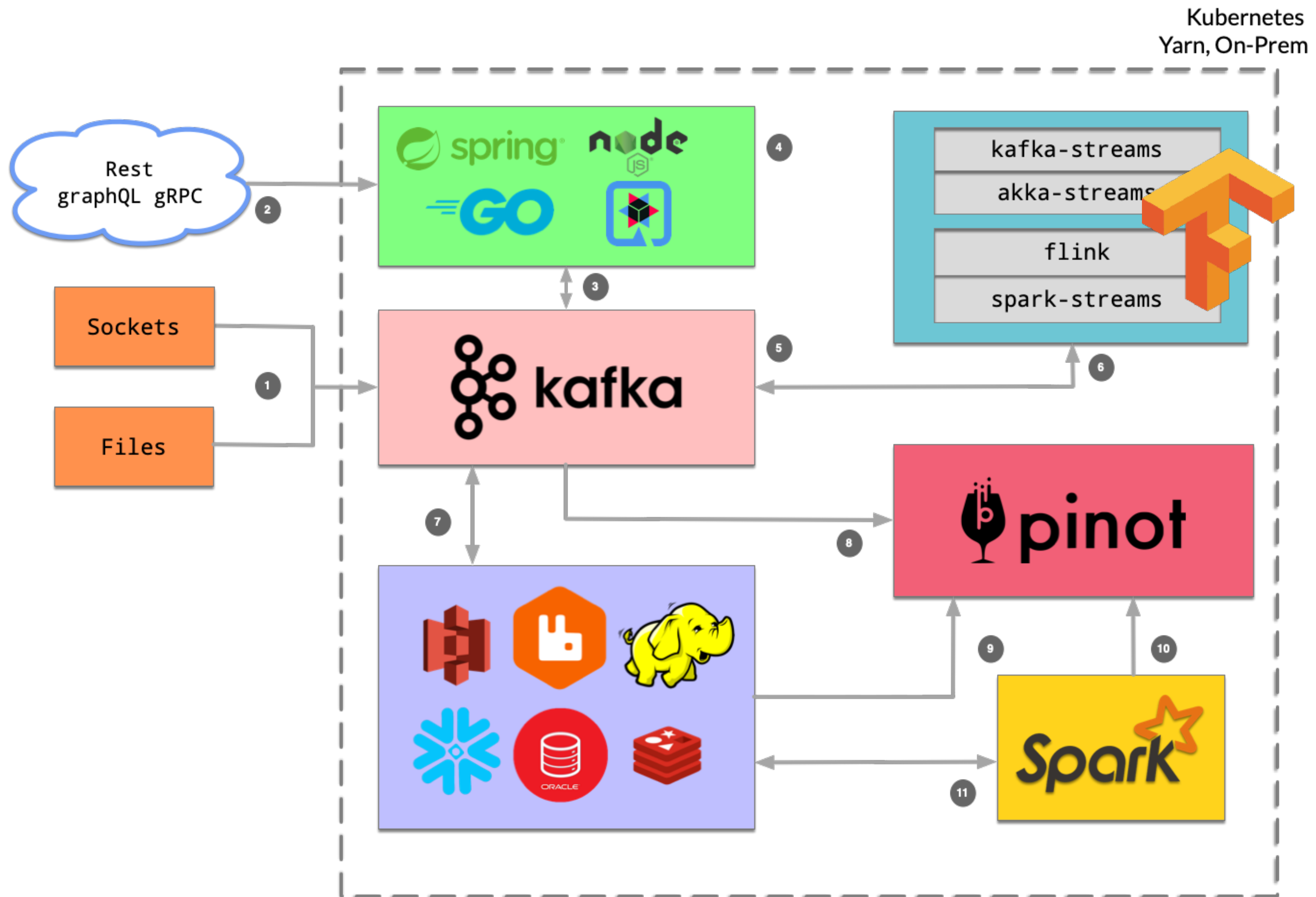
Note: A Producer can be a Consumer

About Kafka

- Handles millions of messages per seconds, high throughput, high volume
- Distributed and Replicated Commit-log
- Real Time Data Processing
- Stream processing

A Kafkaesque Architecture

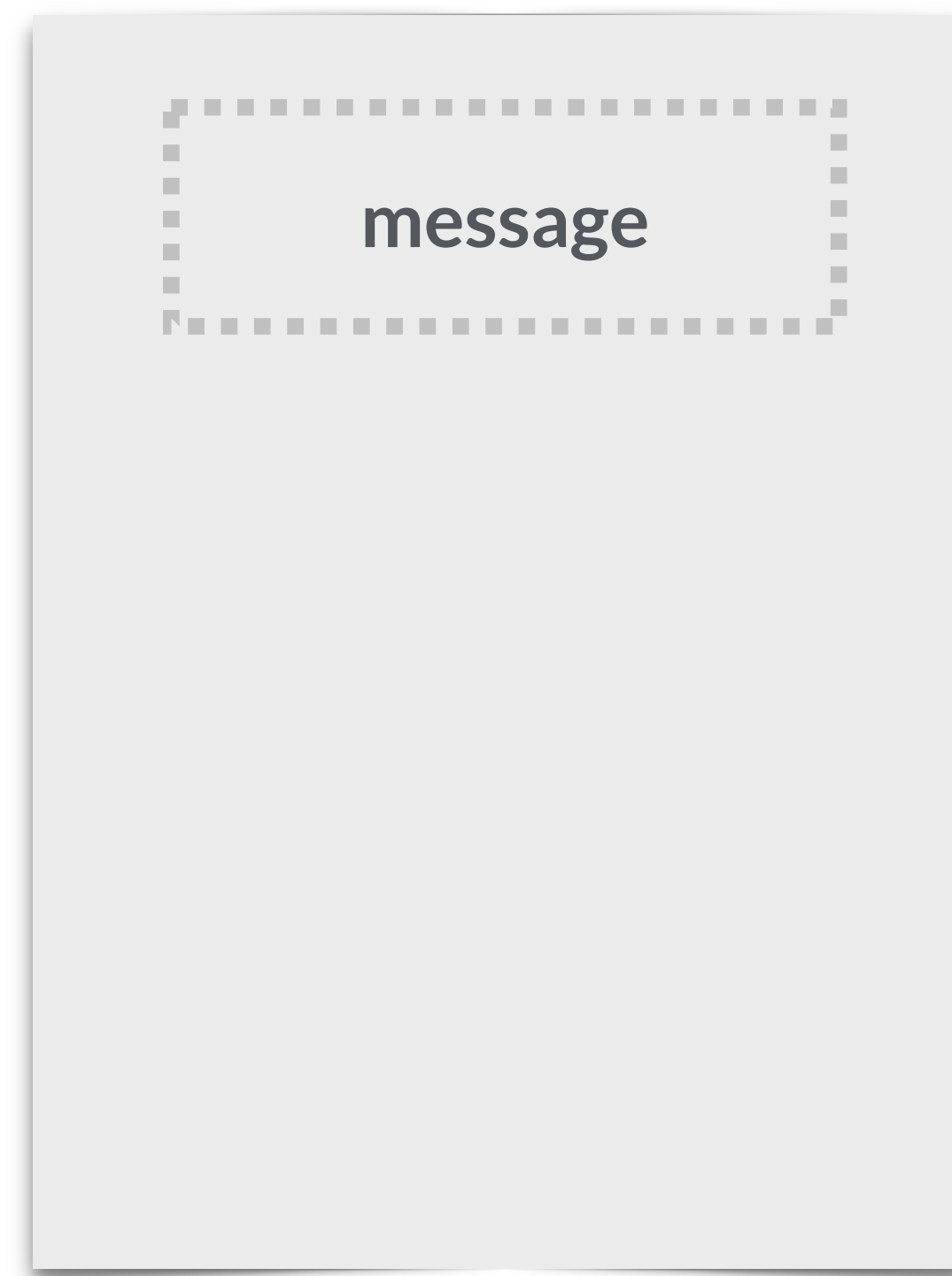




Kafka Messages

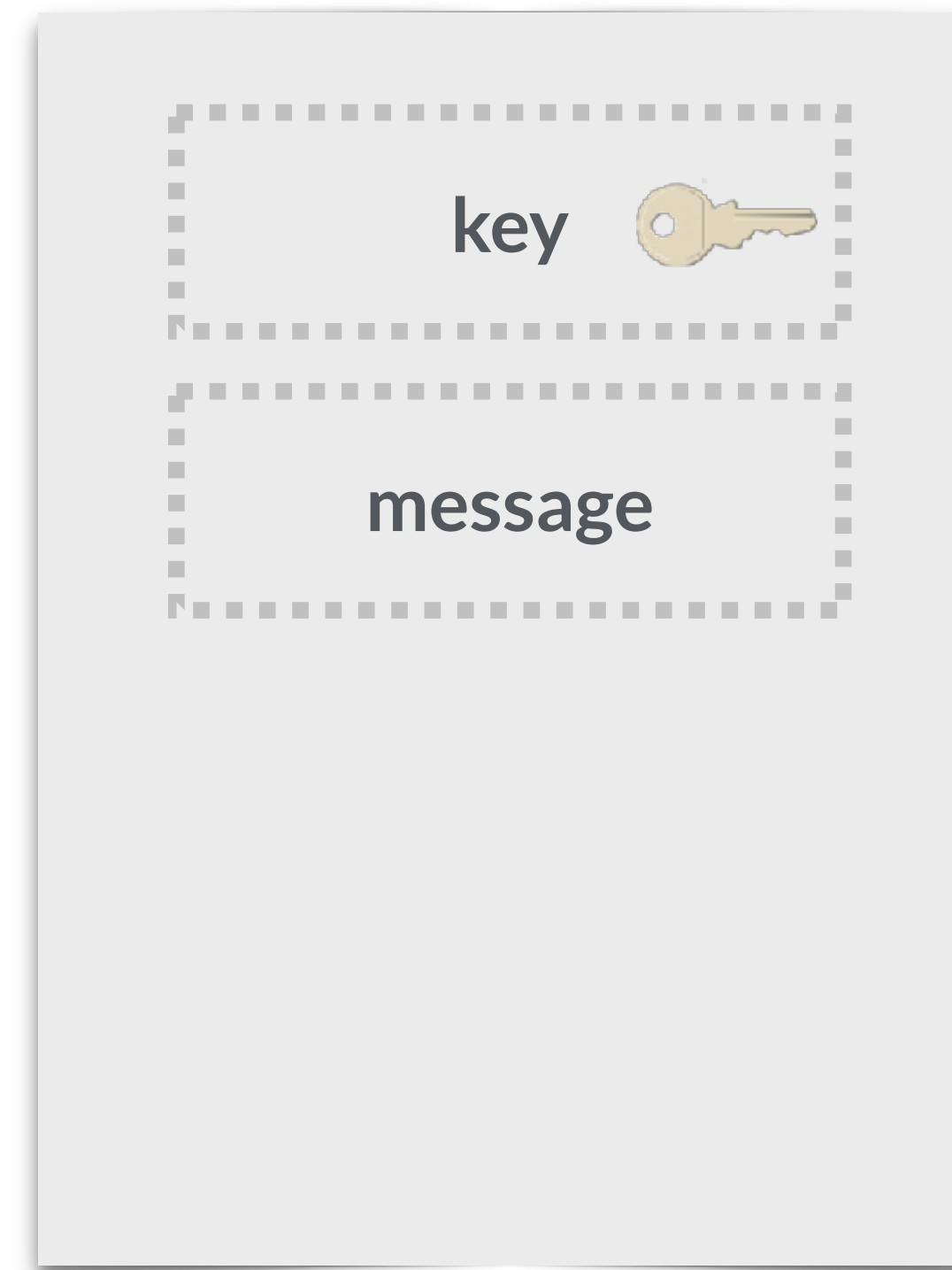
A Kafka Message

- Similar to a *row* or a *record*
- Message is an array of bytes
- No special serialization, that is done at the producer or consumer

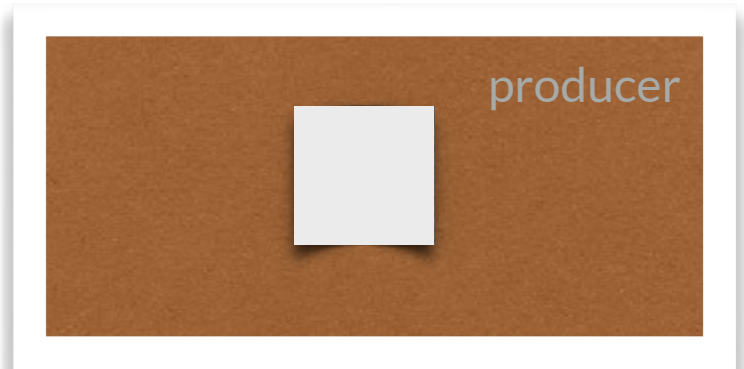


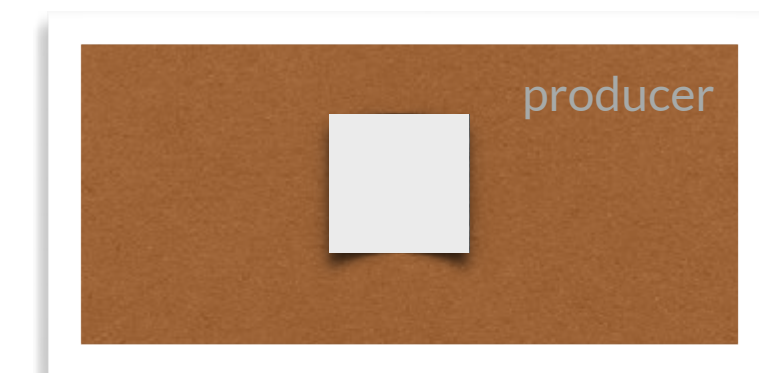
A Kafka Message Key

- Message may contain a *key* for better distribution to partitions
- The *key* is also an array of bytes
- If a *key* is provided, a partitioner will hash the key and map it to a single partition
- Therefore it is the only time that something is guaranteed to be in order



