Kafka API

Lesson Objectives

- Learn to use Kafka Java API

Kafka API

Kafka Clients

- Java is the 'first class' citizen in Kafka
  - Officially maintained
- Python on par with Java
  - Maintained by Confluent.io
- Other language libraries are independently developed
  - may not have 100% coverage
  - May not be compatible with latest versions of Kafka
  - Do your home work!
- REST proxy provides a language neutral way to access Kafka
- Full list: https://cwiki.apache.org/confluence/display/KAFKA/Clients

Kafka Java API

- Rich library that provides high level abstractions
  - No need to worry about networking / data format ..etc
- Write message / Read message
- Supports native data types
  - String
  - Bytes
  - Primitives (int, long ...etc.)

Java Producer Code (Abbreviated)

```java
// ** 1 **
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.common.serialization.IntegerSerializer;
...

// ** 2 **
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "SimpleProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

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

// ** 3 **
String topic = "test";
Integer key = new Integer(1);
String value = "Hello world";
ProducerRecord < Integer, String > record = new ProducerRecord<> (topic, key, value);
producer.send(record);
producer.close();
```

Producer Code Walkthrough

```
// ** 2 **  Recommended approach: use constants

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.common.serialization.IntegerSerializer

Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "SimpleProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

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

```
// ** 2 ** another approach
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("client.id", "SimpleProducer");
props.put("key.serializer",  "org.apache.kafka.common.serialization.IntegerSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

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

- **bootstrap.servers:** Specify the kafka brokers to connect to.
  - Best practice, specify more than one broker to connect to, so there is no single point of failure

```
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092,broker2:9092");
```

Producer Code Walkthrough

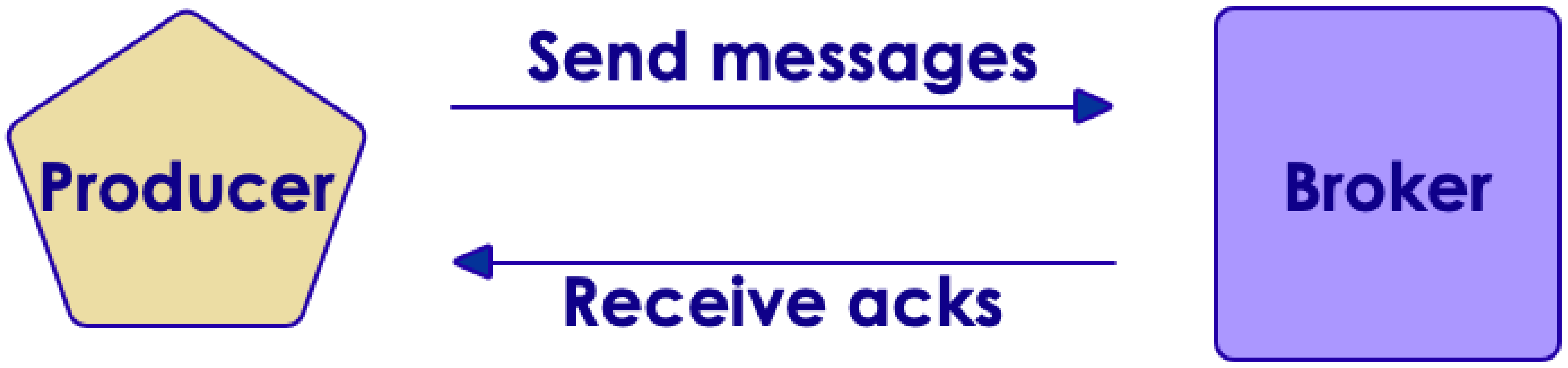
```
// ** 3 **
String topic = "test";
Integer key = new Integer(1);
String value = "Hello world";
ProducerRecord < Integer, String > record = new ProducerRecord<> (topic, key, value);
producer.send(record);
producer.close();
```

- Each **record** represents a message
- Here we have a <key,value> message
- send() doesn't wait for confirmation
- We send in batches
  - for increased throughput
  - Minimize network round trips
- Checkout [ProducerRecord API]

Producer Properties

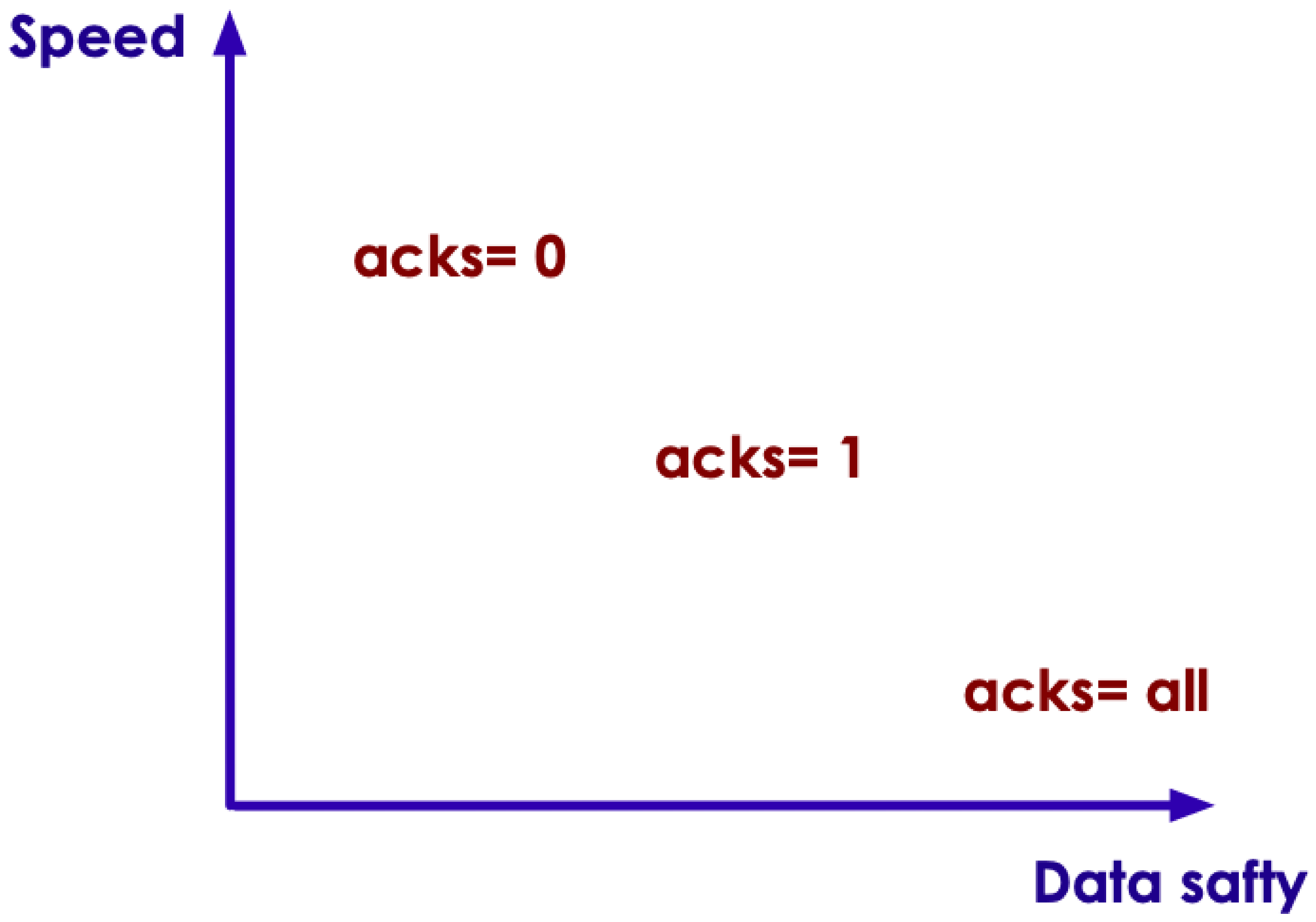
```
Properties props = new Properties();
props.put("boostrap.servers", "localhost:9092");
props.put("client.id", "SimpleProducer");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 1000);
props.put("linger.ms", 100);  // batch timeout 100ms
props.put("buffer.memory", 33554432); // 32 M
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

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

for(int i = 0; i < 100; i++) {
  producer.send(new ProducerRecord < String, String >(
      "my-topic", Integer.toString(i), Integer.toString(i)));
}
producer.close();
```

Producer Acknowledgements



ACK	Description	Speed	Data safety
acks = 0	- Producer doesn't wait for any acks from broker - Producer won't know of any errors	High	Low No guarantee that broker received the message
acks = 1 (default in Kafka 2)	- Broker will write the message to local log - Does not wait for replicas to complete	Medium	Medium Message is at least persisted on lead broker
acks = all (default in kafka 3)	- Message is persisted on lead broker and in replicas - Lead broker will wait for in-sync replicas to acknowledge the write	Low	High Message is persisted in multiple brokers

Producer Acknowledgements



Consumer Code (Abbreviated)

```java
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.IntegerDeSerializer

...

Properties props = new Properties(); // ** 1 **
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
props.put(ConsumerConfig.CLIENT_ID_CONFIG, "Simple Consumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeSerializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

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

consumer.subscribe(Arrays.asList("topic1")); // ** 2 **

try {
    while (true) {
       ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000)); // ** 3 **
       System.out.println("Got " + records.count() + " messages");
       for (ConsumerRecord < Integer, String > record : records) {
          System.out.println("Received message : " + record);
       }
   }
}
finally {
    consumer.close(Duration.OfSeconds(60));
}
```

Consumer Code Walkthrough

```
Properties props = new Properties(); // ** 1 **
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
props.put(ConsumerConfig.CLIENT_ID_CONFIG, "Simple Consumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeSerializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeSerializer.class.getName());


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

consumer.subscribe(Arrays.asList("topic1")); // ** 2 **
```

- **bootstrap,servers:** "broker1:9092,broker2:9092"
    - Connect to multiple brokers to avoid single point of failure
- **group.id:** consumers belong in a Consumer Group
- We are using standard serializers
- Consumers can subscribe to one or more subjects *// ** 2 ***

Consumer Code Walkthrough

```
try {
   while (true) {
      ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000)); // ** 3 **
      System.out.println("Got " + records.count() + " messages");
      for (ConsumerRecord < Integer, String > record : records) {
        System.out.println("Received message : " + record);
      }
   }
}
finally {
   consumer.close();
}
```

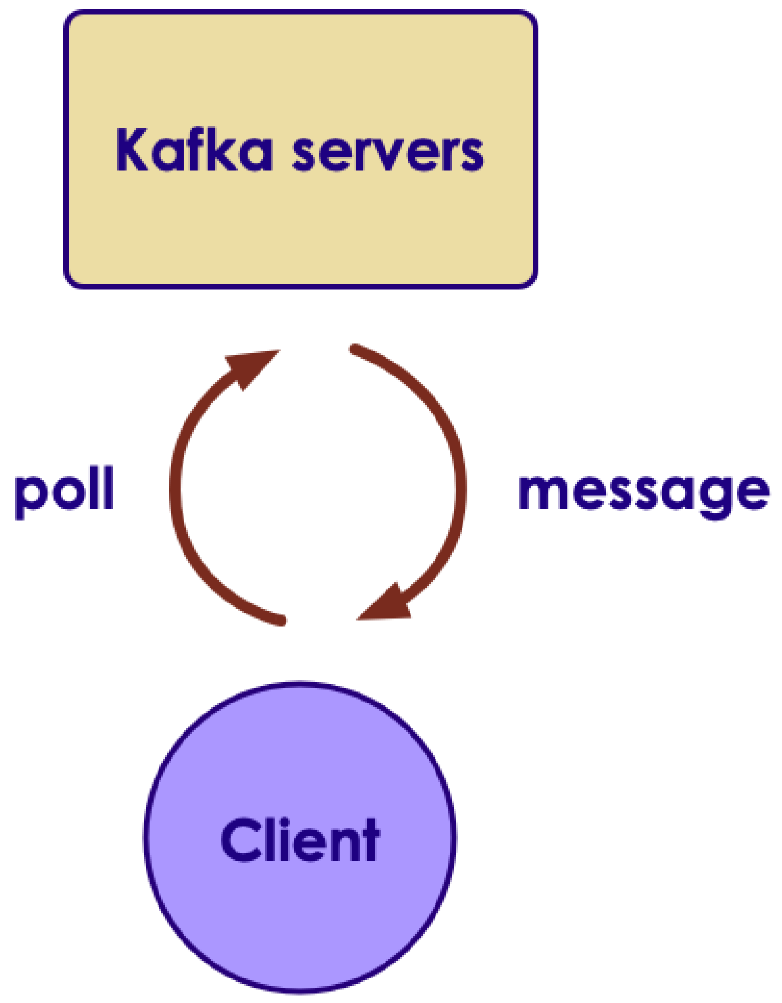
- Consumers must subscribe to topics before starting polling
    - Consumer.subscribe ("test.*") // wildcard subscribe
- Poll: This call will return in 1000 ms, with or without records
- Must keep polling, otherwise consumer is deemed dead and the partition is handed off to another consumer

Consumer Poll Loop

- Polling is usually done in an infinite loop.

```
while (keepRunning) {
    do poll
}
```

- First time poll is called
  - Finds the GroupCoordinator
  - Joining Consumer Group
  - Receiving partition assignment
- Work done in poll loop
  - Usually involves some processing
  - Saving data to a store
  - Don't do high latency work between polls; otherwise the consumer could be deemed dead.
  - Do heavy lifting in a seperate thread



ConsumerRecord

- **org.apache.kafka.clients.consumer.ConsumerRecord <K,V>**
- **K key():** key for record (type K), can be null
- **V value():** record value (type V - String / Integer ..etc)
- **String topic():** Topic where this record came from
- **int partition():** partition number
- **long offset():** long offset in

```
ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000));
  for (ConsumerRecord < Integer, String > record : records) {
     System.out.printf("topic = %s, partition = %d, offset = %d,
            key= %s, value = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
  }
```

Configuring Consumers

```
Properties props = new Properties(); // ** 1 **

...
props.put("session.timeout.ms", 30000); // 30 secs
props.put("max.partition.fetch.bytes", 5 * 1024 * 1024); // 5 M

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

- **max.partition.fetch.bytes**
  - default: 1048576 (1M)
  - Max message size to fetch. Also see **message.max.bytes** broker config
- **session.timeout.ms**
  - default: 30000 (30 secs) in Kafka 2, 45000 (45 secs) in Kafka 3
  - If no heartbeats are not received by this window, consumer will be deemed dead and a partition rebalance will be triggered

Clean Shutdown Of Consumers

- Consumers poll in a tight, infinite loop
- Call **' consumer.wakeup () '** from another thread
- This will cause the poll loop to exit with **' WakeupException '**

