Real-Time Processing with Kafka

Learning Goals

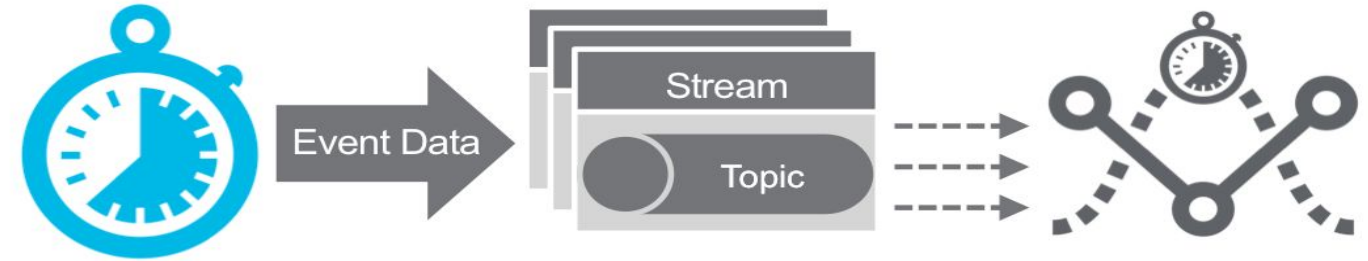
Learning Goals

- Summarize the motivation behind Kafka Streams
- Define core components of Kafka Streams
- Summarize the life of a message in Kafka Streams
- Build a simple producer and consumer application

Prerequisites

- Basic understanding of big data concepts
- Basic understanding of the platform
- Basic understanding of application development principles

Many Big Data sources are Event Oriented

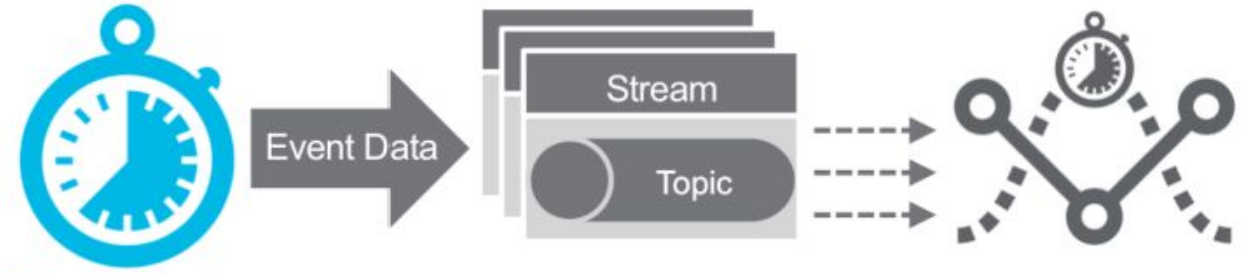


Trigger Events:

- Stock Prices
- User Activity
- Sensor Data

What is Kafka?

Many Big Data sources are Event Oriented



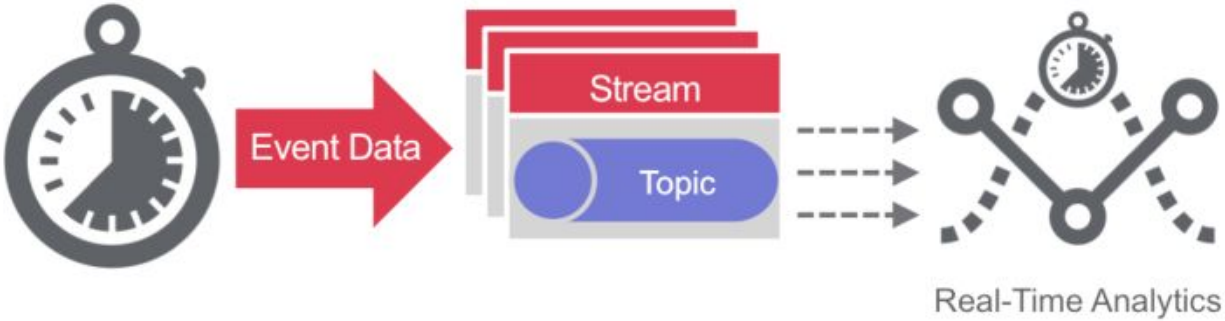
Trigger Events:

- Stock Prices
- User Activity
- Sensor Data

Why Kafka?

Today's Applications need to:

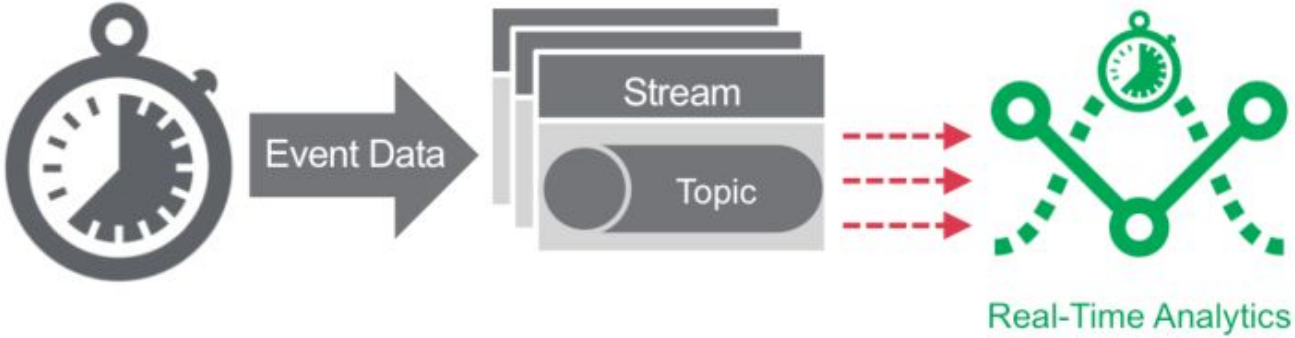
- process high-velocity data as soon as possible
- handle high-volume workloads



Why Kafka?

Today's applications need to:

- process high-velocity data as soon as possible
- handle high-volume workloads
- provide real-time results with extremely low latency



Diverse Data Sources

Social Media Feeds

Financial Transactions

Geotagging Data

Internet of Things

Web Browser Clickstreams

Application Metrics

Diverse Data Sources

Social Media Feeds

Financial Transactions

Geotagging Data

Internet of Things

Web Browser Clickstreams

Application Metrics

Diverse Data Sources

Social Media Feeds

Financial Transactions

Geotagging Data

Internet of Things

Web Browser Clickstreams

Application Metrics

Diverse Data Sources

Social Media Feeds

Financial Transactions

Geotagging Data

Internet of Things

Web Browser Clickstreams

Application Metrics

Diverse Data Sources

Social Media Feeds

Financial Transactions

Geotagging Data

Internet of Things

Web Browser Clickstreams

Application Metrics

Diverse Data Sources

Social Media Feeds

Financial Transactions

Geotagging Data

Internet of Things

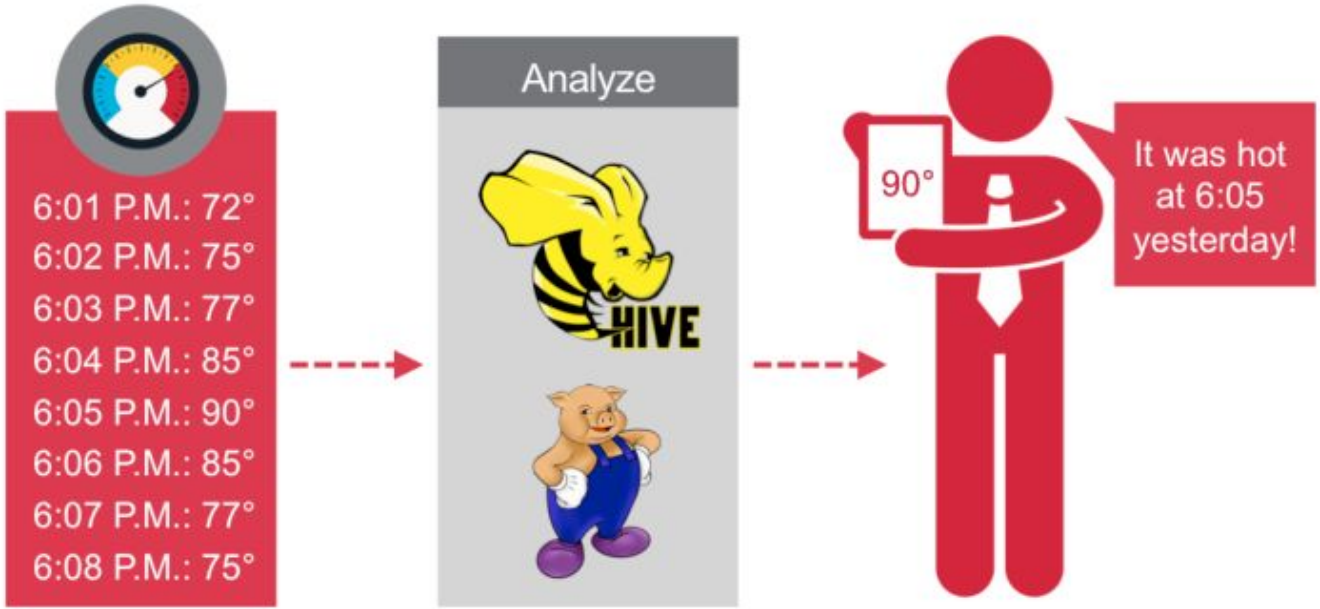
Web Browser Clickstreams

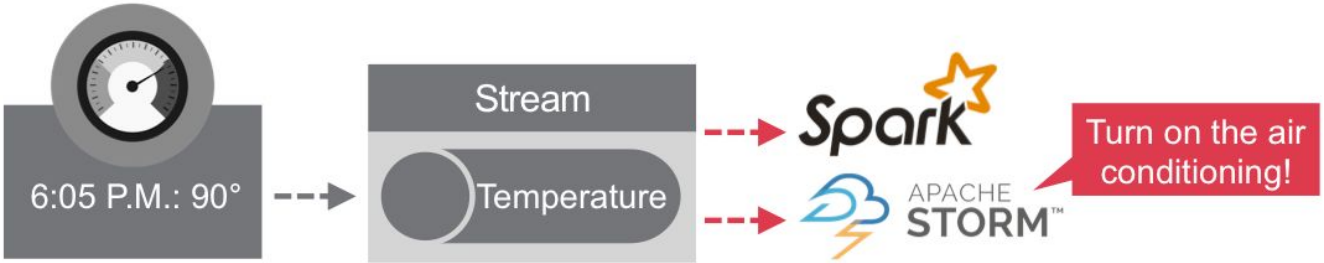
Application Metrics

Analyze Data

What if you need to analyze data as it arrives?

Batch Processing with HDFS




6:05 P.M.: 90°
Stream
Temperature
Spark
APACHE
STORM™
Turn on the air conditioning!

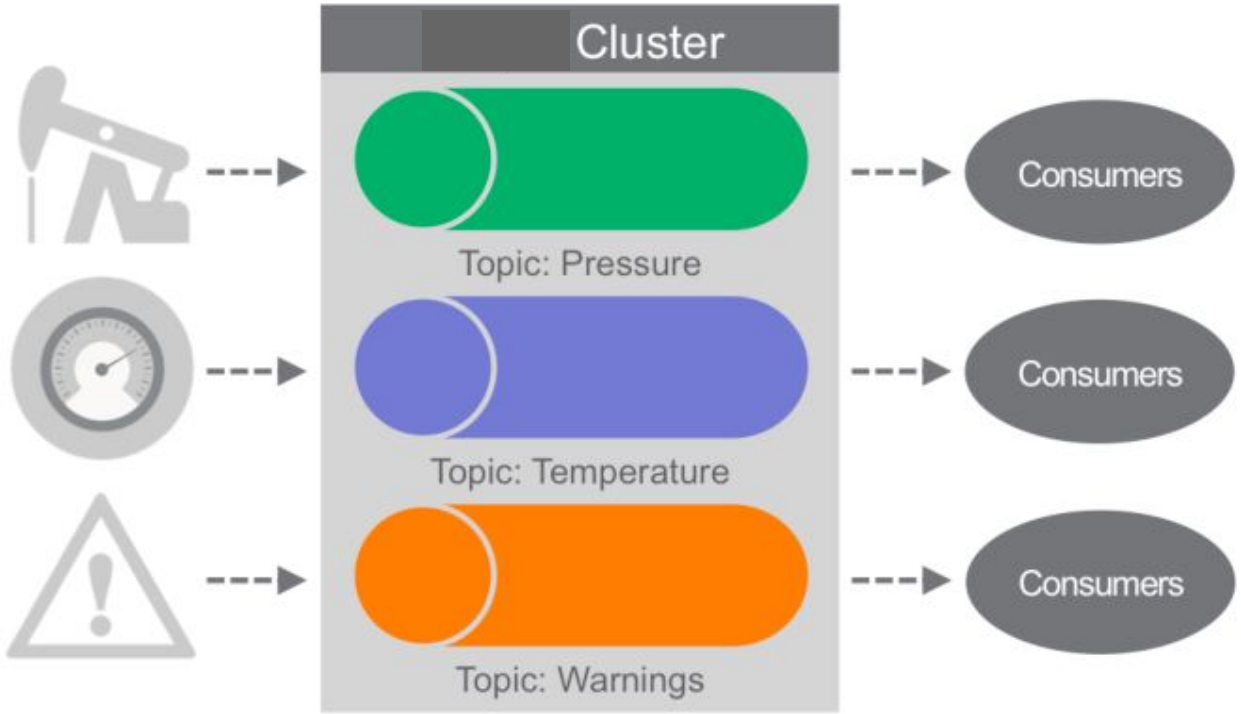
Organize Data

What if you need to organize data as it arrives?

Many Sources of Data

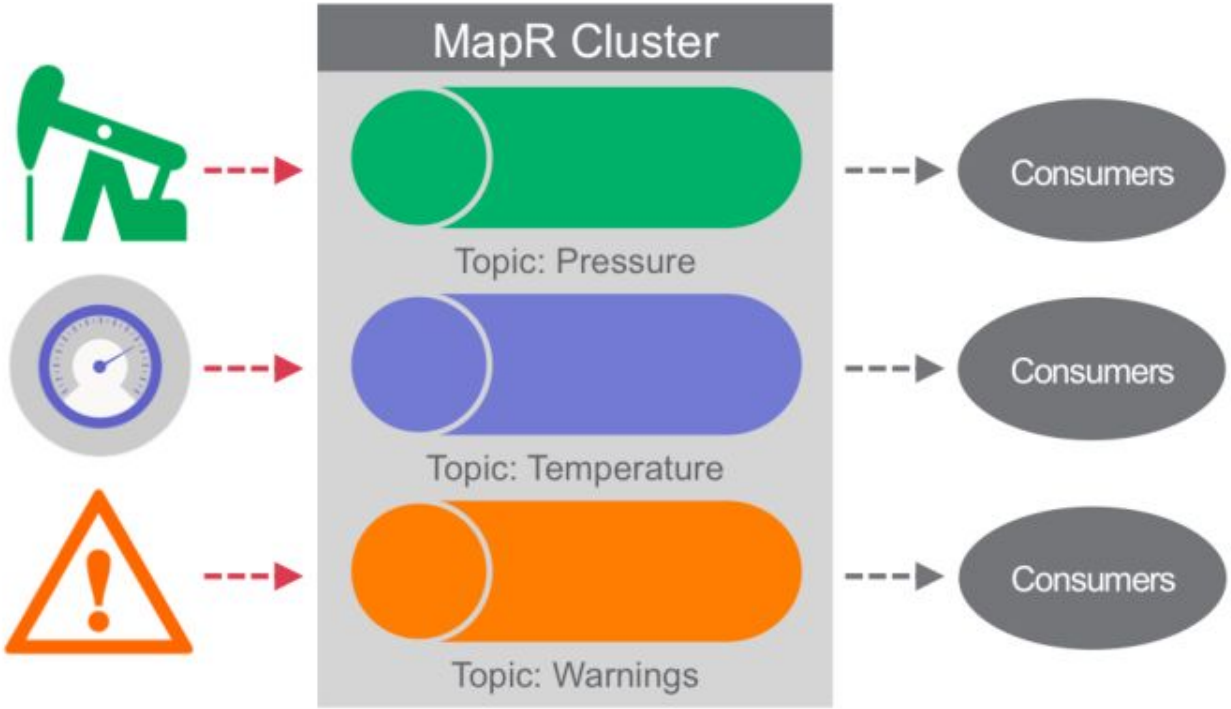
Organize Data into Topics with Kafka Streams

Topics organize events into categories



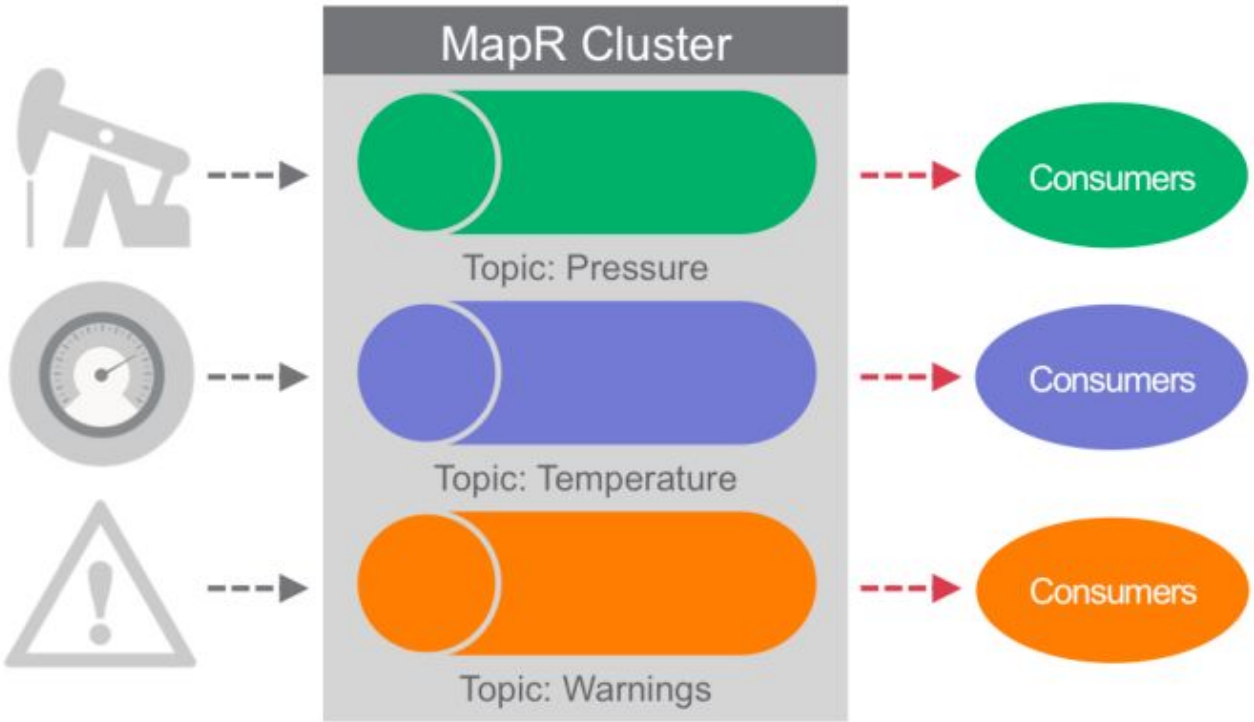
Organize Data into Topics with MapR Streams

Producers publish to topics



Organize Data into Topics with MapR Streams

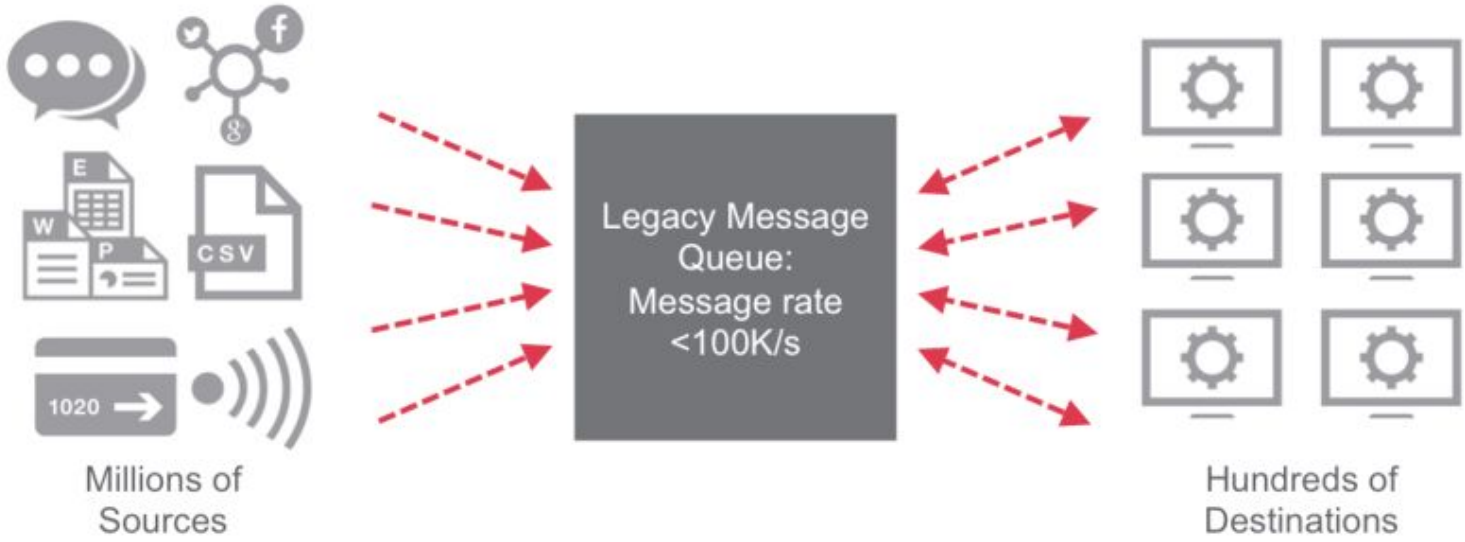
Consumers subscribe to topics



Process High Volume of Data

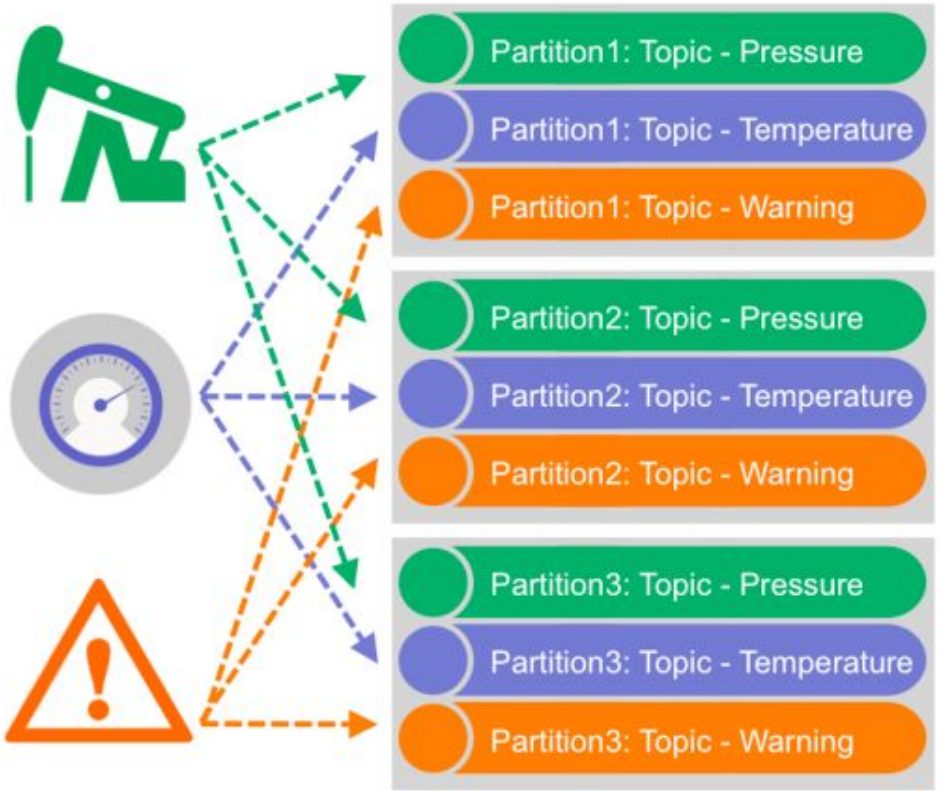
What if you need to process a high volume of data as it arrives?