Kafka Producers

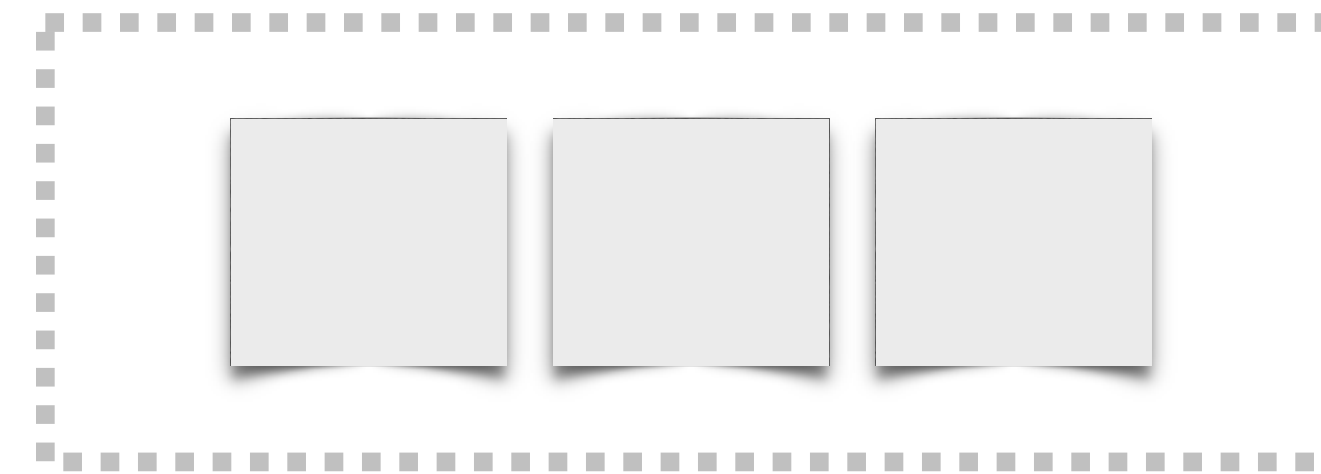


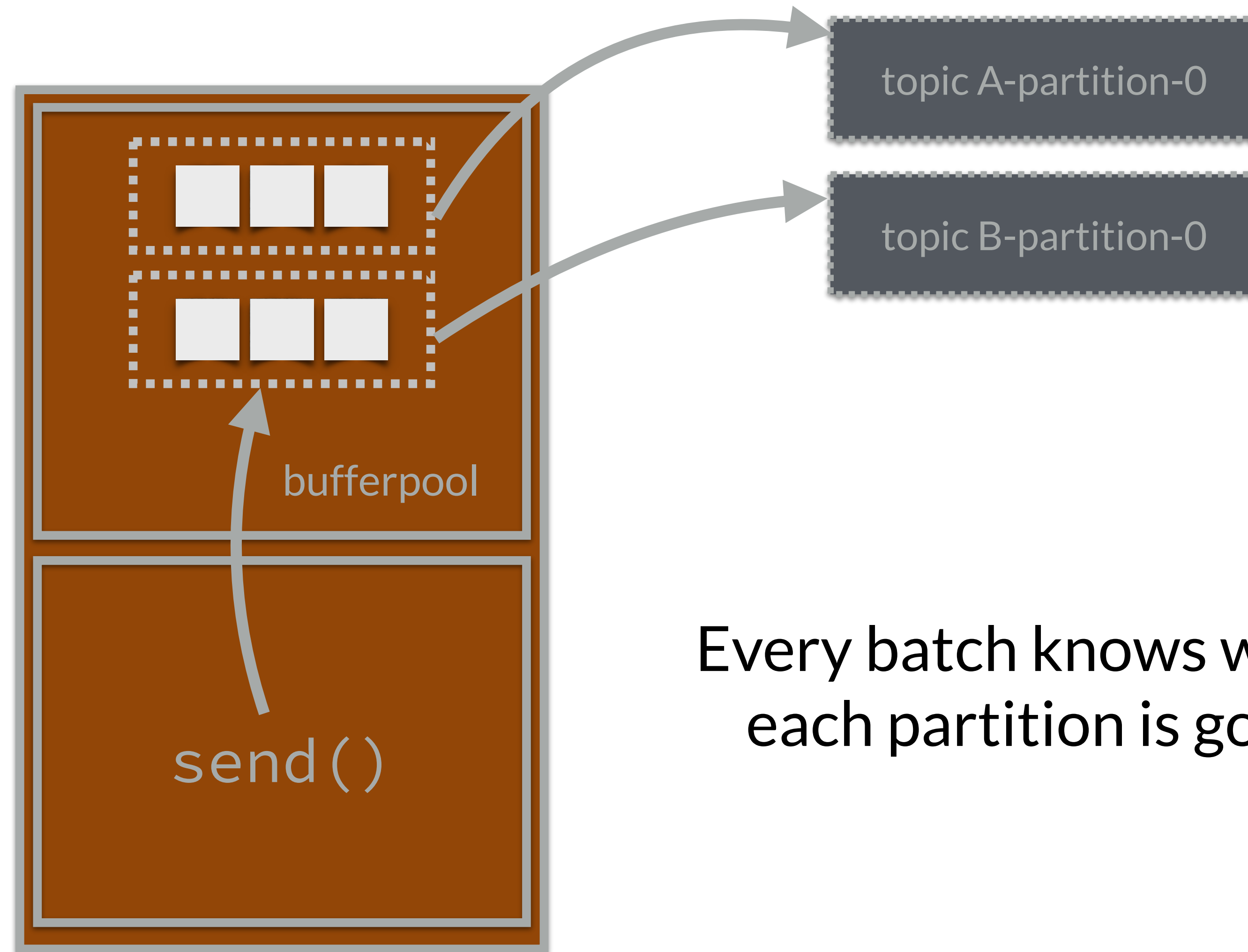


Retention: The **data is temporary**

Kafka Batch

- A collection of messages, that is sent, configured in *bytes*
- Sent to the same *topic* and the same *partition*
- *Avoids overhead of sending multiple message over the wire*

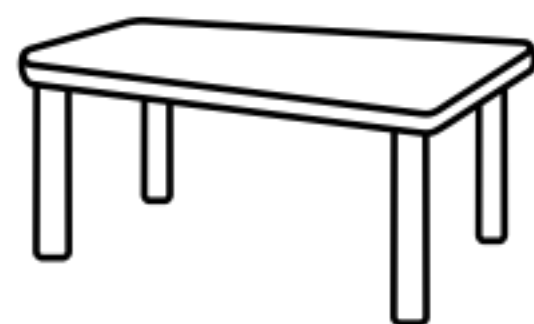




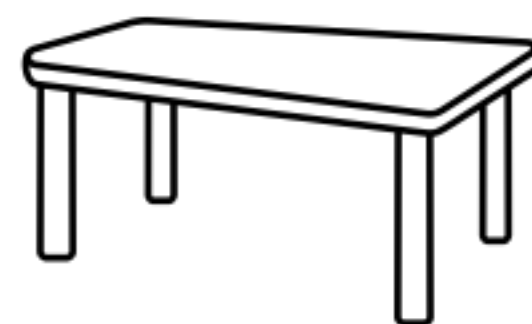
Every batch knows where
each partition is going