```
try {
  while (true) {
   ConsumerRecords < Integer, String > records = consumer.poll(100);
    // handle events
 }
}
catch (WakeupException ex) {
   // no special handling needed, just exit the poll loop
}
finally {
   // close will commit the offsets
   consumer.close();
}
```

Signaling Consumer To Shutdown

- Can be done from another thread or shutdown hook
- **' consumer.wakeup () '** is safe to call from another thread

```
Runtime.getRuntime().addShutdownHook(new Thread() {
   public void run() {
      System.out.println("Starting exit...");
      consumer.wakeup(); // signal poll loop to exit
      try {
           mainThread.join(); // wait for threads to shutdown
      } catch (InterruptedException e) {
           e.printStackTrace();
  }
 }
}
```

Lab : Kafka Producer / Consumer

- **Overview:**
  Use Kafka Java API to write Producer and Consumer
- **Builds on previous labs:**
  - 1-install Kafka
- **Approximate Time:**
  - 30 - 40 mins
- **Instructions:**
  - Please follow: 3.1, 3.2, 3.3
- **To Instructor:**

Producer Send Modes

Producer Send Modes

- **1: Fire and Forget**
  - Send message, doesn't wait for confirmation from Kafka
  - Writes messages to broker in batches (minimize network round-trips)
  - Risk of some messages being lost
  - Default and fastest
- **2: Sync**
  - Send message and wait for confirmation from Kafka
  - Each message is sent out individually
  - Usually lowest throughput
- **3: Async**
  - Registers a callback function while sending
  - Does not wait for confirmation
  - Kafka will call this function with confirmation or exception
  - Higher throughput

Producer Send Mode: Fire and Forget

```
String topic = "test";
Integer key = new Integer(1);
String value = "Hello world";
ProducerRecord < Integer, String > record =
        new ProducerRecord<> (topic, key, value);
producer.send(record); // <-- fire away, not waiting for results
```

- The 'record' is placed in the send buffer
- It will be sent to Kafka in a separate thread
- Send() returns a Java Future object (that we are not checking)
- Some messages can be dropped
- Use cases:
  - Metrics data
  - Low important data

Producer Send Mode: Sync

```
ProducerRecord < Integer, String > record =
    new ProducerRecord<> (topic, key, value);

RecordMetadata recordMetaData = producer.send(record).get(); // <-- wait for result
```

- `get` method is a waiting call
- Inspect `RecordMetaData` for success / error

Producer Send Mode: Async

```java
import org.apache.kafka.clients.producer.Callback;

public class Producer implements Callback { // <-- implement callback

  // .. normal producer code

  // here is the call back function
  @Override
  public void onCompletion(RecordMetadata meta, Exception ex) {
    if (ex != null) // error
      ex.printStackTrace();

    if (meta != null) // success
      System.out.println("send success");
  }
}
...
producer.send(record, this);  // <-- supply callback
```

- Kafka will callback with meta or exception

Producer Send Modes

```
// fire and forget
producer.send(record);

// sync
producer.send(record).get();

// async
producer.send(record, this); // producer has callback implemented
```

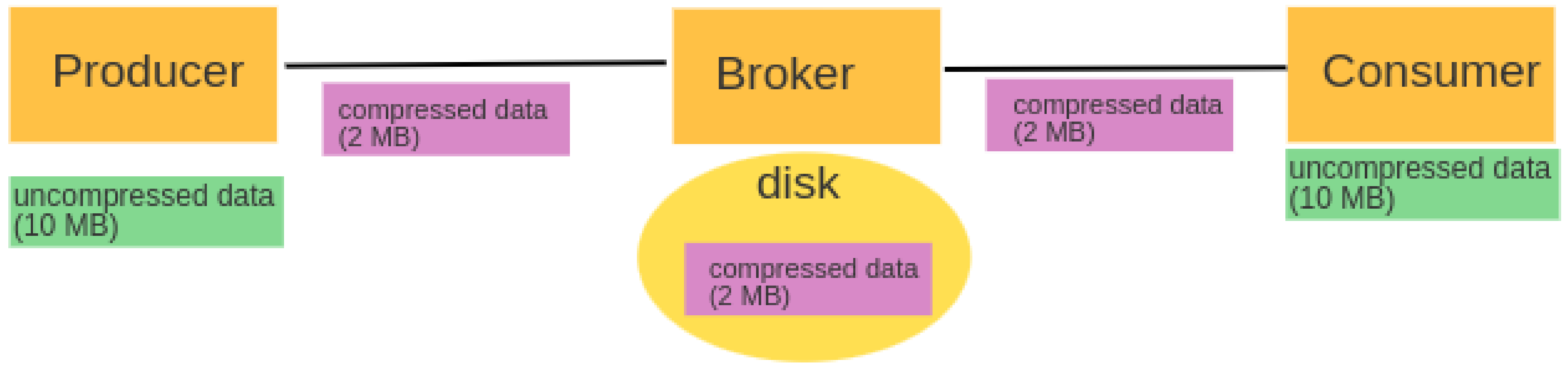
Lab 4.1: Kafka Producer Benchmark

- **Overview:** Try different send methods in Producer
- **Builds on previous labs:** lab 3
- **Approximate Time:** 20 - 30 mins
- **Instructions:**
  - Please follow: lab 4.1
- **To Instructor:**

Message Compression

Compression

- Benefits of compression
  - Reduces the data size goes on network --> faster throughput
  - Reduces data footprint on disk --> less data to write to disk -> faster
- Compression is performed on a batch
  - Larger batch size -> better compression
- Kafka API will automatically
  - On Producer: Compress messages before sending on wire
  - On Broker: Messages remain in compressed state in partitions
  - On Consumer: De-compress messages on the fly



Enabling Compression

```java
import java.util.Properties
import org.apache.kafka.clients.producer.KafkaProducer;

props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("client.id", "CompressedProducer");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

// viable codecs : "none"  or "uncompressed" (default), "snappy", "gzip", "lz4", "zstd"
// https://kafka.apache.org/documentation/#configuration
// https://kafka.apache.org/documentation/#brokerconfigs_compression.type

props.put("compression.type", "snappy"); // <-- **enable compression**

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

- Compression is enabled on **Producer** side. No need to setup on Consumer side. Consumers can automatically figure out the compression scheme

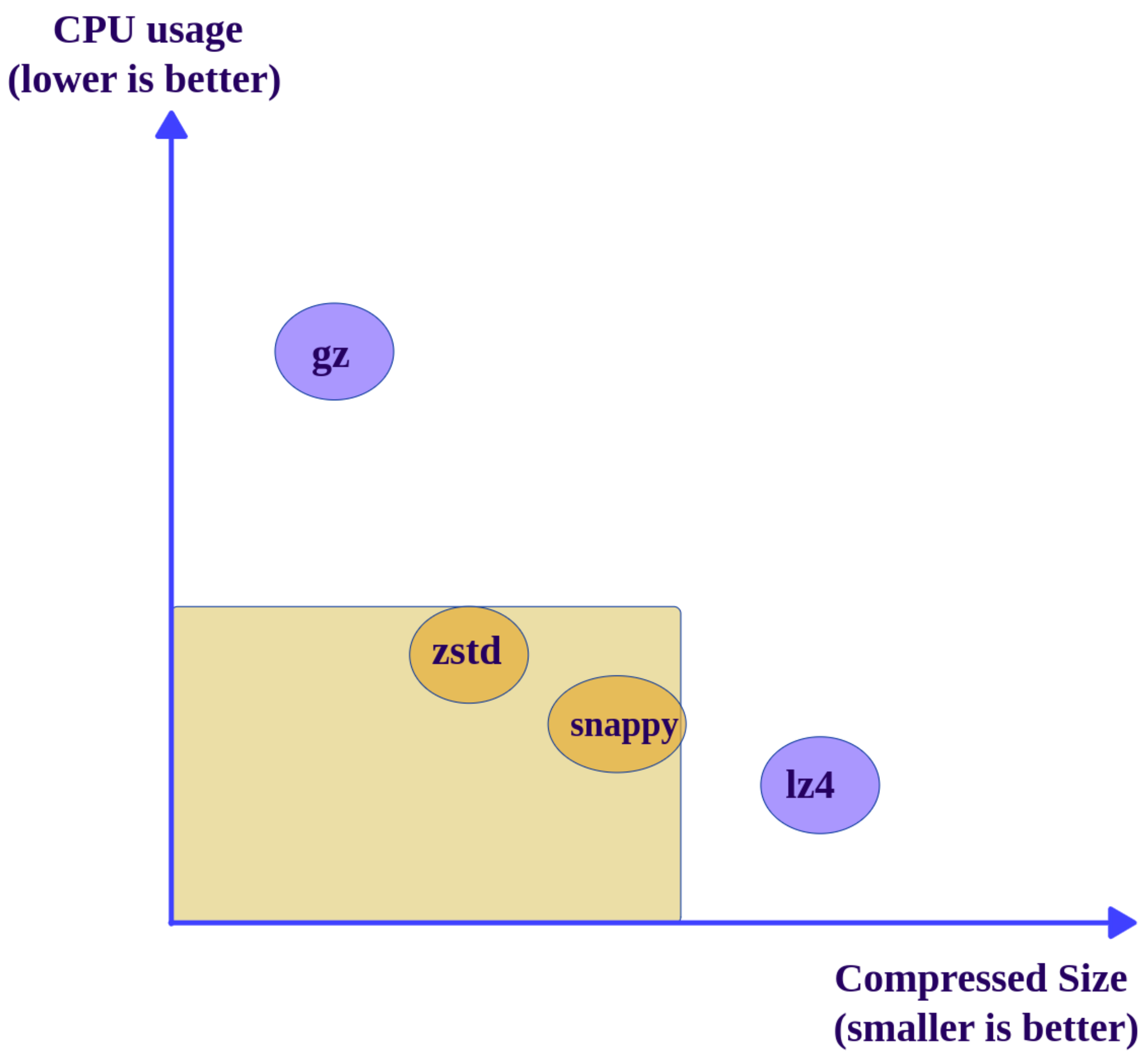
Compression

- **Compression will slightly increase CPU usage**
  - How ever, this is well worth the trade-off, as CPUs are very fast and we usually have plenty of CPU power to spare.
  - Plus modern CPUs have compression algorithms built-in silicone
- Here are some benchmark stats from Message compression in Apache Kafka
- We can see Snappy (from Google) and zstd (from Facebook) giving a good balance of CPU usage, compression ratio, speed and network utilization

Metrics	Uncompressed	Gzip	Snappy	lz4	Zstd
Avg latency (ms)	65	10.4	10.1	9.2	10.7
Disk space (mb)	10	0.92	2.2	2.8	1.5
Effective compression ratio	1	0.09	0.21	0.28	0.15
Process CPU usage %	2.35	11.46	7.25	5.89	8.93

Compression

- Supported compression codecs
  - Gzip, Snappy, LZ4, Zstd
- **Snappy** (from Google) is a pretty good light weight compressor; Easy on GPU and produces medium level compresison
- Current favorite is **Zstd** (Facebook) - Good speed and produces compact size
- Configured via Producer property **`compression.type`** (see next slide for code sample)
- Reference



Compression Data Formats

- **Text data (XML, JSON, CSV) formats are very good candidates for compression**. Since they have lot of repeating elements, they compress well.
- JSON data

```
{"id" : 1, "first_name" : "John", "last_name" : "Smith", "age" : 34, "email" : "john@me.com"}
```

- XML data

```
<CATALOG>
    <CD>
        <TITLE>Empire Burlesque</TITLE>
        <ARTIST>Bob Dylan</ARTIST>
        <YEAR>1985</YEAR>
    </CD>
</CATALOG>
```

- Also **server logs** are good candidates, as they have well defined structure

```
1.1.1.1 - [2020-09-01::19:12:06 +0000] "GET /index.html HTTP/1.1" 200  532"
2.2.2.2 - [2020-09-01::19:12:46 +0000] "GET /contact.html HTTP/1.1" 200  702"
```

- **Binary data formats won't compress well**. E.g. images, base64 encoded strings. Don't enable compression.
- Reference: Message compression in Apache Kafka

Lab: Compression

- **Overview:**
  - Try different compression codecs in Producer
- **Approximate Time:**
  - 15 mins
- **Instructions:**
  - lab 4.2
- **To Instructor:**
  - Demo this lab

Lab: Compression Benchmark

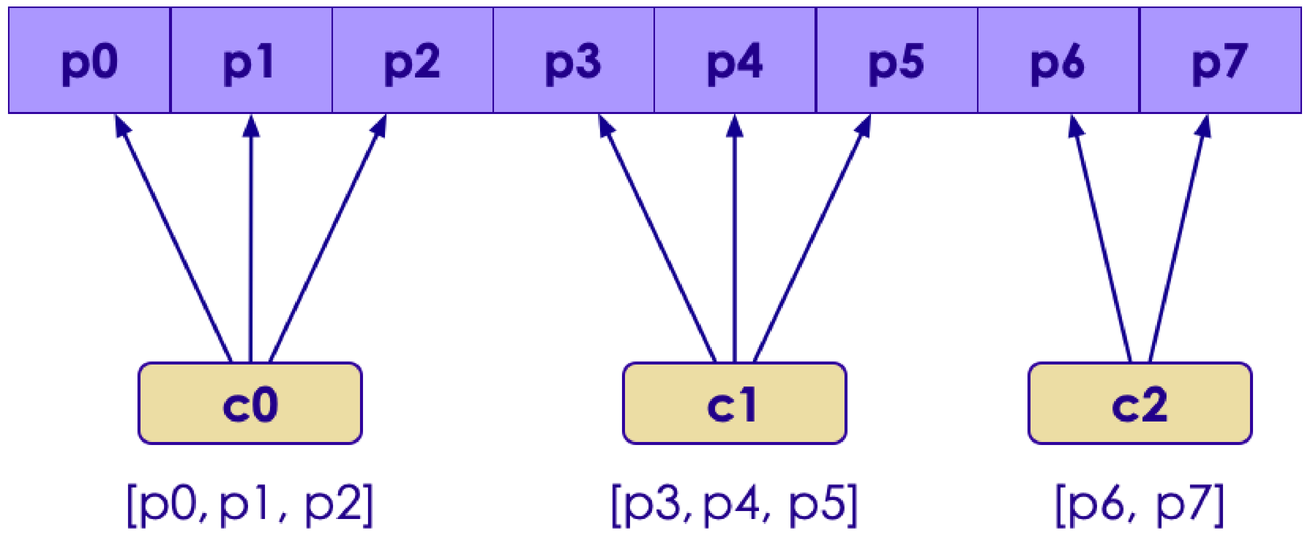
- **Overview:** Try different compression codecs in Producer
- **Builds on previous labs:** lab 4.2
- **Approximate Time:** 20 - 30 mins
- **Instructions:**
    - Lab 4.3
- **To Instructor:**