Legacy Messaging

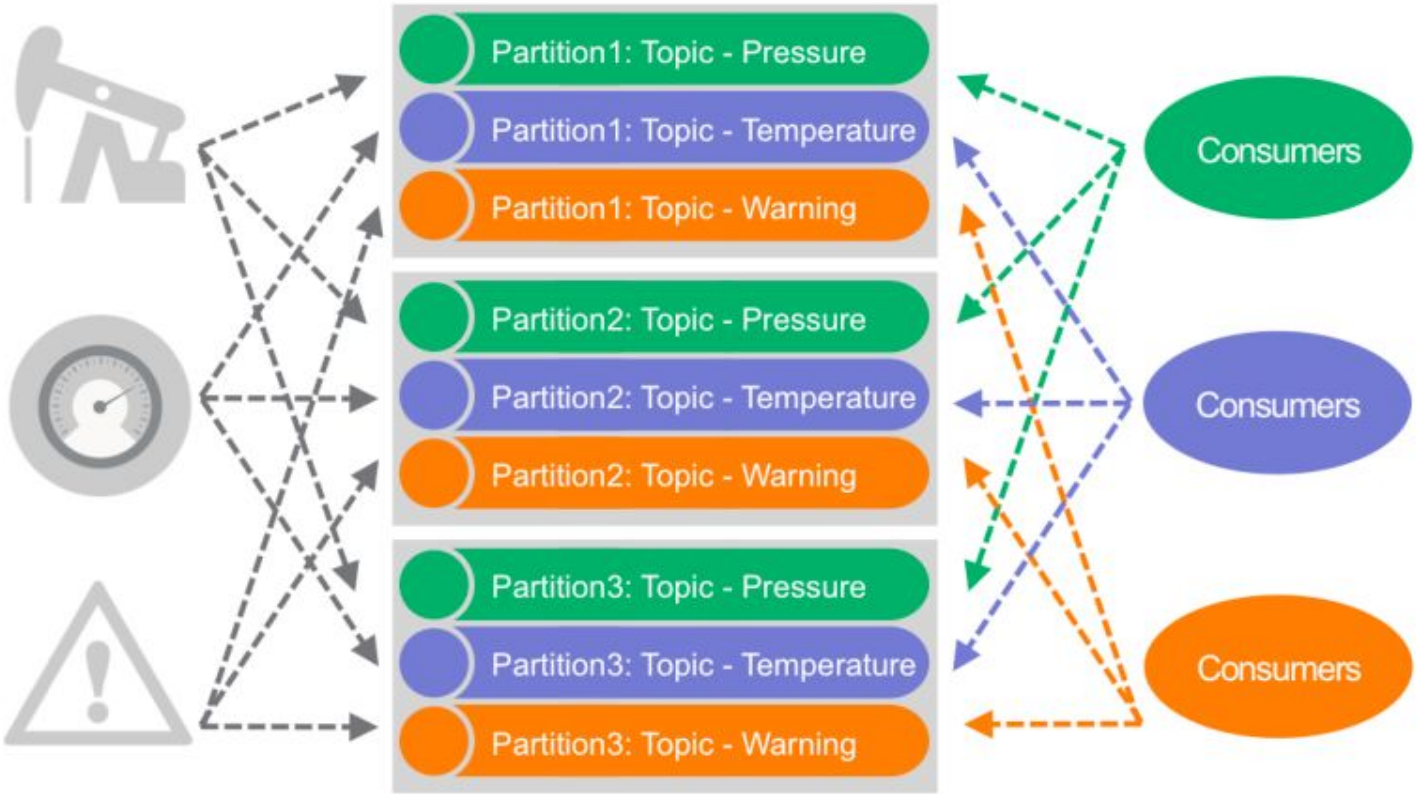



Partition1: Topic - Pressure
Partition1: Topic - Temperature
Partition1: Topic - Warning
Server 1
Partition2: Topic - Pressure
Partition2: Topic - Temperature
Partition2: Topic - Warning
Server 2
Partition3: Topic - Pressure
Partition3: Topic - Temperature
Partition3: Topic - Warning
Server 3

Scalable Messaging with MapR Streams



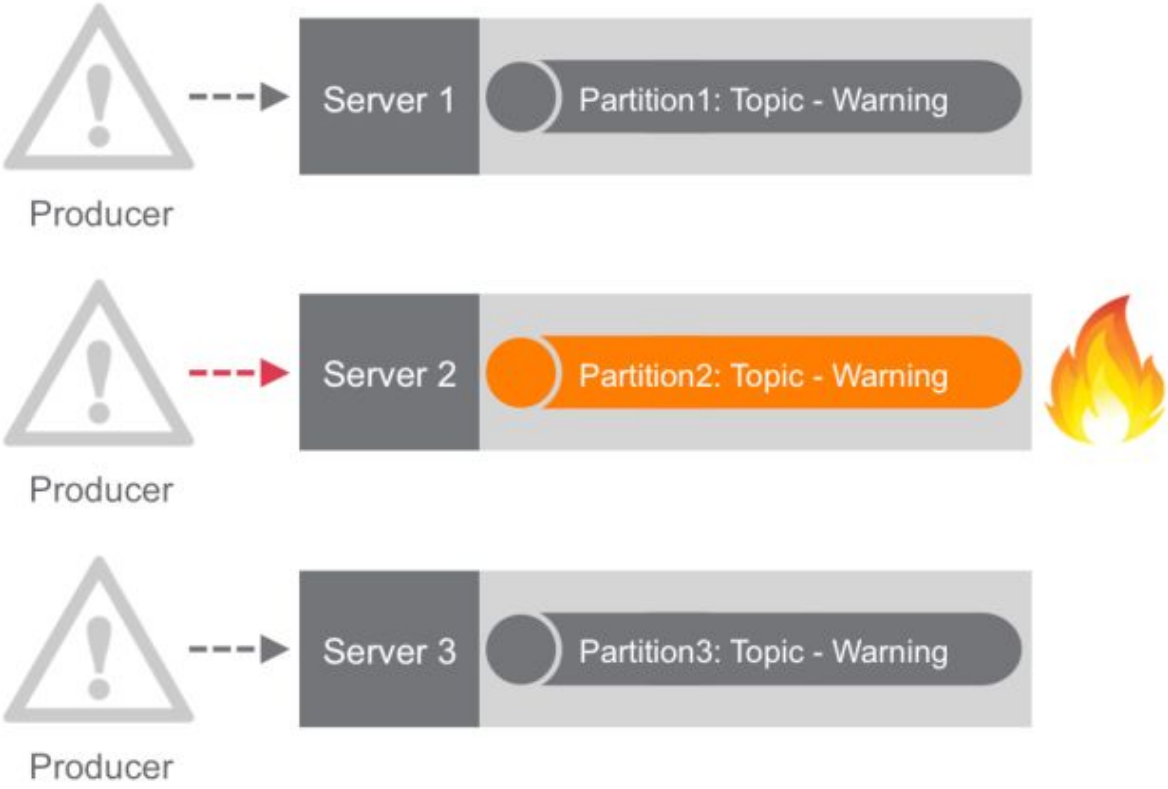
Scalable Messaging with MapR Streams



Message Recovery

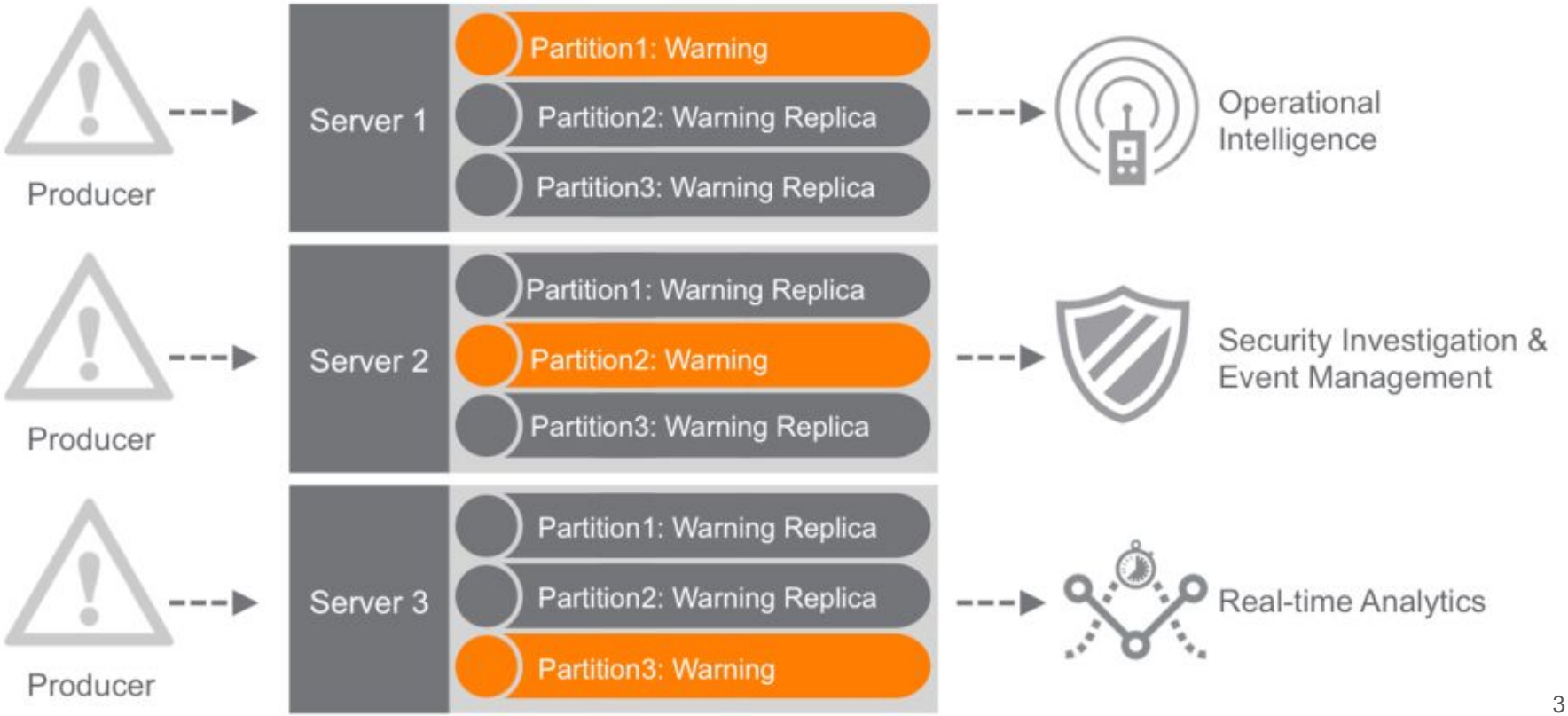
What if you need to recover messages in case of server failure?

Lack of Replication

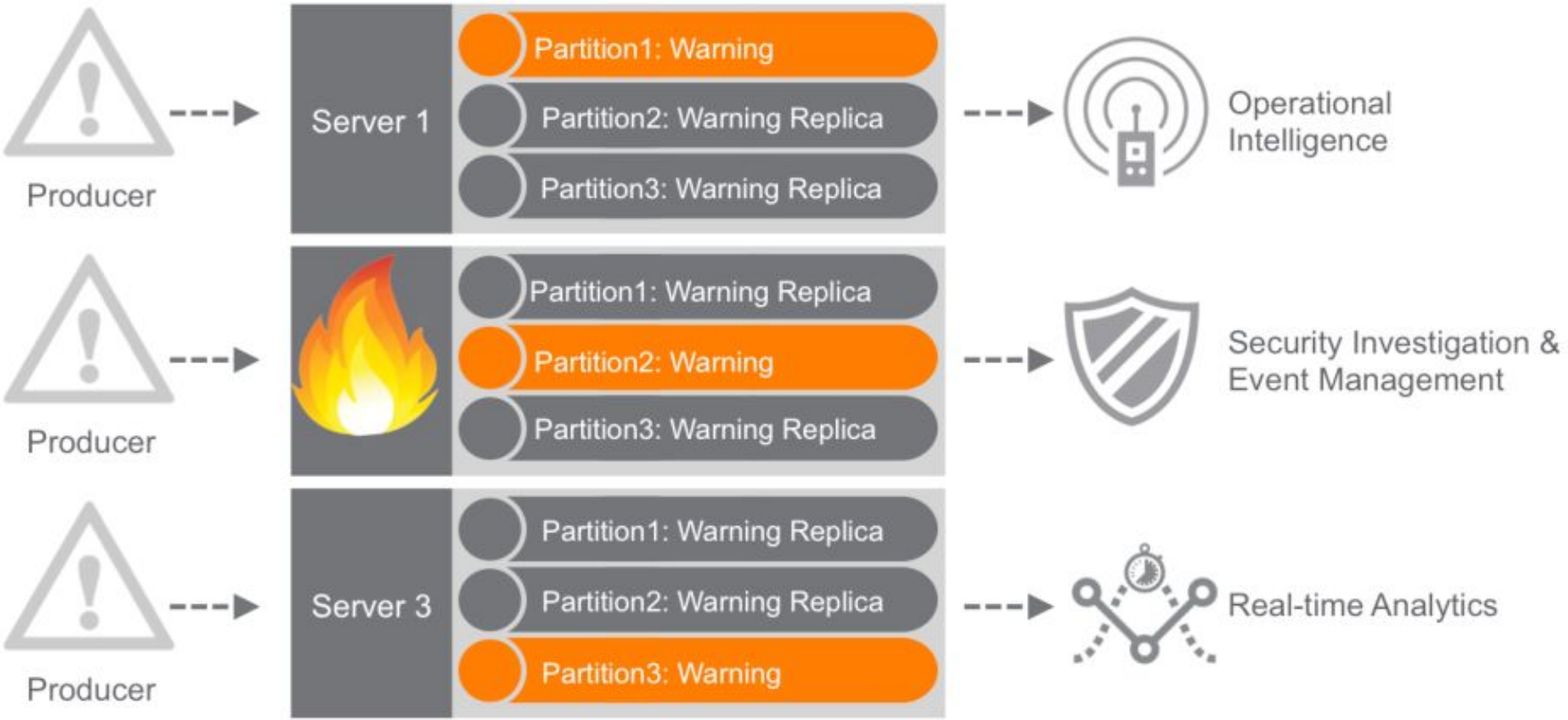


Partitions are Replicated for Fault Tolerance

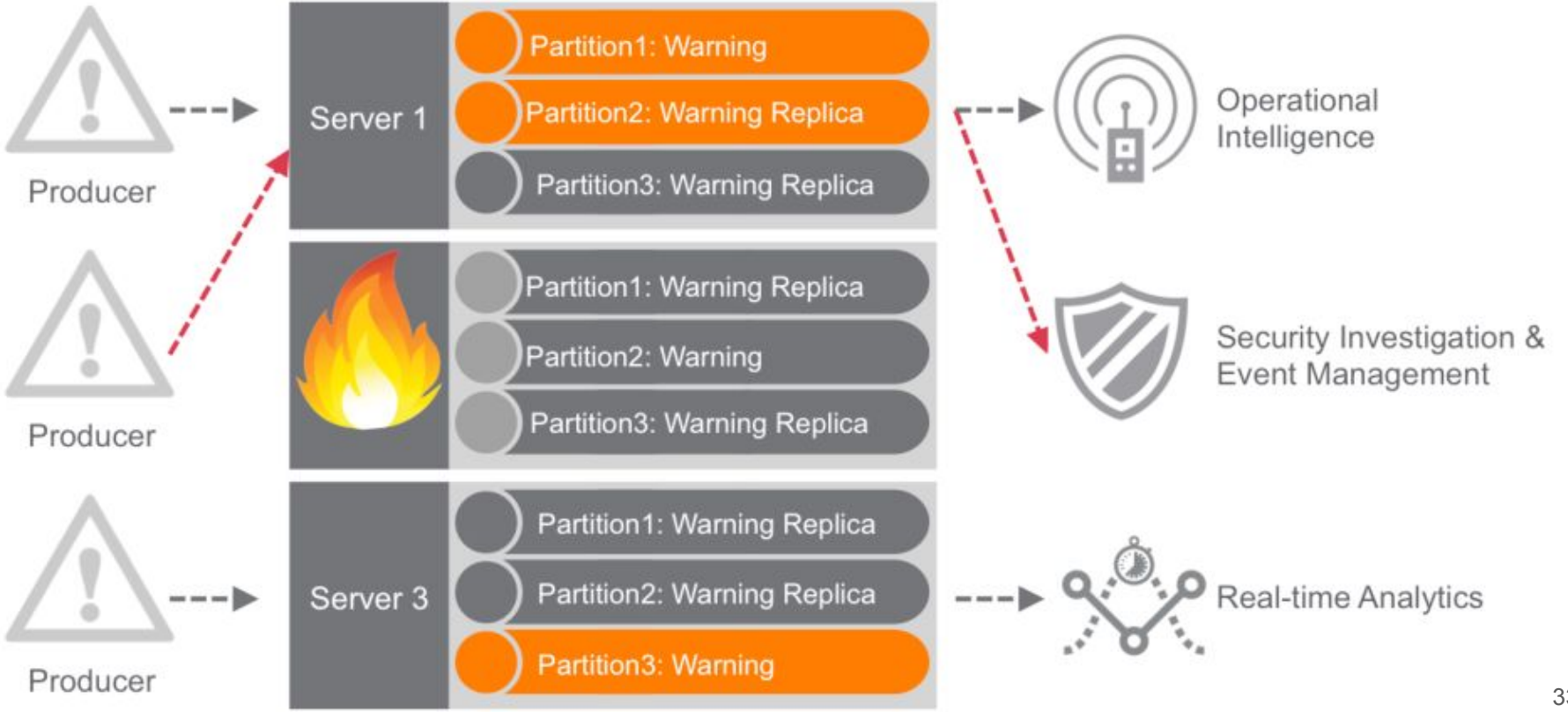
Partitions are Replicated for Fault Tolerance



Partitions are Replicated for Fault Tolerance



Partitions are Replicated for Fault tolerance



Real-time Access

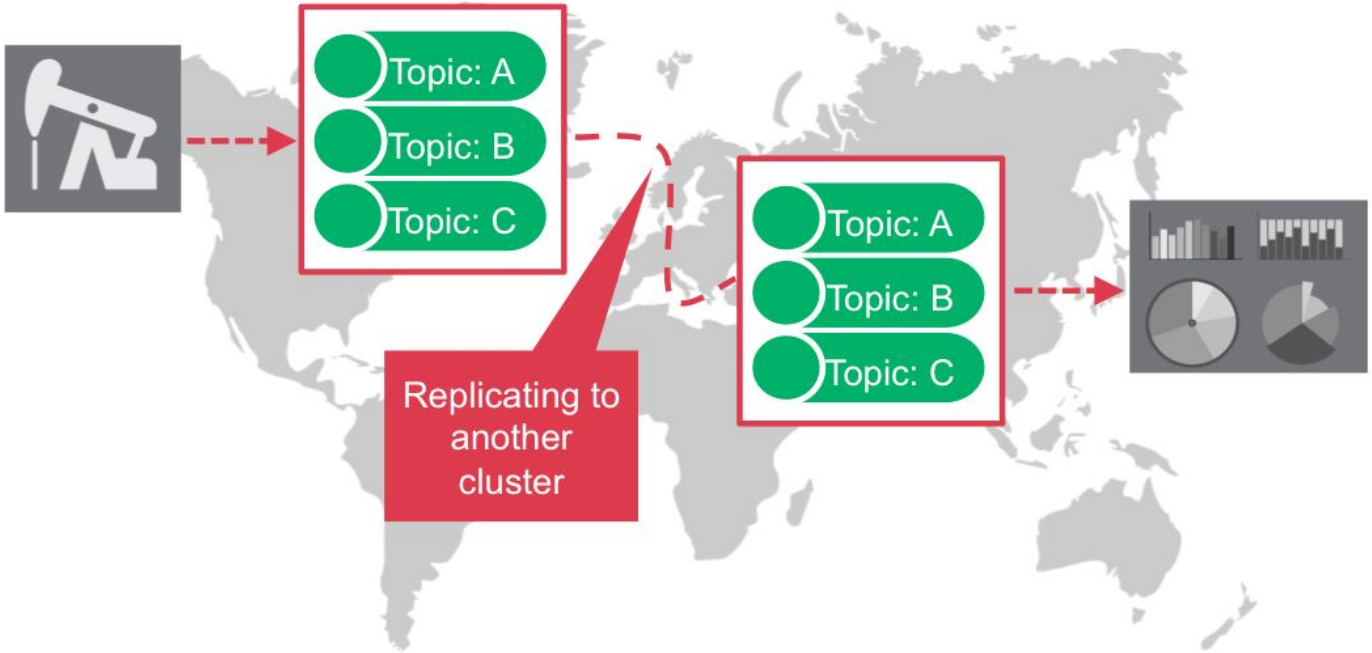
What if you need real-time access to live data distributed across multiple clusters and multiple data centers?

Lack of Global Replication

Streams and Replication

Streams:

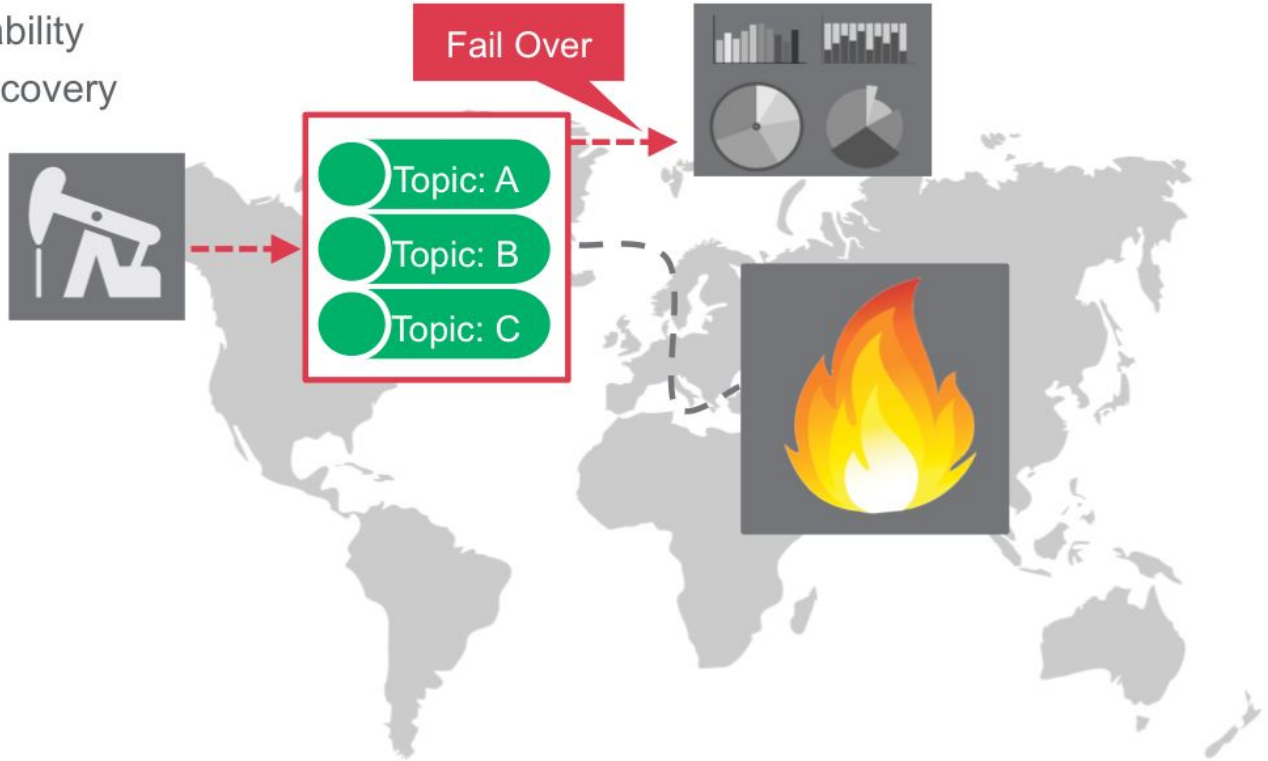
- are a collection of topics
- can be replicated worldwide