`murmur2(bytes) % number partitions`

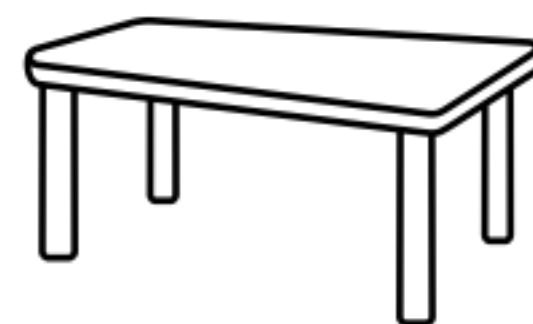
A-E



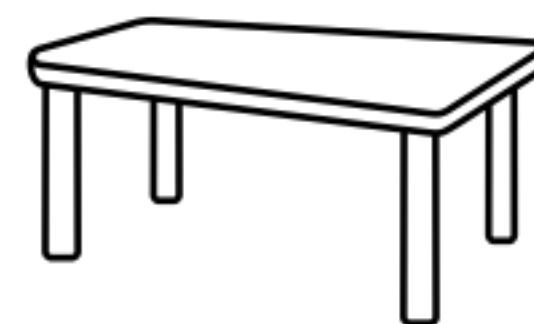
F-K



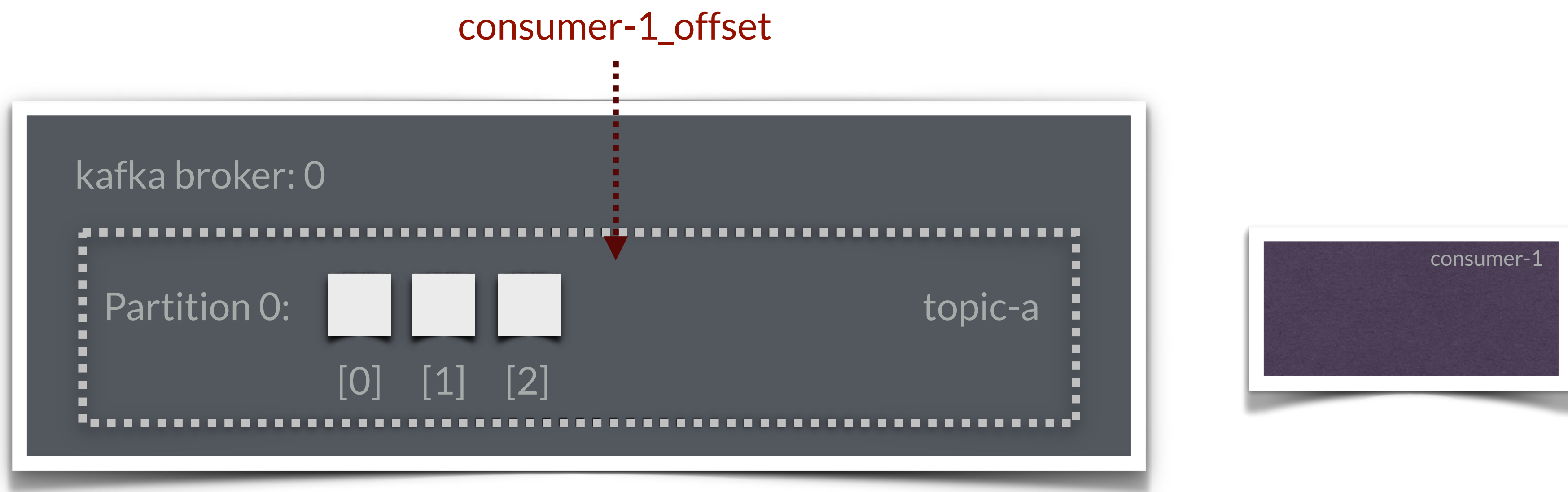
L-S

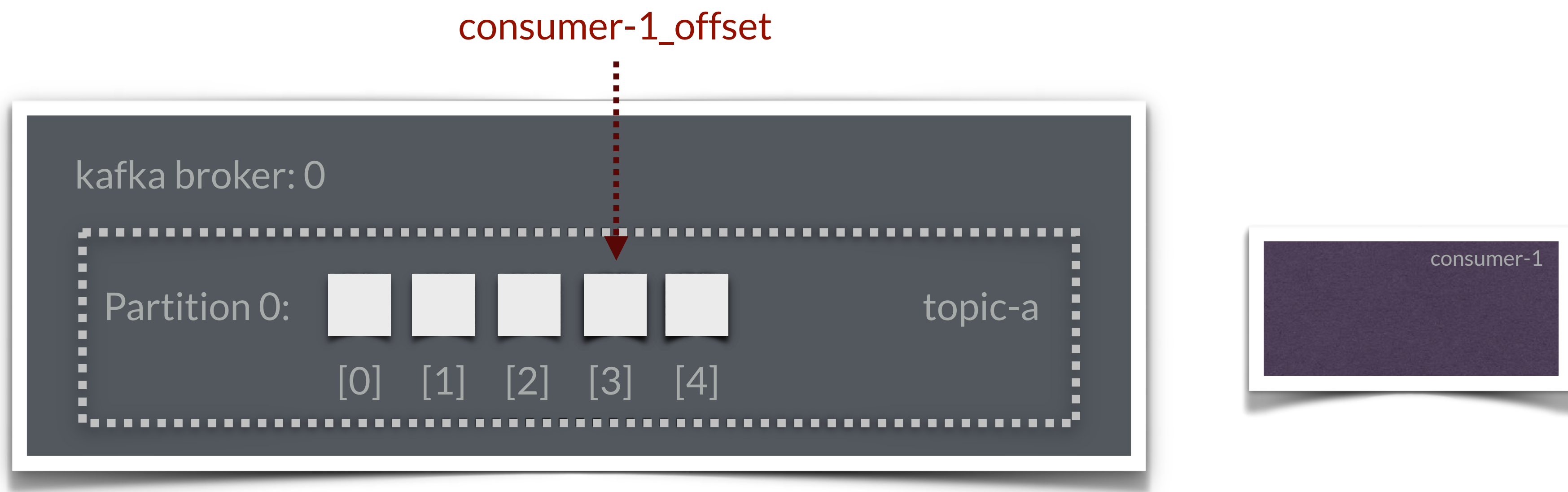


T-Z



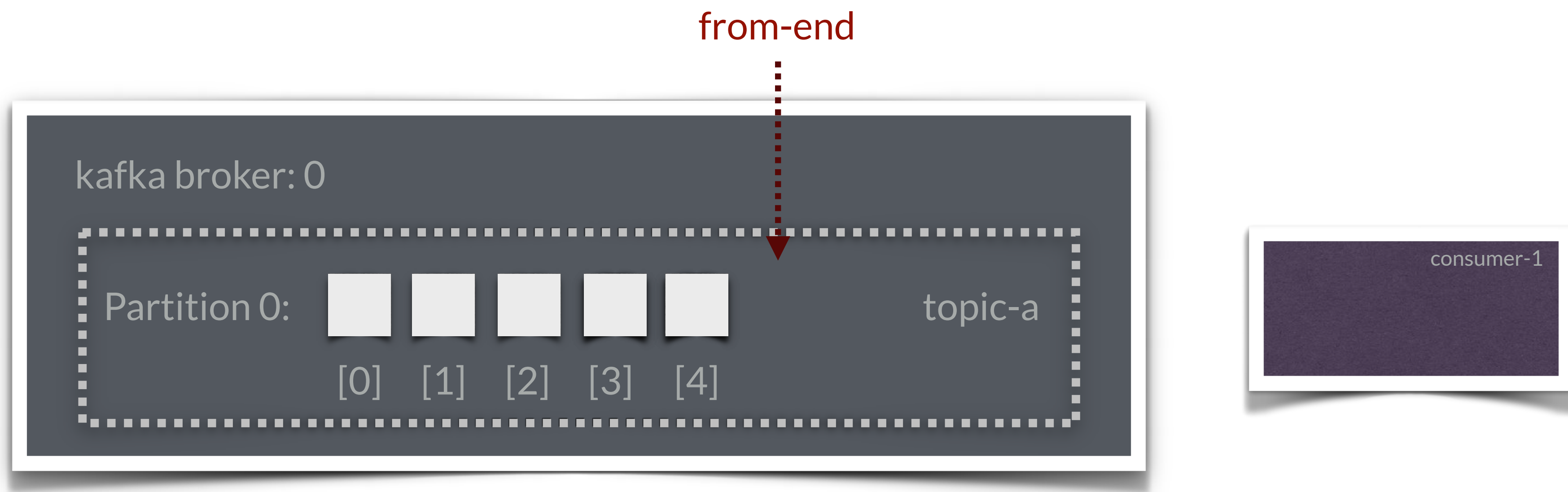
Kafka Consumers

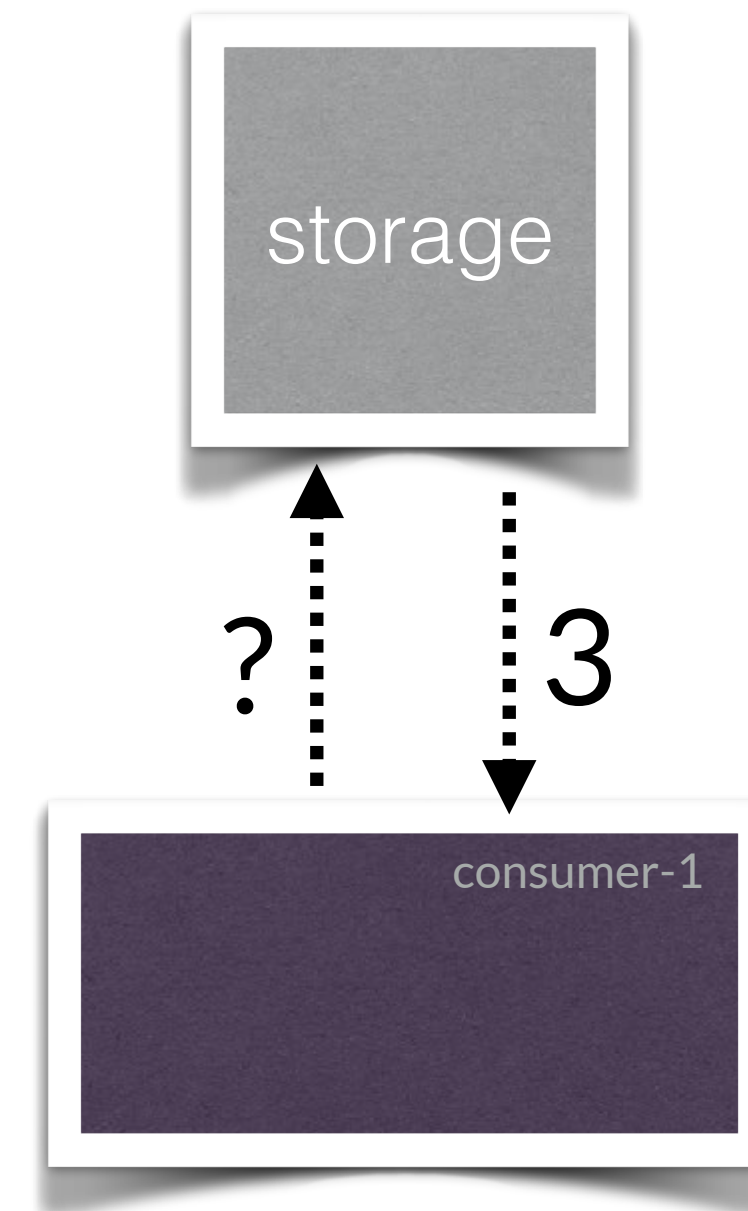
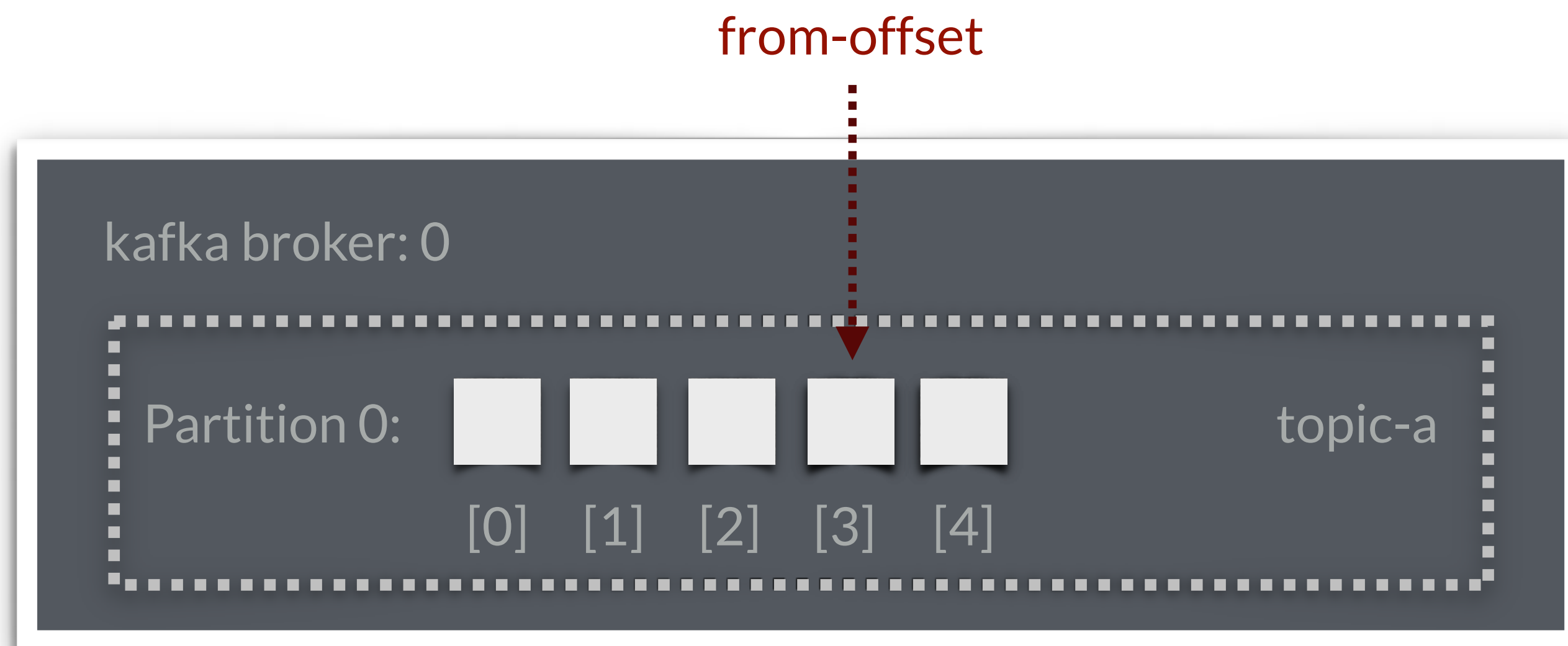




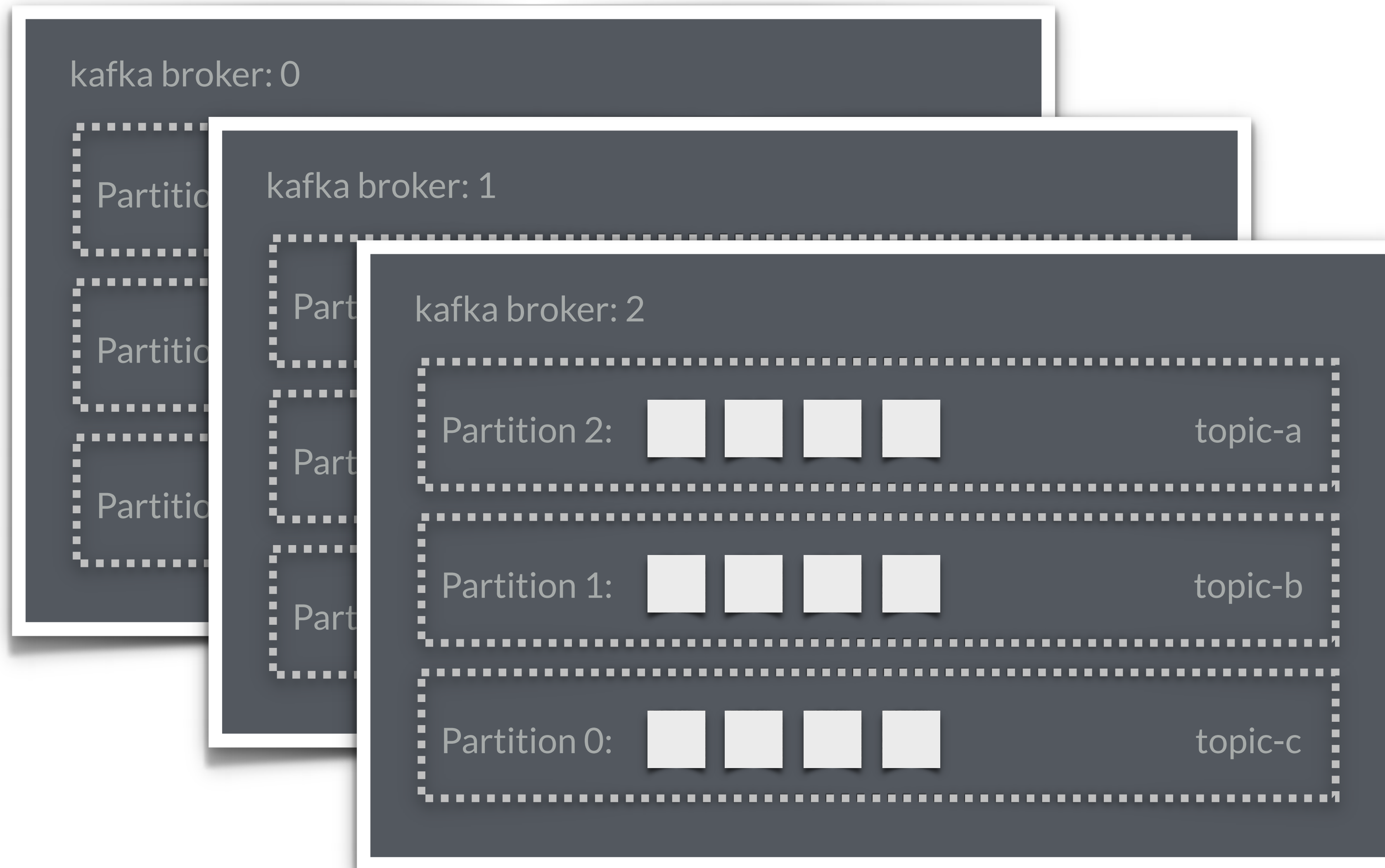
from-beginning







Kafka Partitions



Each partition is on a different broker,
therefore a single topic is scaled

kafka bro

Partition

Partition

Partition

Partition

Partition

kafka bro

Partition

Partition

Partition

Partition

Partition

kafka broker: 2

Partition 0:  topic-a

Partition 2:  topic-a

Partition 1:  topic-b

Partition 0:  topic-c

Partition 2:  topic-c

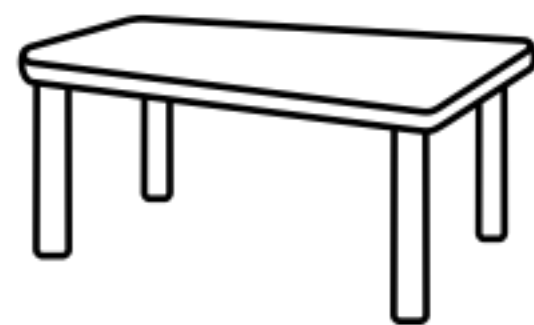
Partition 1:  topic-a

Partition 0:  topic-b

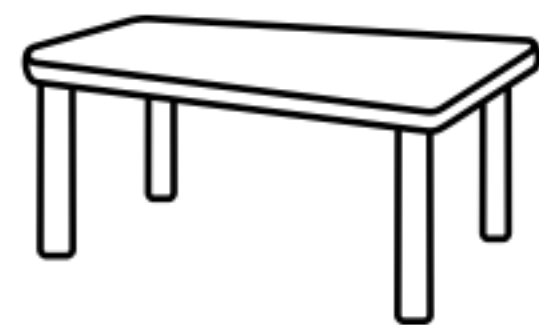
Partition 2:  topic-b

Partition 1:  topic-c

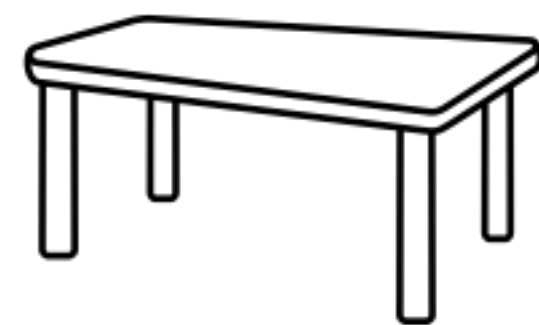
A-E



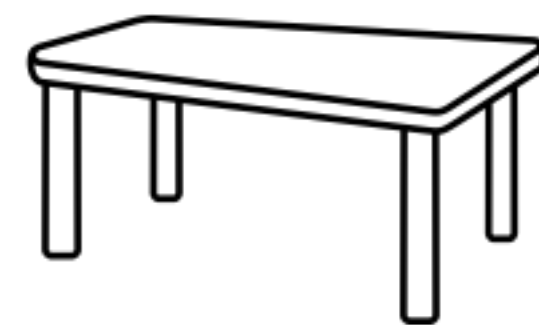
F-K



L-S



T-Z



Kafka Pub/Sub Comparisons

Comparison Pub/Sub

- Kafka as the distributed "git log"
- Ability to replayed in a consistent manner
- Kafka is also stored durable, in order, and deterministic, if key is available
- Data can also be distributed for resiliency
- Scalable and Elastic

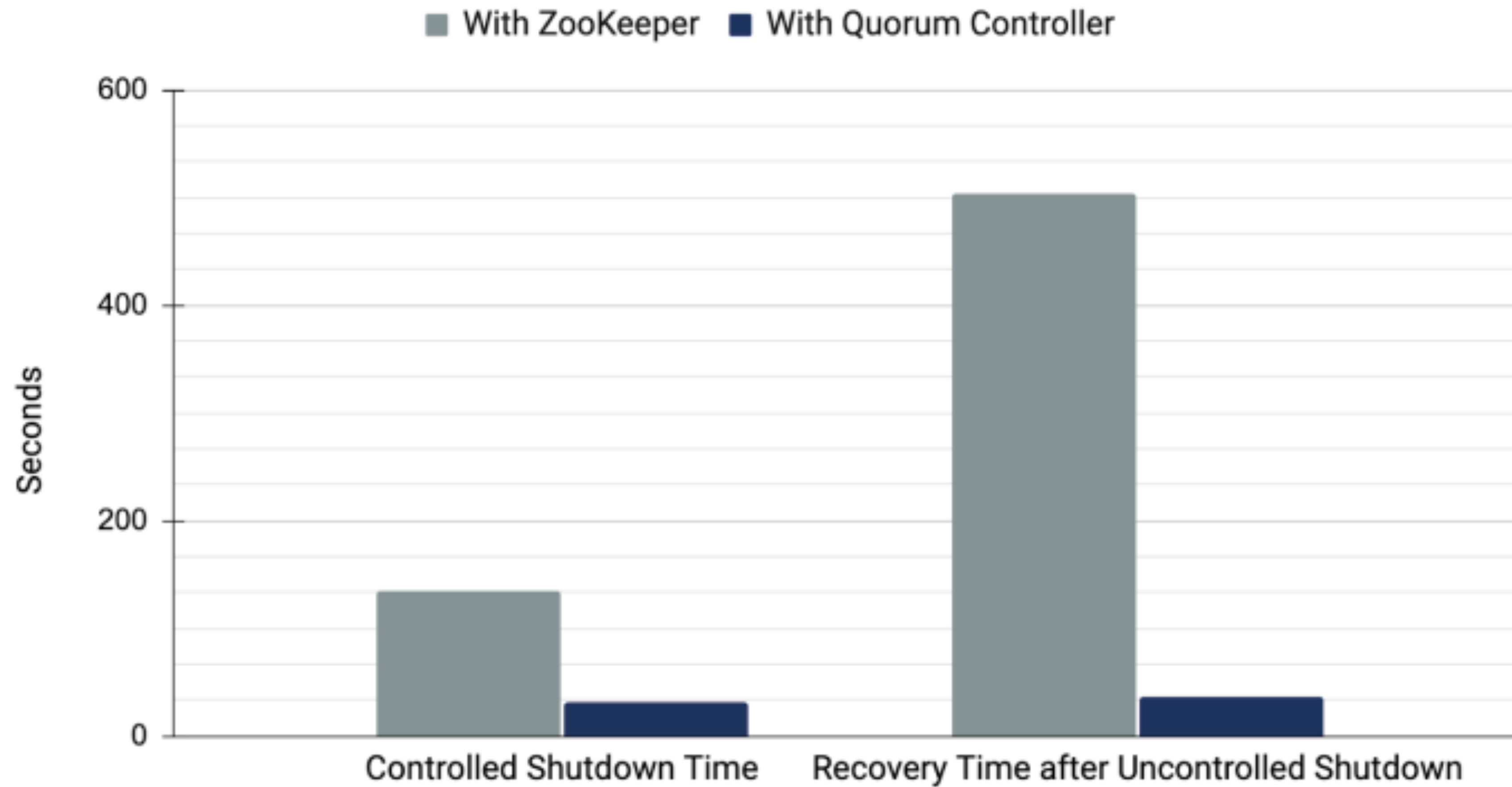
kRaft

kRaft

- Consensus Protocol, introduced in KIP-500.
- Used to replace Zookeeper with its own management.
- kRaft mode makes use of a new quorum controller service in Kafka which replaces the previous controller.
- Makes use of an event-based variant of the Raft consensus protocol.
- Combined mode, used for development is a Kafka node acts as a broker and also a kRaft controller