Advanced Consumer Properties

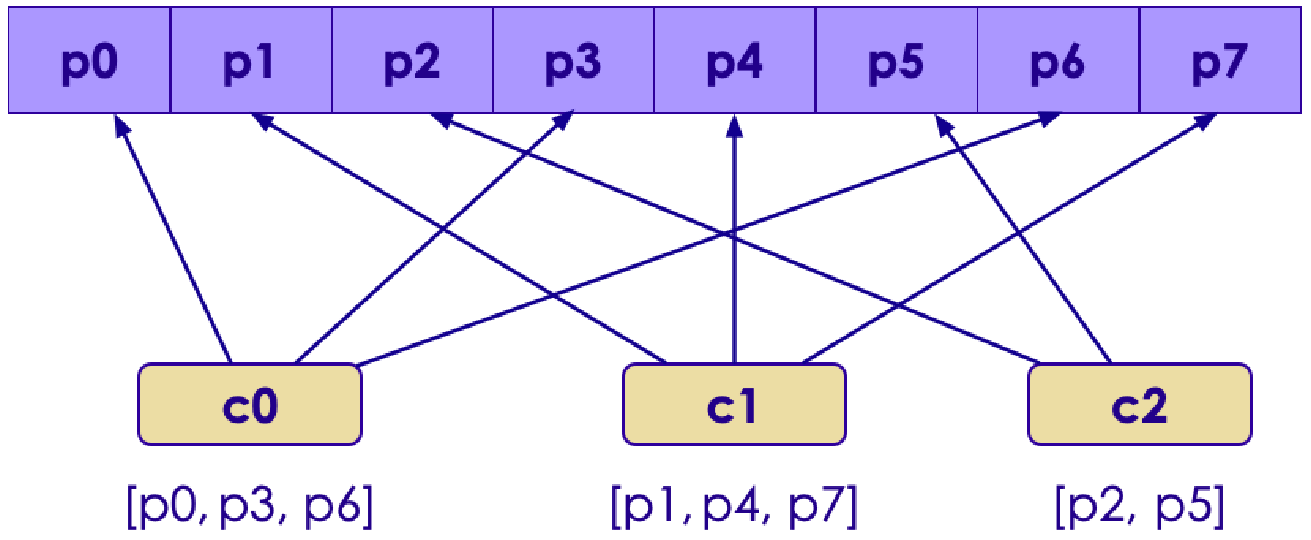
Property	Description	Default Value
fetch.min.bytes	Min. data to fetch.	
fetch.max.wait.ms	Max. wait time	500 ms
Session.timeout.ms	Time after which consumer is deemed dead if it doesn't contact broker	30 seconds
Heartbeat.interval.ms	Intervals in which heartbeats are sent	1 second
Auto.offset.reset	Offset value to use when no committed offset exists	"latest"
Partition.assignment.strategy	Assign partitions by range or round-robin (next slide)	RangeAssignor
Max.poll.records	Max. number of records poll can return	

Partition Assignment Schemes

- Range assignment (default)
  - `partition.assignment.strategy = RangeAssignor`

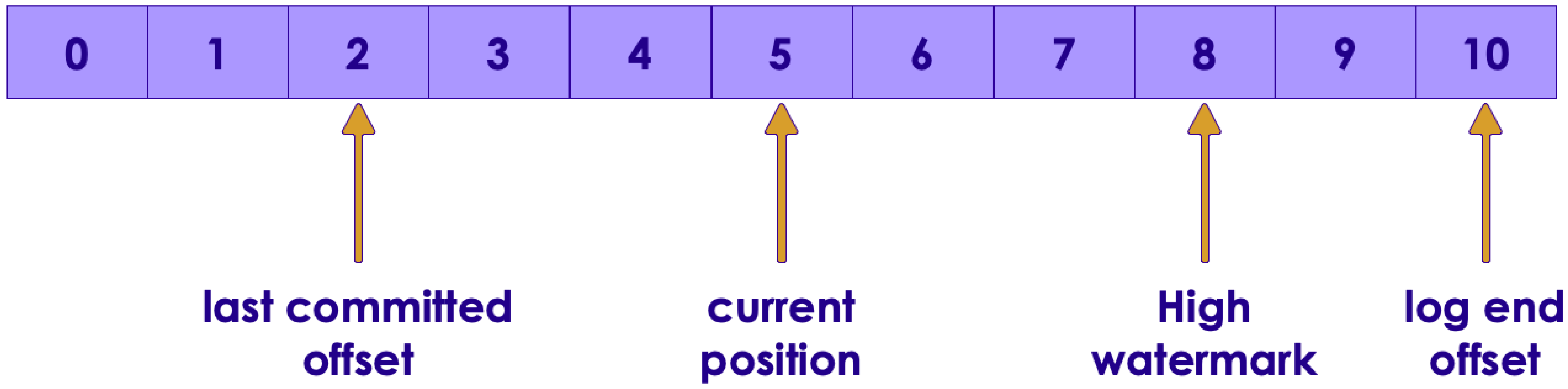


- Random assignment
  - `partition.assignment.strategy = RoundRobinAssignor`



Commits And Offsets

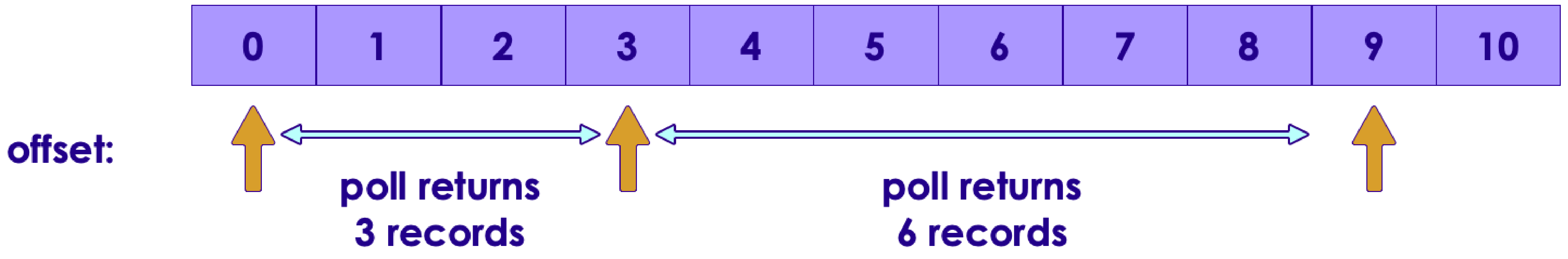
Understanding Offsets



- **Last committed offset:** from client commit (auto or manual)
- **Current position:** where client is reading from
- **High watermark:** latest replicated offset.
- **Log End:** offset of last message
- Consumers can only read up to **high watermark**
  - Otherwise consumers will read un-replicated data, might result in data loss

Commits And Offsets

- Kafka does not track 'read acknowledgements' of messages like other JMS systems
- It tracks the consumer progress using an **offset**
- When a Consumer calls `Poll()` it gets **new records** from the offset
- Offsets are stored in Kafka (in a special topic : **__ consumer_offsets)**
  - Used to be stored in ZK, but now stored in Kafka for performance reasons
- When a consumer crashes..
  - Partitions of that consumer are assigned to another consumer
  - New 'partition owner' consumer resumes from current offset



Offset Management

- **auto commit**: Offsets can be 'moved' automatically by consumer API
  - Convenient
  - But does not give full control to developer
- **Manual commit**
  - Clients handle offset
  - Complete control

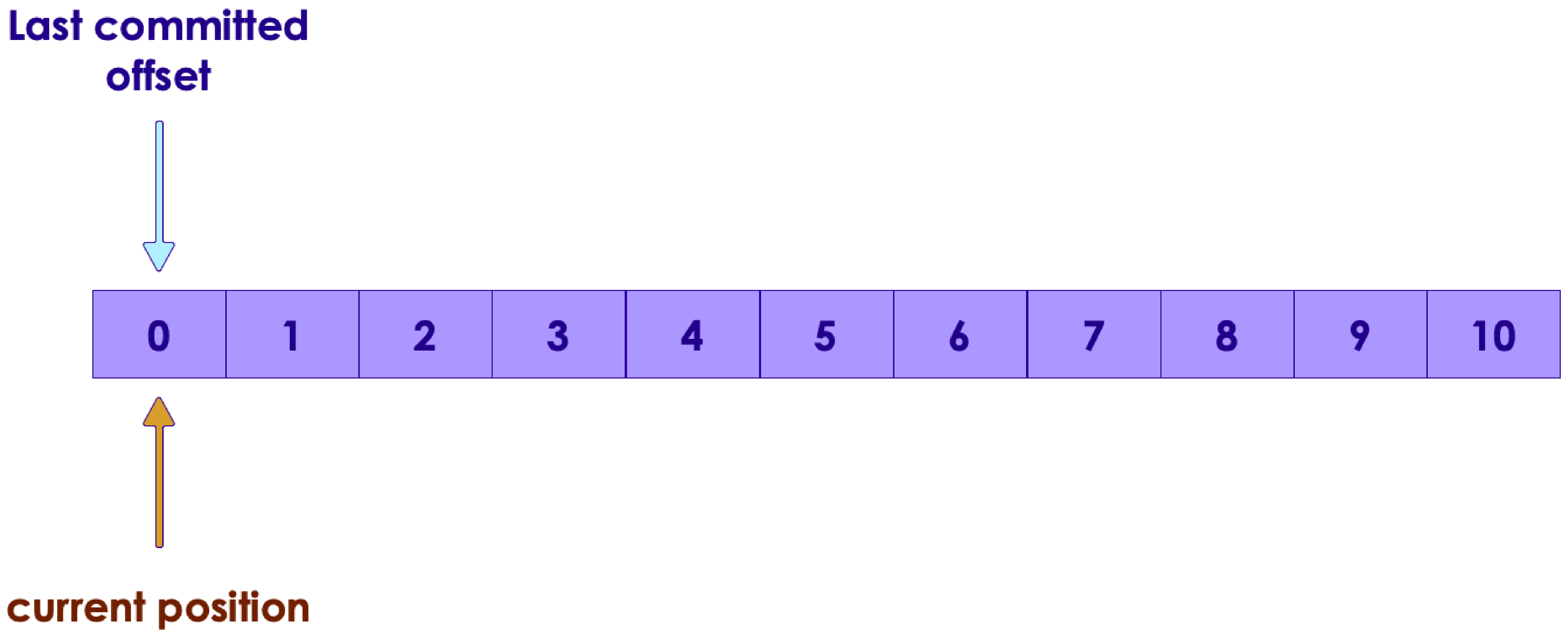
Auto Commit

- We need to set two properties
  - **`enable.auto.commit=true`** so the client will save the offset when poll()
  - Frequency controlled by **`auto.commit.interval.ms`** (default 5 secs)
- auto.commit is enabled by default
- During each poll, consumer checks if the **auto.commit.interval.ms** interval has expired; if yes, commit the **latest offset** returned by the **last poll()**
- Commits are saved to a special Kafka topic called **_consumer_offsets**

```java
props.setProperty ("enable.auto.commit", "true");
props.setProperty ("auto.commit.interval.ms", "5000");
props.setProperty ("session.timeout.ms", "30000"); // set higher for high latency applications

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records)
        // process records
    }
    // make sure all records are processed before next poll
}
```

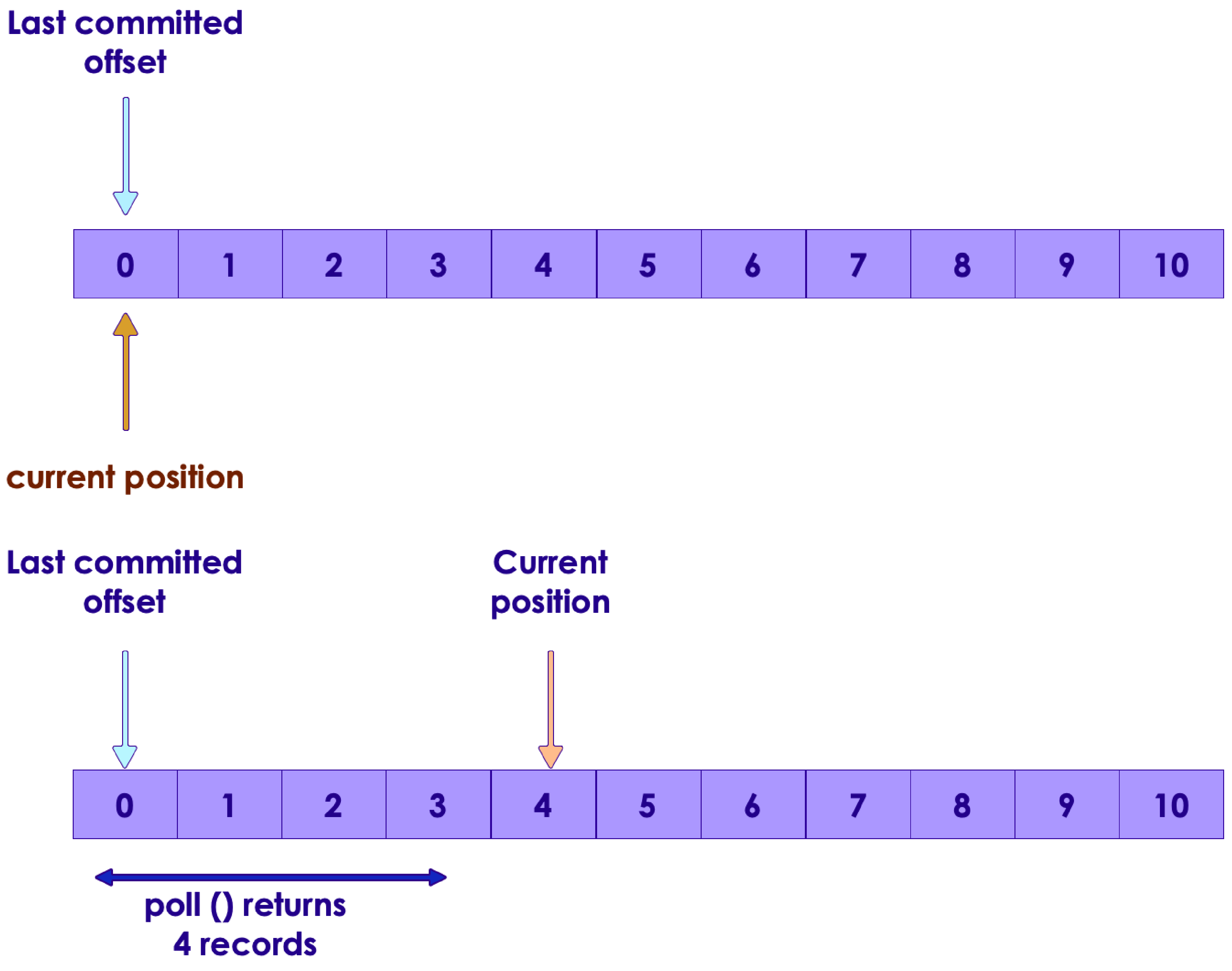
Auto Commit

- Remember this as we walk through these scenarios:
  - **current position** : Pointer for **current consumer**
  - **committed position**: Pointer for **replacement consumer**
- Both pointers are at 0 at the beginning