Streams and Replication

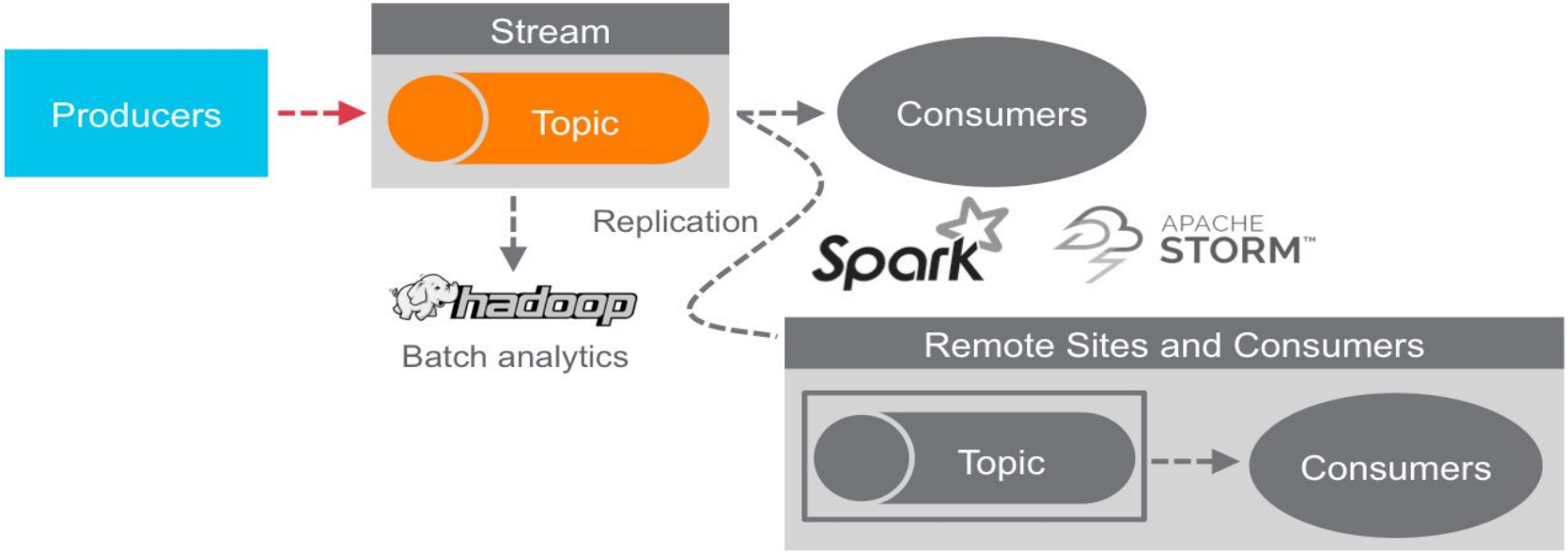
Streams:

- high availability
- disaster recovery

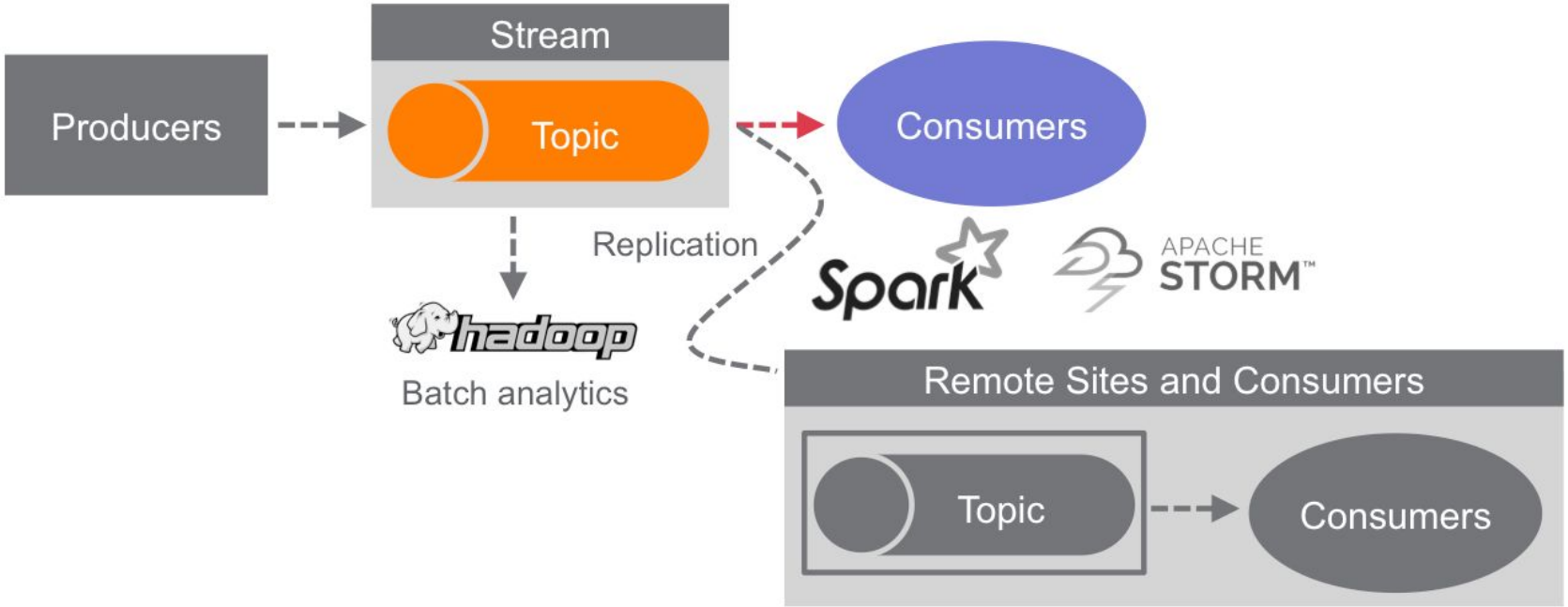


Kafka: A Global Pub-Sub System for Big Data

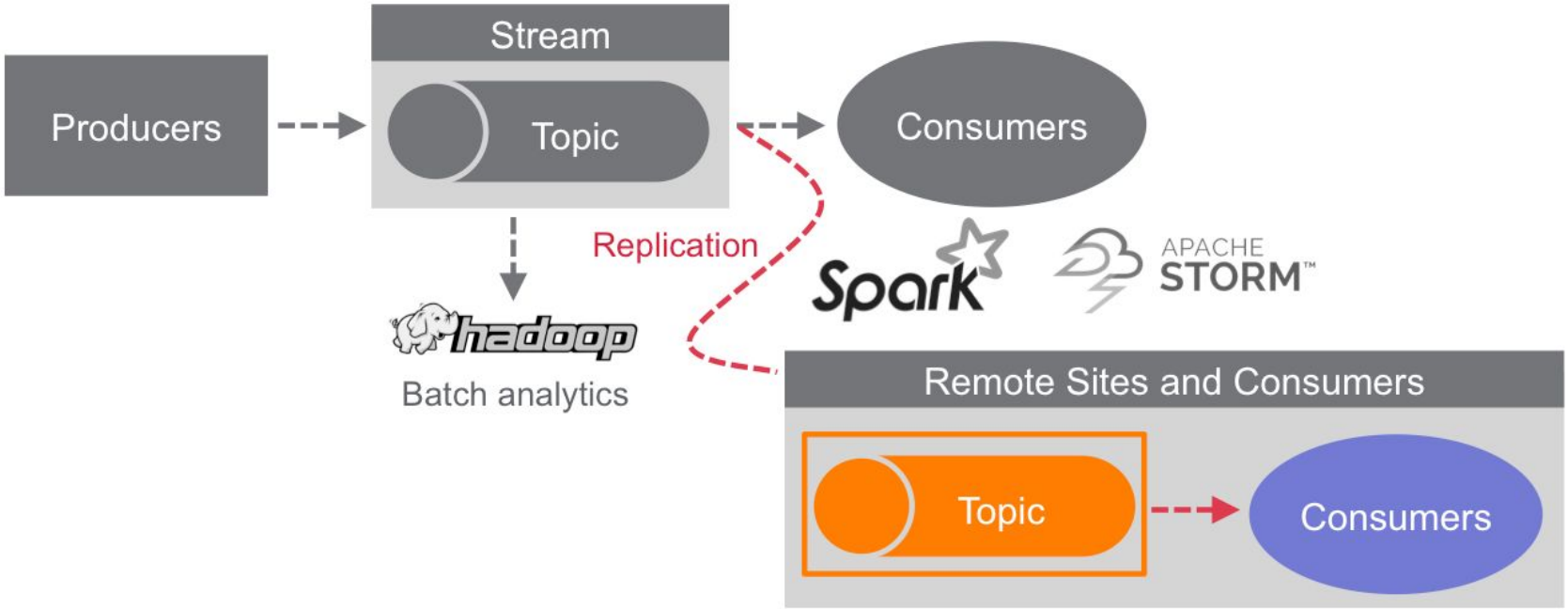
Producers publish billions of events/sec



Reliable delivery to all consumers

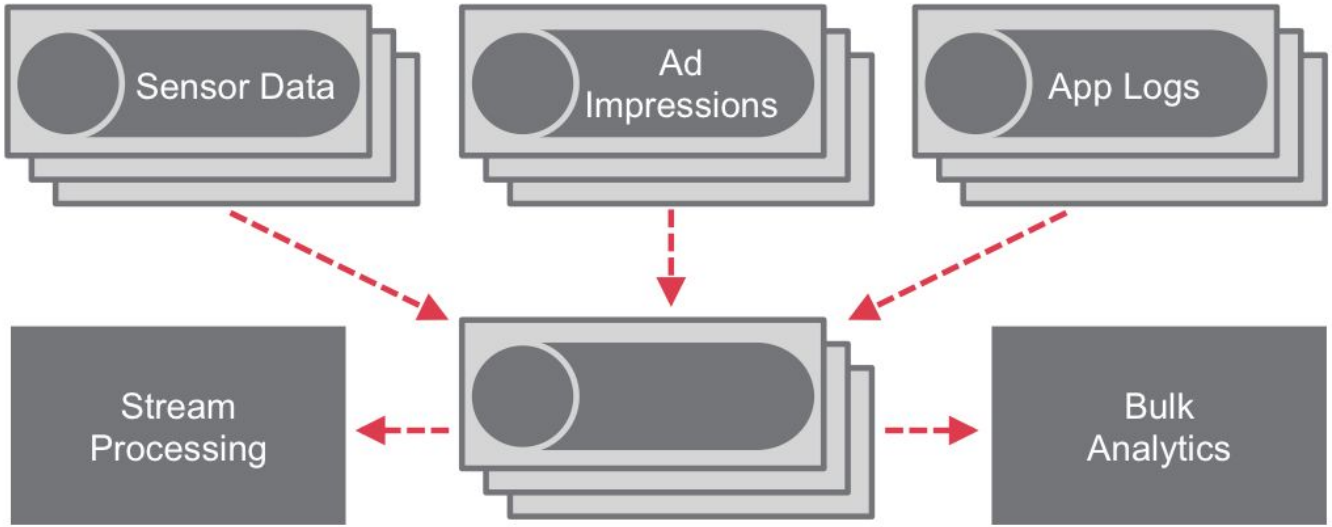


Tie together geo-dispersed clusters worldwide



- Combines:
  - Massively scalable messaging
  - Consistent, reliable delivery
  - Real-time analytics
  - Security and fault-tolerance
- Standard real-time API (Kafka).
- OJAI API – direct data access from analytics frameworks

- Scalable to handle Internet of Things
- Global producers and consumers
- Secure and fault-tolerant
- Bulk and real-time analytics



Health Care

Advertising

Credit Card Security

Internet of Things

Use Case: Streaming System of Record for Healthcare

Objective:

- Build a flexible, secure healthcare database

Challenges:

- Many different data models
- Security and privacy issues
- HIPAA compliance

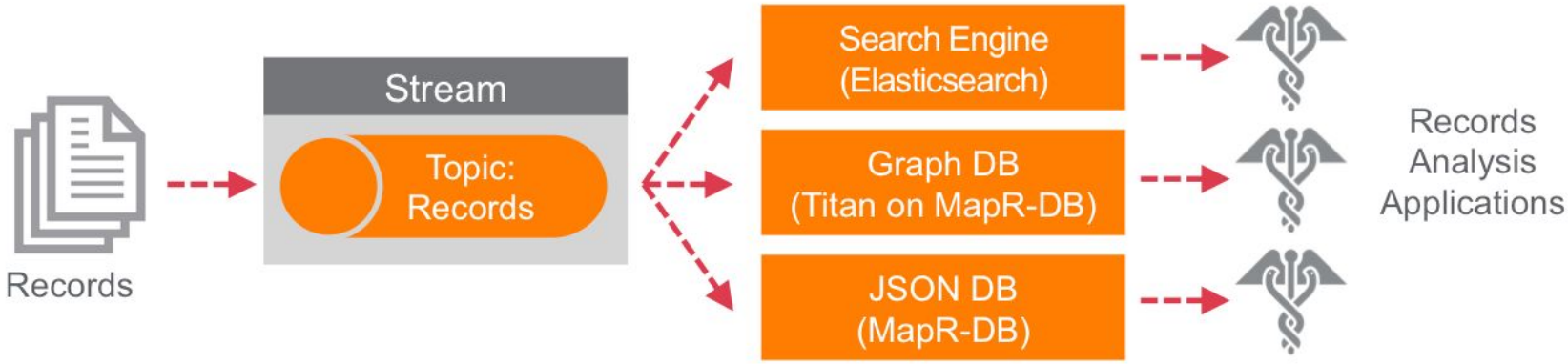
Use Case: Streaming System of Record for Healthcare

Solution:

- Streaming system of record
  - secure
  - immutable
  - rewindable

Auditable

- Materialized views continuously computed
- Selective cross data center replication



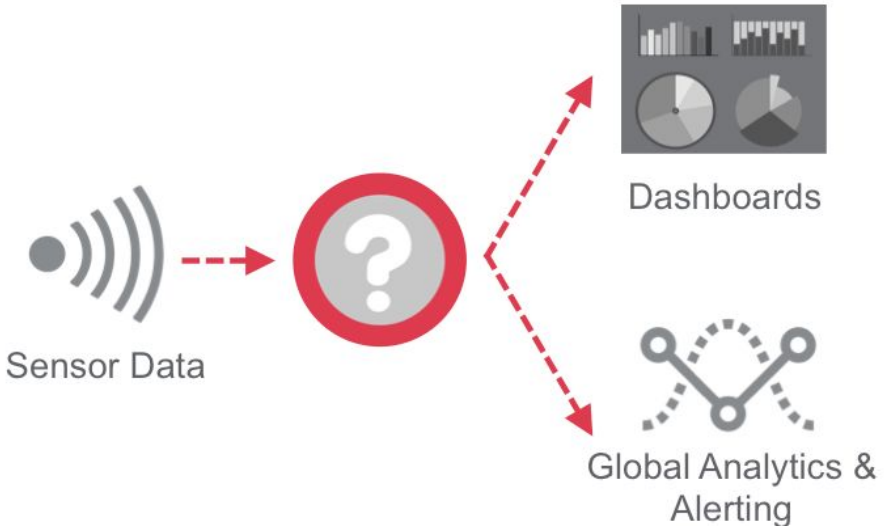
Use Case: Monitoring the Internet of Things in Real-Time

Objective:

- Monitor oil rig sensor data and create real-time alerts

Challenges:

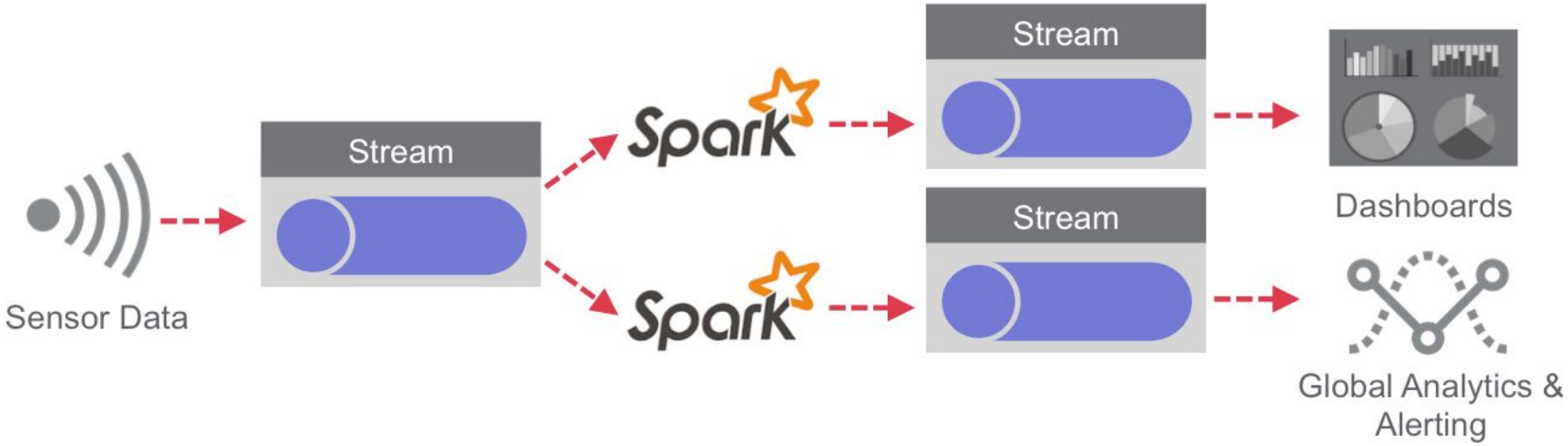
- Disperse, global data sources
- Need alerts in real-time
- Need to audit data



Use Case: Monitoring the Internet of Things in Real-Time

Solution:

- Synchronize global servers
- Streaming analytics
- Replicated streams



Use Case: Global Analytics in Targeted Advertising

Objective:

- Optimize global ad market with real-time analytics

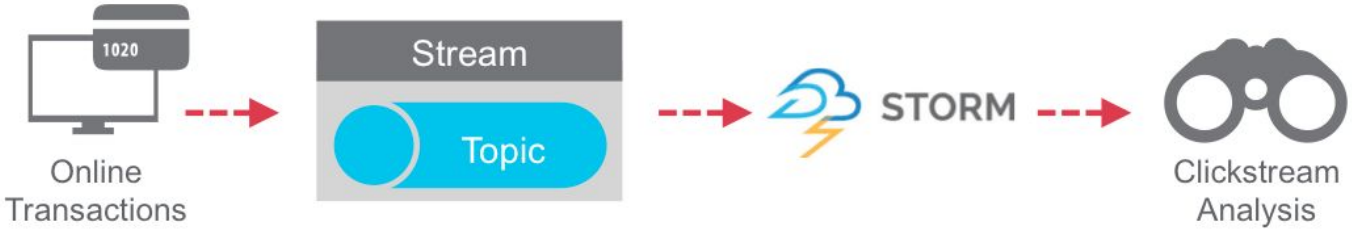
Challenges:

- Data consistency
- Global insights
- Complex pipeline

Use Case: Global Analytics in Targeted Advertising

Solution:

- Real-time analytics
- Global message streams
- Streaming ETL



Use Case: Intelligent Credit Card Processing

Objective:

- Build reliable, real-time service bus for financial transactions

Challenges:

- Accurate data
- Sensitive data
- Ad-hoc querying

Use Case: Intelligent Credit Card Processing

Solution:

- Reliable, replicated streams
- Security settings

Goals

Core Components: Message

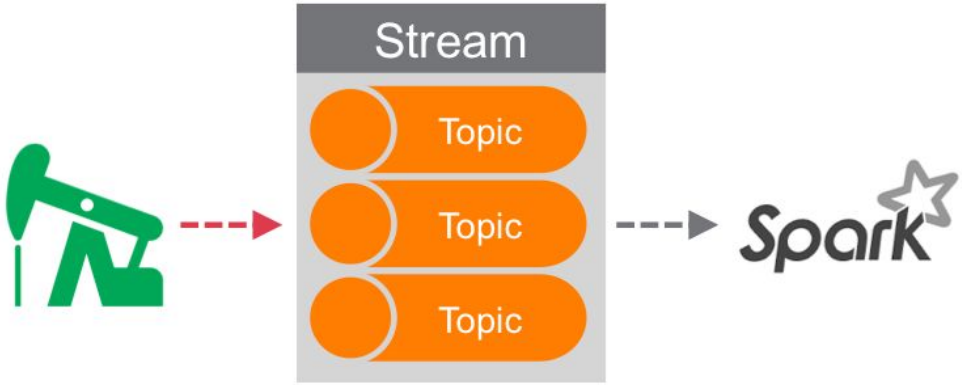
- Messages are key/value pairs
- Keys are optional
- Values can be any type of data

key	value
	{"event_ID":122, "hcid":63, "timestamp": "2015-01-01 00:00:02", "production":47, "sensor_readings": {"pressure":107, "temperature":30, "oil_percentage":18}, "injection_vol": 15367}

Core Components: Producers and Consumers

Producers

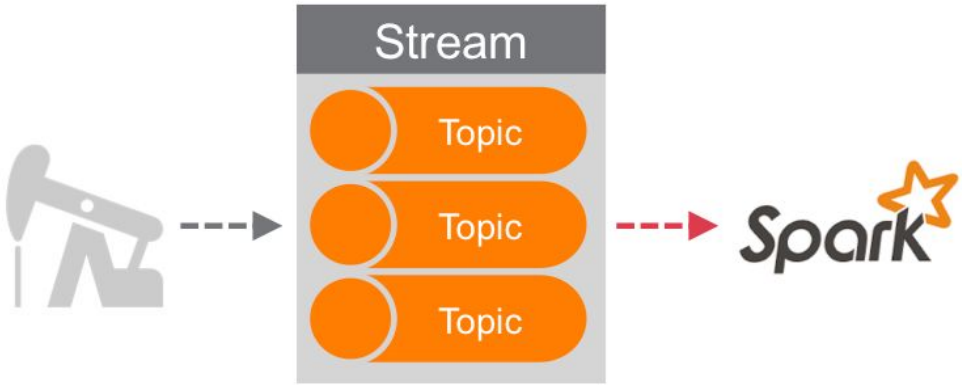
- publish to topics
- example: networked sensors on an oil rig

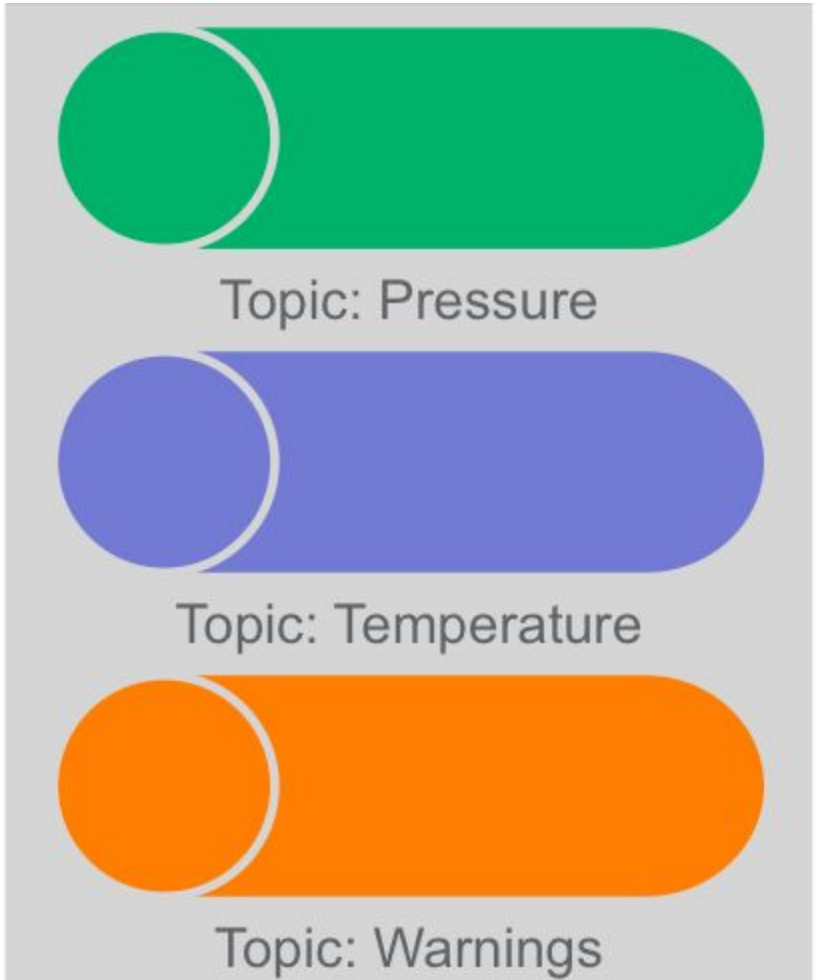


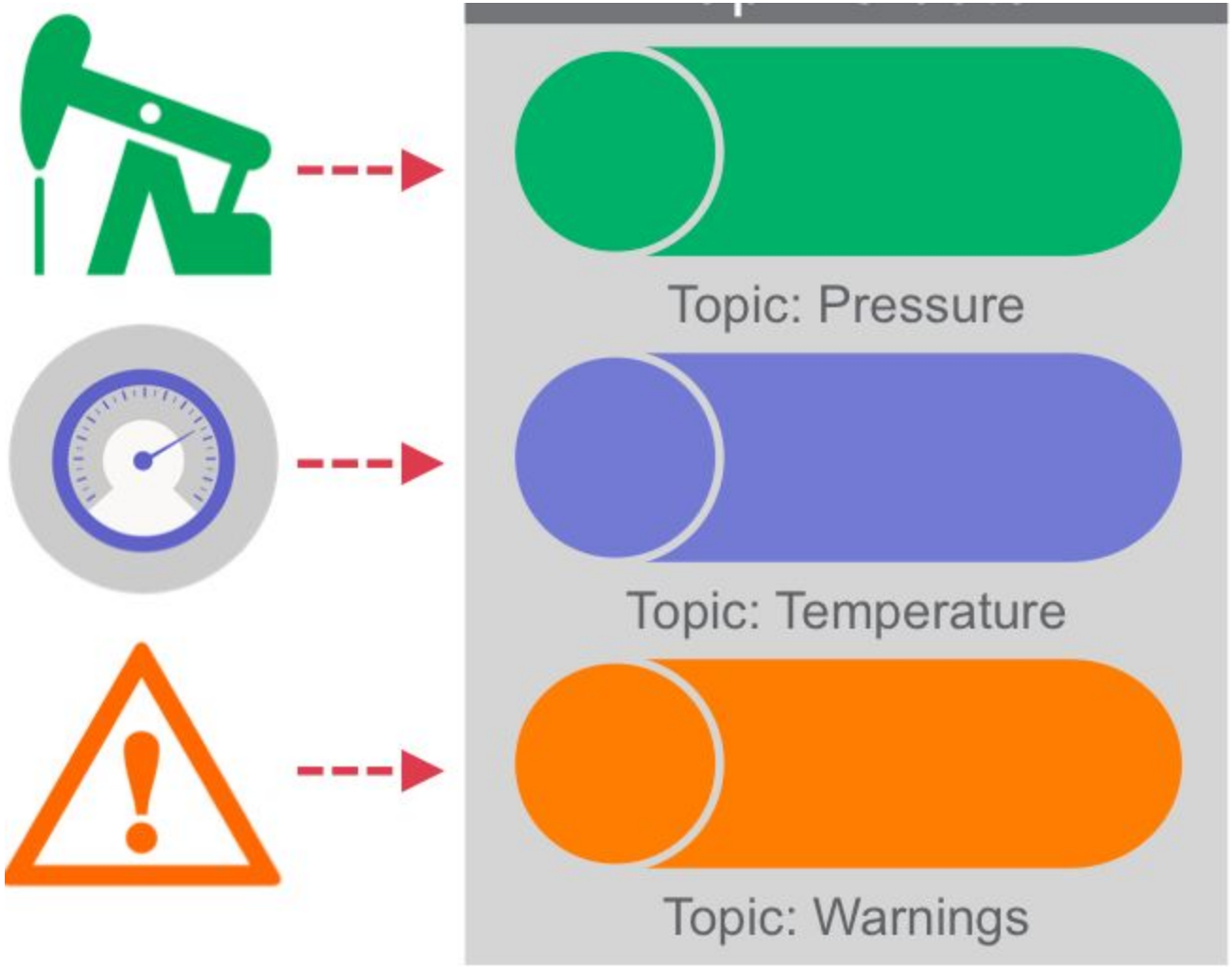
Core Components: Producers and Consumers