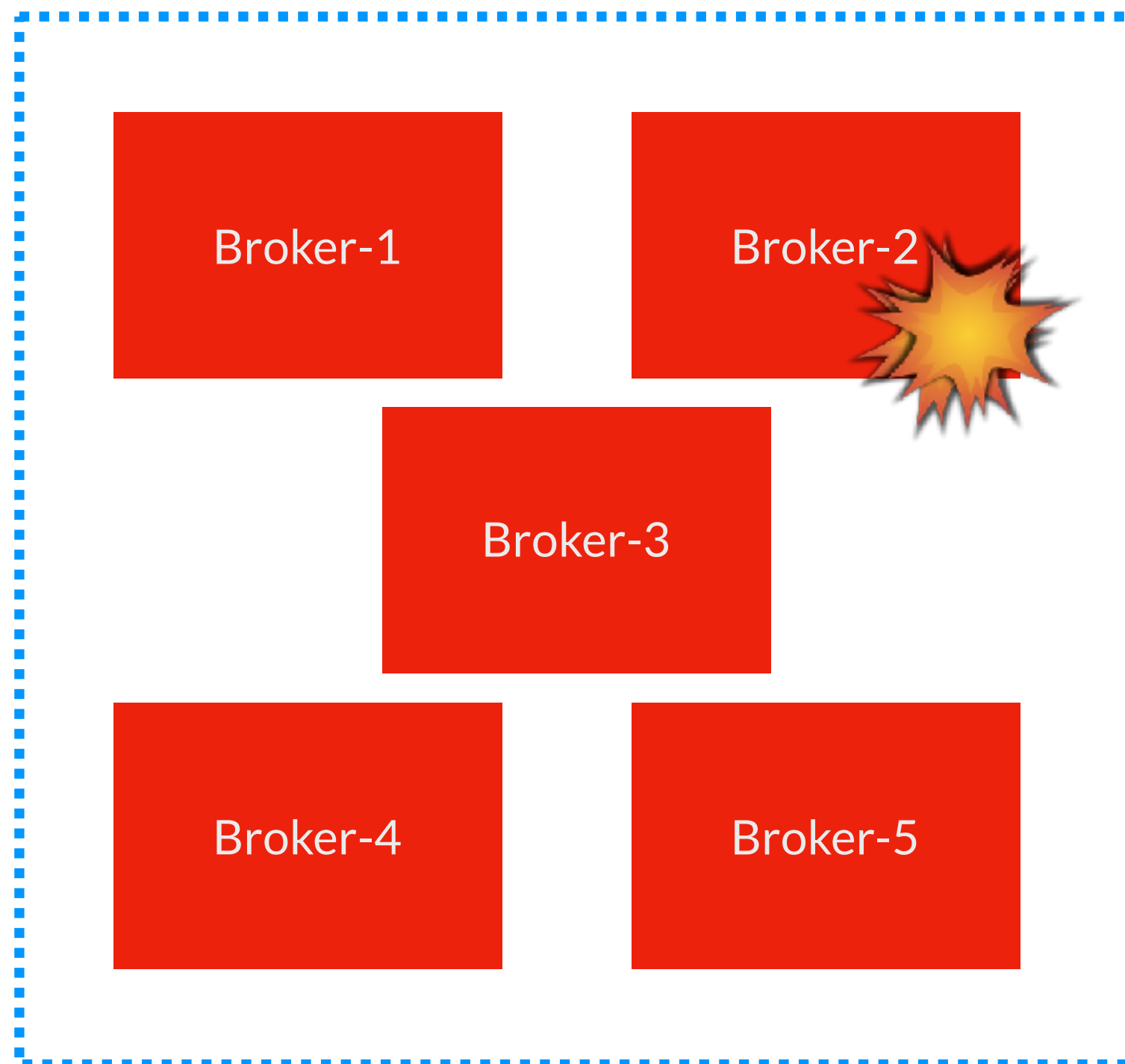
Timed Shutdown Operations In Apache Kafka with 2 Million Partitions

Faster is better

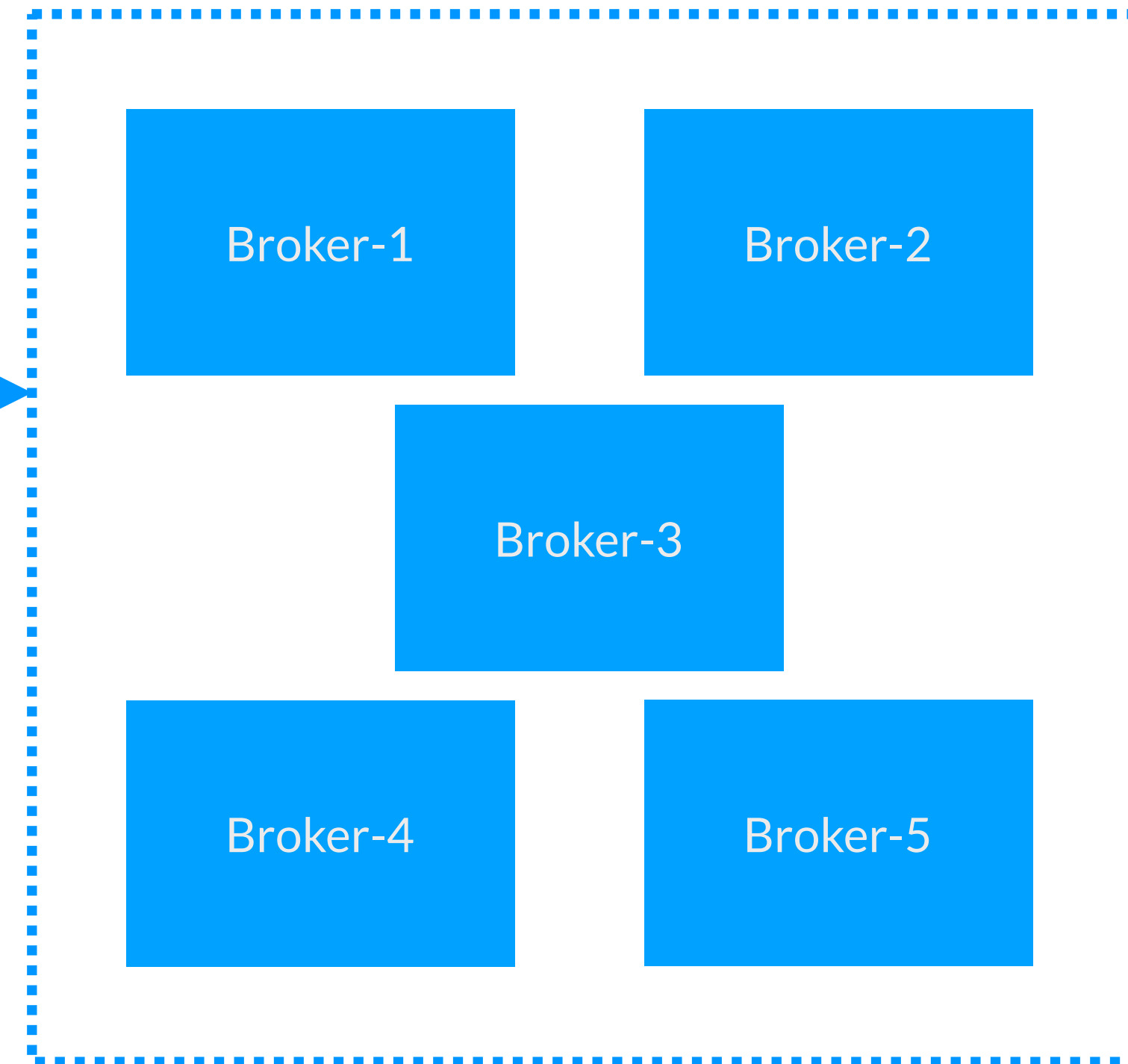


<https://developer.confluent.io/learn/kraft/>

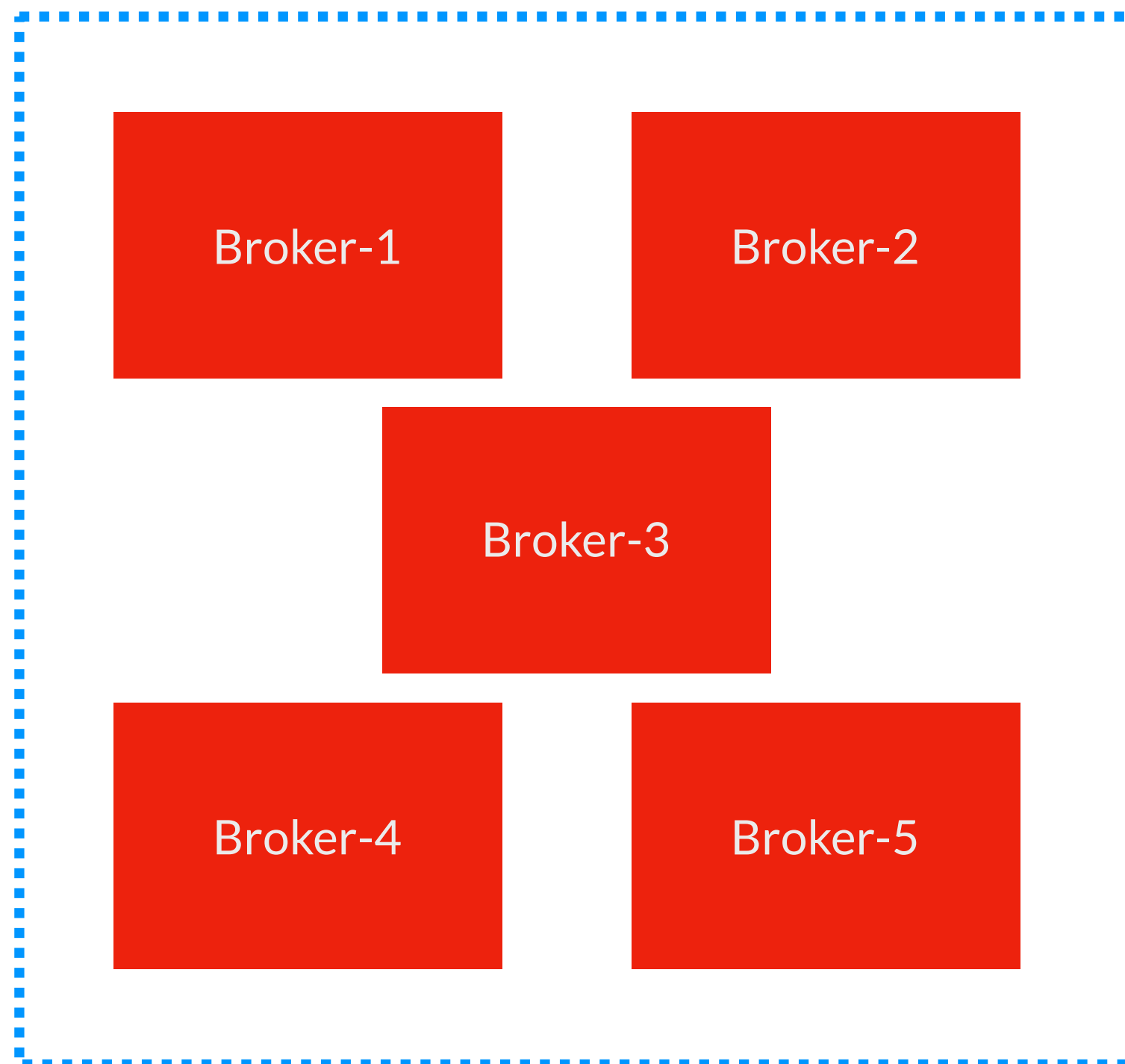
Kafka Cluster



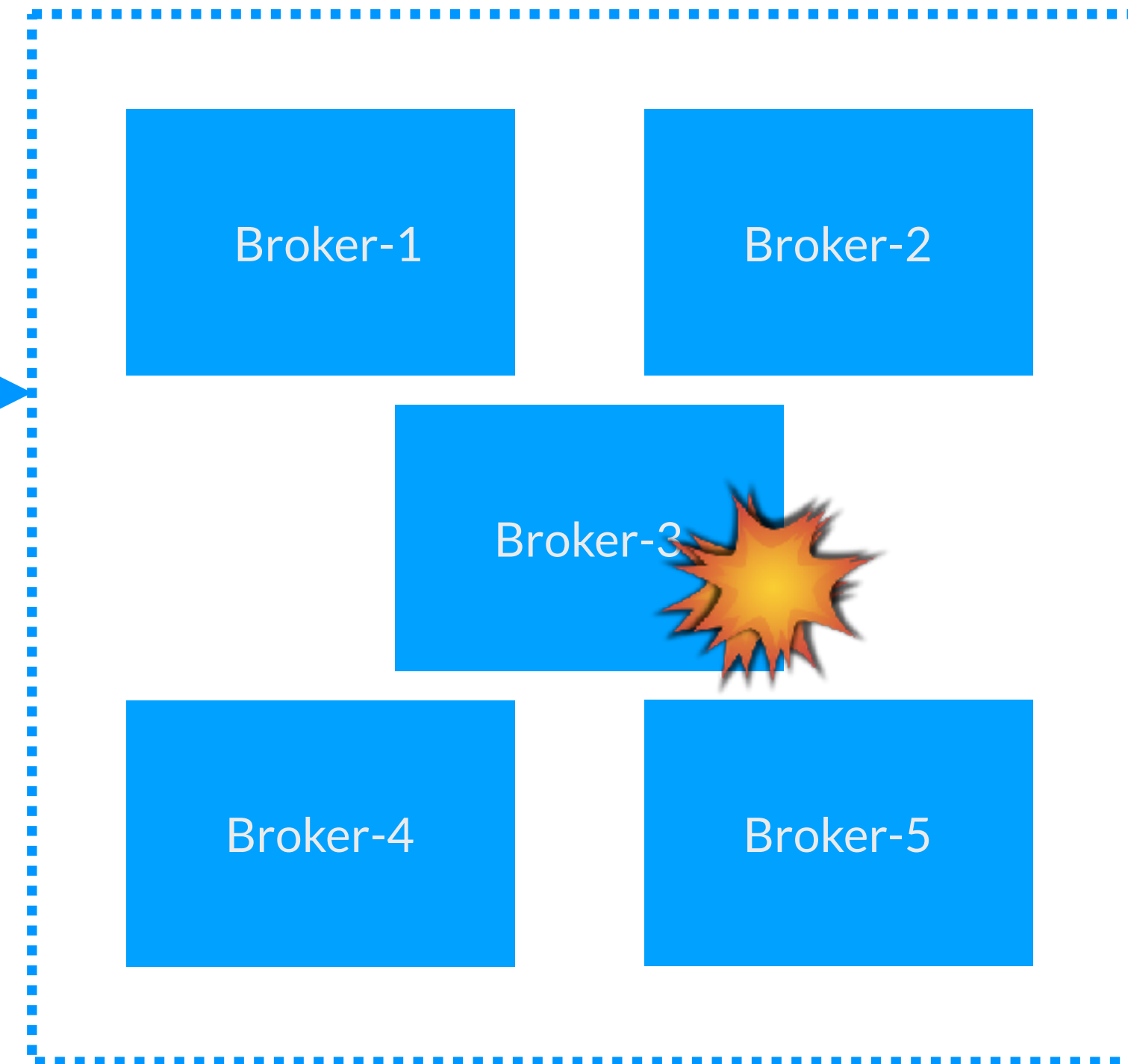
kRaft Quorum



Kafka Cluster



kRaft Quorum



Requirements



Hardware Requirements

- Do not co-locate other applications due to memory page cache pollution and will degrade performance
- Performant drive is required
- Storage capacity will need to be calculated by the expected messages per day and retention
- Slower networks can degrade the rate in which messages are produced