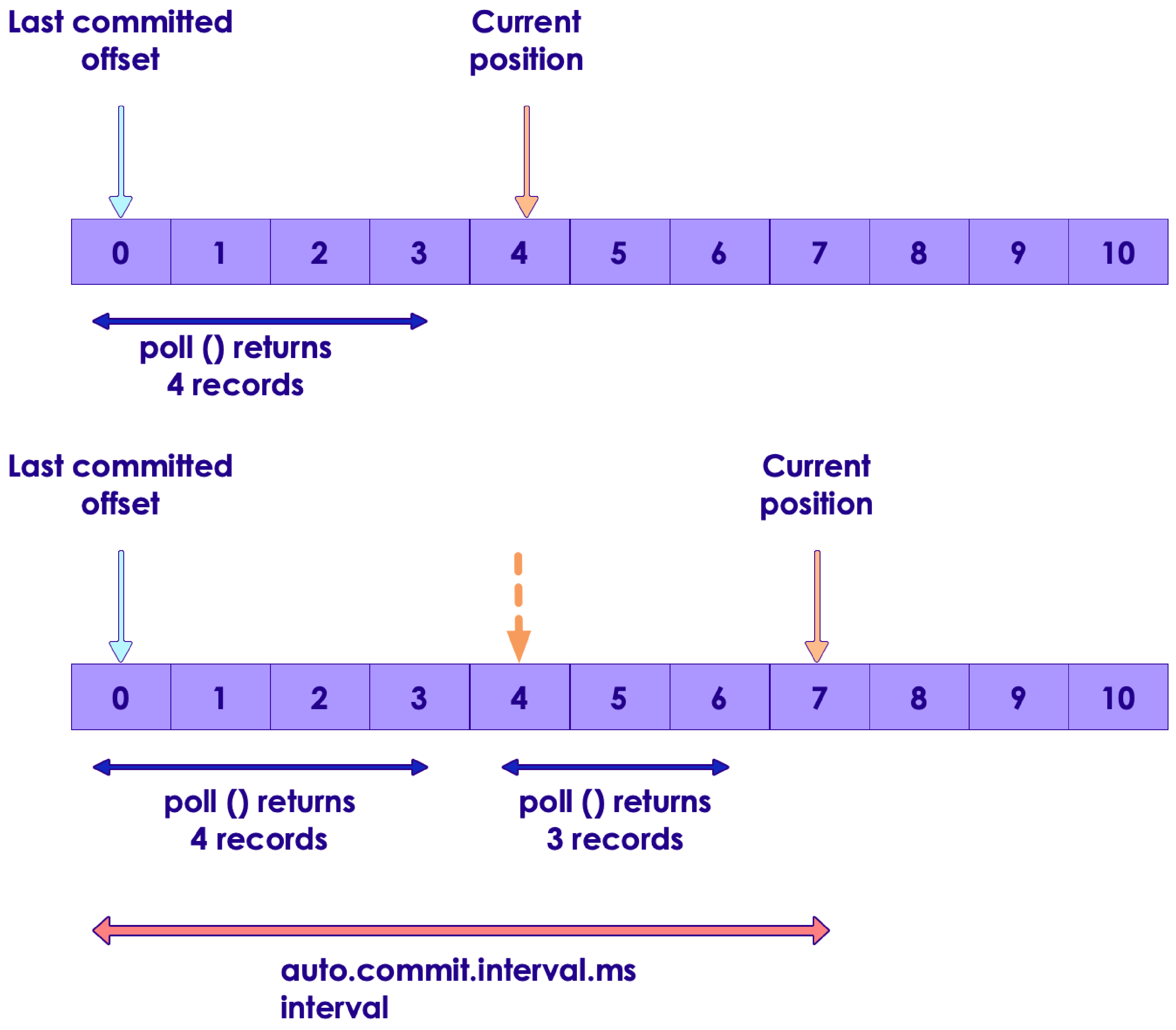
Auto Commit

- First poll gets 4 records.
  Current position = 0 + 4 = 4
- **Committed offset still at 0**
- Top diagram (previous state), bottom diagram (current state)



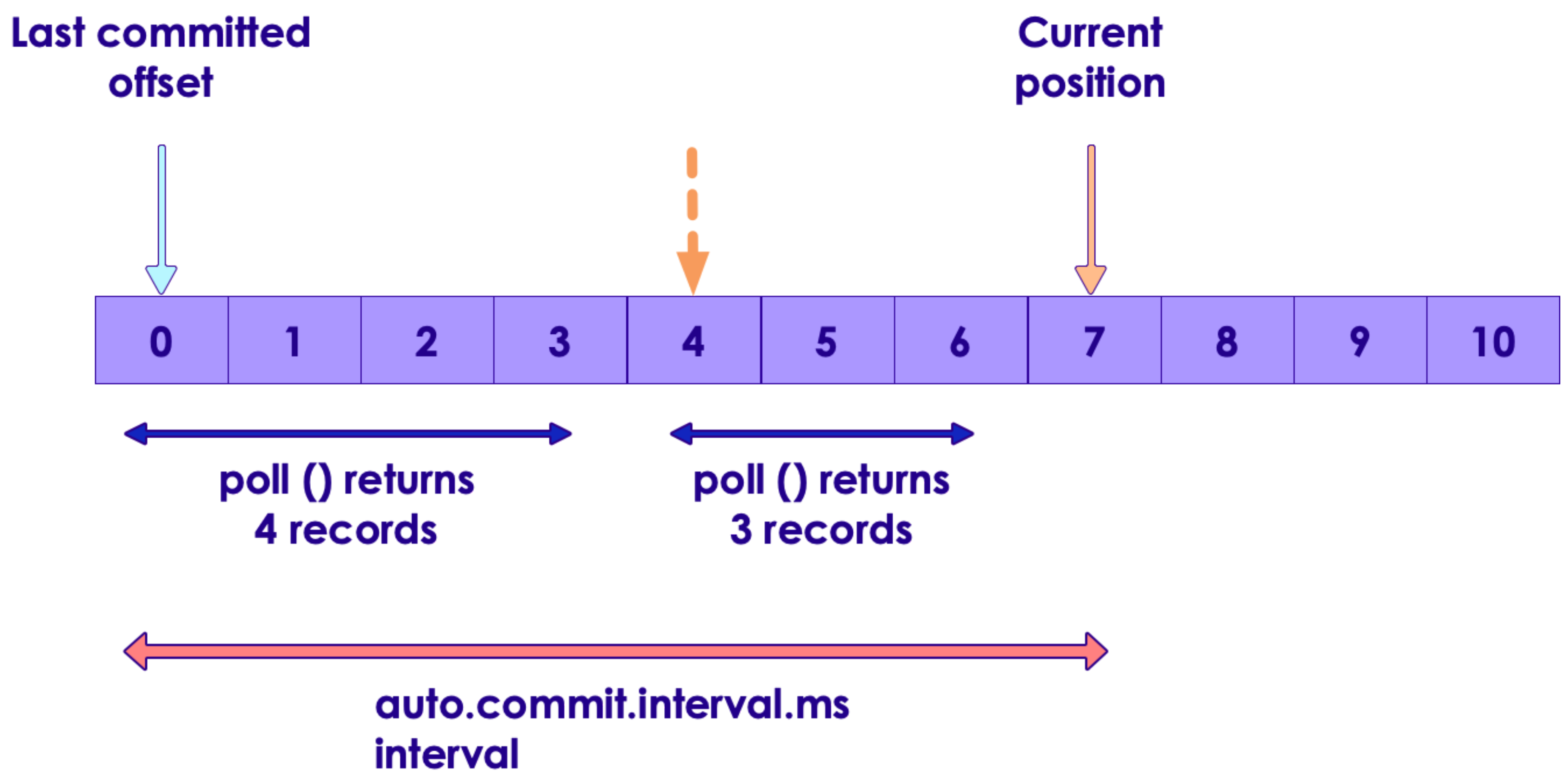
Auto Commit

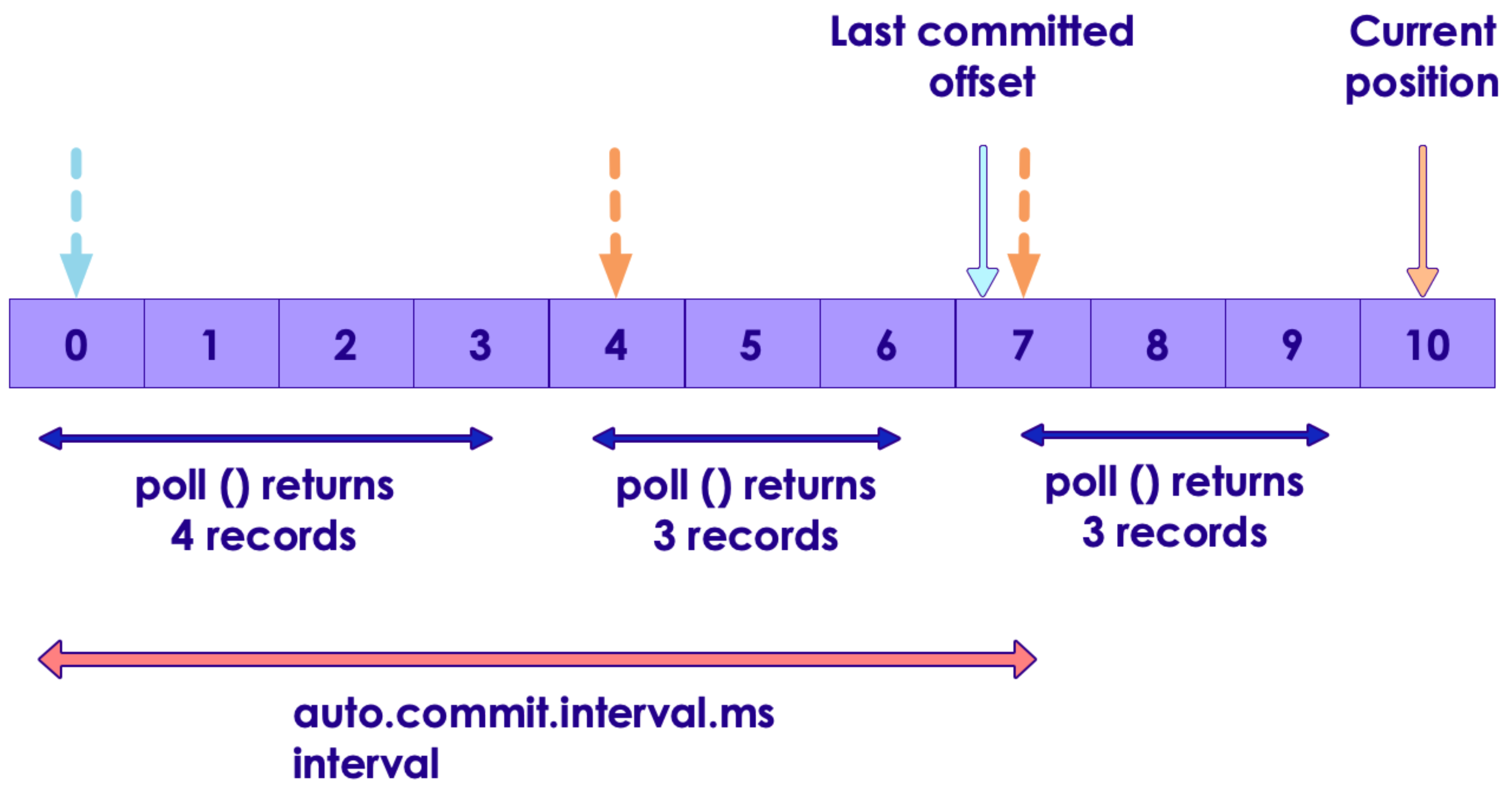
- Second poll gets 3 records
- **Current position = 4 + 3 = 7**
- **Committed offset still at 0**. Because elapsed time still under 'auto.commit.interval.ms' window
- Top diagram (previous state), bottom diagram (current state)