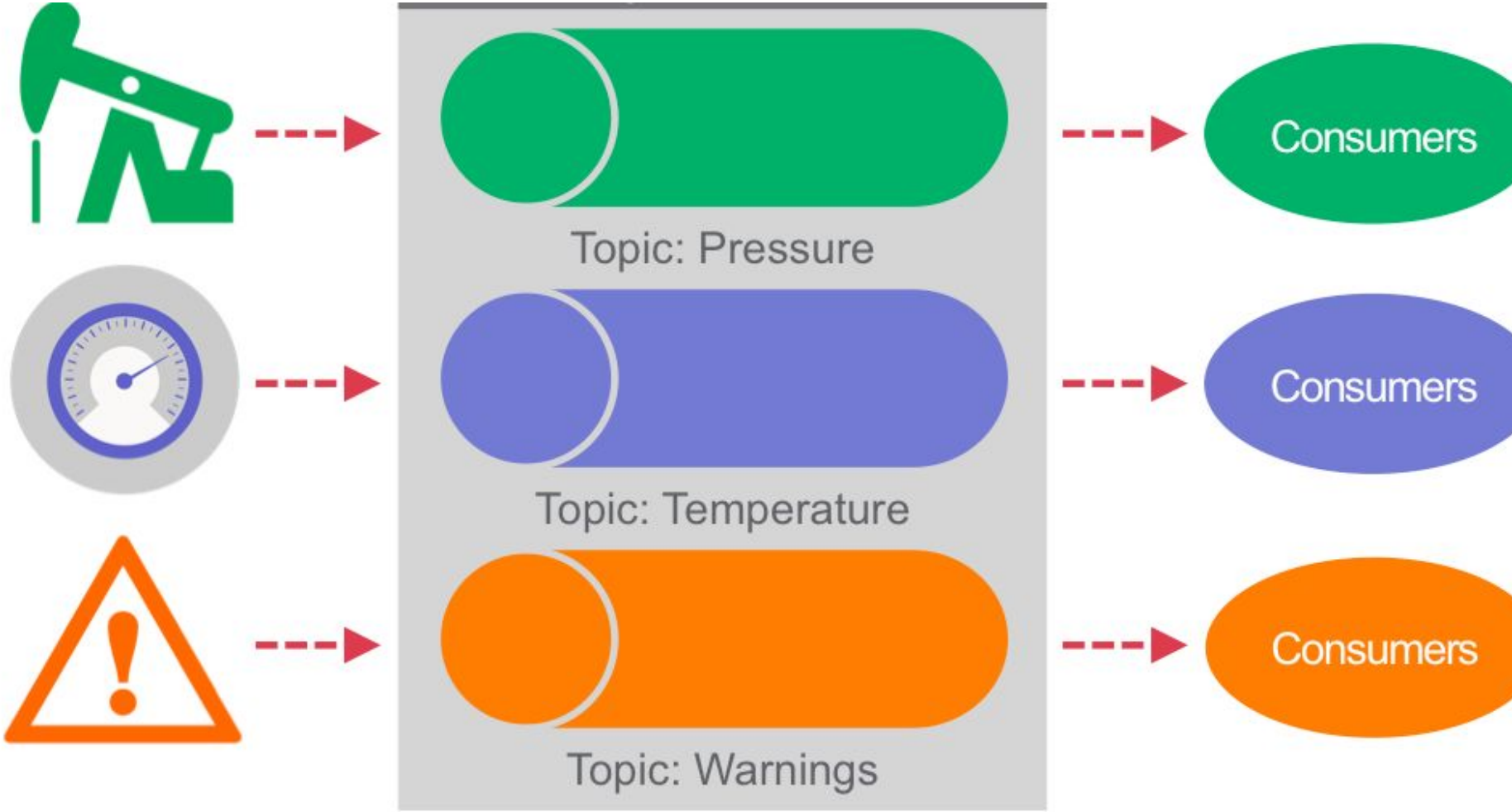
Consumers

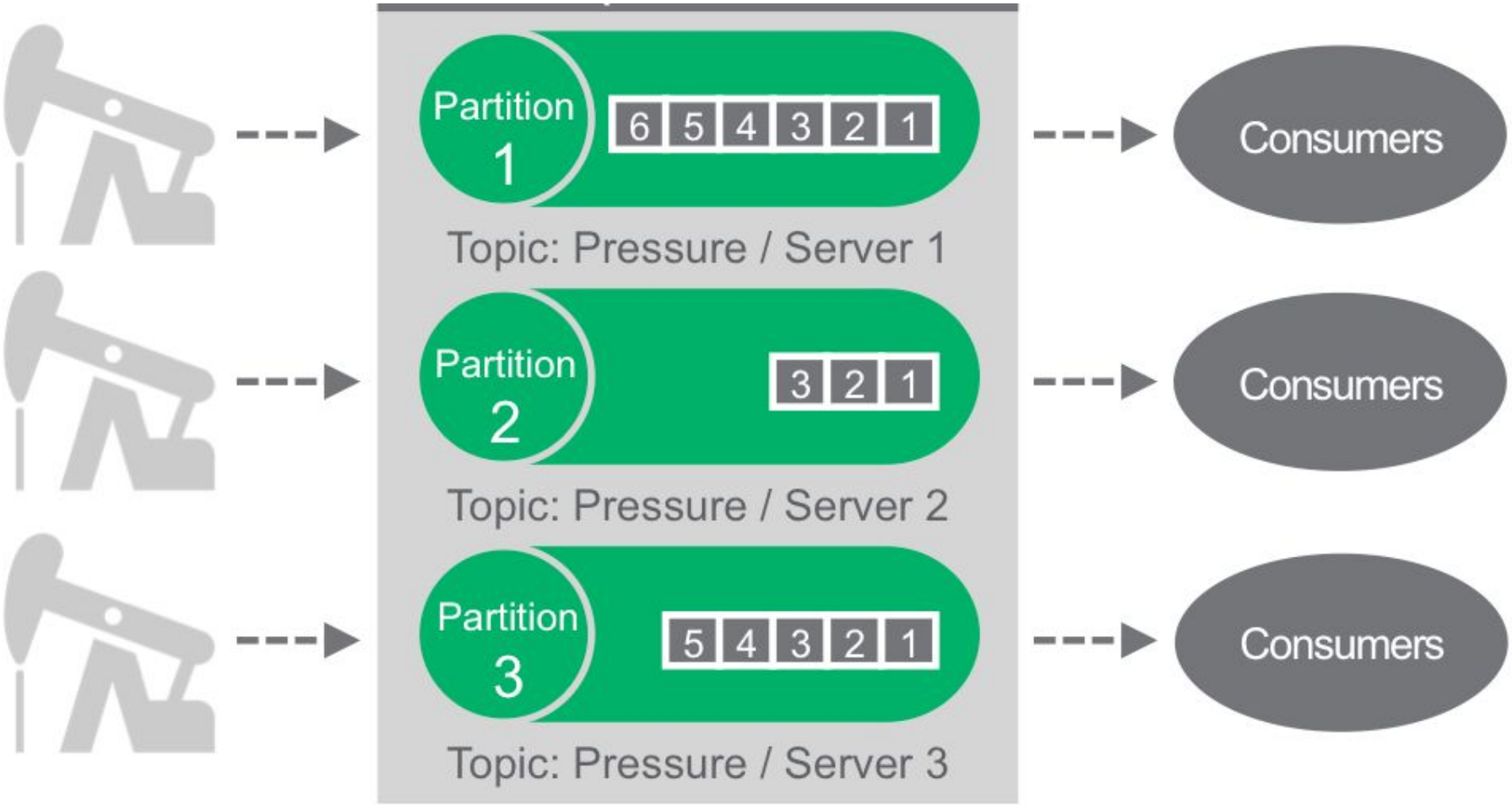
- subscribe to topics
- example: applications using Apache Spark

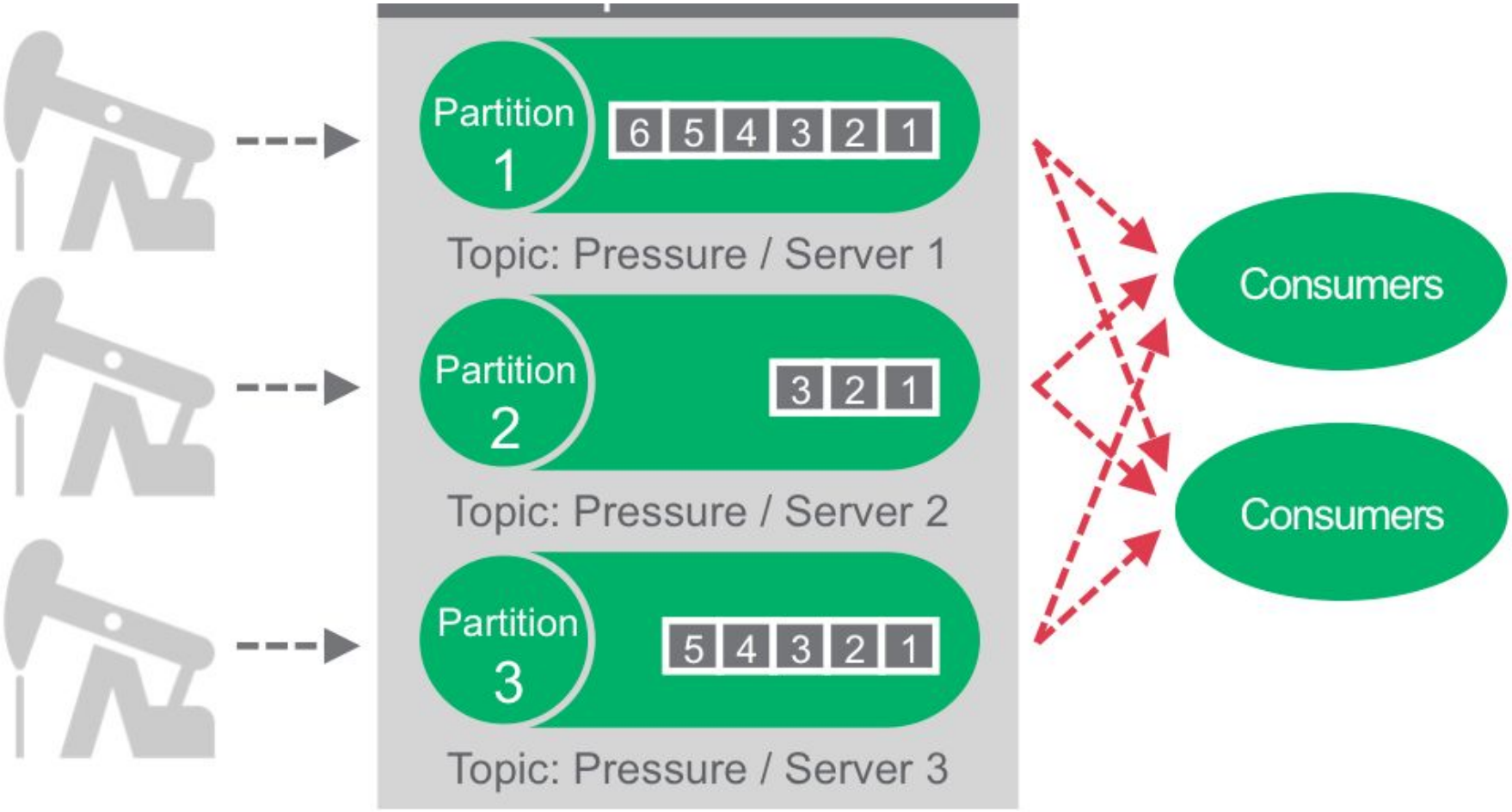


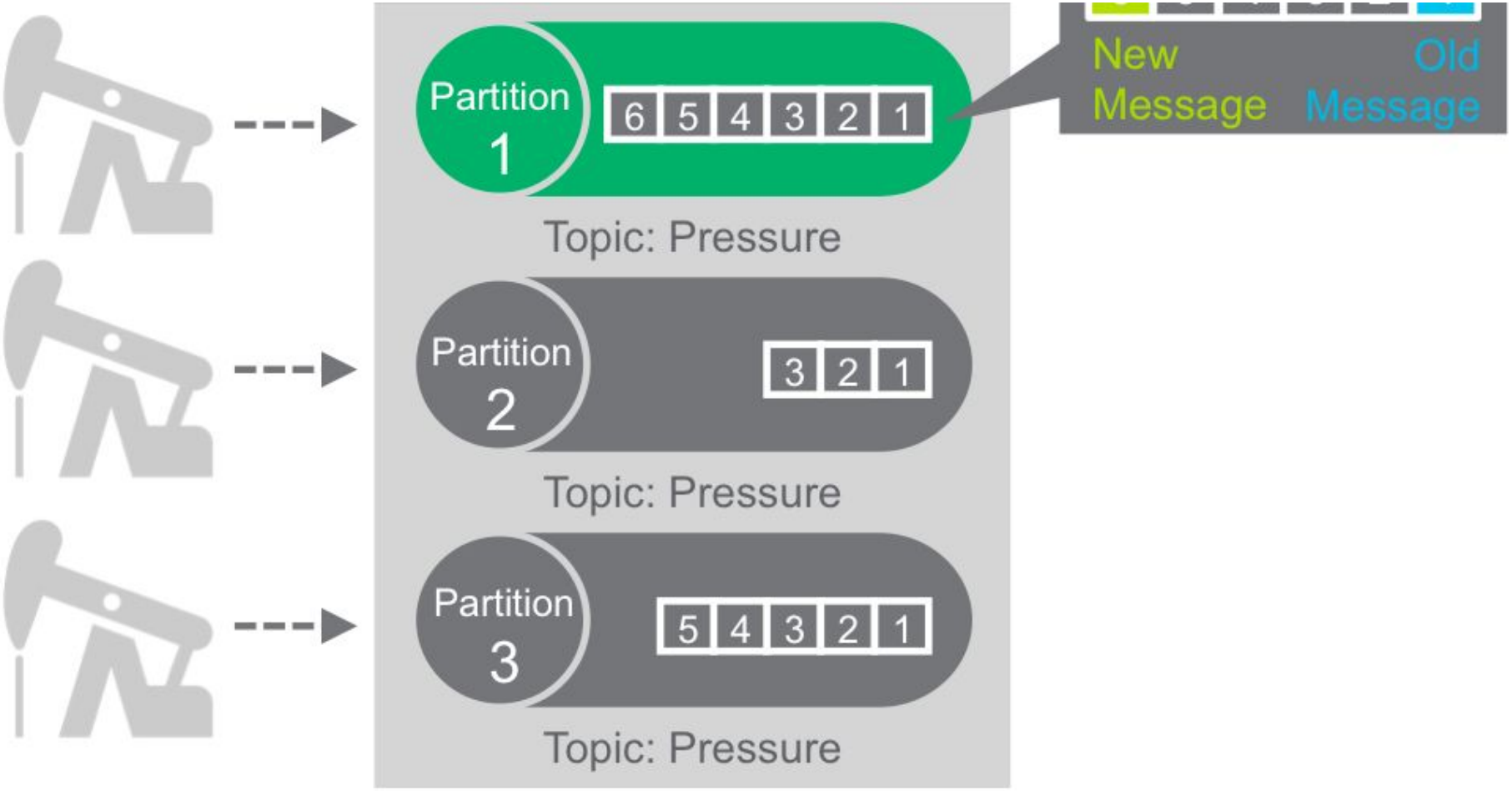

Topic: Pressure
Topic: Temperature
Topic: Warnings


Topic: Pressure
Topic: Temperature
Topic: Warnings


Topic: Pressure
Topic: Temperature
Topic: Warnings
Consumers
Consumers
Consumers

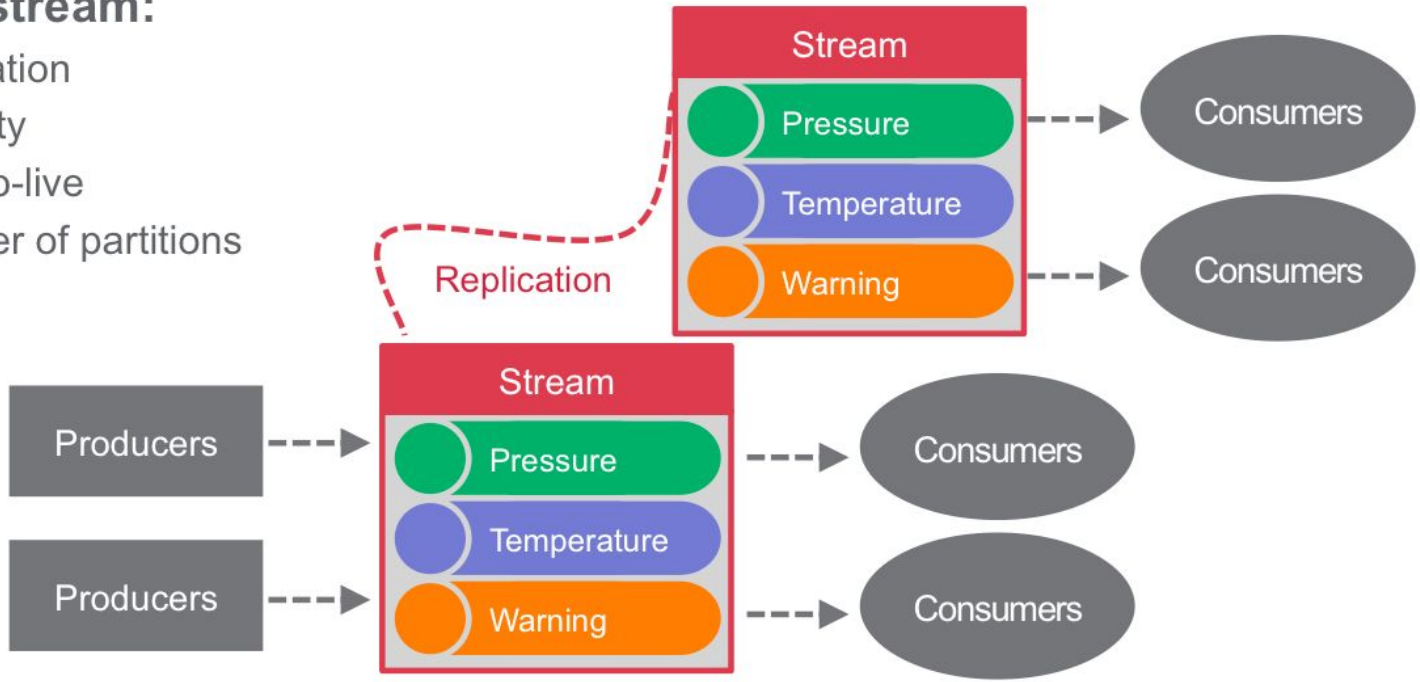

Partition 1
6 5 4 3 2 1
Topic: Pressure / Server 1
Partition 2
3 2 1
Topic: Pressure / Server 2
Partition 3
5 4 3 2 1
Topic: Pressure / Server 3
Consumers
Consumers
Consumers


Partition 1
6 5 4 3 2 1
Topic: Pressure / Server 1
Partition 2
3 2 1
Topic: Pressure / Server 2
Partition 3
5 4 3 2 1
Topic: Pressure / Server 3
Consumers
Consumers


Partition 1
6 5 4 3 2 1
Topic: Pressure
Partition 2
3 2 1
Topic: Pressure
Partition 3
5 4 3 2 1
Topic: Pressure
New Message
Old Message

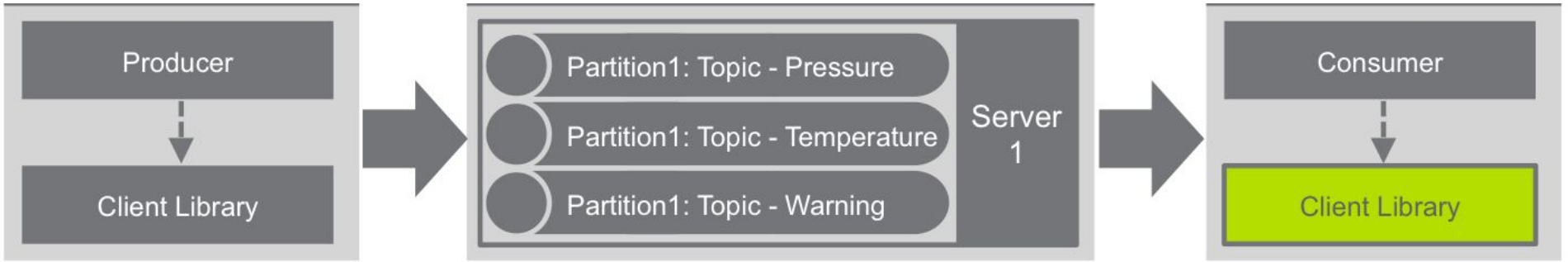
Core Components: Streams

- **Stream:**
  - collection of topics managed together
- **Manage stream:**
  - replication
  - security
  - time-to-live
  - number of partitions

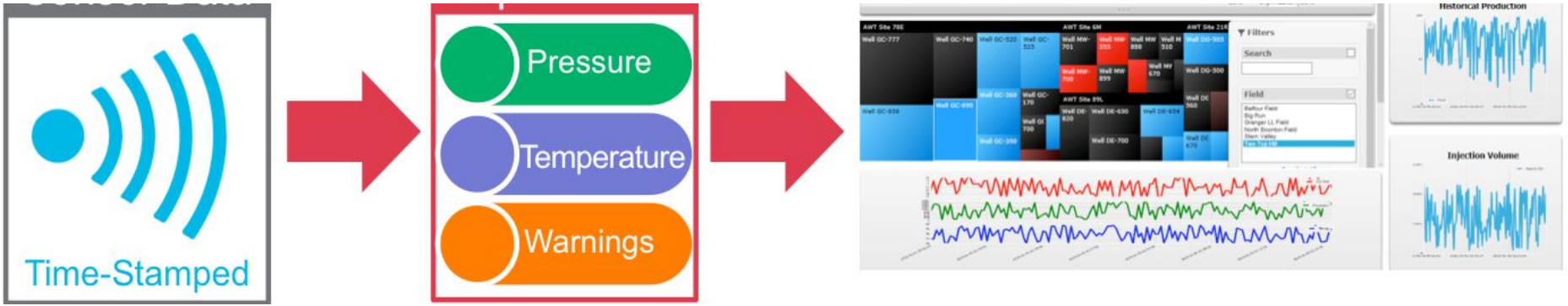


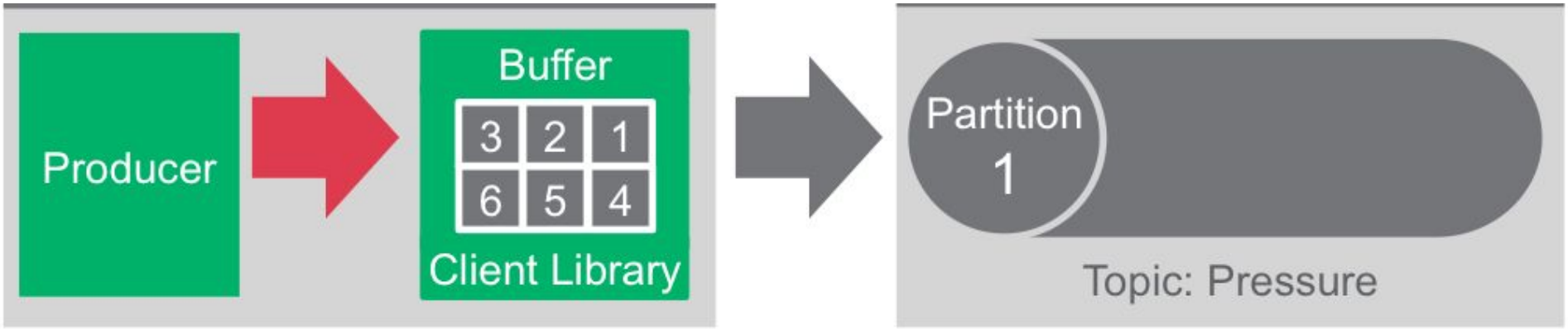

Producer
Client Library
Partition1: Topic - Pressure
Partition1: Topic - Temperature
Partition1: Topic - Warning
Server 1
Consumer
Client Library