Cloud Requirements

- Analyze by data retention
- Analyze performance need by the producers
- If low latency is required, SSD should be considered
- Ephemeral Storage may be required (Elastic Block Storage)

Kafka Guarantees



Kafka Guarantees

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



Kafka CLI

Creating a Topic

Creating the topic orders with replication factor 2 and 4 partitions

```
$ /usr/bin/kafka-topics --create --bootstrap-server localhost:9092 \  
  --replication-factor 2 --partitions 4 --topic orders
```

Listing Topics

List the topics using one of the zookeeper nodes

```
$ /usr/bin/kafka-topics --bootstrap-server localhost:9092 --list
```


Sending a Message

Send a message to list of brokers for a particular topic

```
$ /usr/local/kafka/bin/kafka-console-producer.sh --broker-list <kafka-broker>:9092 --topic <topic>  
> Hello  
> I  
> am  
> sending  
> six  
> messages
```

Receiving a Message

Receiving the messages from the topic that was posted by the CLI producer

```
$ /usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server <kafka-broker>:9092 \  
--topic <topic> \  
--from-beginning  
  
> Hello  
> I  
> am  
> sending  
> six  
> messages
```

Showing Distributed Partitions

Showing how partitions are distributed

```
$ /usr/local/kafka/bin/kafka-topics.sh --describe \  
  --topic <topic-name> \  
  --bootstrap-server localhost:9092
```

Kafka Programming

Producers

Establishing Properties

- Construct a `java.util.Properties` object
- Provide two or more locations where the bootstrap servers are located
- Provide a Serializer for the key
- Provide a Serializer for the value

```
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);
```

Create a Producer Object

- Construct a `org.apache.kafka.clients.producer.KafkaProducer` object
- A Kafka Producer is thread-safe
- Inject the `Properties` into the Producer object

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

Creating a Record/Message

- Create a `org.apache.kafka.clients.producer.ProducerRecord`
- Accepts many parameters but the main ones are:
 - Topic
 - Key (if applicable)
 - Value

```
ProducerRecord<String, Integer> producerRecord =  
    new ProducerRecord<>("my_orders", state, amount);
```

Sending a message

- Send the record by calling `send` on the `Producer`
- Returns a `Future` object to process the results on another Thread.
- The `Future` will contain `RecordMetadata`, an object that has information about your send.

```
Future<RecordMetadata> send = producer.send(producerRecord);
```


RecordMetadata

Contains information about your send including the messages

```
if (metadata.hasOffset()) {  
    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());
```

Sending with a Callback

- Alternately, you can send with a Callback (lambda)
- Callback is an interface, can be use as a lambda
- If RecordMetadata is null there was an error, if Exception is null the send was successful

```
producer.send(producerRecord, new Callback() {  
    @Override  
    public void onCompletion(RecordMetadata metadata,  
                             Exception e) {  
        ...  
    }  
})
```

Using a Closure to capture Key and Value

- RecordMetadata does not have information on key and value
- Using a closure you can obtain that in the block of your lambda

```
producer.send(producerRecord, (metadata, e) -> {  
    if (metadata != null) {  
        System.out.println(producerRecord.key());  
        System.out.println(producerRecord.value());  
    }  
})
```

Be a good citizen, close your resources

- When you need to terminate, flush messages from the bufferpool
- Close the Producer

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

Closing Resources in a Shutdown Hook

- `Runtime.getRuntime().addShutdownHook(...)` will listen for SIGTERM (CTRL+C)
- This would make an excellent place to flush and close, and close any loops that you may have created

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {  
    done.set(true);  
    producer.flush();  
    producer.close();  
}));
```

Acknowledgements & Retries

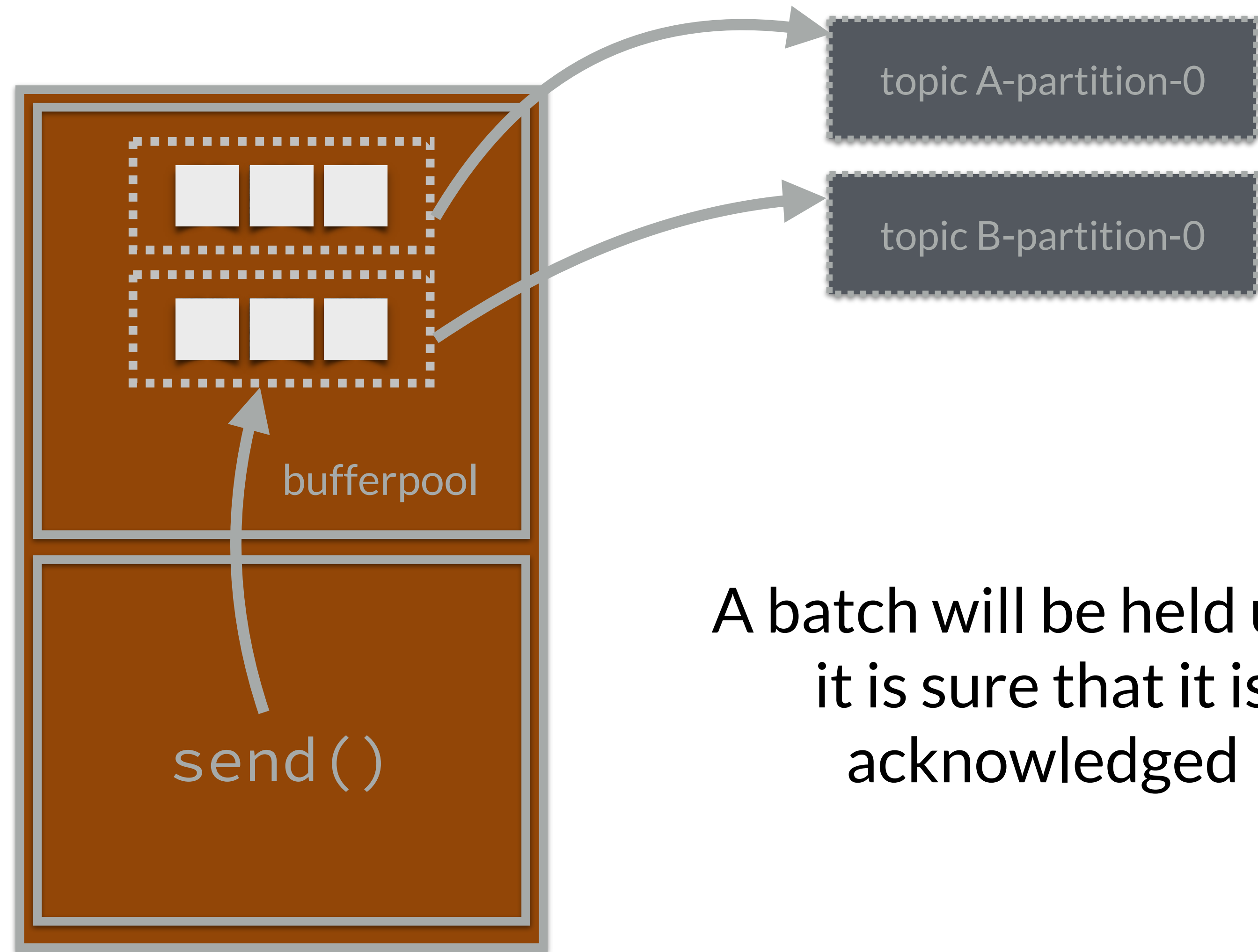
Acks

acks controls how many partition replicas must receive the record before the write is considered a success.

```
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, 20);
```

Acks

acks	description
0	No acknowledgment, assume all is well
1	Leader will acknowledge, Not the replicas
all	All replicas must acknowledge. Higher latency, safest



A batch will be held until
it is sure that it is
acknowledged

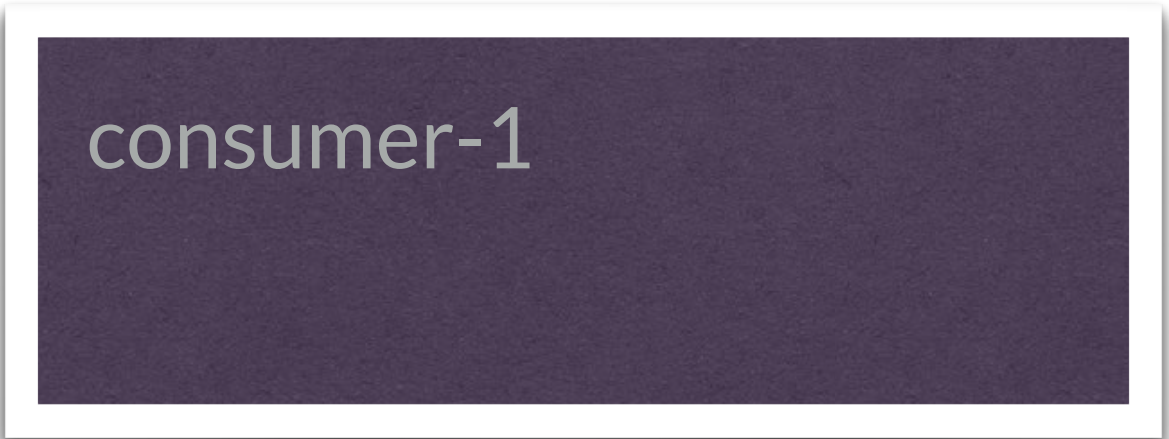
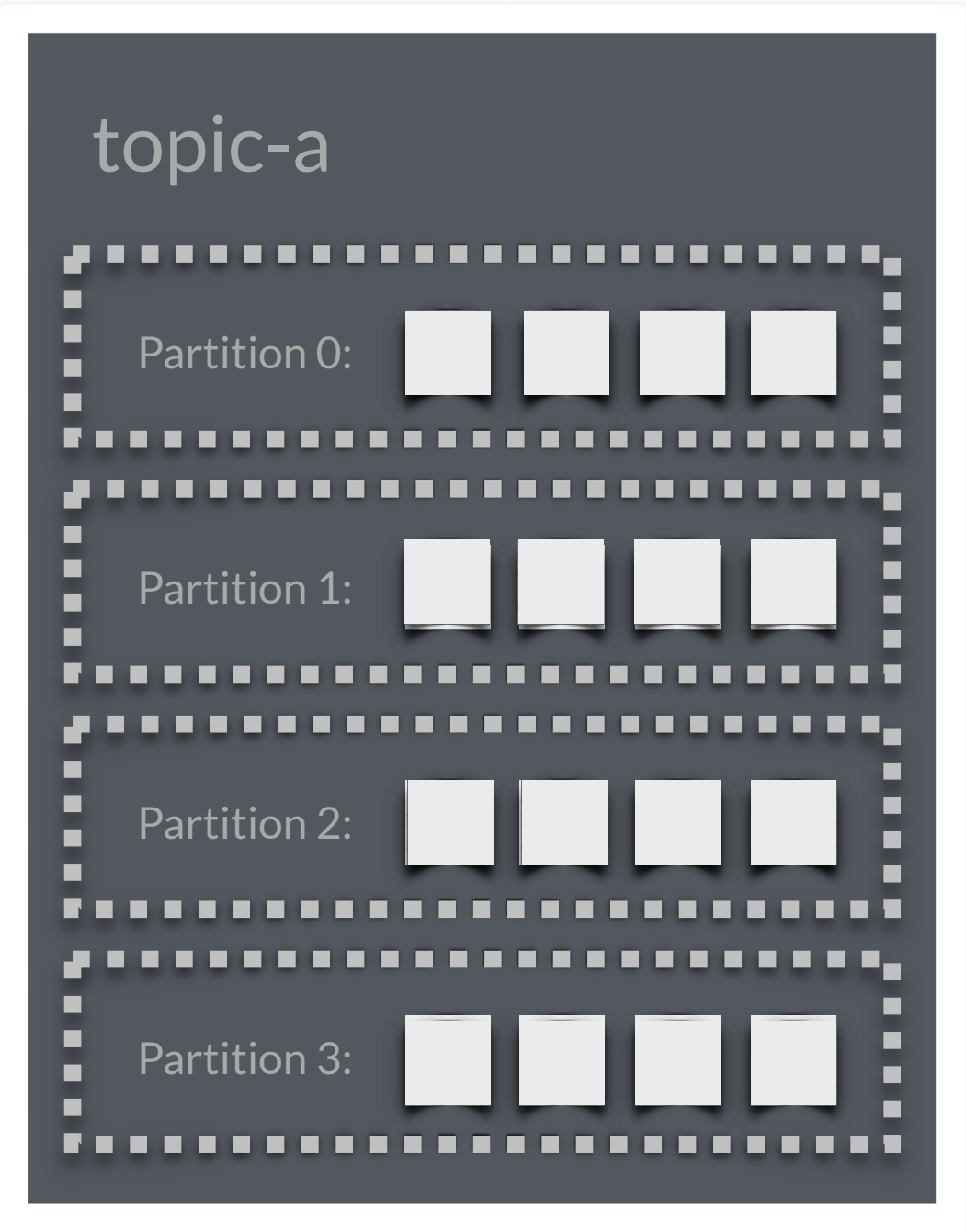
Demo: Producers

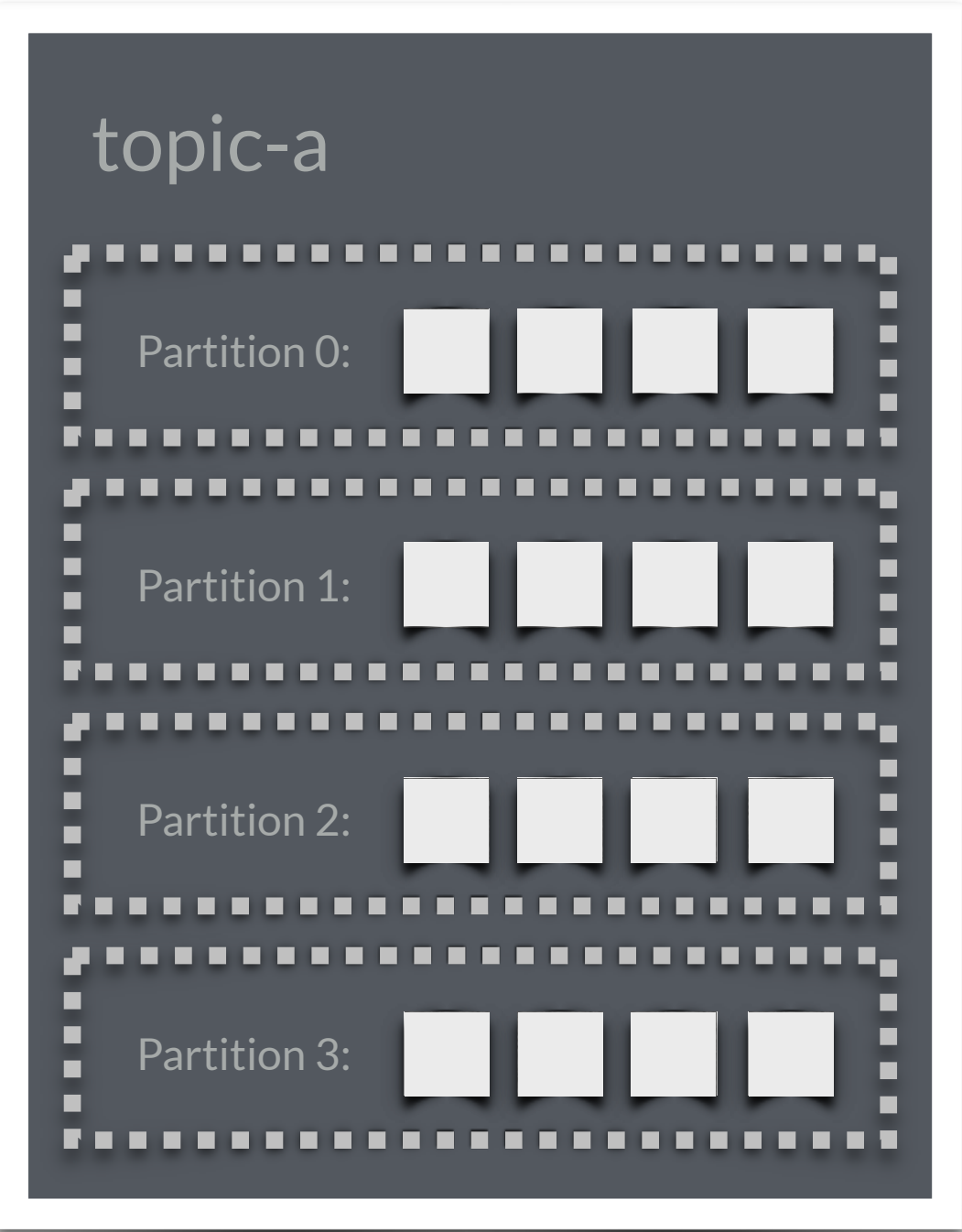
Kafka Programming

Consumers & Groups

Kafka Consumer Groups

- Consumers are typically done as a group
- A single consumer will end up inefficient with large amounts of data
- A consumer may never catch up
- **Every consumer should be on it's own machine, instance, pod**





consumer-1

consumer-2

consumer-3

consumer-4

Kafka's Goal

Kafka scales to large amount of different consumers without affecting performance

Kafka Consumer Threading

- There are no multiple consumers that belong to the same group in one thread. **One consumer per one thread.**
- There are not multiple threads running one consumer, **One consumer per one thread.**

__consumer_offsets

- Topic on Kafka brokers that contain information about consumer and their offsets, stored in Kafka's data directory

Establishing Properties

- Construct a `java.util.Properties` object
- Provide two or more locations where the bootstrap servers are located
- Provide a "Team Name", officially called a `group.id`
- Provide a `DeSerializer` for the key
- Provide a `DeSerializer` for the value