Auto Commit

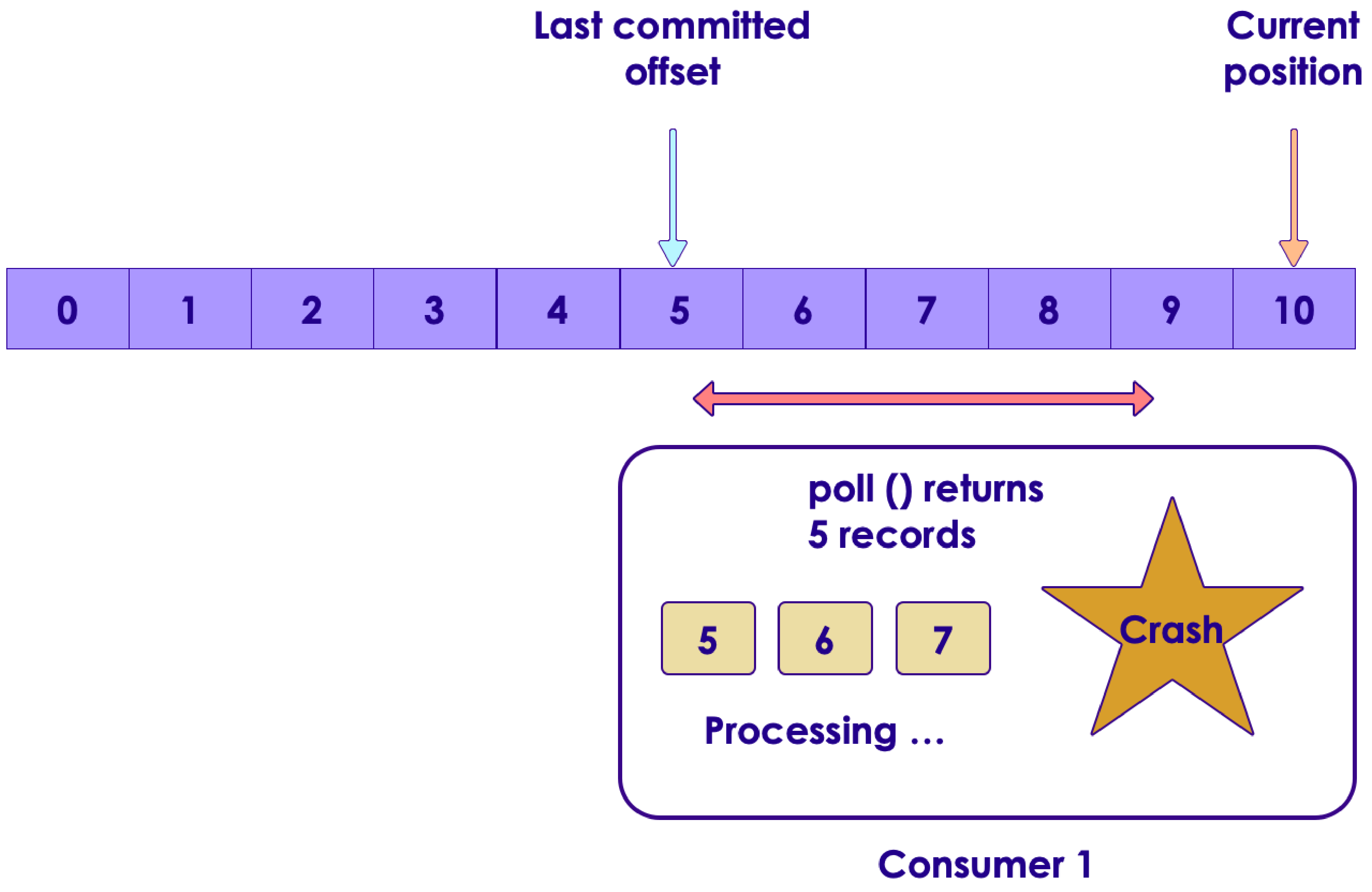
- Third poll gets 3 records
- **Current position = 7 + 3 = 10**
- **Committed offset is updated to 7 (offset from last poll)**. Because elapsed time has surpassed 'auto.commit.interval.ms' window
- Top diagram (previous state), bottom diagram (current state)


Last committed offset
Current position
0
1
2
3
4
5
6
7
8
9
10
poll () returns 4 records
poll () returns 3 records
auto.commit.interval.ms interval


Last committed offset
Current position
0
1
2
3
4
5
6
7
8
9
10
poll () returns 4 records
poll () returns 3 records
poll () returns 3 records
auto.commit.interval.ms interval

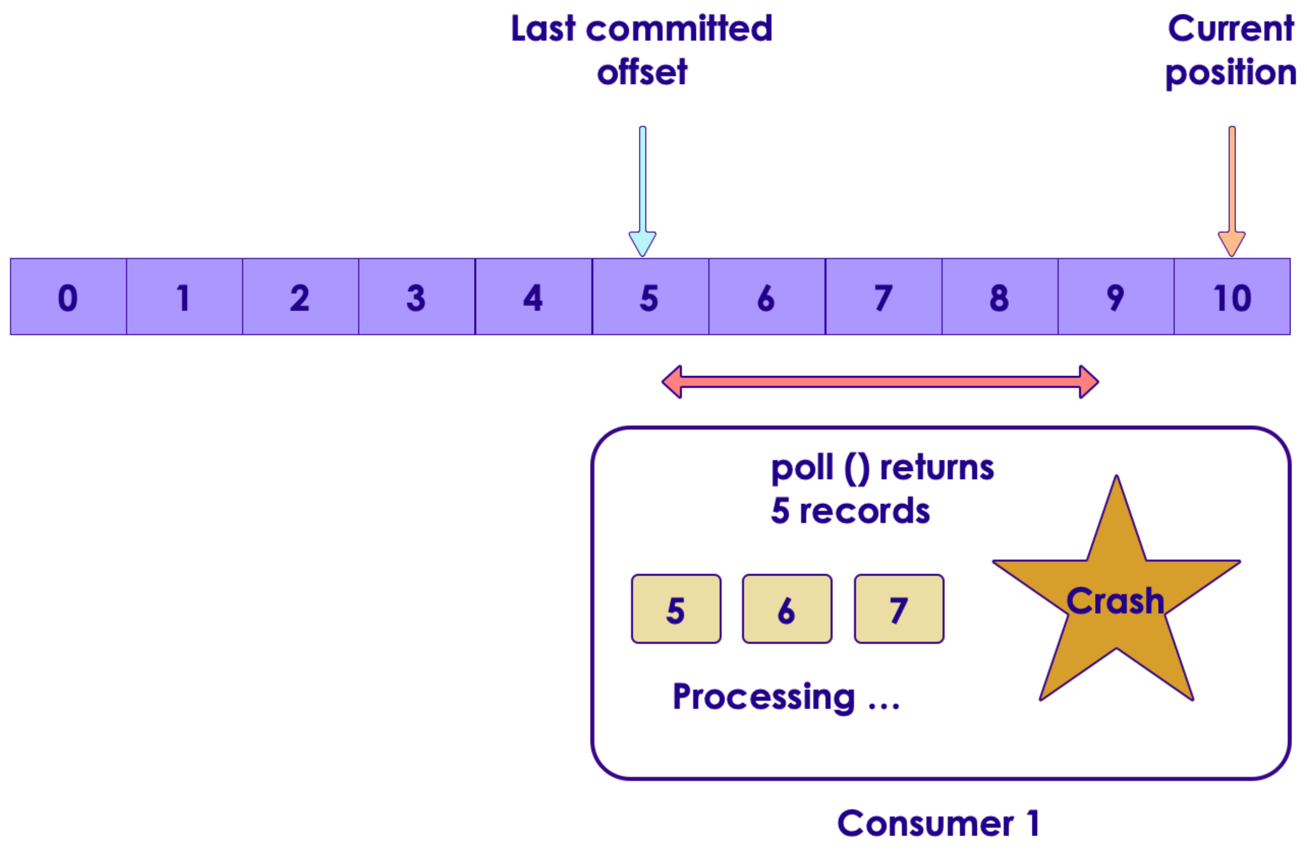
Auto Commit & Duplicate Events

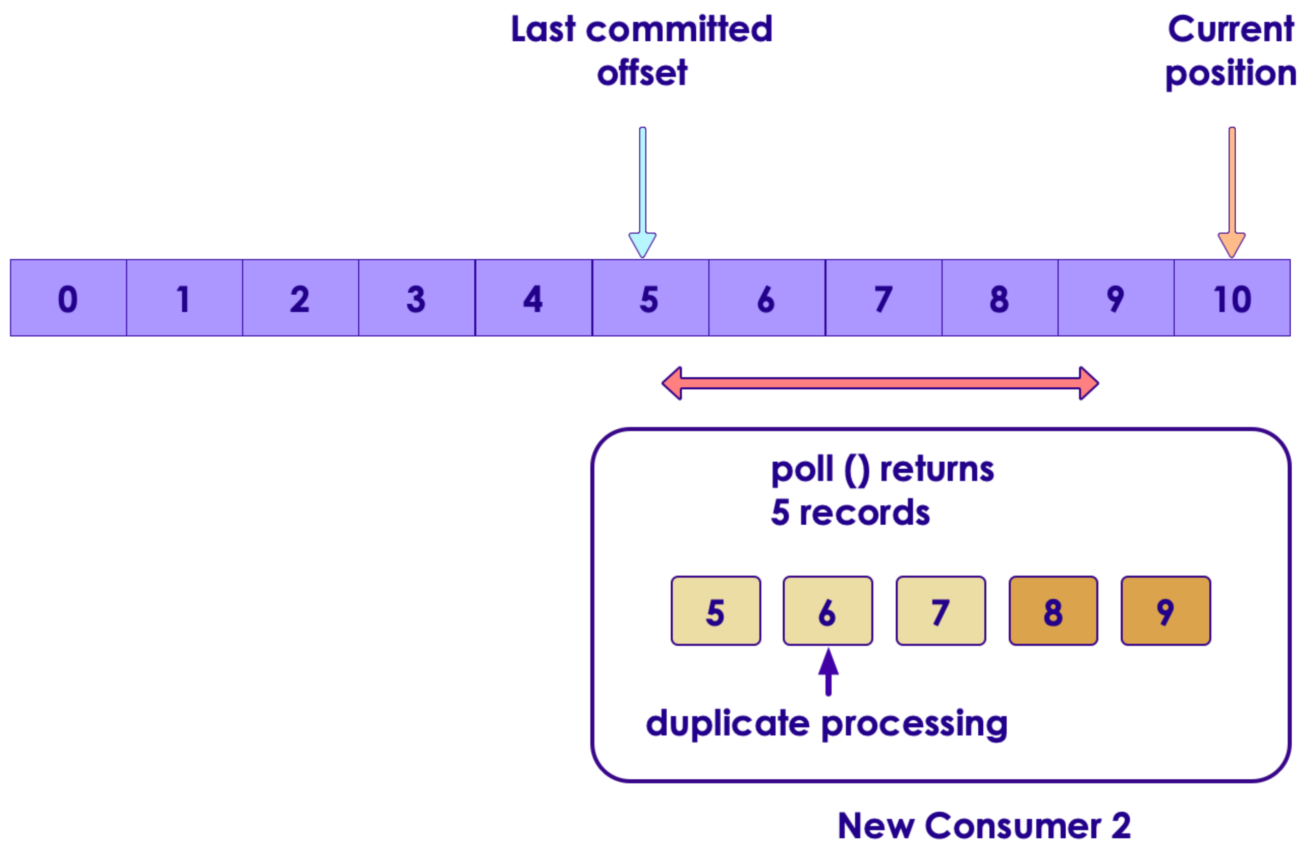
- Consumer 1 reads records, but crashes in the middle of processing (after processing messages 5, 6 and 7)



Auto Commit & Duplicate Events

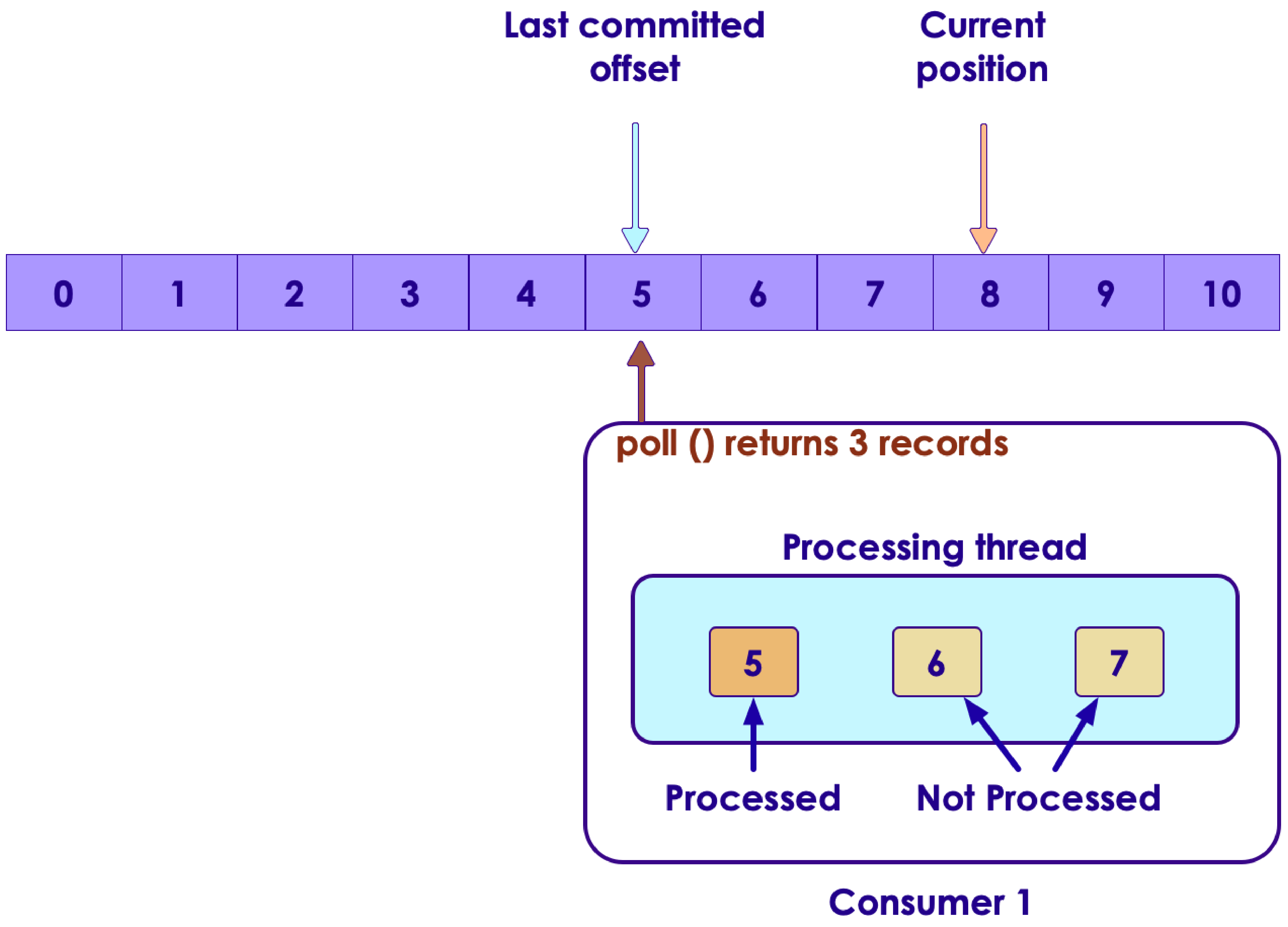
- Replacement consumer2 starts reading from **`committed offset`**
- And processes messages 5, 6 and 7 again. This is **duplicate processing**
- **At-least-once** semantics