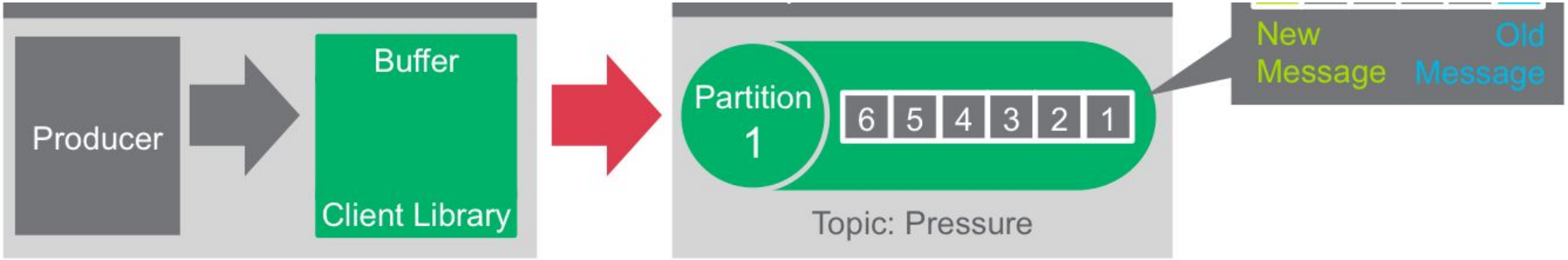

Producer
Client Library
Partition1: Topic - Pressure
Partition1: Topic - Temperature
Partition1: Topic - Warning
Server 1
Consumer
Client Library

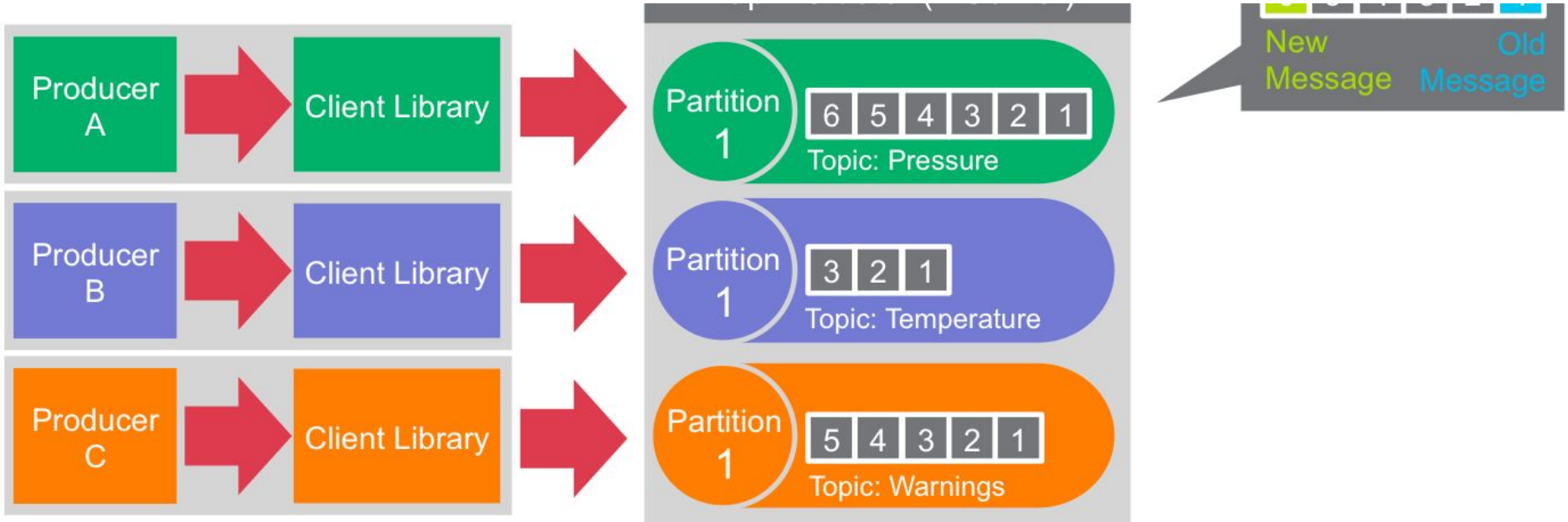

Producer
Client Library
Partition1: Topic - Pressure
Partition1: Topic - Temperature
Partition1: Topic - Warning
Server 1
Consumer
Client Library

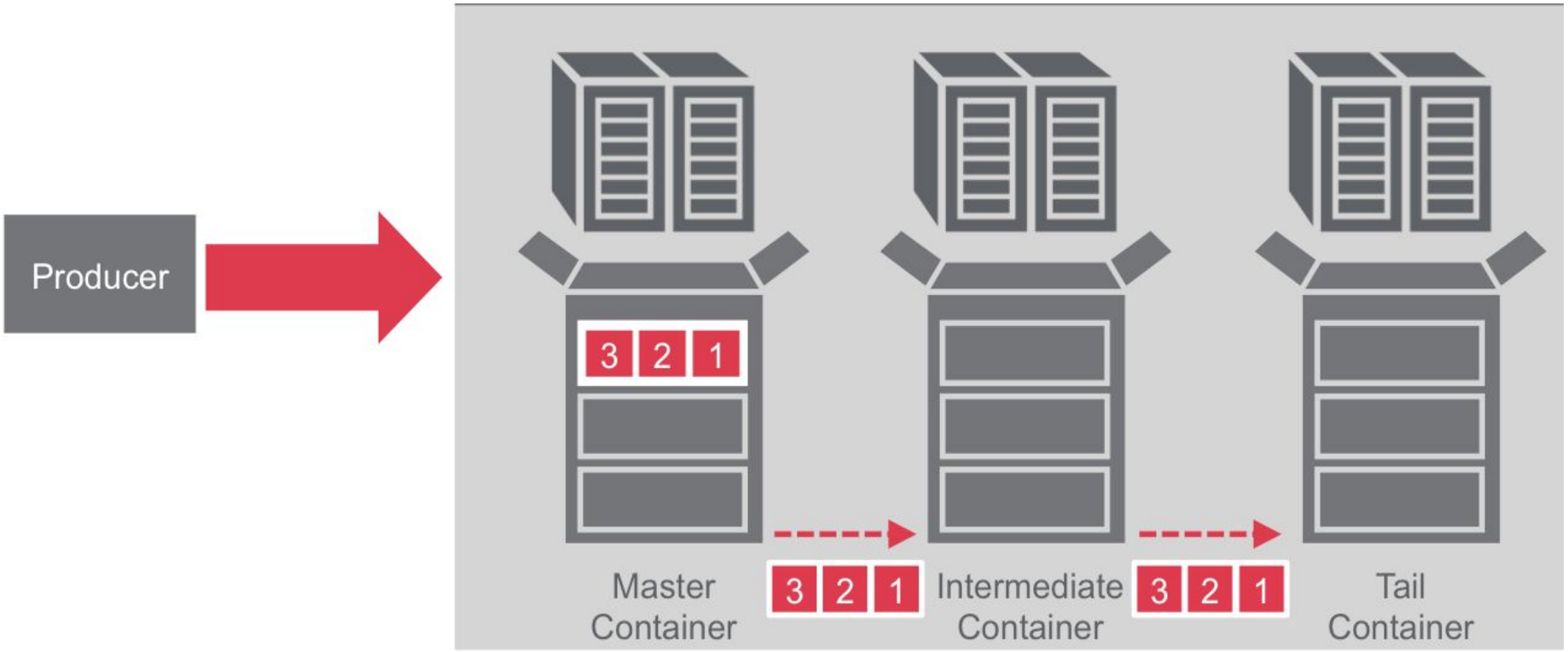
Goals

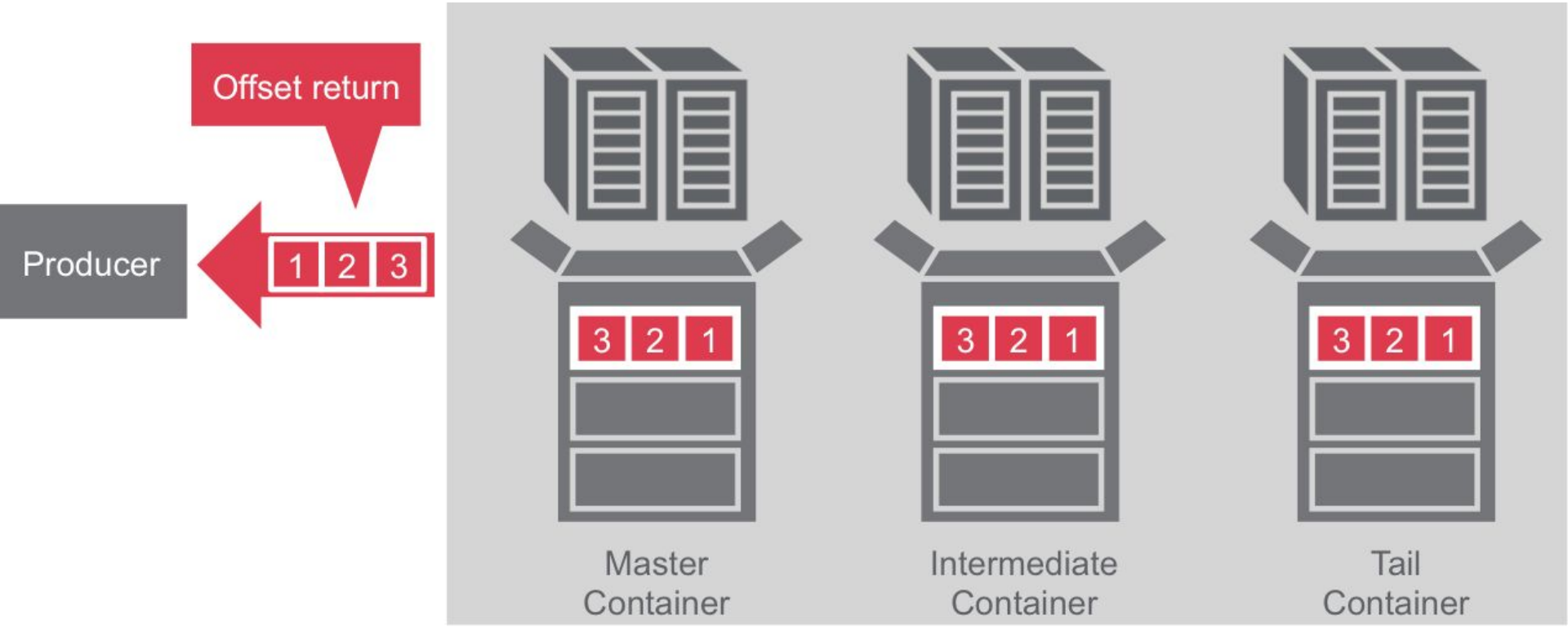

Time-Stamped
Pressure
Temperature
Warnings
Filters
Search
Field
Historical Production
Injection Volume


Producer
Buffer
3
2
1
6
5
4
Client Library
Partition
1
Topic: Pressure

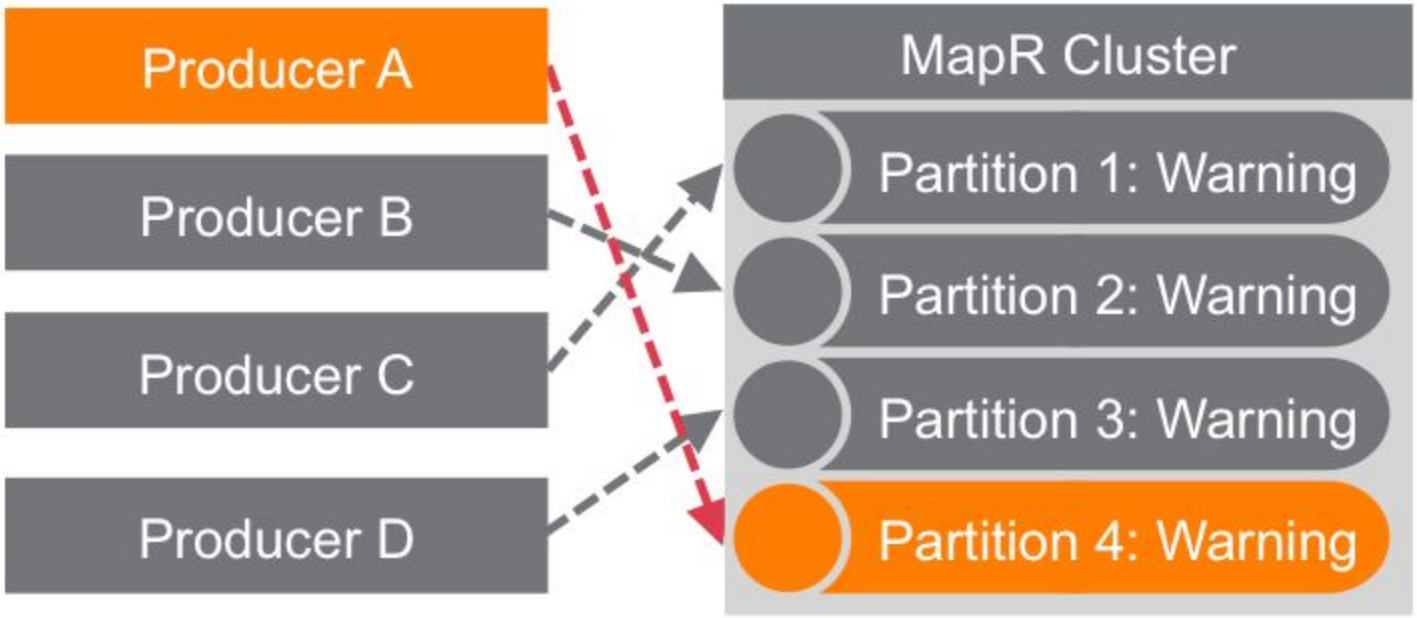

Producer
Buffer
Client Library
Partition
1
6
5
4
3
2
1
Topic: Pressure
New
Message
Old
Message


Producer A
Client Library
Producer B
Client Library
Producer C
Client Library
Partition 1
6 5 4 3 2 1
Topic: Pressure
Partition 1
3 2 1
Topic: Temperature
Partition 1
5 4 3 2 1
Topic: Warnings
New Message
Old Message


Producer
3 2 1
Master Container
3 2 1
Intermediate Container
3 2 1
Tail Container

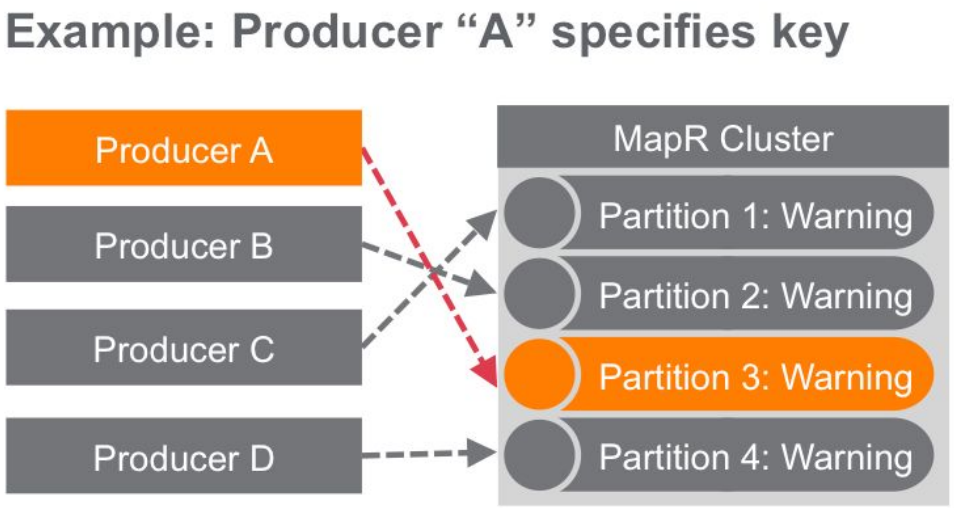

Offset return
Producer
1 2 3
3 2 1
3 2 1
3 2 1
Master Container
Intermediate Container
Tail Container

Example: Producer “A” specifies Partition ID=4



Parallelism When Publishing

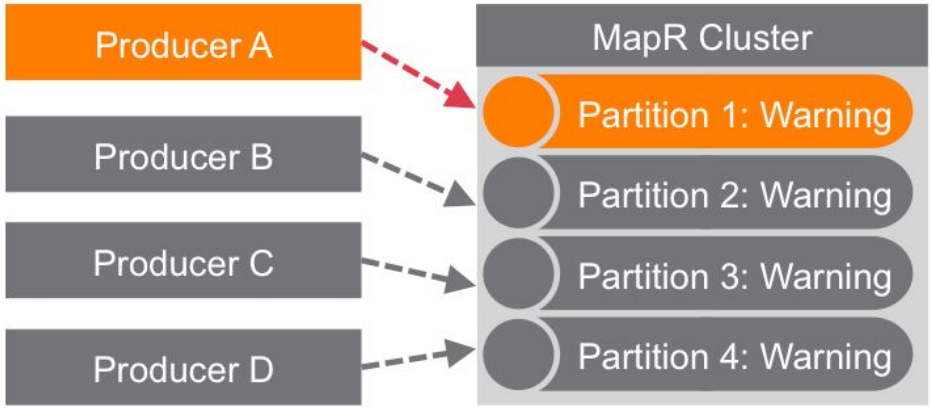
How do servers choose which partition to publish to?



Parallelism When Publishing

How do servers choose which partition to publish to?

Partitions assigned randomly, sticky round robin



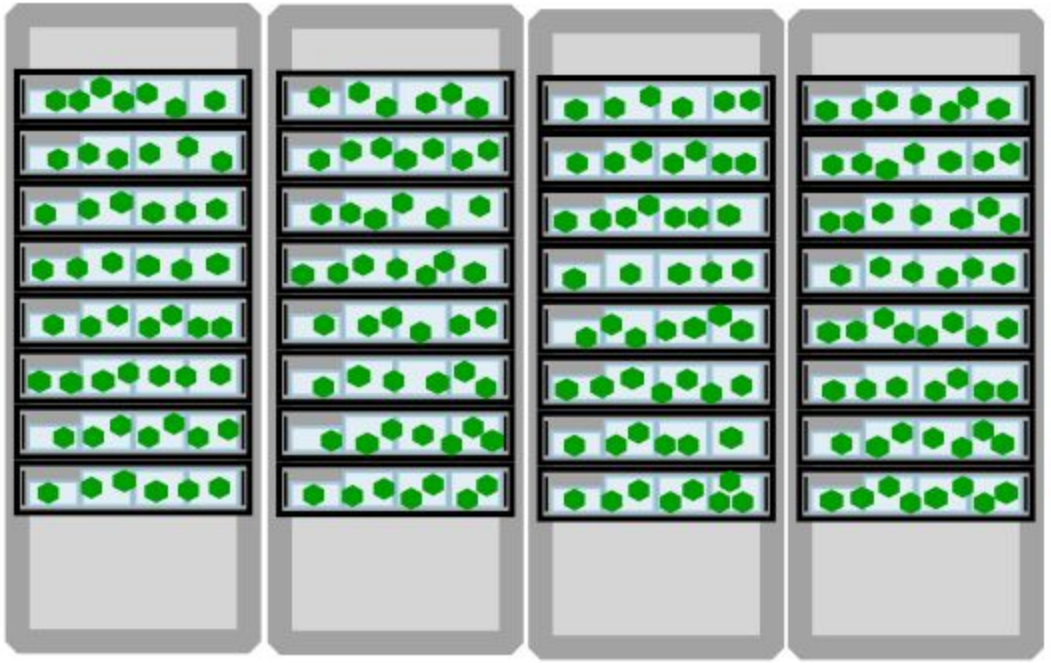

Partition
1
3
2
1
Topic: Warning
Get Unread
Client Library
Poll
Consumer


Partition
1
3 2 1
Topic: Warning
Unread Returned
3 2 1
Client Library
Consumer
Extracts
Data

When Are Messages Deleted?