```
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,
    "org.apache.kafka.common.serialization.IntegerDeserializer");
```

Create a Consumer Object

- Construct a `org.apache.kafka.clients.consumer.KafkaConsumer` object
- A Kafka Consumer is **not thread-safe**
- Inject the `Properties` into the Consumer object

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

Processing Messages

- Use the consumer that you have constructed, and call `poll` with pulse time
- The poll is a max sleep time, if a message is ready it will download a batch, `ConsumerRecords`
- Iterate through each record in the batch with a for-loop, process the message

```
while (!done.get()) {
    ConsumerRecords<String, String> records =
        consumer.poll(Duration.of(500, ChronoUnit.MILLIS));
    for (ConsumerRecord<String, Integer> 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());
    }
}
```

Be a good citizen, close your resources

- When you need to terminate, let the *Kafka group coordinator* know you are done
- Close the Consumer

```
consumer.close();
```

Consumer Offset Reset

Consumer Offset Reset

- When a consumer is assigned it must determine what offset for the partition to start
- Position is set according to your consumer's reset policy, either `earliest` or `latest`
- This is only applicable if there is no valid offset in the `__consumer_offset__` topic

```
Properties properties = new Properties();  
...  
properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
  
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Manual Commits

Manual Commits

- Consumers will commit automatically every 5 seconds
- If your consumer fails within at minimum for 5 seconds, another consumer may get those messages
- Might be too long before commit, in such case you may want to handle commits yourself
- Turn off your auto-commit by setting `ENABLE_AUTO_COMMIT_CONFIG` to `false`

```
Properties properties = new Properties();  
...  
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);  
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Synchronous Commit

- After turning off auto-commit
- Committing and blocking synchronously yourself
- Synchronous commits will block

Synchronous Commit

- After turning off auto-commit
- Committing and blocking synchronously yourself
- Synchronous commits will block

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

```
...
```

```
consumer.commitSync(); //Block
```

Asynchronous Commit

- After turning off auto-commit
- Committing and letting it commit asynchronously
- Advantage: Speed, Higher throughput
- Disadvantage: You will get error reports asynchronously

```
consumer.commitAsync((offsets, exception) -> {  
    //offsets  
    //exception  
});
```

Using both kinds of commits

- You can commit asynchronously within the loop
- Then commit synchronously outside the loop
- Reason, if you are out of the loop, likely you want to shut down

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> done.set(true)));

while (!done.get()) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.of(500, ChronoUnit.MILLIS));
    for (ConsumerRecord<String, String> record : records) {
        ...
    }
    consumer.commitAsync((offsets, exception) -> {
        //offsets
        //exception
    });
}
consumer.commitSync(); //Block
consumer.close();
```

Consumer Rebalancing

Consumer Rebalance

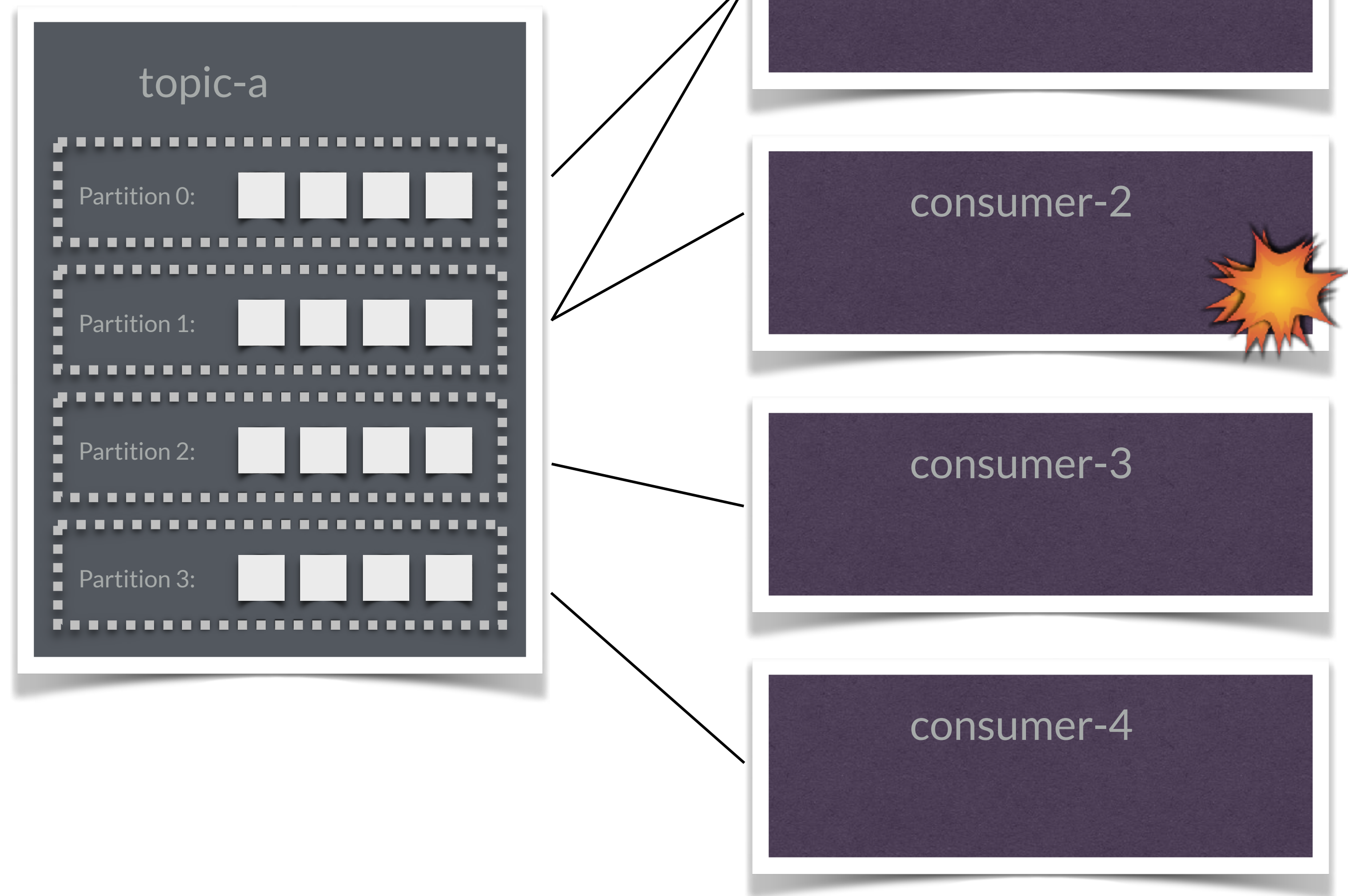
- When one partition is moved from one consumer to another, this is known as a *rebalance*.
- A way to mitigate when either consumers go down or when consumers are added.
- Although unavoidable, this will cause an unfortunate pause, and it ***will lose state***.

Groups and Heartbeats

- At regular intervals, consumers will send heartbeats to the broker, to let it know it is alive and still reading data
- Heartbeats are sent to a Kafka broker called the *group coordinator*
- They are sent in a separate band, aka it's own thread.

Leaving the Group

- When a consumer leaves the group, it lets the *group coordinator* know it is done
- The Kafka broker then triggers a rebalance of the group.



Consumer Rebalance Listener


- You can implement a `ConsumerRebalanceListener` and be alerted when a rebalance has occurred
- The rebalance listener requires that you implement `onPartitionsRevoked`, and `onPartitionsAssigned`

```
consumer.subscribe(Collections.singletonList("my_orders"),
    new ConsumerRebalanceListener() {
        @Override
        public void onPartitionsRevoked(Collection<TopicPartition> collection) {
            System.out.println("Partition revoked:" +
                collectionTopicPartitionToString(collection));
            consumer.commitAsync();
        }


        @Override
        public void onPartitionsAssigned(Collection<TopicPartition> collection) {
            System.out.println("Partition assigned:" +
                collectionTopicPartitionToString(collection));
        }
    });
```

Lab: Consumers

Kafka Visualization



Kafka Visualization

 **Apache Kafka** is a distributed event streaming platform. Using the tool below you can simulate how data flows through a replicated Kafka topic, to gain a better understanding of the message processing model.

Configuration

Choose the number of partitions - between which data will be evenly distributed. Experiment with various counts of brokers, turning them on and off, and seeing how the system adapts. Make sure to store data in replicas, so that they are not lost! Simulate load by increasing the consume interval. Finally, verify how offsets are committed, and see how this impacts redelivery when consumers or brokers are added/removed.

Partitions :

12345

Brokers :

12345


Replication factor :

12345

Producer

Producing interval : ticks

Consumers

sumer 1

Consume interval: ticks

← Hide Configuration

≡ Hide Descriptions

Animation speed:

5s

4s

3s

2s

1s

⏸ Pause

🔄 Restart

> What is Apache Kafka?

Producer:

P

7

8

9

10

11

12

13

14

15

16

17

> What is a Kafka producer?

6

Brokers:

1

4

2

2

5

3

1

1

4

2

2

5

3

1

> What is a Kafka broker?

Consumers:

<https://softwaremill.com/kafka-visualisation/>

Retention Settings

Retention

- Durable Storage over a Period of Time
- Can either be configured in time or size
- Once reached messages are expunged

Retention Formula

Who administers Kafka can update the default retention time or size

```
log.retention.hours = 120  
log.retention.bytes = 3221225472
```

Default Hours are set to (168 hours) or one week

You may also use **log.retention.minutes** or **log.retention.ms**, whichever is smallest wins.