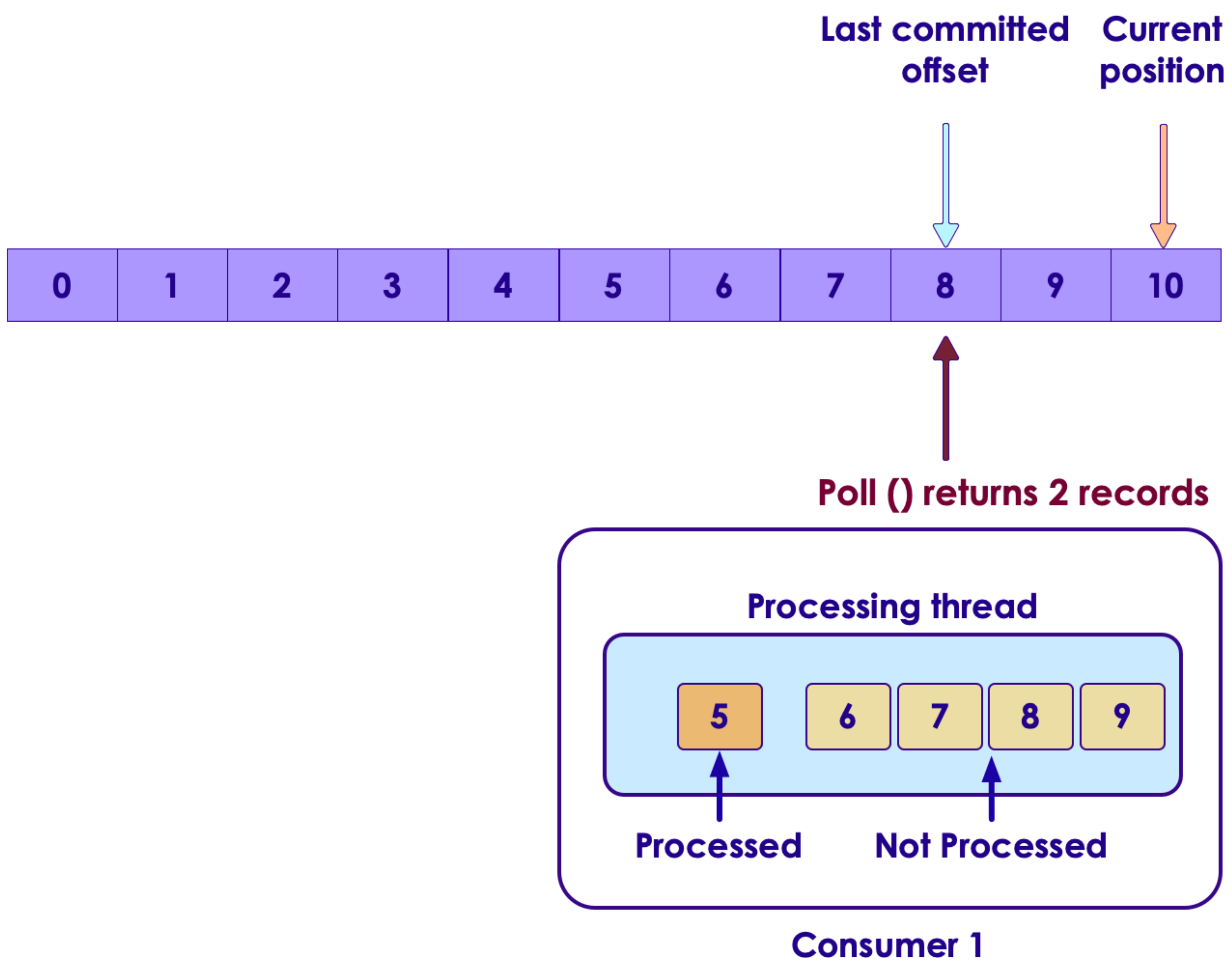
Auto Commit & Skipped Events

- Consumer1 polls the events and hands them off to another thread for processing



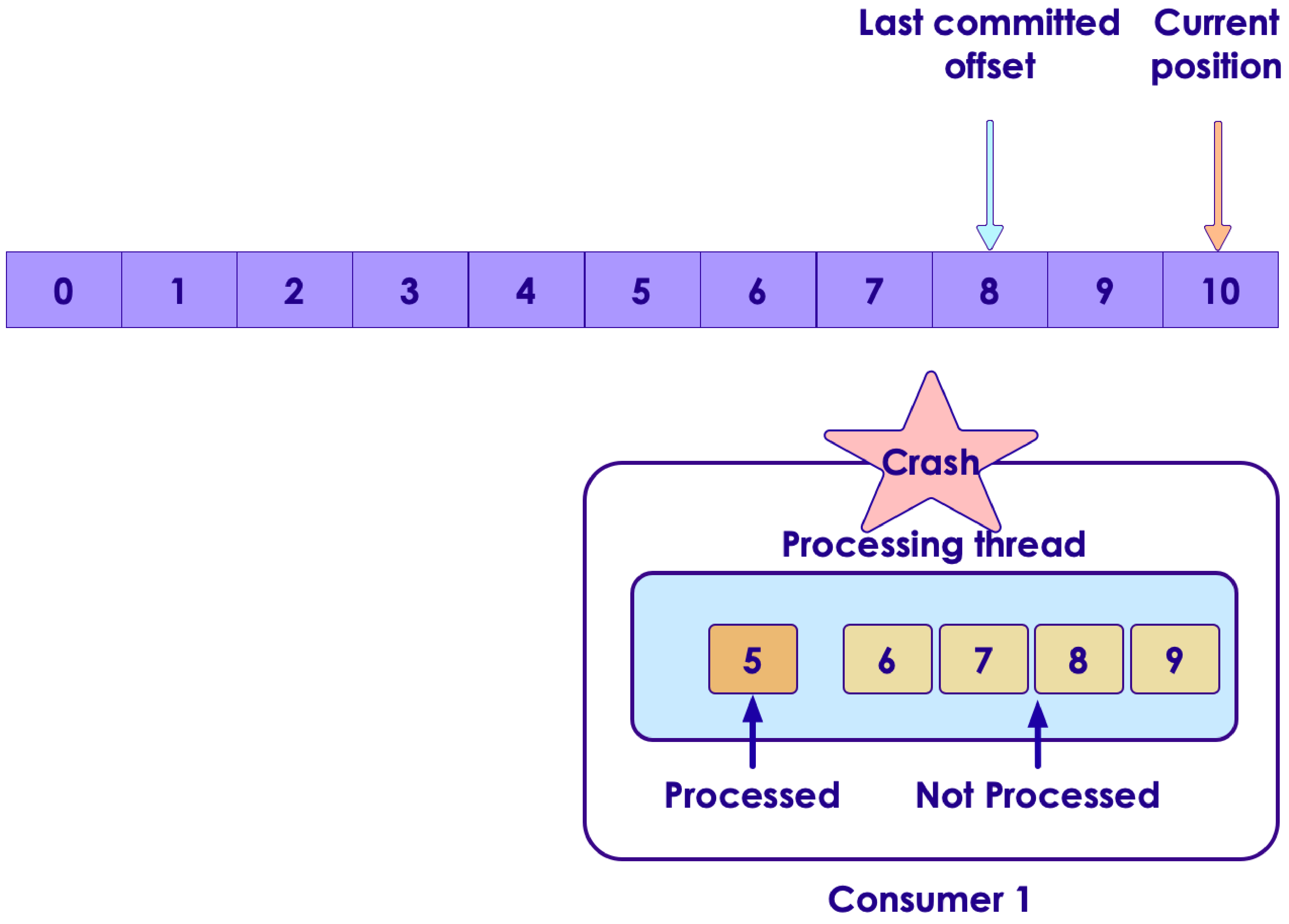
Auto Commit & Skipped Events

- Before 'processing thread' has a chance to finish processing all messages, main polling thread initiates another poll
- This commits the offset to 8



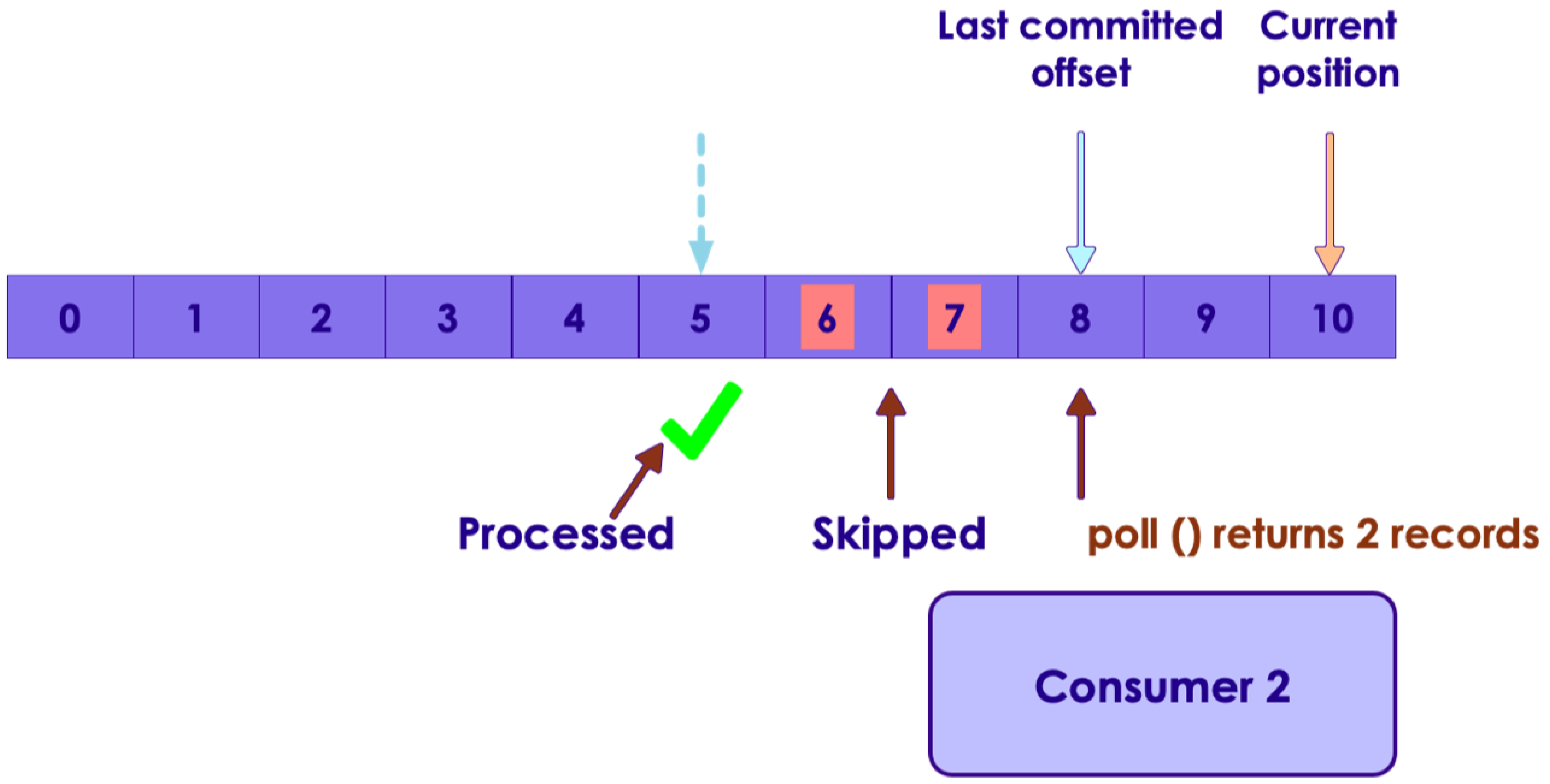
Auto Commit & Skipped Events

- Consumer1 crashes



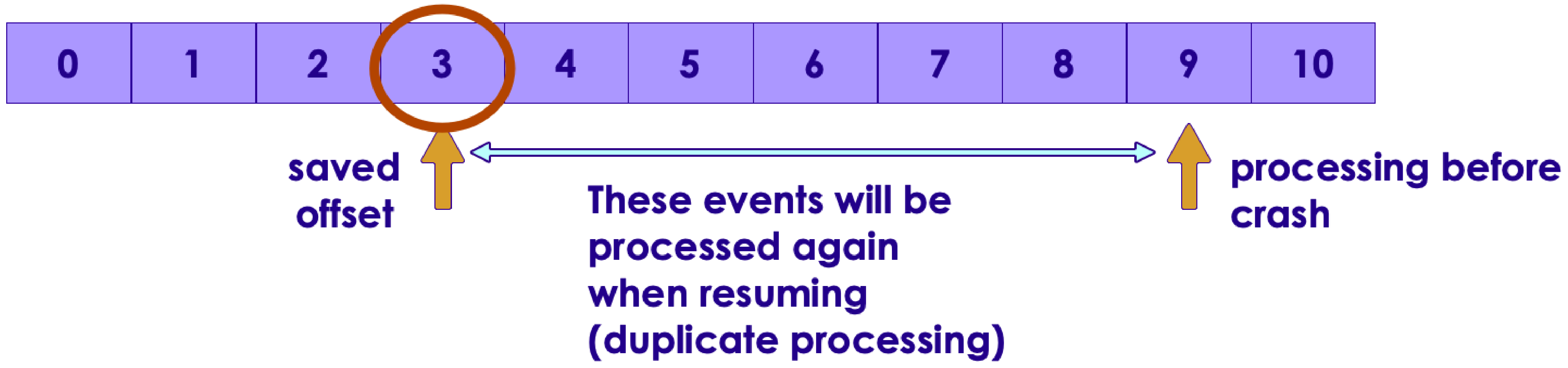
Auto Commit & Skipped Events

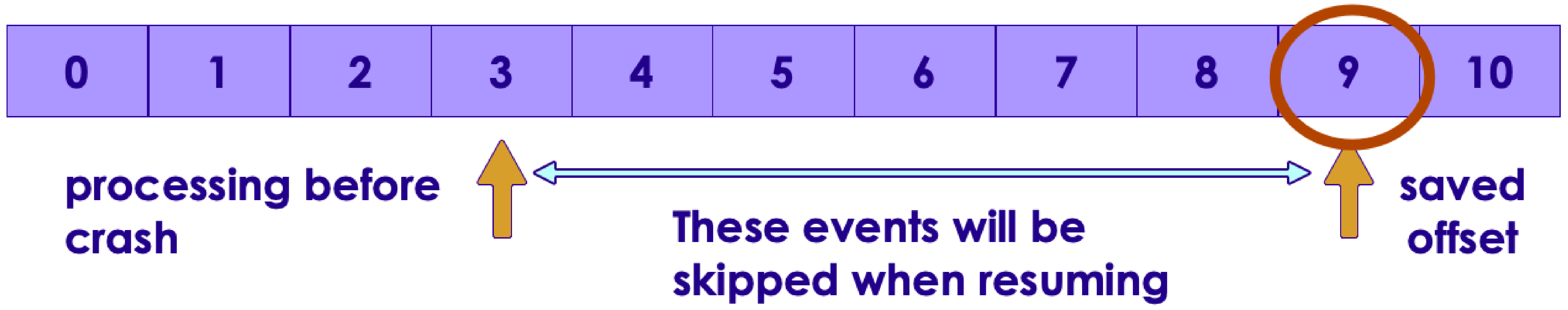
- A new consumer2 starts processing
- But when it polls, it is retrieving messages from last-committed-offset (8 onwards)
- Messages 6 & 7 are skipped