- Older messages deleted based on time to live
- Time-to-live set when stream is created
- Messages don't expire if time-to-live is zero
- Expired messages deleted automatically

Partitions Are Automatically Distributed



Parallelism When Reading

To read messages in parallel:

- create consumer groups
- consumers with same `group.id`
- partitions assigned dynamically round-robin

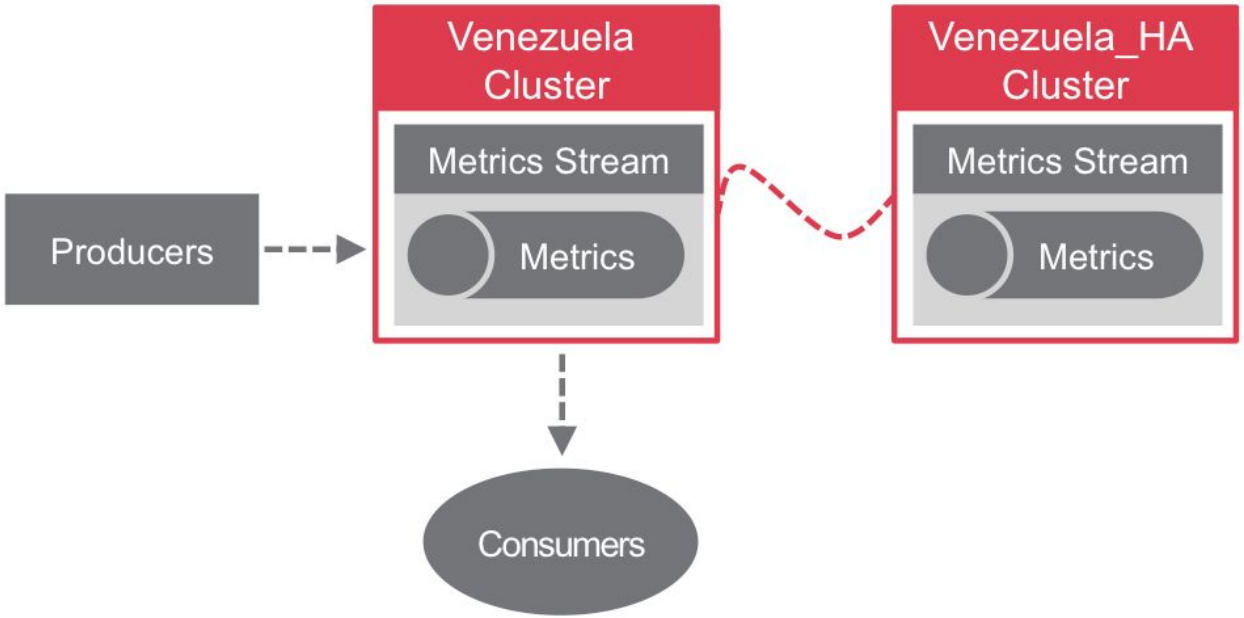
Partitions Re-Assigned Dynamically

If consumer goes offline, partitions re-assigned

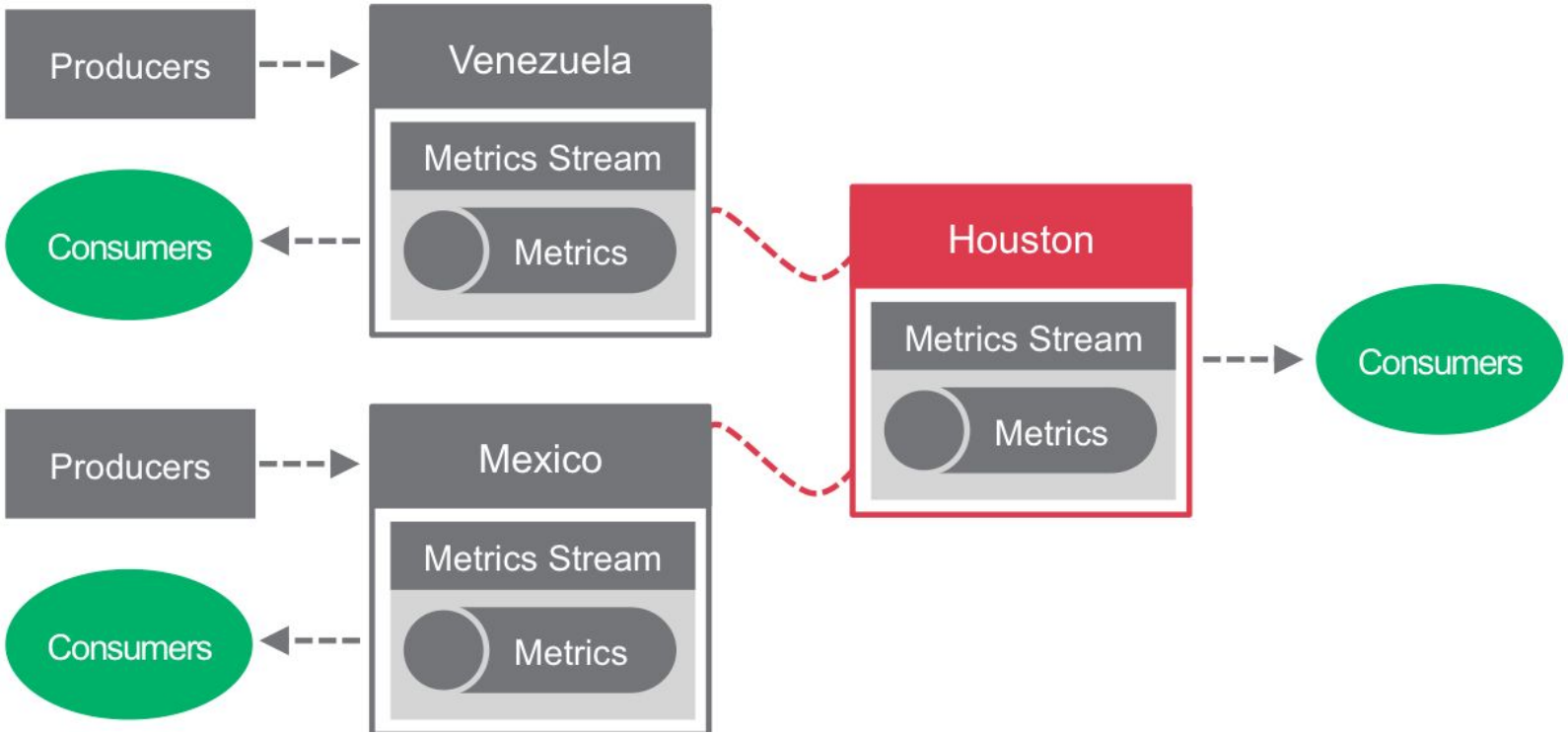
Saving of Cursor Position: Read Cursor

- Cursors keep track of messages read
- Read cursor: offset ID of most recent message sent to consumer

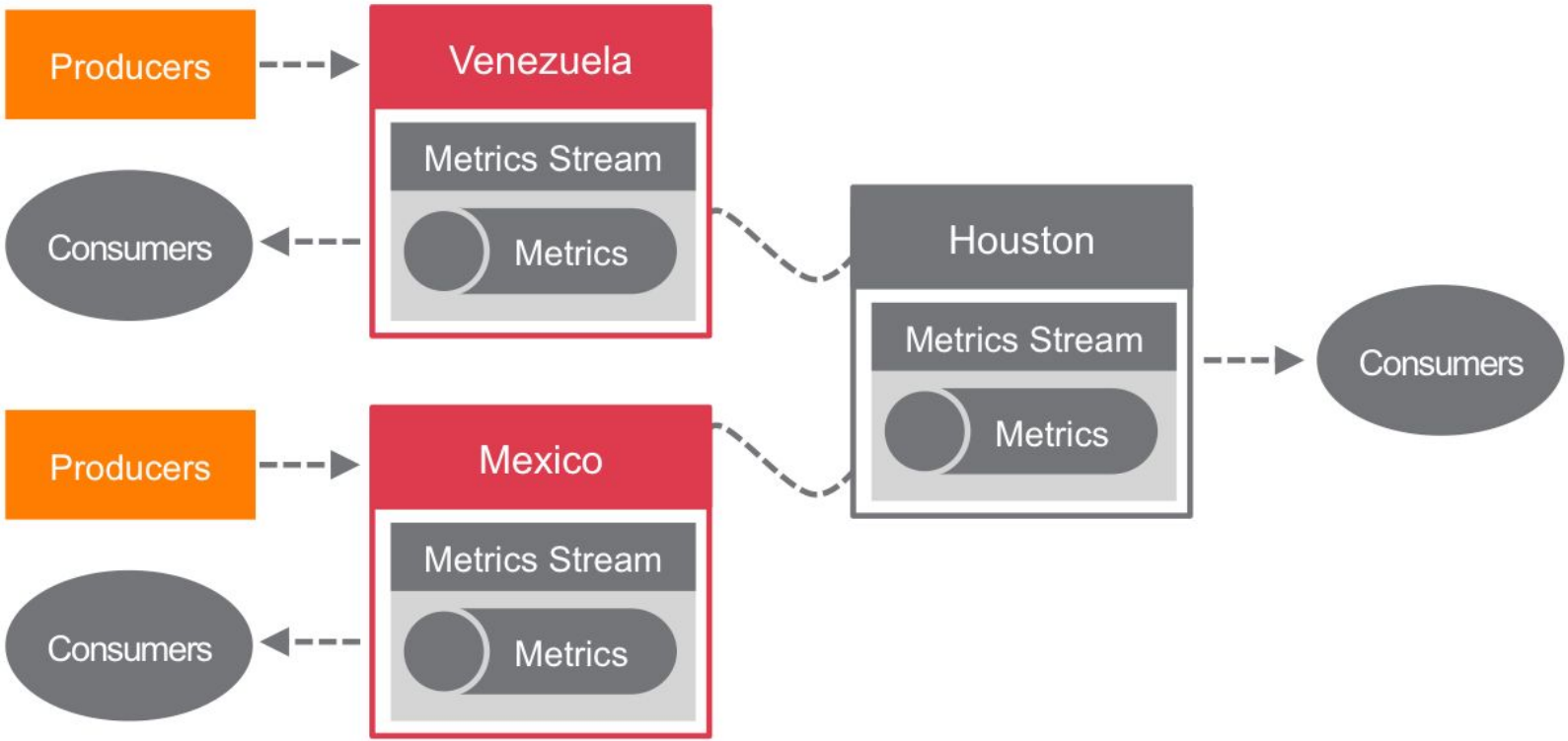
Replicating Streams: Master-Slave Replication



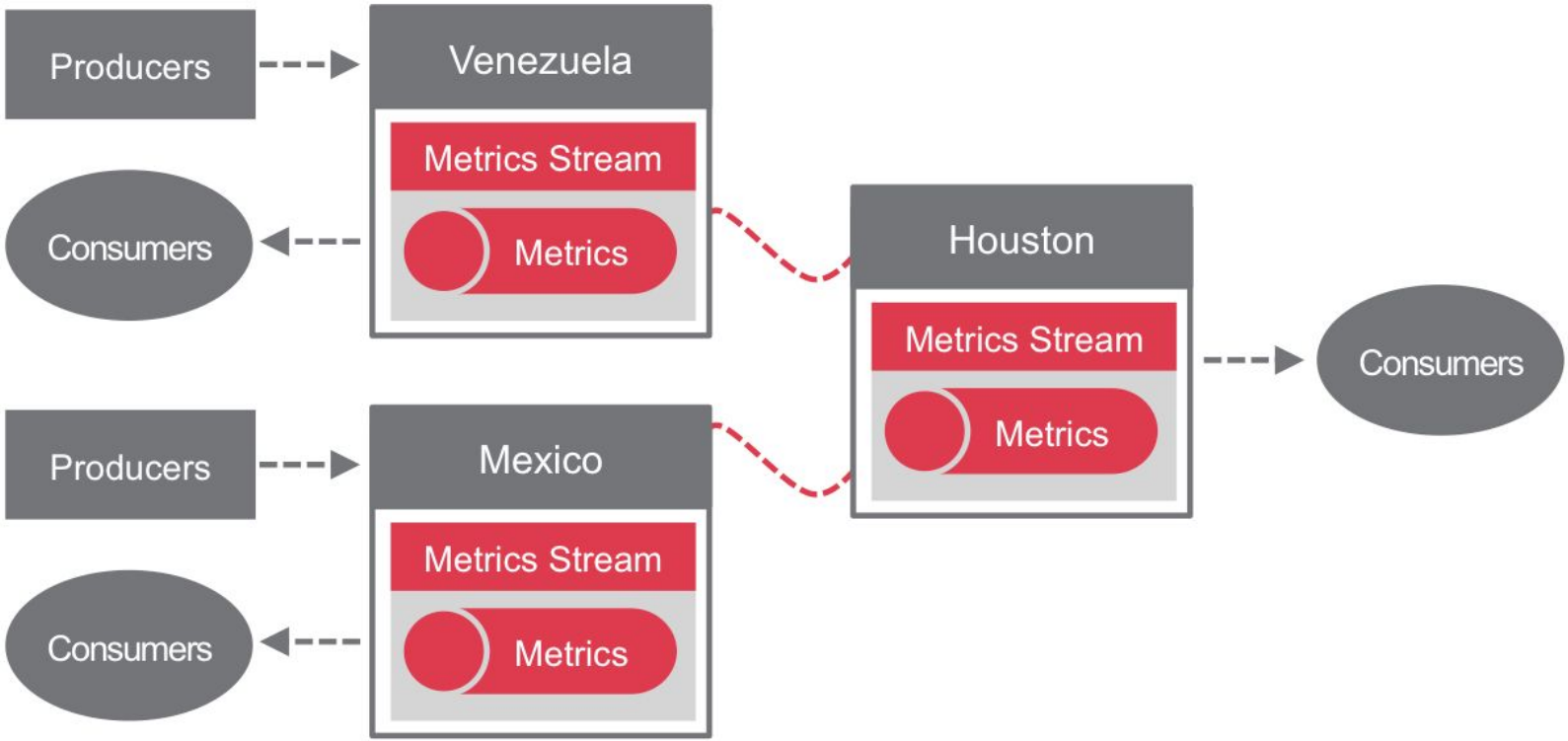
Replicating Streams: Many-to-One Replication



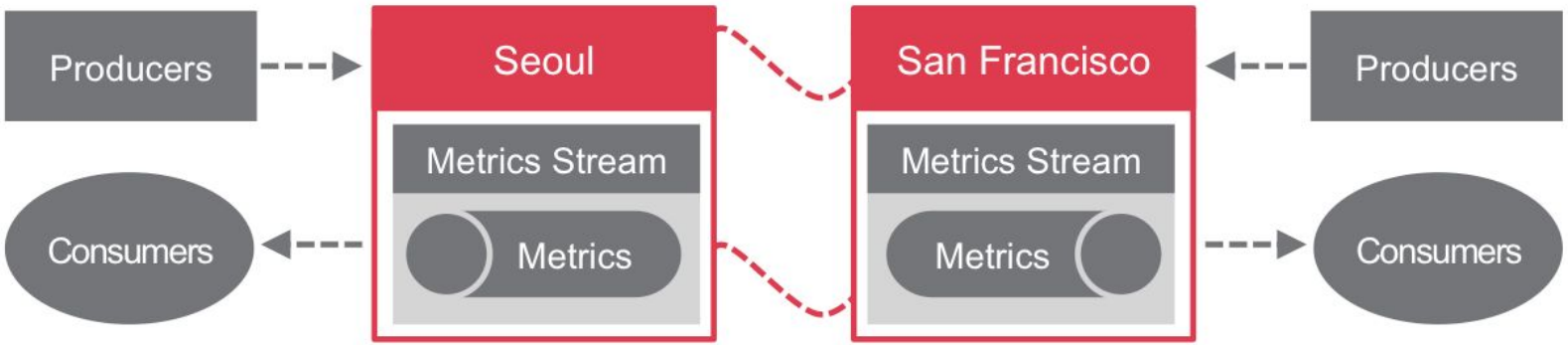
Replicating Streams: Many-to-One Replication



Replicating Streams: Many-to-One Replication

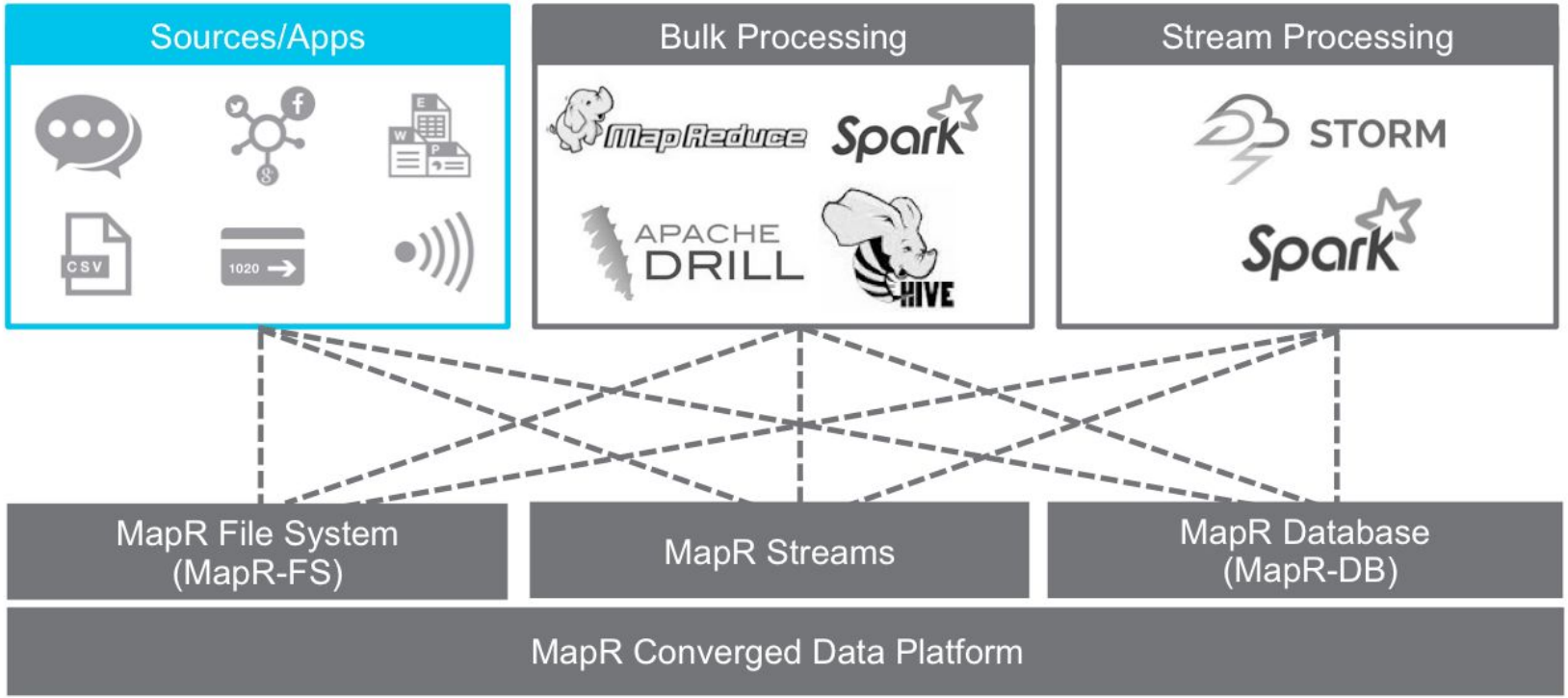


Replicating Streams: Multi-Master Replication

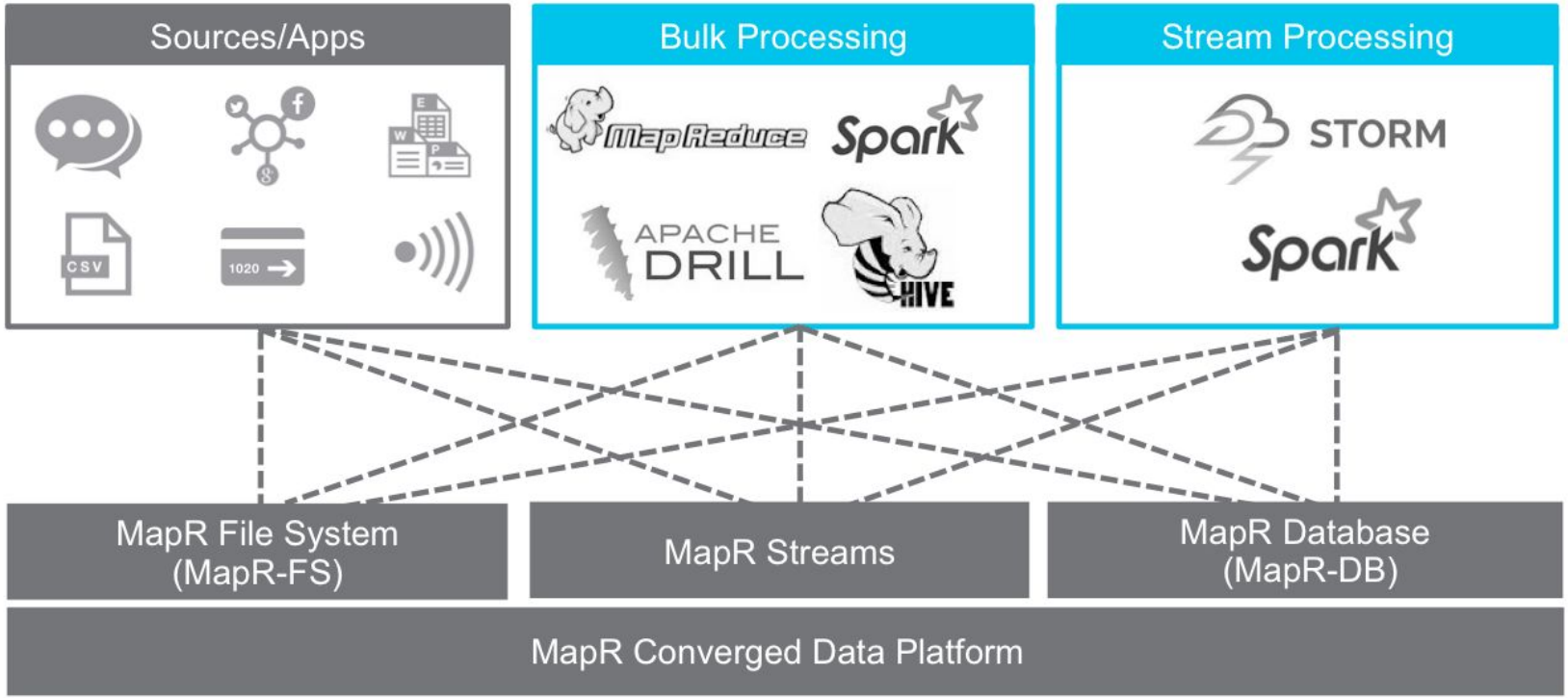


Stream Replication

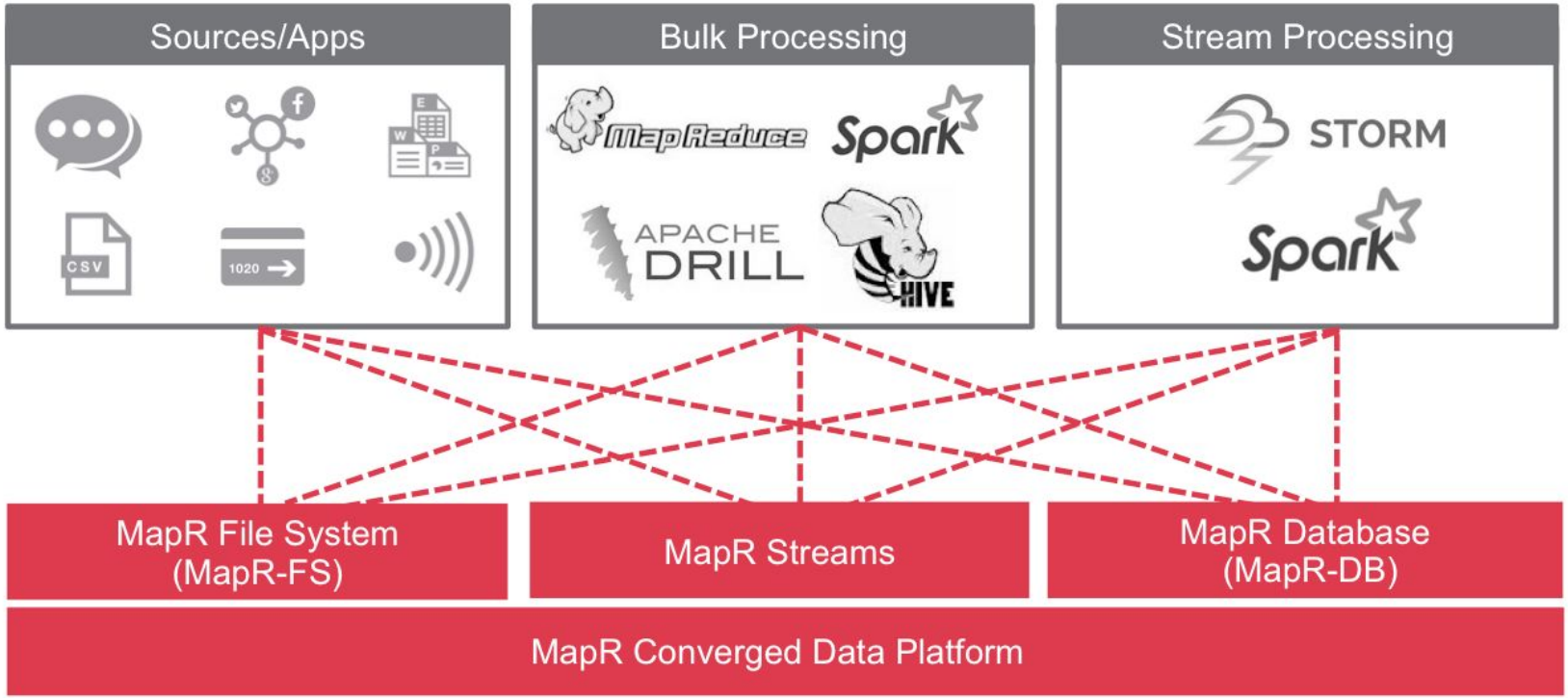
Building a Complete Data Architecture



Building a Complete Data Architecture



Building a Complete Data Architecture



Goals

Learning Goals

- Create a stream
- Develop a Java producer
- Develop a Java consumer

Learning Goals

- **Create a stream**
- Develop a Java producer
- Develop a Java consumer

Create a Stream

maprcli command for creating a stream

```
maprcli stream create -path /<filepath & name> \

  -consumeperm u:<userID> -produceperm u:<userID> \

  -topicperm u:<userID>
```

Create a Stream

Optional parameters to grant security permissions

```
maprcli stream create -path /<filepath & name> \

  -consumeperm u:<userID> -produceperm u:<userID> \

  -topicperm u:<userID>
```

Create a Stream

`produceperm` determines which users can publish messages to topics in a stream

```
maprcli stream create -path /<filepath & name> \

  -consumeperm u:<userID> -produceperm u:<userID> \

  -topicperm u:<userID>
```

Create a Stream

`consumeperm` determines which users can read topics from a stream

```
maprcli stream create -path /<filepath & name> \

  -consumeperm u:<userID> -produceperm u:<userID> \

  -topicperm u:<userID>
```

Create a Stream

`topicperm` determines which users can create/remove topics

```
maprcli stream create -path /<filepath & name> \

  -consumeperm u:<userID> -produceperm u:<userID> \

  -topicperm u:<userID>
```

Create a Stream

`defaultpartitions` determines how many partitions are created in a topic

```
maprcli stream create -path /<filepath & name> \

  -consumeperm u:<userID> -produceperm u:<userID> \

  -defaultpartitions 3
```

Optionally Create a Topic

Create topics manually

```
maprcli stream topic create \
  -path <path and name of the stream> \
  -topic <name of the topic>
```

Goals

Learning Goals

- Create a stream
- **Develop a Java producer**
- Develop a Java consumer

Producer Sending Messages

1. **Set producer properties**
2. Create a producer
3. Build the message
4. Send the message

Sample Producer: Set Producer Properties

```
// Set up KafkaProducer properties

Properties properties = new Properties();

properties.put(”key.serializer",

 "org.apache.kafka.common.serialization.StringSerializer");

properties.put("value.serializer",

 "org.apache.kafka.common.serialization.StringSerializer");

. . .
```

Some Producer Properties

Property	Description
`key.serializer`	Java class to use to serialize the key of each message
`value.serializer`	Java class to use to serialize the value of each message
`client.id`	ID that identifies the producer (optional)