You can also configure retention per topic, like the following

```
bin/kafka-configs --zookeeper localhost:2181 --entity-type topics --entity-name my-topic --alter --add-config retention.ms=128000
```

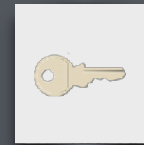
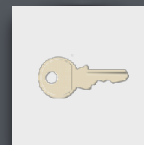
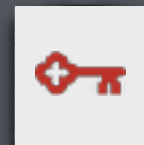
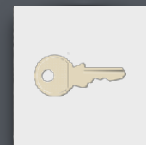
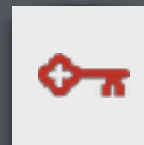
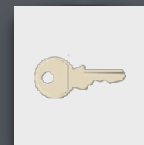
Compaction

Compaction

- A form of retention where messages of the same key where only the latest message will be retained.
- Compaction is performed by a *cleaner thread*

kafka broker: 0

Partition 0:



[0]

[1]

[2]

[3]

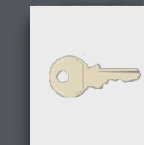
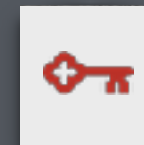
[4]

[5]



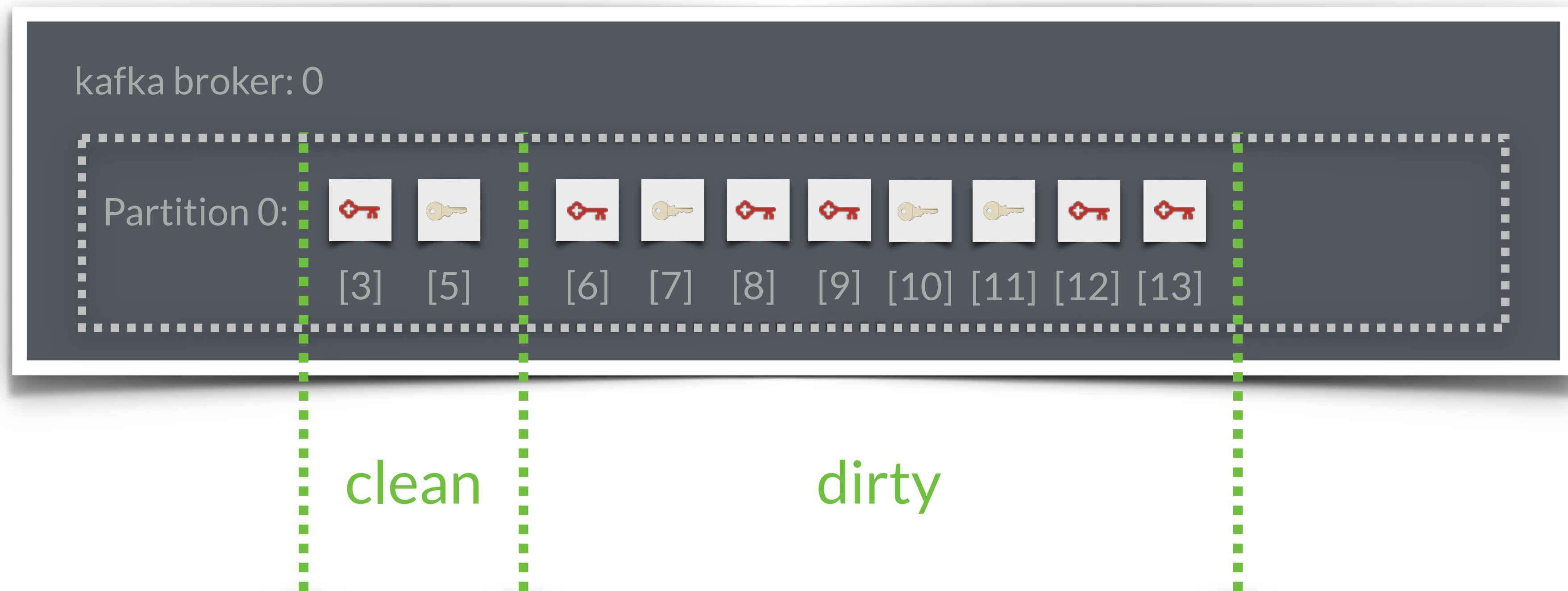
kafka broker: 0

Partition 0:



[3]

[5]



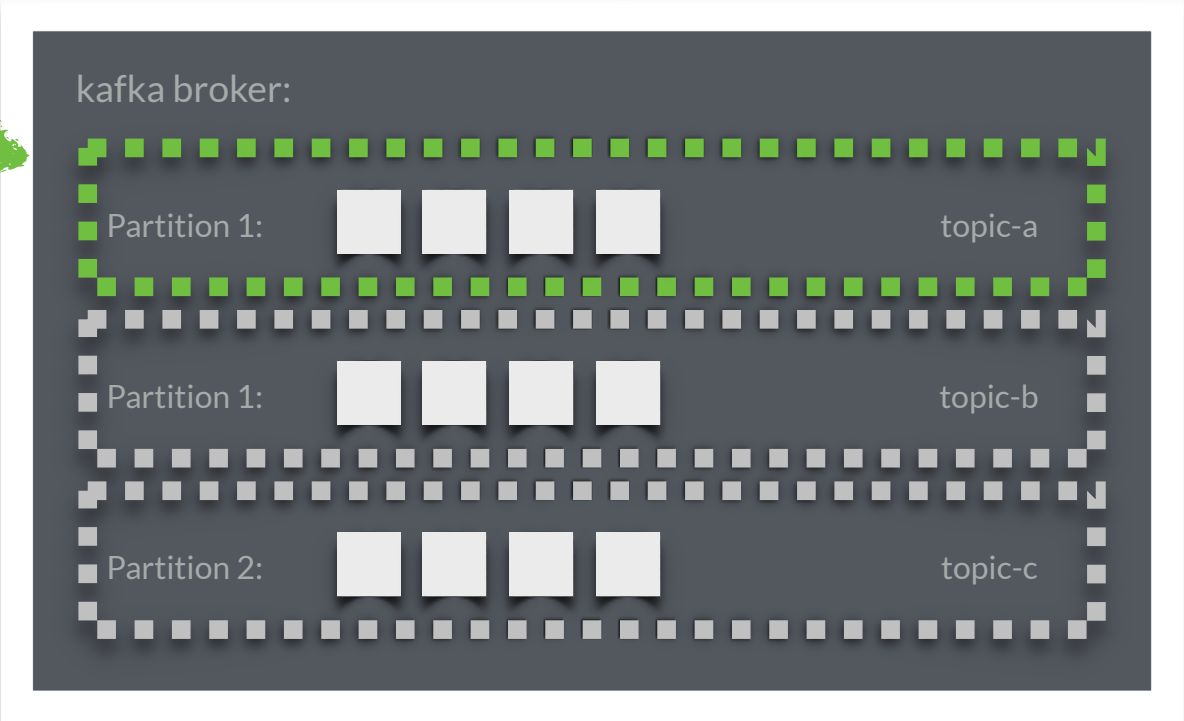
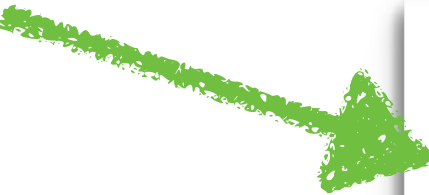
Kafka will start compacting when 50% of the topic contains dirty records

Resiliency

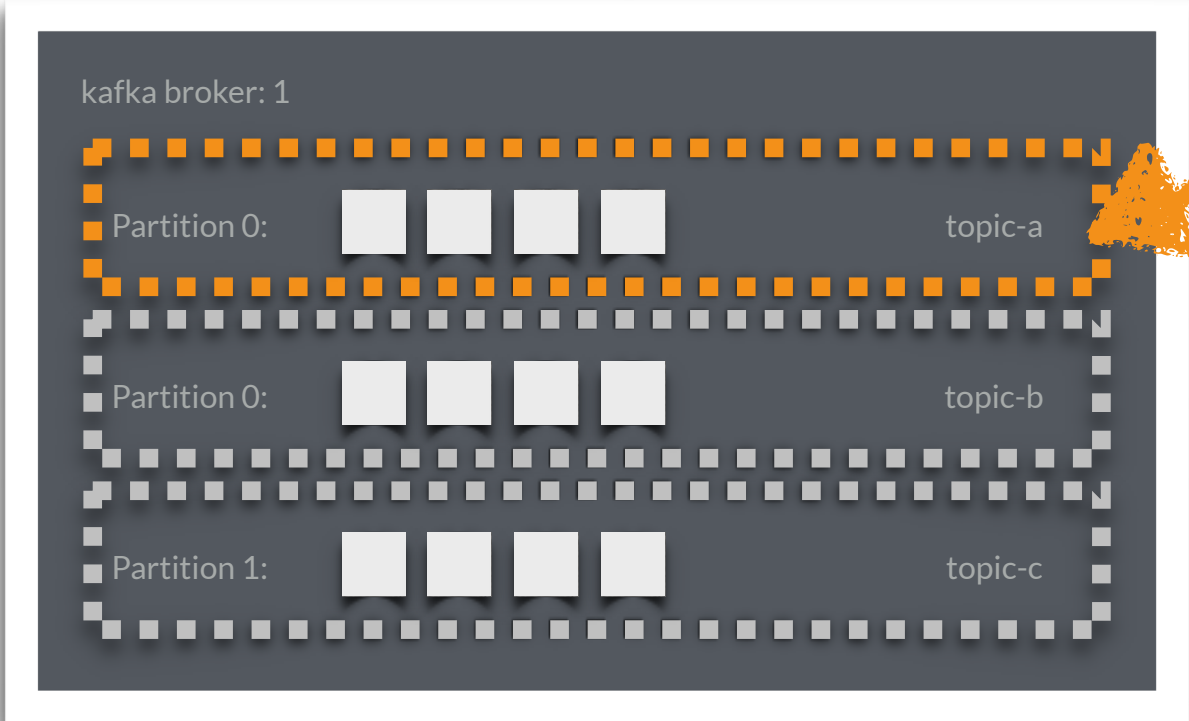
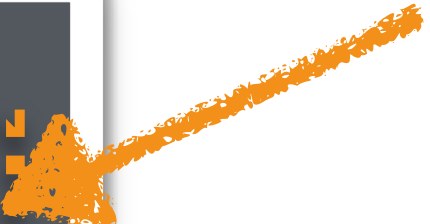
ISR (In sync replicas)

- Given a leader partition, an in sync replica is one that has been kept up to date within the last 10 seconds
- This is configurable
- During a crash, the closest ISR will retain control for failover

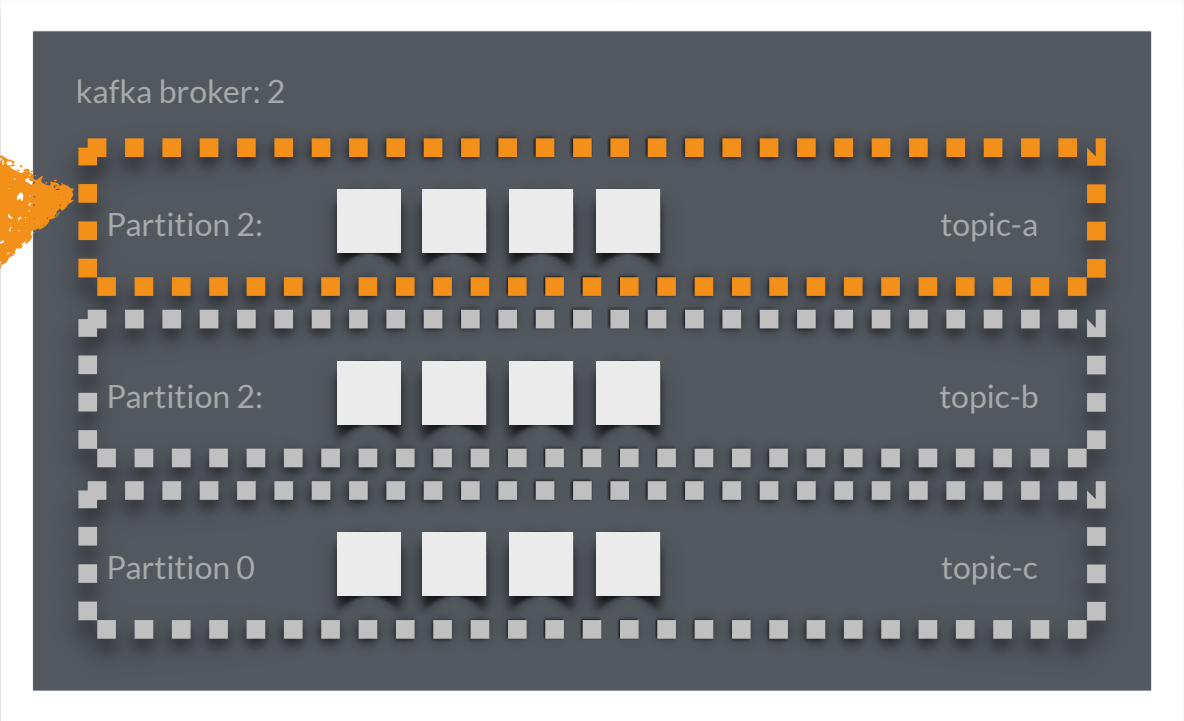
Leader



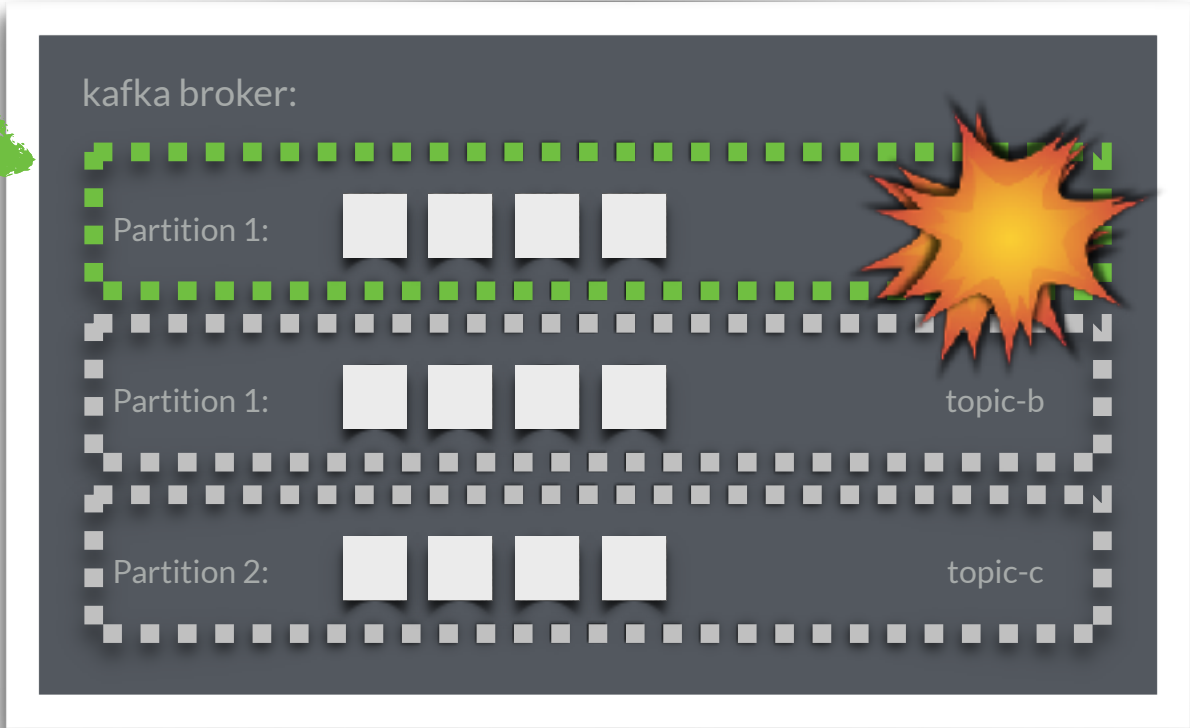
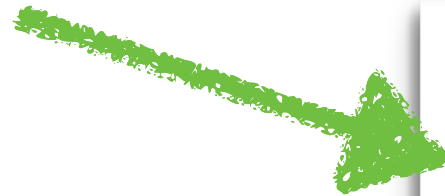
ISR