Duplicate Events vs. Skipped Events

- Processed but Offset not updated -> Duplicate processing
- Offset updated but not processed -> Skipped events


0
1
2
3
4
5
6
7
8
9
10
saved offset
These events will be processed again when resuming (duplicate processing)
processing before crash



Auto Commit Best Practices

- Auto commit can lead to duplicate processing or skipped events
- Dealing with **duplicate events**
  - Make sure processing is **'idempotent'** (duplicate processing does not have any side effects)
  - Example, **insert** vs **upsert**
- Dealing with **skipped events**
  - Before calling poll() make sure all events returned from previous poll are processed

Manual Commit - Example 1

- To do manual-commit, do the following
  - Set **`enable.auto.commit = false`**
  - call **`consumer.commitSync()`** or **`consumer.commitASync()`**
- Here we are marking **all received records as committed**

```
Properties props = new Properties();
...
props.put("enable.auto.commit", "false"); // must disable
KafkaConsumer < Integer, String > consumer = new KafkaConsumer<>(props);

while (true) {
  ConsumerRecords < Integer, String > records = consumer.poll(100);
  for (ConsumerRecord < Integer, String > record : records)
  {
     System.out.println(String.format(
    "topic = %s, partition = %s, offset = %d",
    record.topic(), record.partition(), record.offset()));
  }
  try {
    consumer.commitSync(); // <--- 1 - wait for commit to be done
    consumer.commitASync(); // <--- or  ** 2 ** - send a commit message and move on

  } catch (CommitFailedException e) {
    e.printStackTrace();
  }
}
```

Manual Commit

- **`commitSync()`** will wait for confirmation
    - Might slow down processing
- **`commitAsync()`** is 'fire and forget'
    - Commits can be lost
    - But the next calls can catchup
- Both will commit the **latest offset** returned by **last poll** ()
- To explicitly set an offset, supply offset to commitSync() or commitAsync() calls
    - (see documentation)

Manual Commit - Example 2

- Here we are going to commit **specific records** that were processed
- Both commitSync and commitAsync can take a Map of partitions & offsets
- We are committing every 100 records to minimize the overhead

```
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
Map < TopicPartition, OffsetAndMetadata > currentOffsets = new HashMap<>();
int count = 0;
while (true) {
   ConsumerRecords < Integer, String > records = consumer.poll(100);
   for (ConsumerRecord < Integer, String > record : records)
   {
      count++;
      // process message and update offset map
      currentOffsets.put(
          new TopicPartition(record.topic(), record.partition()),
          new OffsetAndMetadata(record.offset()+1, "no metadata"));

      if (count % 100 == 0) { // commit every 100 records
          consumer.commitSync(currentOffsets, null);
          currentOffsets.clear();
    }
  }
}
```

- References
  - org.apache.kafka.clients.consumer.KafkaConsumer
  - org.apache.kafka.common.TopicPartition

Manual Commit - Example 3

- Here we have finer control on which records to mark as committed
- We commit offset after we finish handling the records in each partition.

```
try {
    while(running) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(500));

        // loop over partitions returned
        for (TopicPartition partition : records.partitions()) {
            // grab records for this partition
            List<ConsumerRecord<String, String>> partitionRecords = records.records(partition);

            // go over these records
            for (ConsumerRecord<String, String> record : partitionRecords) {
                System.out.println(record.offset() + ": " + record.value());
            }
            long lastOffset = partitionRecords.get(partitionRecords.size() - 1).offset();

            // Mark records from this partition as done
            // The committed offset should always be the offset of the next message
            // that your application will read. Thus, when calling commitSync(offsets)
            // you should add one to the offset of the last message processed.
            consumer.commitSync(Collections.singletonMap(partition,
                    new OffsetAndMetadata(lastOffset + 1)));
        }
    }
} finally {
consumer.close();
}
```

Consumer Configurations

- We are going to see a few code samples of different consumers
- **At-most-once** consumer
  - OK to loose messages, but no duplicate processing
- **At-least-once** consumer
  - Do not loose messages, duplicate processing is OK
- **Exactly-once** consumer
  - Process only once
  - No duplicate events
  - No dropped events
- References:
  - Meaning of at-least once, at-most once and exactly-once delivery
  - Kafka Clients (At-Most-Once, At-Least-Once, Exactly-Once, and Avro Client)

At-least-once Consumer With Auto Commit

- Simplest implementation, we let Kafka keep track of offsets
- We finish processing all records before next poll
- Analyze what would happen if the consumer crashes
    - Replacement consumer starts up
    - It resumes from 'committed offset' (this is lagging from consumer offset)
    - It will receive duplicate messages

```
props.setProperty ("enable.auto.commit", "true");
props.setProperty ("auto.commit.interval.ms", "5000");
props.setProperty ("session.timeout.ms", "30000"); // set higher for high latency applications

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records)
        // process records
    }
    // all records are processed before next poll
}
```

At-least-once Consumer with Manual Commit

- Here auto commit is disabled, we are committing manually at the end of processing all records
- What would happen if consumer crashed at **point A** (just before committing)
    - Replacement consumer starts up
    - It resumes from 'committed offset' (this is lagging from consumer offset)
    - It will receive duplicate messages

```
props.setProperty ("enable.auto.commit", "false");

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records)
        // process records
    }
    // ** A **
    // commit all records are processed
    consumer.commitSync()
}
```

At-most-once Consumer

- Here we keep polling Kafka and building a 'batch' in consumer
- And saving the batch once it exceeds 1000 records
- Every time we poll, Kafka will commit the offsets
    - The small value of `auto.commit.interval.ms` will ensure the commit happens
- Now think about what happens when the consumer crashes before saving database (next slide)

```
props.put("enable.auto.commit", "true");  // auto commit on
props.put("auto.commit.interval.ms", "100"); //  keep it small

// internal buffer
final int minBatchSize = 1000;
List<ConsumerRecord<String, String>> buffer = new ArrayList<>();
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    // add records to buffer
    for (ConsumerRecord<String, String> record : records) {
        buffer.add(record);
    }
    if (buffer.size() >= minBatchSize) {
        insertIntoDb(buffer);
        buffer.clear();
    }
}
```

At-most-once Consumer

- Here is the **at-most-once scenario**
- As we keep polling, Kafka will auto commit the offsets
- Let's say consumer crashed before saving to database
- Replacement consumer comes up, it resumes from 'last-commited-offset' - which is moved to current-pointer
- So a few messages are skipped

Exactly Once Processing in Kafka

- Exactly-once is the coveted scenario
- In Kafka, it is **possible to do exactly-once given a few restrictions**
- We need to implement changes **in Broker, Producer and Consumer**
- Kafka Streams API implements Exactly Once Processing
- References
    - Exactly-Once Semantics Are Possible: Here's How Kafka Does It
    - Exactly Once Processing in Kafka with Java
    - KafkaProducer API

Exactly Once - Broker Setup

- **`replication.factor = 3`**
  - 3 Replicas has been proven at scale to provide very durable data safety
  - This is why Hadoop/HDFS uses 3 replicas for durable data
- **`min.insync.replicas = 2`**
  - This specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful, When a producer sets acks to "all" (or "-1")

Exactly Once - Producer Setup

- Enable **idempotent**
- Enable **atomic transactions**
- Enable **acks=all** for durable writes

```java
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("enable.idempotence", "true"); // <-- *1*
producerProps.put("acks", "all"); // <--  *2*
producerProps.put("transactional.id", "my-transactional-id"); // <-- *3*
Producer<String, String> producer = new KafkaProducer<> ...

producer.initTransactions(); // <-- *4*
try
{
    producer.beginTransaction(); // <-- *5* begin atomic batch

    // send a bunch of messages
    producer.send(record1);   // <-- *6*
    producer.send(record2);
    producer.send(record3);

    producer.commitTransaction(); // <-- *7* commit batch
}
catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
     // We can't recover from these exceptions, so our only option is to close the producer and exit.
     producer.close();
} catch (KafkaException e) {
     // For all other exceptions, just abort the transaction and try again.
     producer.abortTransaction(); // <-- *8*
}
producer.close();
```

Exactly Once - Consumer

- Finally, in order for transactional guarantees to be realized from end-to-end, the consumers must be configured to read only committed messages as well
- **`isolation.level="read_committed"`**
- This Controls how to read messages written transactionally. If set to **read_committed**, consumer.poll() will only return transactional messages which have been committed.
- If set to **read_uncommitted** (the default), consumer.poll() will return all messages, even transactional messages which have been aborted. Non-transactional messages will be returned unconditionally in either mode.

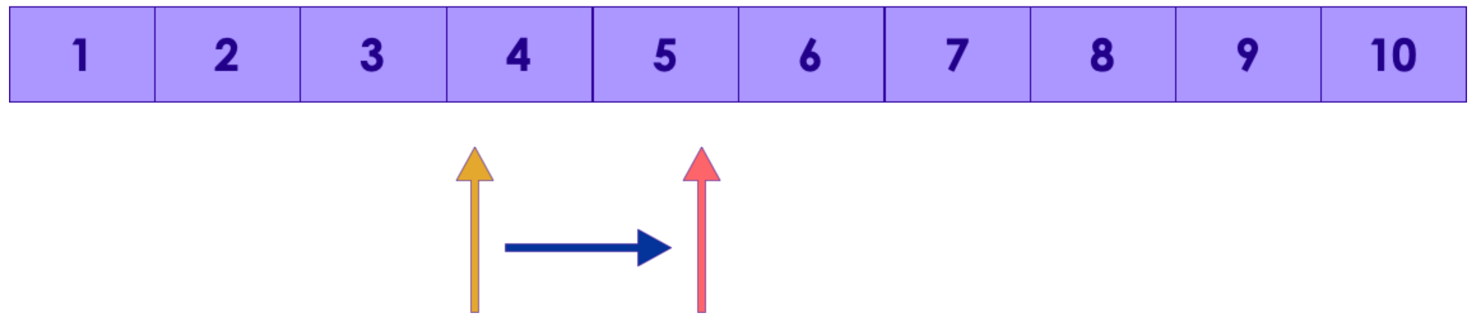
```
Properties consumerProps = new Properties();
consumerProps.put("bootstrap.servers", "localhost:9092");
consumerProps.put("group.id", "my-group-id");
consumerProps.put("enable.auto.commit", "false");
consumerProps.put("isolation.level", "read_committed"); // <-- *1*

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(singleton("topic1"));
```

Lab : Consumer Commits

- **Overview:** Try different commit methods in Consumers
  - At-most once, At-least once
- **Builds on previous labs:** lab 3
- **Approximate Time:** 20 - 30 mins
- **Instructions:**
  - Please follow lab 5.1
- **To Instructor:**

Seeking To An Offset

- Consumers can skip to any offset in Kafka
  - To go to end of partition
  - To go to beginning of partition
  - Seek to any offset in partition

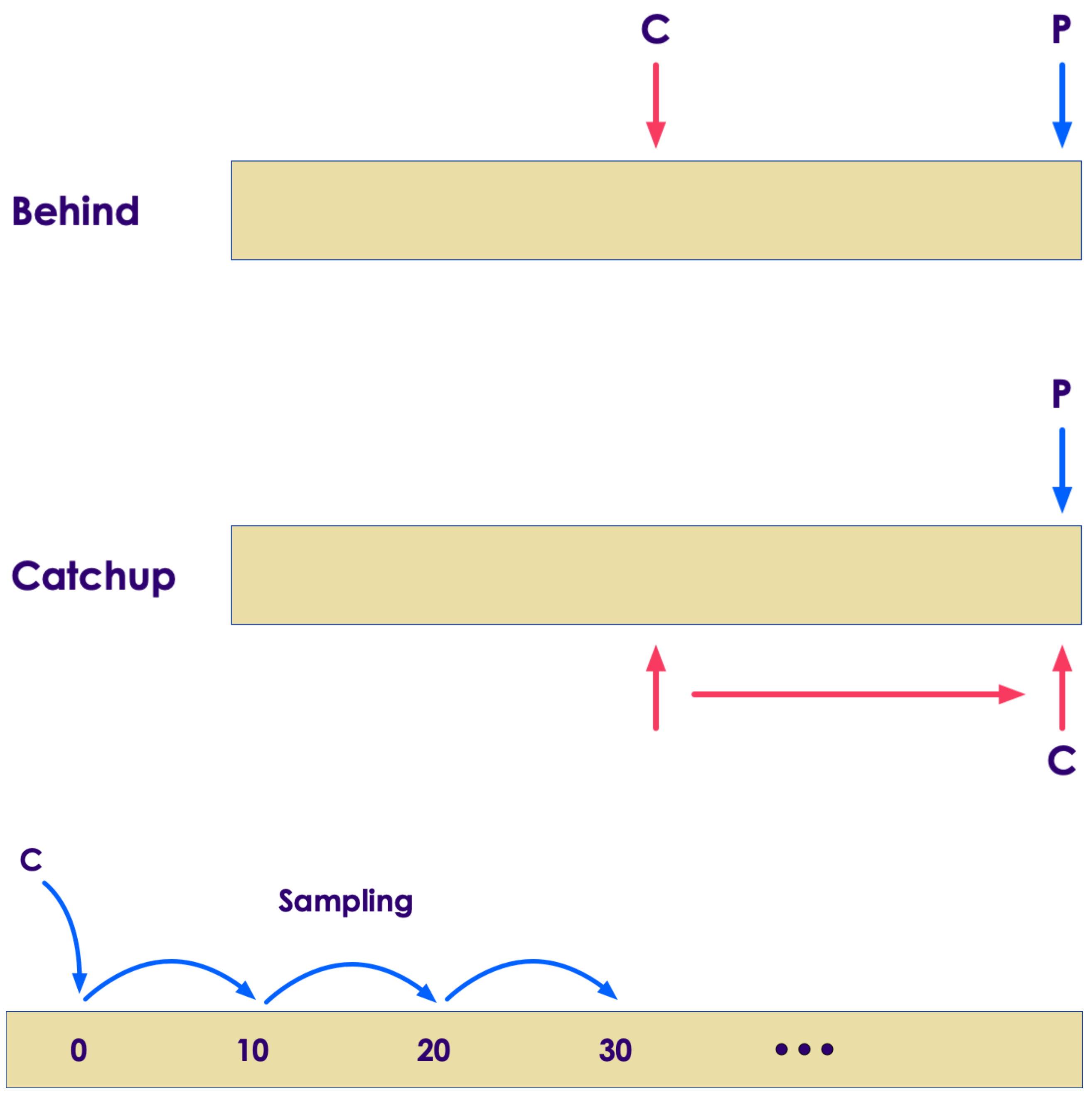


- Use cases:
  - Catching up to latest events first (and then processing the backlog). See demo

```java
import java.util.Collections;
import org.apache.kafka.common.TopicPartition;

// topic: topic1,  partition : 0
TopicPartition partition = new TopicPartition("topic1", 0);

// go to beginning
consumer.seekToBeginning(Collections.singletonList(partition));

// go to end
consumer.seekToEnd(Collections.singletonList(partition));

// seek to offset #5 on partition[0]
consumer.seek(partition, 5);
```