Producer Sending Messages

1. Set producer properties
2. **Create a producer**
3. Build the message
4. Send the message

Sample Producer: Create a Producer

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

public static KafkaProducer producer;

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

Producer Sending Messages

1. Set producer properties
2. Create a producer
3. **Build the message**
   - Specify:
     - topic – either already exists or is implicitly created
     - value of some type V
   - Optionally specify:
     - key of type K
     - partition ID of type integer
4. Send the message

Sample Producer: Build Message

```
String topic="<streamname>:<topicname>";

for(int i = 0; i < 3; i++) {
 // message text
 String txt = "msg " + i;

 // create ProducerRecord with topic and text
 ProducerRecord<String, String> record =
   new ProducerRecord<String, String>(topic, txt);
 . . .
}
```

Producer Sending Messages

1. Set producer properties
2. Create a producer
3. Build the message
4. **Send the message**
   - `producer.send(record)`

Sample Producer: Send the Message

```
for(int i = 0; i < 3; i++) {
   . . .
   Callback cb = new ProducerCallback();
   producer.send(record, cb);
}
producer.close()
```

Sample Producer: Send the Message

```
for(int i = 0; i < 3; i++) {
   . . .
   Callback cb = new ProducerCallback();
   producer.send(record, cb);
}
producer.close()
```

Sample Producer: Send the Message

```
for(int i = 0; i < 3; i++) {
   . . .
   Callback cb = new ProducerCallback();
   producer.send(record, cb);
}
producer.close()
```

Sample Producer: Send the Message

Producer callback

```
class ProducerCallback implements Callback {

 public void onCompletion(RecordMetadata meta, Exception e){
    if(e != null)
      e.printStackTrace();
    System.out.println(”offset is: " +meta.offset());
 }
}
```

Sample Producer: All Together

```
public class SampleProducer {
 String topic="/streams/pump:warning";
 public static KafkaProducer producer;

 public static void main(String[] args) {
   producer=setUpProducer();
   for(int i = 0; i < 3; i++) {
     String txt = "msg " + i;
     ProducerRecord<String, String> rec = new
       ProducerRecord<String, String>(topic, txt);
     producer.send(rec);
     System.out.println("Sent msg number " + i);
   }
  producer.close();
 }
```

Goals

Learning Goals

- Create a stream
- Develop a Java producer
- **Develop a Java consumer**

Consumer Getting Messages

1. **Set consumer properties**
2. Create a consumer
3. Subscribe to topics
4. Polls from topics
5. Process returned messages

Sample Consumer: Set Consumer Properties

```
// Set up KafkaConsumer properties
Properties properties = new Properties();
properties.put(”key.deserializer”,
 "org.apache.kafka.common.serialization.StringDeserializer");
properties.put("value.deserializer”,
 "org.apache.kafka.common.serialization.StringDeserializer");
```

Sample Consumer: Set Consumer Properties

```
// Set up KafkaConsumer properties
Properties properties = new Properties();
properties.put(”key.deserializer”,
 "org.apache.kafka.common.serialization.StringDeserializer");
properties.put("value.deserializer”,
 "org.apache.kafka.common.serialization.StringDeserializer");
```

Some Consumer Properties

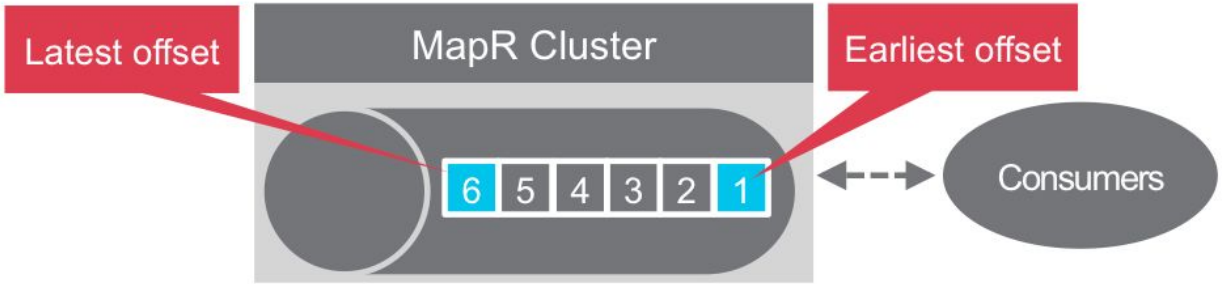
Property	Description
`key.deserializer` `value.deserializer`	Used to deserialize keys, values
`auto.offset.reset`	• **Earliest**: Reset offset to the offset of earliest message in the partition. • **Latest (default):** Reset offset to the offset of latest message in the partition. • **None:** Throws NoOffsetForPartitionException exception when consumer next polls for messages in subscription and no offset exists. Consumer must unsubscribe from partition before polling functions correctly

Consumer Properties

`auto.offset.reset`:

`earliest`: reset cursor to the offset of the earliest message in the partition

`latest`: reset the offset to the offset of the latest message in the partition.



Consumer Getting Messages

1. Set consumer properties
2. **Create a consumer**
3. Subscribe to topics
4. Polls from topics
5. Process returned messages

Sample Consumer: Set up `KafkaConsumer`

```
import org.apache.kafka.clients.consumer.KafkaConsumer;

public static KafkaConsumer consumer;


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

Consumer Getting Messages

1. Set consumer properties
2. Create a consumer
3. **Subscribe to topics**
4. Polls from topics
5. Process returned messages

Sample Consumer: Subscribe

```
public static String topic = ”/stream/pump:warning";

// Subscribe to the topic
consumer.subscribe(Arrays.asList(topic));
```

Consumer Getting Messages

1. Set consumer properties
2. Create a consumer
3. Subscribe to topics
4. **Polls from topics**
5. Process returned messages

Sample Consumer: Poll

```
long pollTimeOut = 1000;
while (true) {
  // Request unread messages from the topic
  ConsumerRecords<String, String> msgs =
    consumer.poll(pollTimeOut);
  . . .
}
```

Consumer Getting Messages

1. Set consumer properties
2. Create a consumer
3. Subscribe to topics
4. Polls from topics
5. **Process returned messages**
   - **One for each message found in `poll()` `ConsumerRecords<K, V>`**
   - **Key of type K**
   - **Value of type V**
   - **Partition and topic as well**

Sample Consumer: Process Returned Messages

```
ConsumerRecords<String, String> msgs =
   consumer.poll(pollTimeOut);

// Process returned messages
Iterator<ConsumerRecord<String, String>> iter =
   msgs.iterator();
```

Sample Consumer: Process Returned Messages

```
Iterator<ConsumerRecord<String, String>> iter =
   msgs.iterator();

// get ConsumerRecord from ConsumerRecord iterator
 while (iter.hasNext()) {
     ConsumerRecord<String, String> record = iter.next();
     System.out.println(”read msg" + record.toString());
 }
```

Example `record.toString()`

```
topic=/events:sensor,partition=0,offset=192,
    key=null,value=Msg1
```

Sample Consumer: All Together

```
public class MyConsumer {
 public static String topic = "/stream/pump:warning”;
 public static KafkaConsumer consumer;
 public static void main(String[] args) {
  configureConsumer(args);
  consumer.subscribe(topic);
  while (true) {
    ConsumerRecords<String, String> msg=
      consumer.poll(pollTimeOut);
    Iterator<ConsumerRecord<String, String>> iter =
      msg.iterator();
    while (iter.hasNext()) {
       ConsumerRecord<String, String> record = iter.next();
       System.out.println(”read " + record.toString());
    }
   }
  consumer.close();
 }
}
```

Sample Consumer: Run the Consumer

```
$java -cp my-app.jar:`mapr classpath` solution.MyConsumer

read msg(topic=/
events:sensor,partition=0,offset=192,key=null,value=Msg1)
read msg(topic=/
events:sensor,partition=0,offset=193,key=null,value=Msg2)
read msg(topic=/
events:sensor,partition=0,offset=194,key=null,value=Msg3)
```

Goals

Learning Goals

- Describe producer properties and options for:
  - buffering and batching of messages
  - publishing to partitions
- Describe consumer properties and options for:
  - fetching data
  - consumer groups
  - read cursors
- Explain messaging semantics

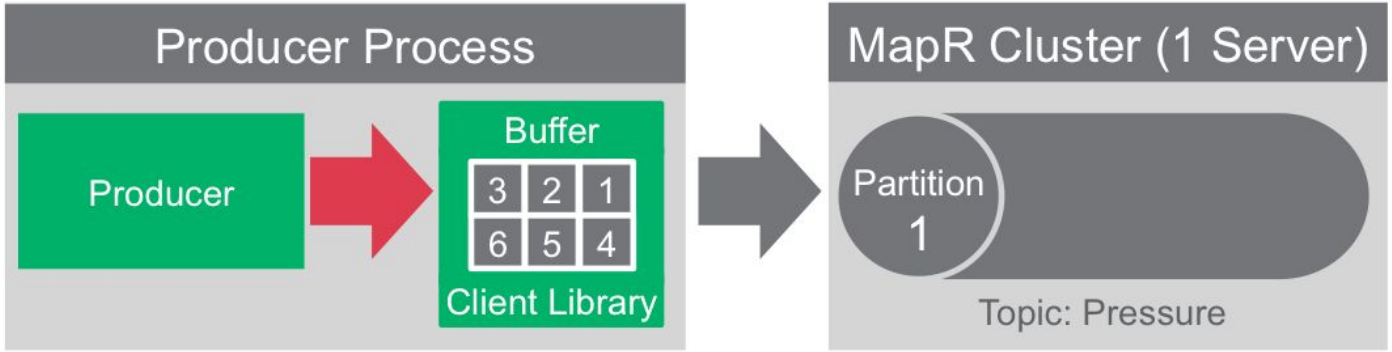
Learning Goals

- **Describe producer properties and options for:**
  - **buffering and batching of messages**
  - publishing to partitions
- Describe consumer properties and options for:
  - fetching data
  - consumer groups
  - read cursors
- Explain messaging semantics

Review

Review: How Are Messages Sent?

- Producers create and send messages to client library
- Client library buffers the messages



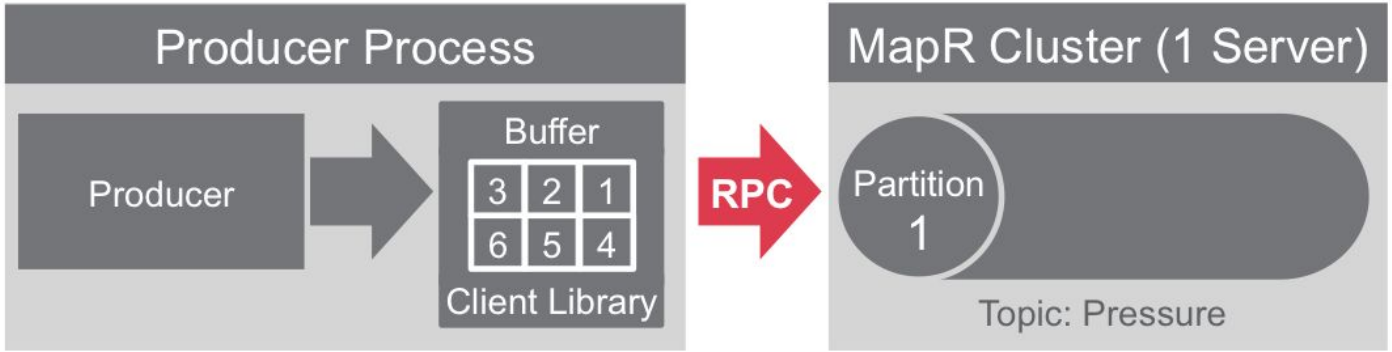
Producer Buffering and Batching Properties

Property	Description
`buffer.memory` (supported from Apache Kafka)	(default: 33554432) The total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are generated faster than they can be delivered to the server the producer will block.

When Are Messages Published?

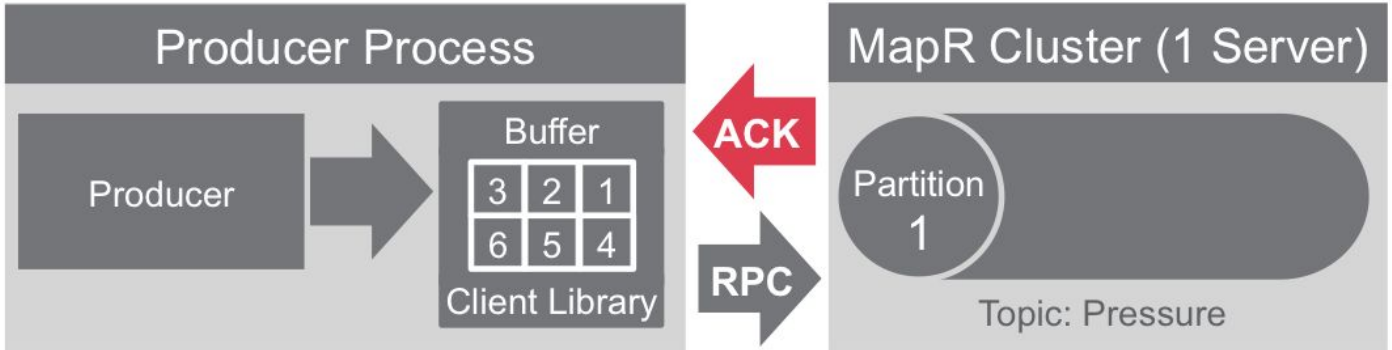
When any of four conditions met:

- Messages buffered is enough to make efficient RPC to Server
- Messages buffered = `streams.buffer.max.time.ms` setting
- Bytes buffered = `buffer.memory` setting
- Producer calls `flush()` method



Two Modes of Publishing

- `streams.parallel.flushers.per.partition` default **`true`**:
  - does **not** wait for ACK before sending more messages
  - possible for messages to arrive out of order
- `streams.parallel.flushers.per.partition` set to **`false`**:
  - client library will wait for ACK from server
  - slower than default setting



Guaranteed Ordering

- Messages are stored in partitions in the order they are received
- Messages are always read from a partition in order

Note:

- By default producer buffers are flushed in parallel
- Can cause out of order messages
- If an issue, MapR Streams:

```
streams.parallel.flushers.per.partition=false
```

Goals

Learning Goals

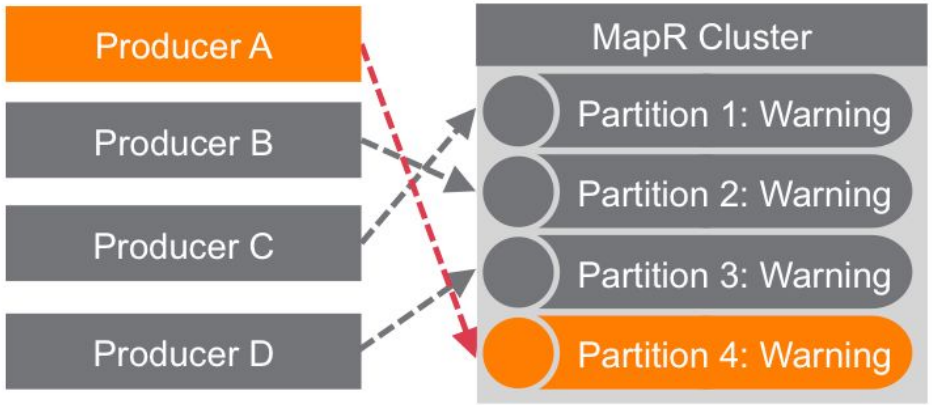
- **Describe producer properties and options for:**
  - buffering and batching of messages
  - **publishing to partitions**
- Describe consumer properties and options for:
  - fetching data
  - consumer groups
  - read cursors
- Explain messaging semantics

Review

Review: Parallelism When Publishing

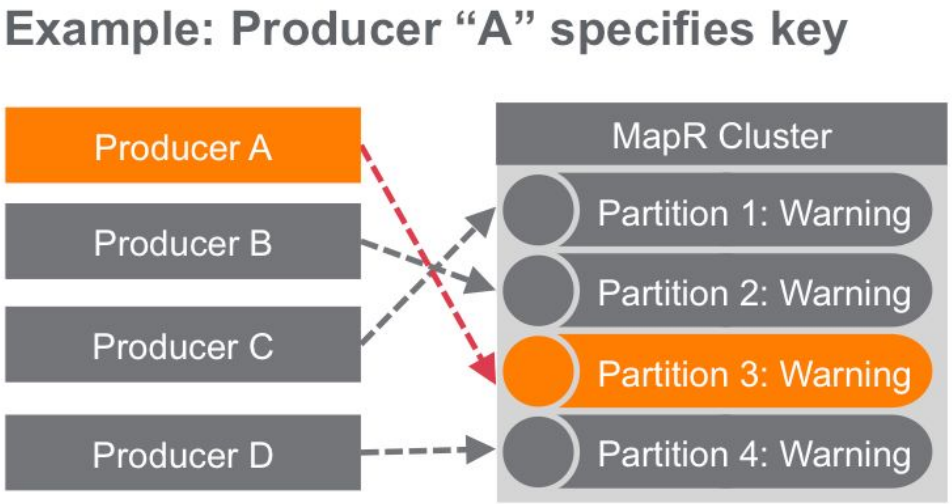
How do servers choose which partition to publish to?

Example: Producer "A" specifies Partition ID=4



Review: Parallelism When Publishing

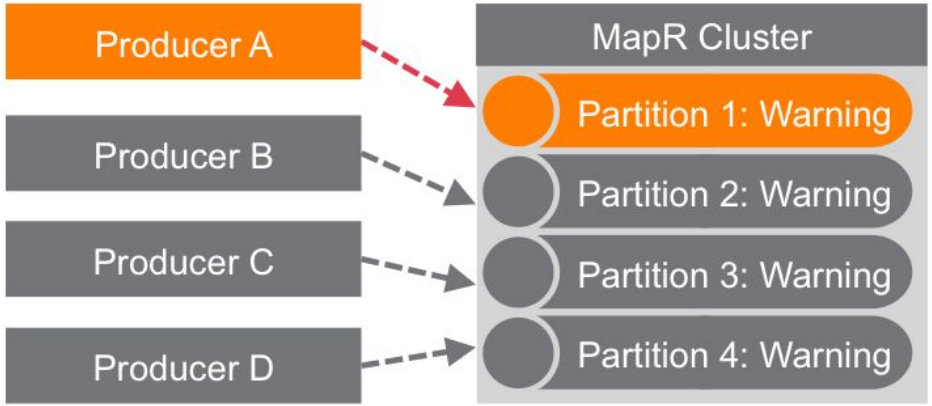
How do servers choose which partition to publish to?



Review: Parallelism When Publishing

How do servers choose which partition to publish to?

Partitions assigned randomly, sticky round robin



Producer Builds Message Specifying Key for Partitioning

```
String key="sensorxyz";
// create ProducerRecord with topic and text
ProducerRecord<String, String> record =
   new ProducerRecord<String,String>(topic,key,txt);

producer.send(record);
```

Producer Builds Message Specifying a Partition

```
Integer partition=1;
 // create ProducerRecord with topic and text
 ProducerRecord<String, String> record =
   new ProducerRecord<String,String>(topic,partition,txt);

producer.send(record);
```

Producer Property for Topic Metadata

Property	Description
`metadata.max.age.ms` (supported from Apache Kafka)	How frequently to fetch metadata, which might discover new topics or partitions (default 600 * 1000 msec)

Producer Property for Streams Partitioner Class

Property	Description
`streams.partitioner.class`	Allows to specify a java class for determining topic partition

```
Properties properties = new Properties();
properties.put(”streams.partitioner.class",
 ”myorg.MyPartitoner.class");
```

Example: Partitioner Implementation

```
public class MyPartitioner implements Partitioner {
  public int partition(String topic, Object key,
    byte[] keyBytes, Object value,
    byte[] valueBytes,Cluster cluster){
    List topicPartitions=cluster.partitionsForTopic(topic);
    int partition;
    partition=key % topicPartitions.size()

    return partition;
  }
}
```

Goals

Learning Goals

- Describe producer properties and options for:
  - buffering and batching of messages
  - publishing to partitions
- **Describe consumer properties and options for:**
  - **fetching data**
  - consumer groups
  - read cursors
- Explain messaging semantics

Review

Review: How are Messages Read?

- Consumers read at their own pace
- Consumers polls topics it subscribes to
- Client library requests unread messages

Some Consumer Properties

Property	Description
`max.partition.fetch.bytes`	Number of bytes of message data to attempt to fetch for each partition in each poll request (default: 64KB)
`fetch.min.bytes`	The minimum amount of data the server should return for a fetch request

Goals

Learning Goals

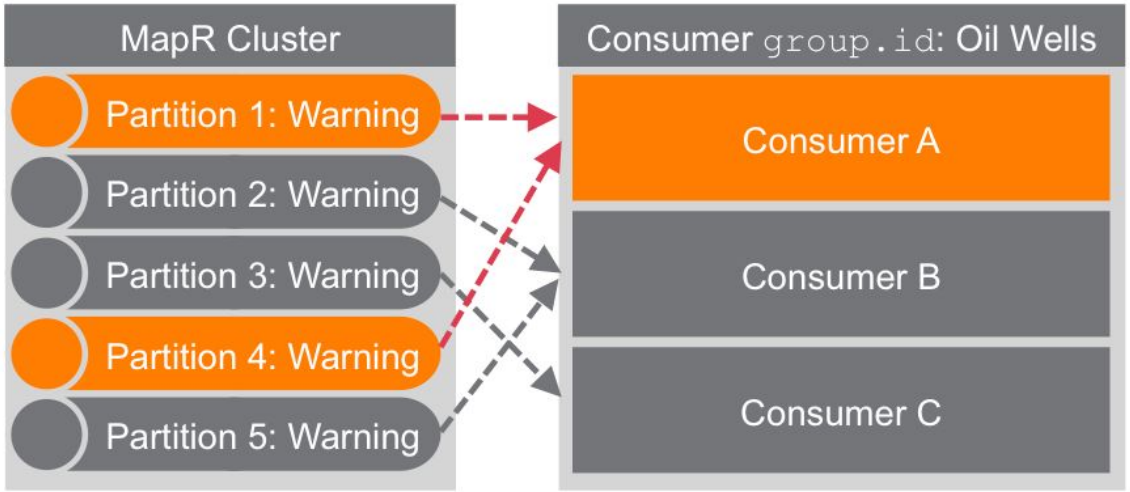
- Describe producer properties and options for:
  - buffering and batching of messages
  - publishing to partitions
- **Describe consumer properties and options for:**
  - fetching data
  - **consumer groups**
  - read cursors
- Explain messaging semantics

Review

Review: Parallelism When Reading

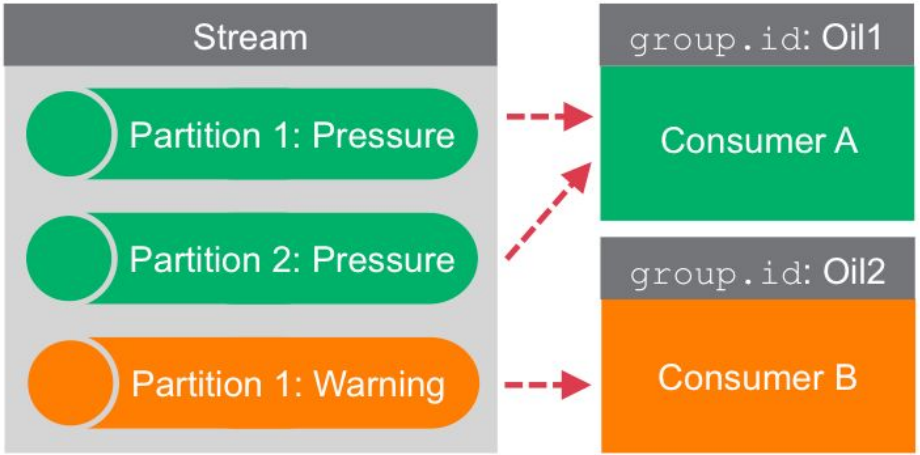
To read messages in parallel:

- create consumer groups
- consumers with same `group.id`
- partitions assigned dynamically round-robin



Consumer Groups

Enable cursor persistence which is useful even if group has only one member



Consumer Group Property

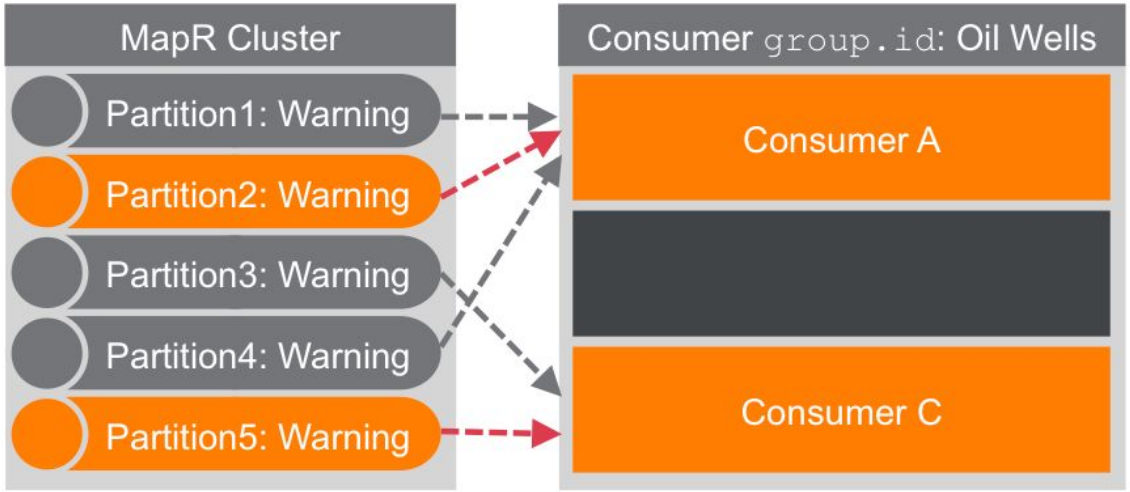
Property	Description
`group.id`	identifies the group of consumer processes to which this consumer belongs