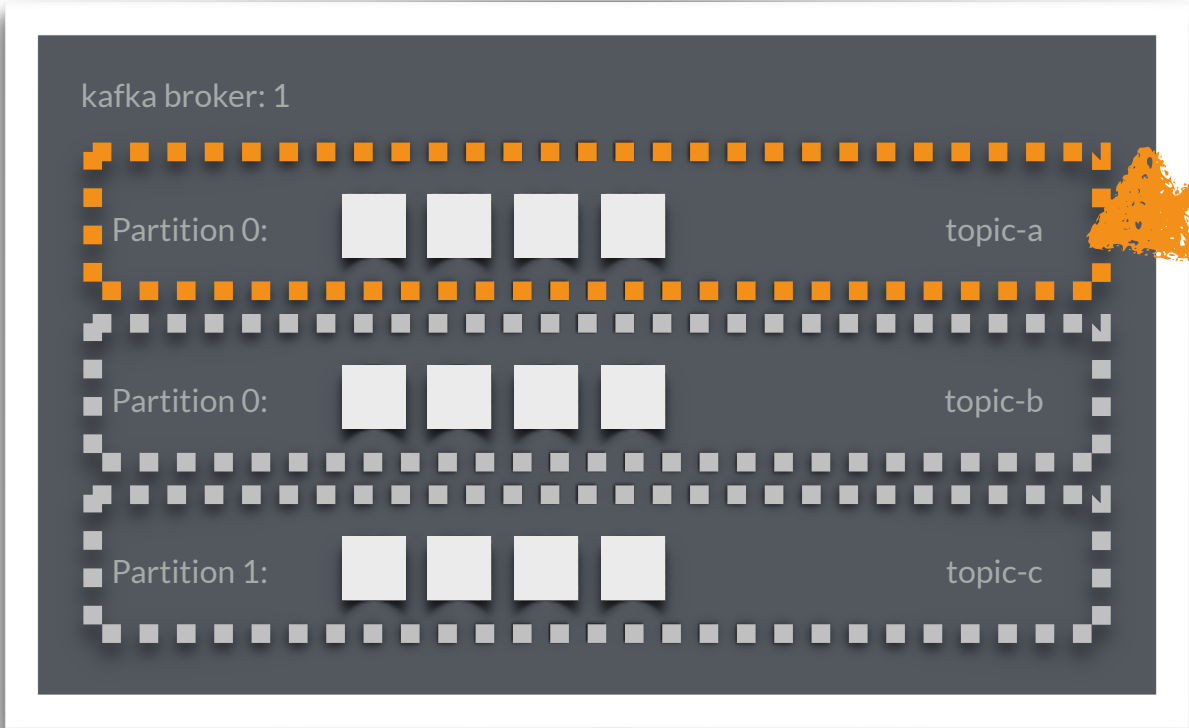
ISR



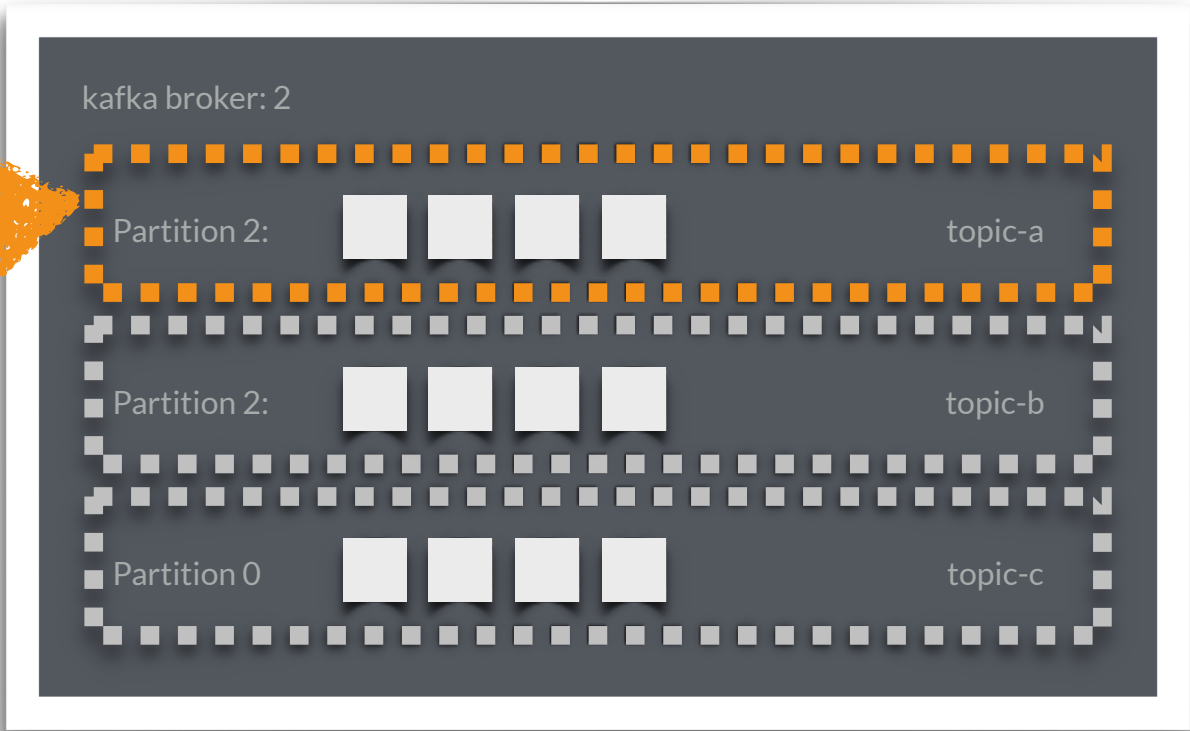
Leader



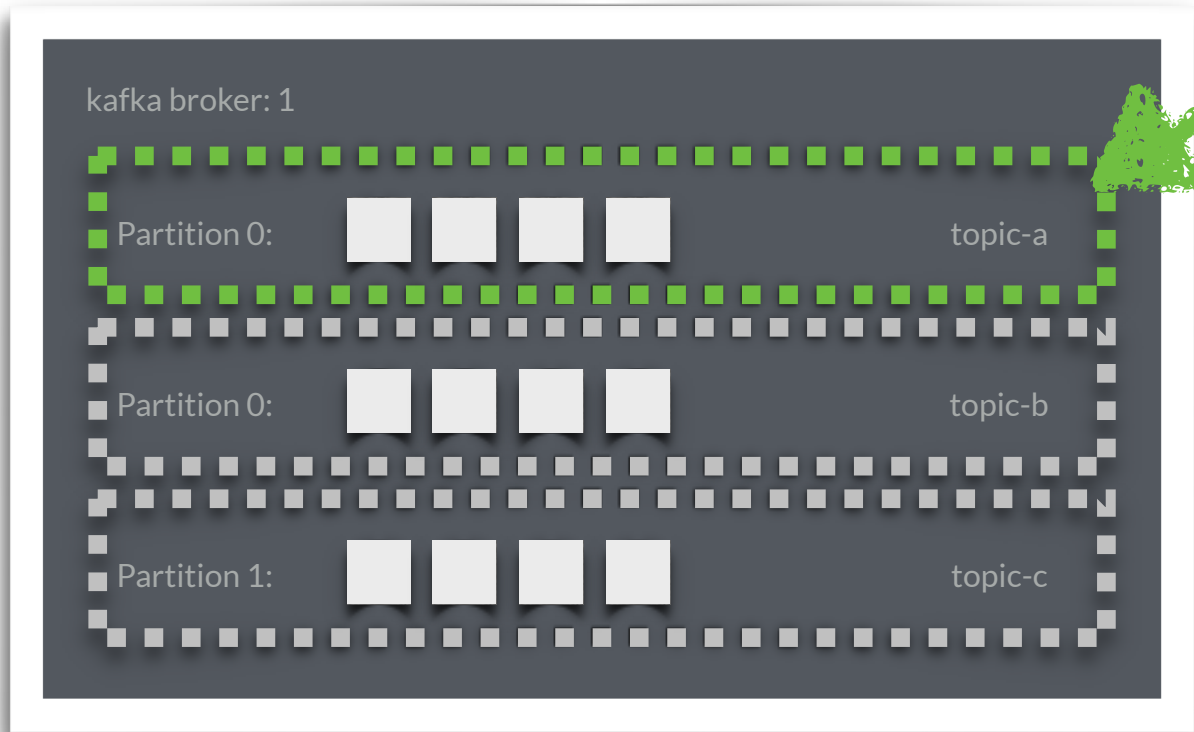
ISR



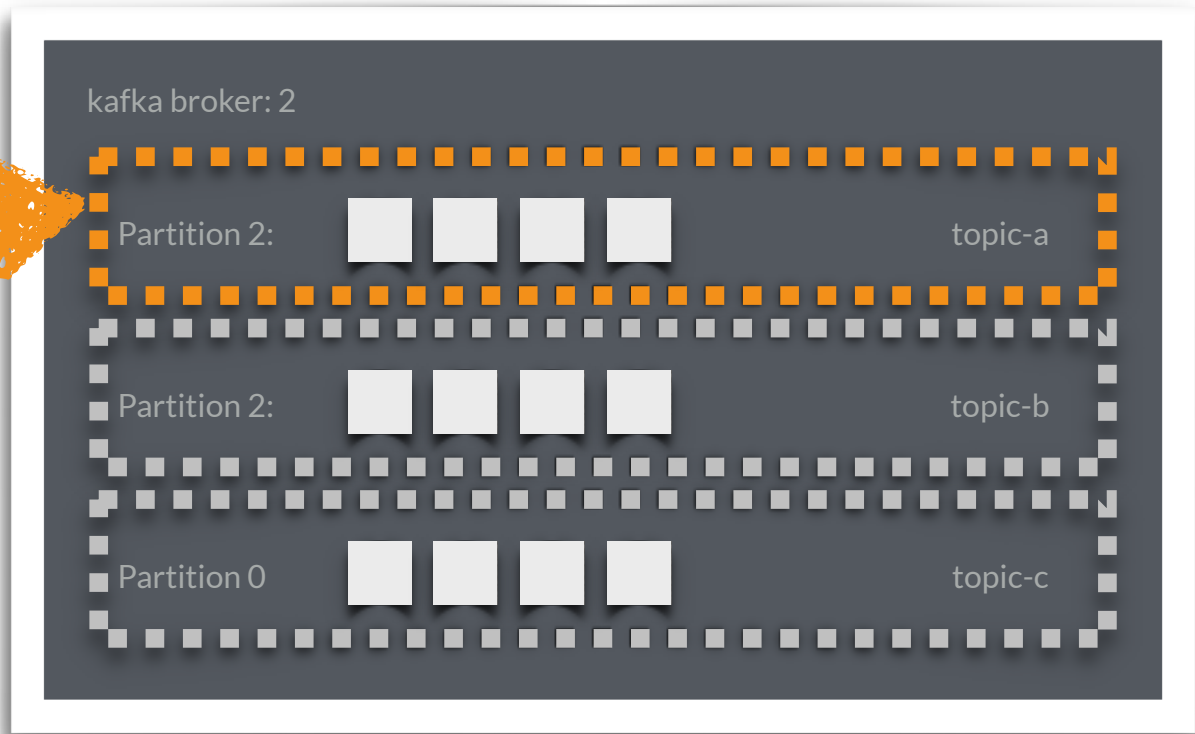
ISR



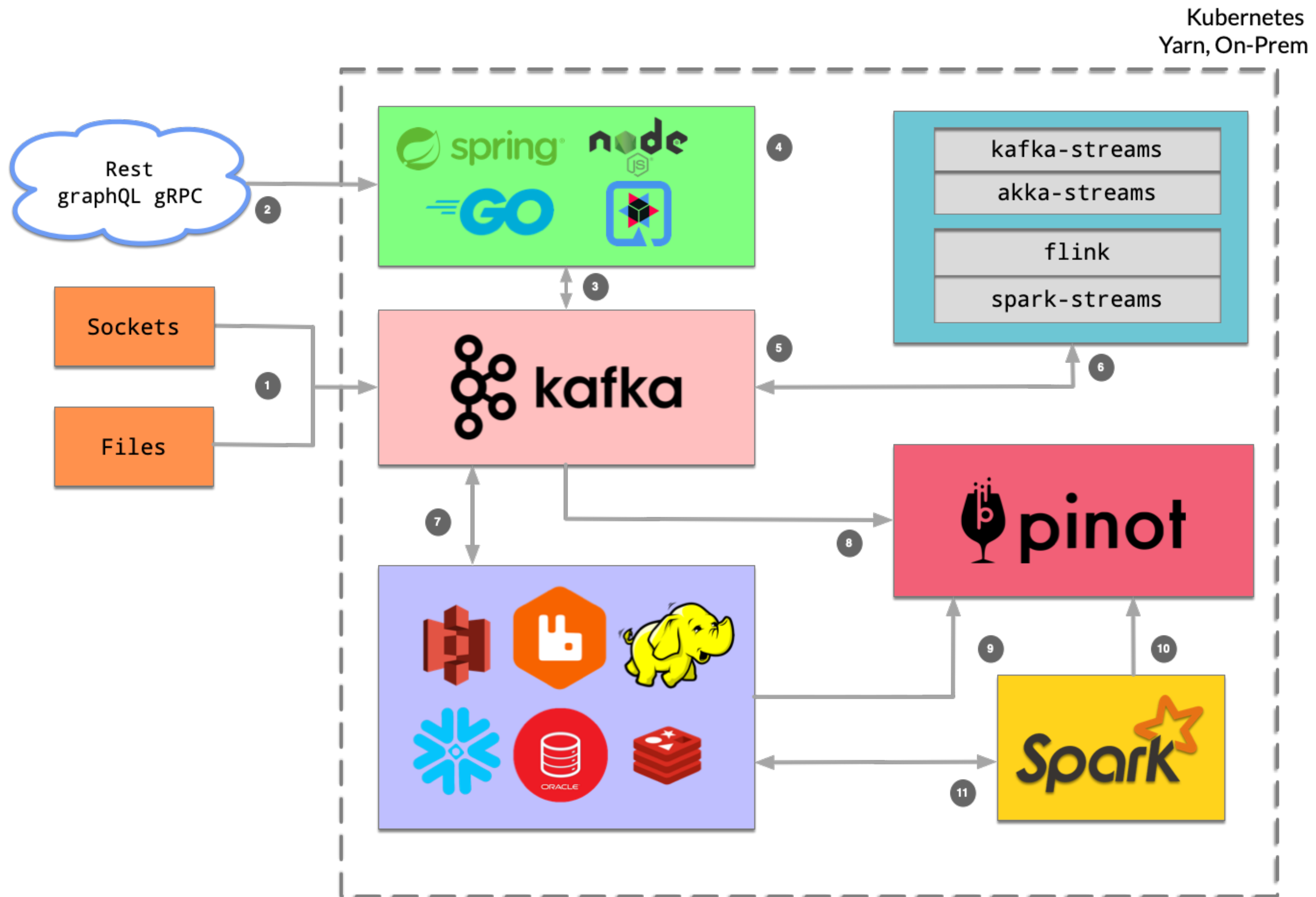
Leader



ISR →



Conclusion



Agenda

- ★ Understand Kafka
- ★ Understand Producer
- ★ Understand Consumer
- ★ Understand Rebalancing