```
// Set up KafkaConsumer properties
Properties props = new Properties();
props.put(”group.id”, ”pumppressure");
consumer = new KafkaConsumer<String, String>(props);
```

Review

Review: Partitions Re-assigned Dynamically

If consumer goes offline, partitions re-assigned



Example: ConsumerRebalanceListener

```
public class Listener implements ConsumerRebalanceListener {
 private Consumer<?,?> consumer;
 // Constructor
 public Listener(Consumer<?,?> consumer) {
   this.consumer = consumer;
 }
 public void onPartitionsAssigned(
    Collection<TopicPartition> partitions) {
  }
 public void onPartitionsRevoked(
   Collection<TopicPartition> partitions) {
  }
}
```

Example: ConsumerRebalanceListener

```
public class Listener implements ConsumerRebalanceListener {
 private Consumer<?,?> consumer;
 // Constructor
 public Listener(Consumer<?,?> consumer) {
   this.consumer = consumer;
 }
 public void onPartitionsAssigned(
    Collection<TopicPartition> partitions) {
  }
 public void onPartitionsRevoked(
   Collection<TopicPartition> partitions) {
   this.consumer.commitSync();
  }
}
```

Example: ConsumerRebalanceListener

```
public class Listener implements ConsumerRebalanceListener {
 private Consumer<?,?> consumer;

 public void onPartitionsAssigned(
    Collection<TopicPartition> partitions) {
 }
 public void onPartitionsRevoked(
   Collection<TopicPartition> partitions) {
   this.consumer.commitSync();
   System.out.print("Stopped Listening to:" + partitions);
 }
}
```

Example: ConsumerRebalanceListener

```
public class Listener implements ConsumerRebalanceListener {
 private Consumer<?,?> consumer;

 public void onPartitionsAssigned(
    Collection<TopicPartition> partitions) {
     System.out.print("Started Listening to:" + partitions);
 }
 public void onPartitionsRevoked(
   Collection<TopicPartition> partitions) {
 }
}
```

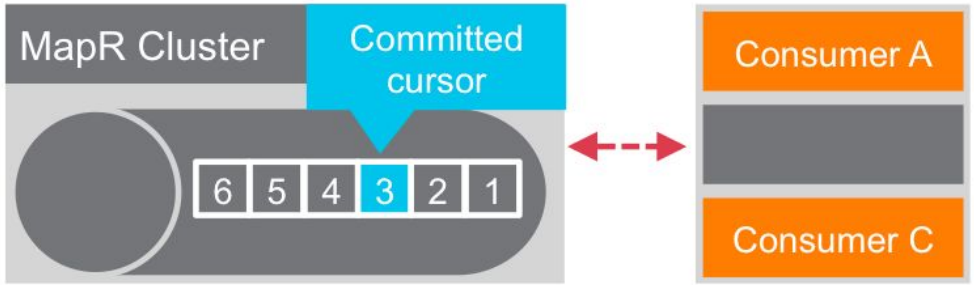
Example: ConsumerRebalanceListener

```
// subscribe to topic, provide ConsumerRebalanceListener
consumer.subscribe(topic,new Listener(consumer));
```

Partition Balancing and the Committed Cursor

When new consumers join, leave or new partitions added, rebalance occurs

- there may be a slight pause in message delivery while this happens
- can cause duplicates if consumers don't properly commit cursors



Goals

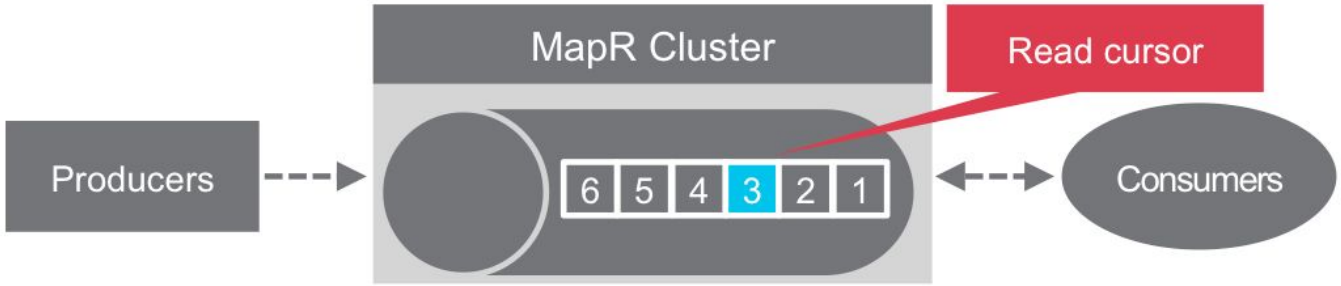
Learning Goals

- Describe producer properties and options for:
  - buffering and batching of messages
  - publishing to partitions
- **Describe consumer properties and options for:**
  - fetching data
  - consumer groups
  - **read cursors**
- Explain messaging semantics

Review

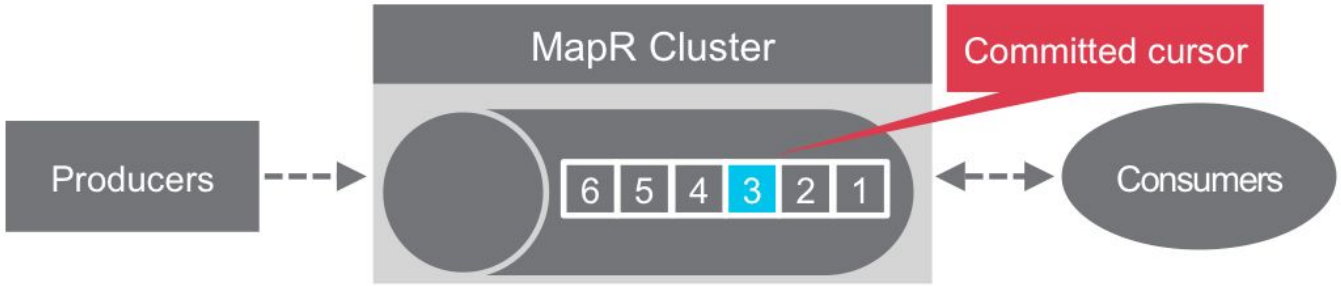
Review: Saving of Read Cursor Position

- Cursors keep track of messages read
- Read cursor: offset ID of most recent message sent to consumer



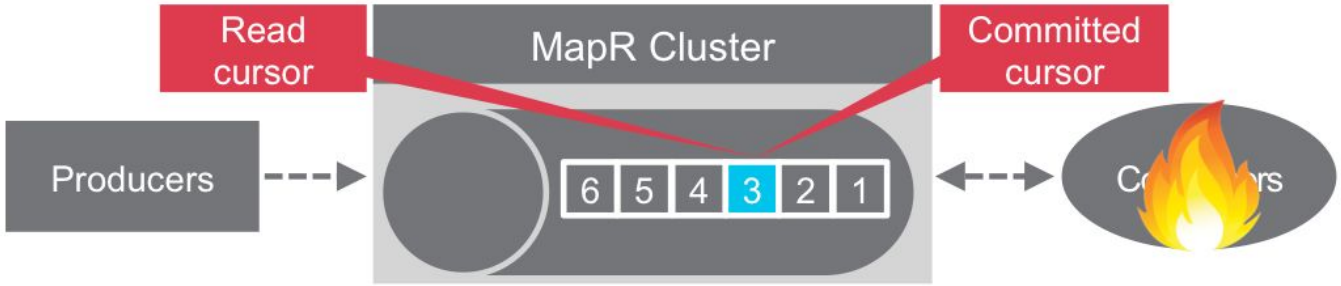
Review: Saving of Committed Cursor Position

Committed cursor: saved current position of read cursor



How Often Should Consumer Commit Cursor?

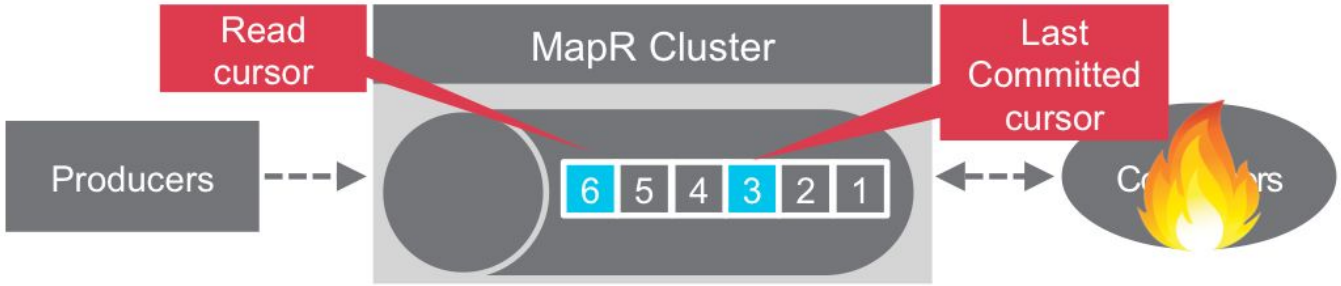
More frequent commits = less read duplication



Read duplication when Consumer fails?

How much read duplication when consumer fails:

- Time since last consumer commit
- Rate messages published

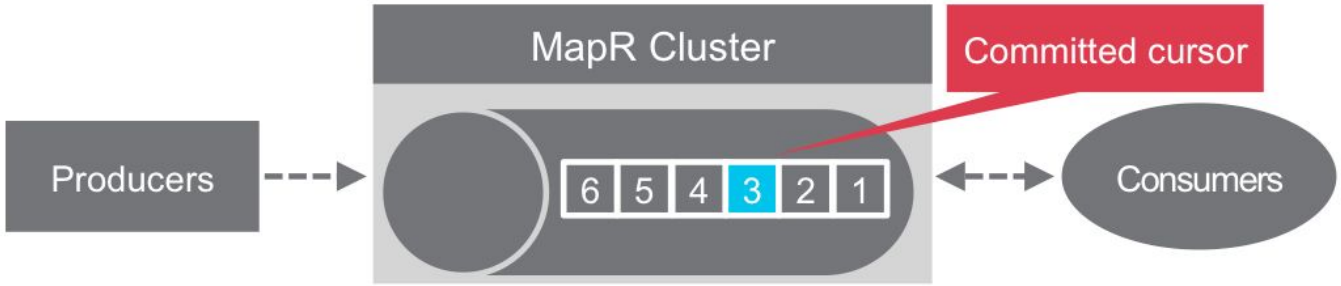


Consumer Properties: Cursor

Property	Description
`enable.auto.commit`	If true, periodically commits the highest offsets of the messages fetched by the consumer (default: true)
`auto.commit.interval.ms`	Frequency in milliseconds that the offsets are committed (default: 1000ms)

Commit Cursor: Automatic or Manual?

`enable.auto.commit` – if cursors should be committed automatically (default true)



Example: Consumer Properties

```
Properties props = new Properties();
props.put(”enable.auto.commit”, ”false");
consumer = new KafkaConsumer<String, String>(props);
. . .
// manually commit offsets of messages received
consumer.commitSync();
```

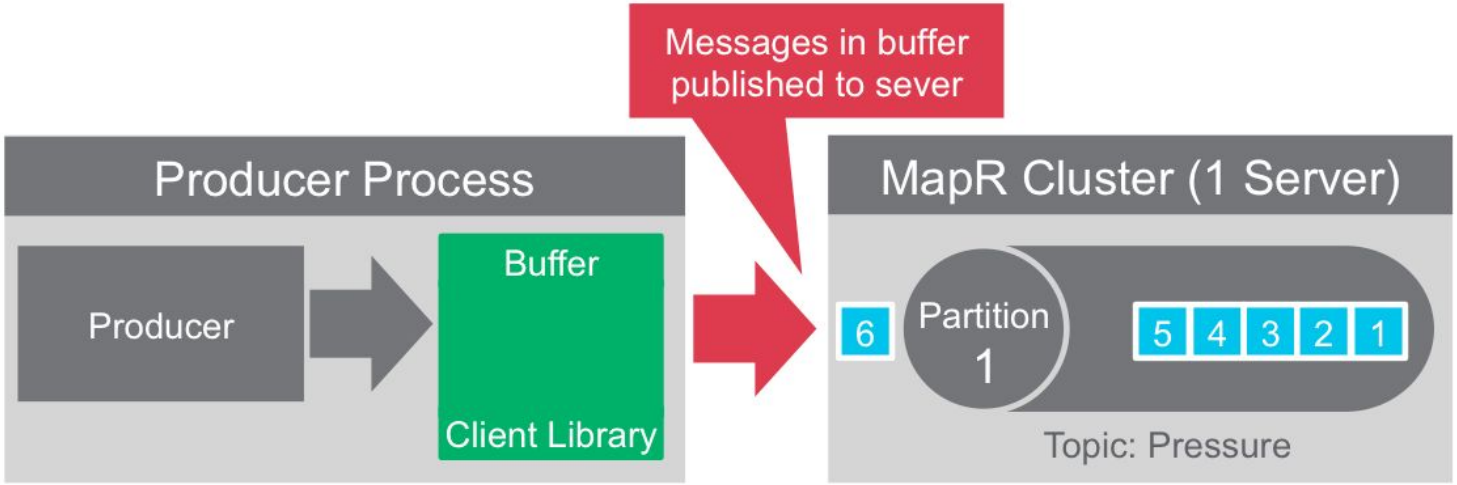
Goals

Learning Goals

- Describe producer properties and options for:
  - buffering and batching of messages
  - publishing to partitions
- Describe consumer properties and options for:
  - fetching data
  - consumer groups
  - read cursors
- **Explain messaging semantics**

Delivery Semantics: At Least Once

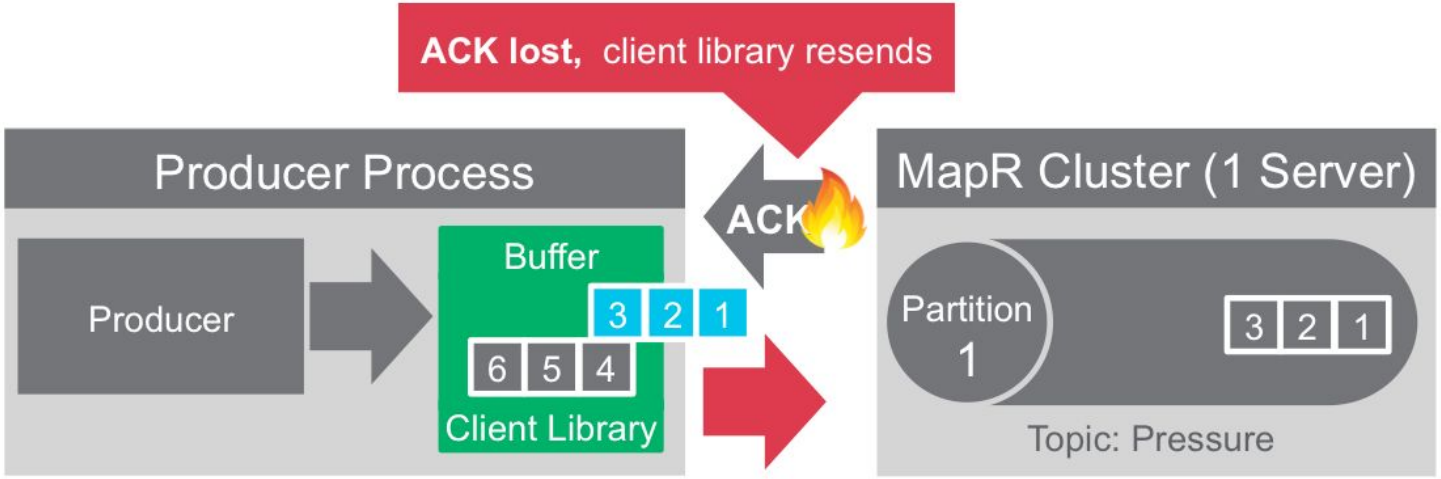
Once published to server, messages will be delivered



Message Duplication: Producer Library Resends

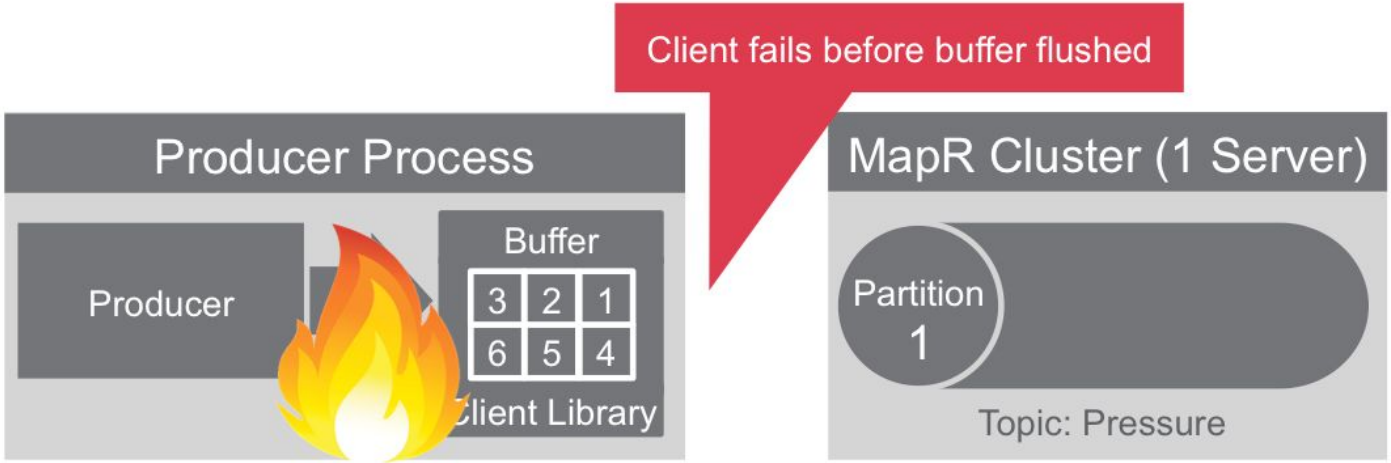
Messages may be duplicated

- producer side
- retransmit on network issue



Avoid Losing Unpublished Messages

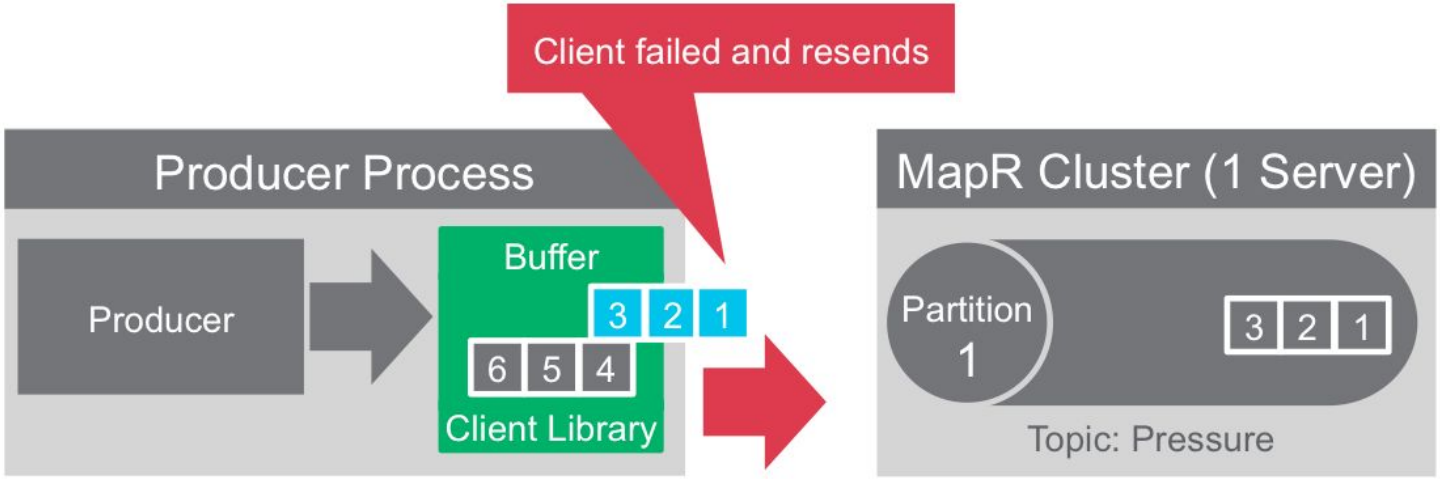
If a client fails before client buffer flushed, message unpublished producer can provide callback, resend on restart



Message Duplication: Producer Resends

Messages may be duplicated

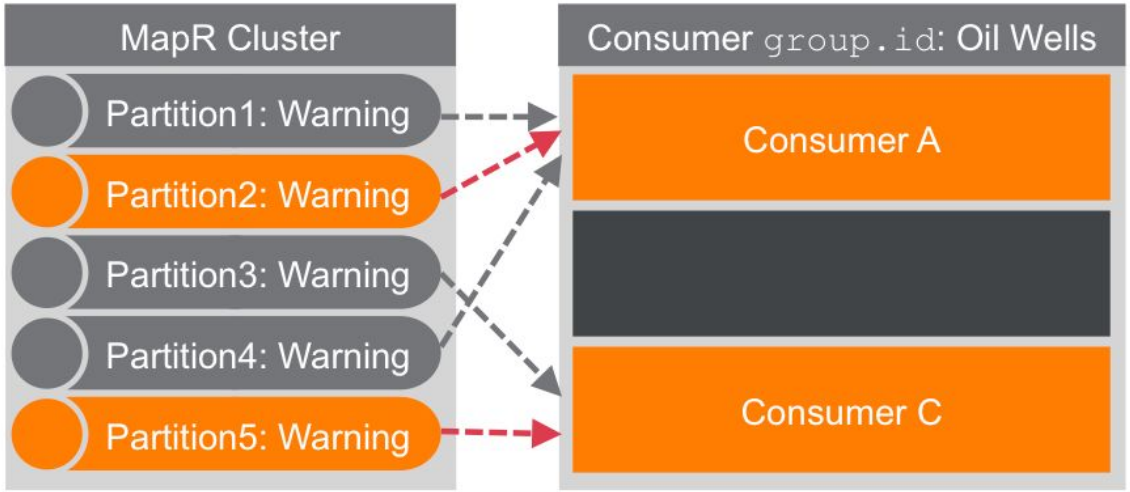
- Producer side
- Producer might fail and resend previously sent messages on restart



Message Duplication: Consumer Cursor Not Committed

Messages may be duplicated

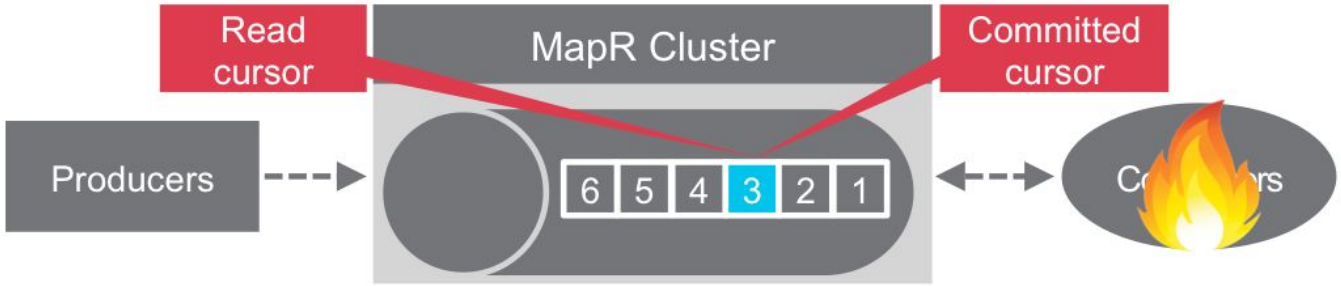
- consumer side
  - consumer partition re-assignment



Messages Duplication: Consumer Fails Before Commit

Messages may be duplicated

- consumer side
  - client might crash after get but before `cursor.commit()`



Delivery Semantics: Message Duplication

Suggestions if duplicate messages are a problem:

- design your applications for idempotent messages
- add unique ID for duplicate detection