Use Cases for Seeking

- Catchup with latest data
  - If our processing is behind, we can skip ahead and catchup with latest data
  - For example, weather data, stock quotes ..etc
- Do sampling
  - We don't need to process all data, let's say we want to do **10% sampling**
  - Seek to offset=0, process
  - Then seek to offset=10, process
  - and so on...



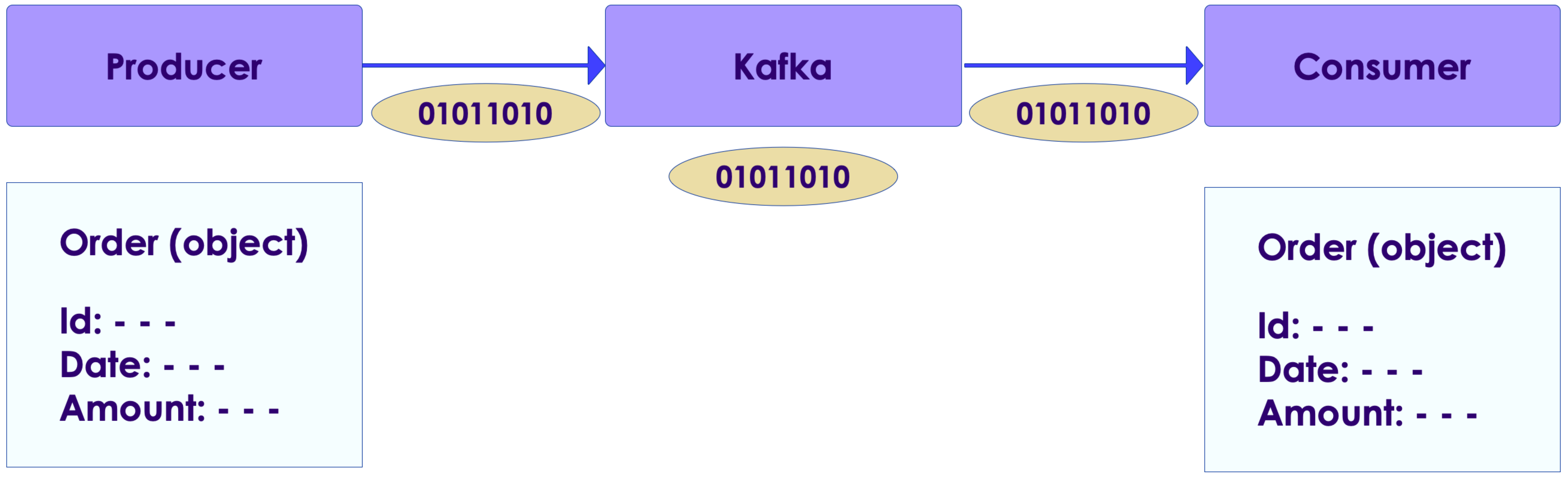
Lab: Seeking to Offsets

- **Overview:** Seek and read various offsets in a partition
- **Builds on previous labs:**
- **Approximate Time:** 20 - 30 mins
- **Instructions:**
  - Please follow: lab 5.2
- **To Instructor:**

Data Schema in Kafka

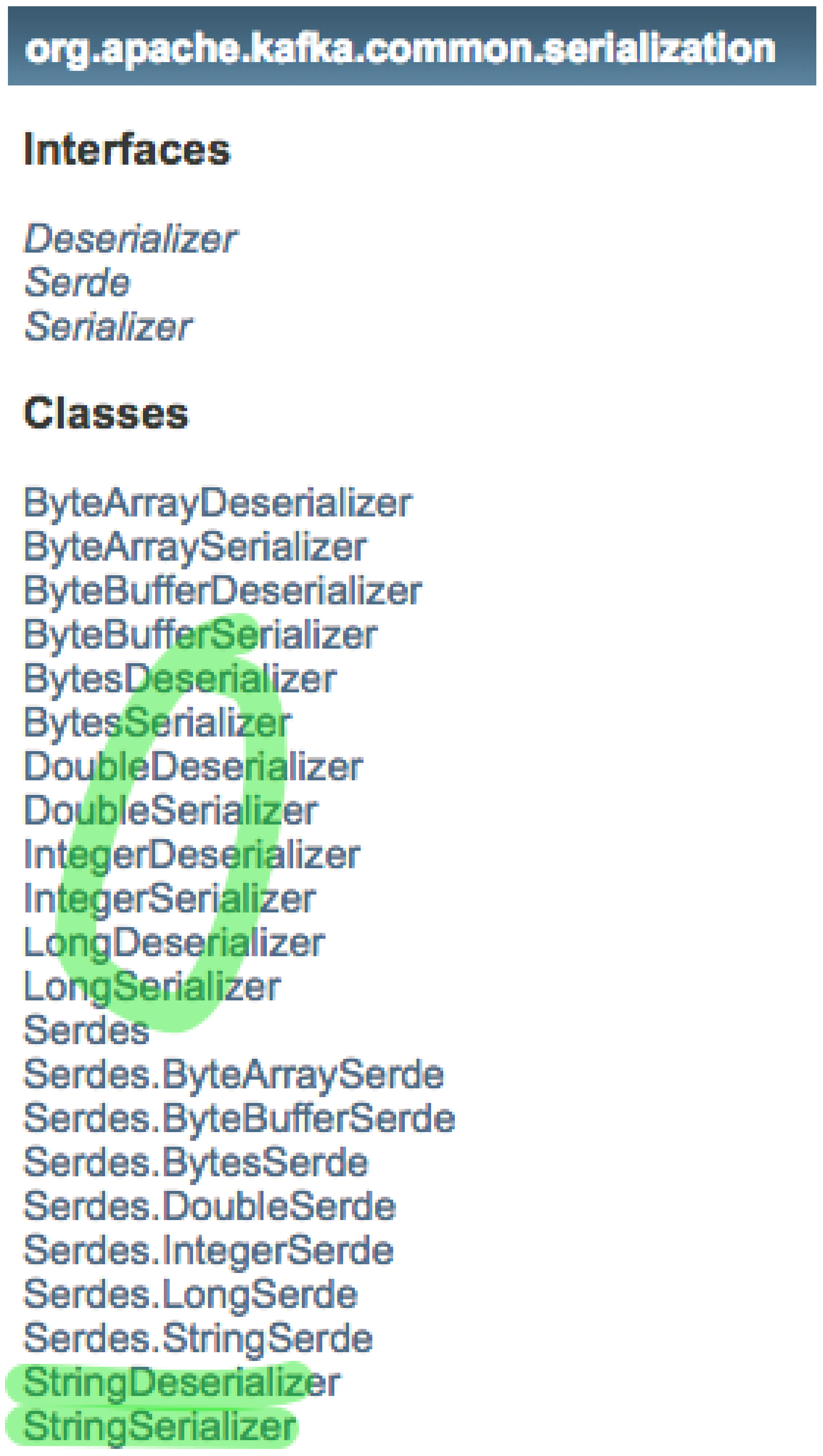
Kafka and Data Schema

- Kafka treats all data as 'binary'; it doesn't care about schema/structure
- How ever, Producer / Consumer like to deal in Java objects



Standard Kafka SerDe

- Kafka comes with Serializers Deserializers for standard Java types
- Can be found in package **org.apache.kafka.common.serialization**
- We need to provide SerDe classes for custom types
- Can use Avro/JSON



org.apache.kafka.common.serialization

Interfaces

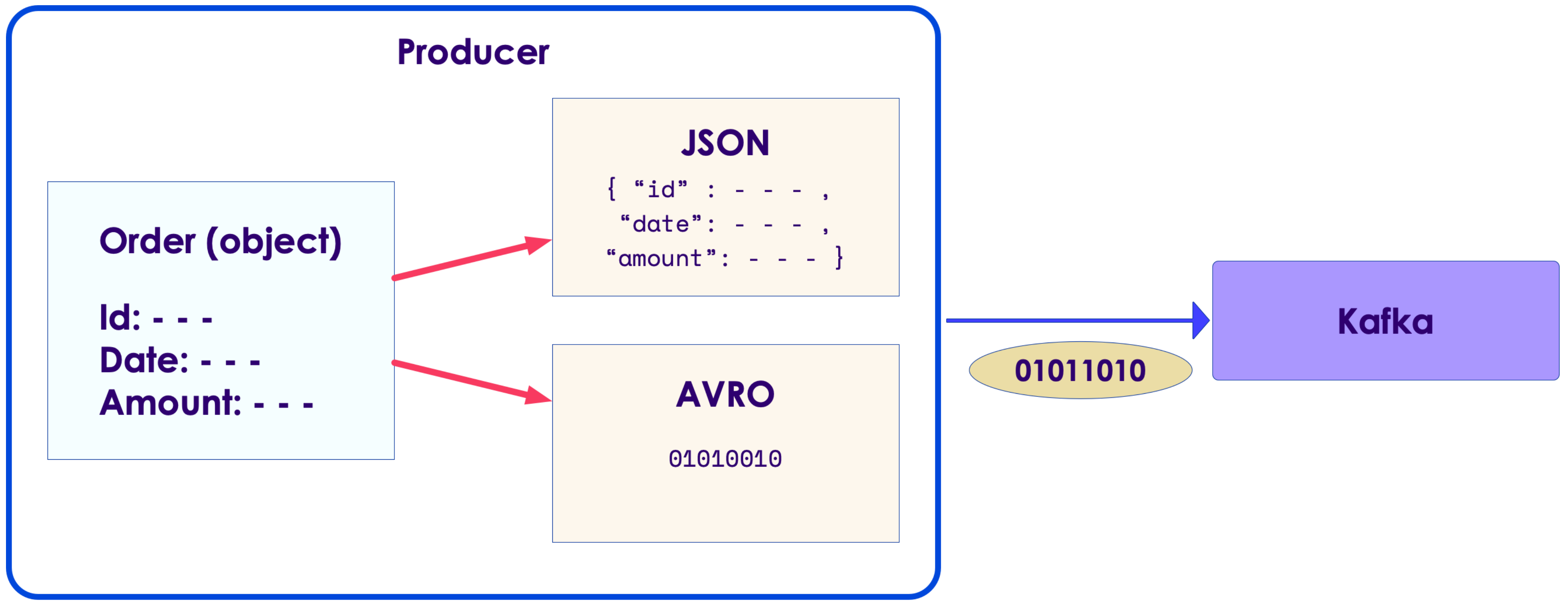
Deserializer
Serde
Serializer

Classes

ByteArrayDeserializer
ByteArraySerializer
ByteBufferDeserializer
ByteBufferSerializer
BytesDeserializer
BytesSerializer
DoubleDeserializer
DoubleSerializer
IntegerDeserializer
IntegerSerializer
LongDeserializer
LongSerializer
Serdes
Serdes.ByteArraySerde
Serdes.ByteBufferSerde
Serdes.BytesSerde
Serdes.DoubleSerde
Serdes.IntegerSerde
Serdes.LongSerde
Serdes.StringSerde
StringDeserializer
StringSerializer

Handling Structured Data

- We can use various data formats to convert from Java objects to serializable formats
- JSON format
  - Text based format, easily readable
- AVRO format
  - Binary format



JSON Format

- JSON is a flexible, text format
- JSON supports nested structures

```
{"id": 123,
 "name": "Homer Simpson",
 "age": 45,
 "address": {
             "street": "742 Evergreen Terrace",
             "city":   "Springfield",
             "state":  "CA" }
}
```

- Java objects can be easily transferred to/from JSON
- JSON text compresses really well; we can get anywhere from 5x to 10x compression; This saves a lot of bandwidth and disk space
- Also since JSON is just text, we can use built-in `StringSerializer` to encode the data

JSON Format

- We can use GSON library to convert to JSON

```java
import com.google.gson.Gson;

public class Customer {
  public int id;
  public String name;
  public String email;
}

Gson gson = new Gson();
Customer customer = new Customer(1, "Homer", "homer@simposon.com")
String json = gson.toJson(customer);

// {"id": 1, "name": "Homer", "email": "homer@simpson.com" }
```

- From JSON

```java
import com.google.gson.Gson;

String jsonStr = "{\"id\": 1, \"name\": \"Homer\", \"email\": \"homer@simpson.com\" }";

Gson gson = new Gson();

Customer customer = gson.fromJson(jsonStr, Customer.class);
// Now 'customer' object has all fields populated
```

Using JSON in Producer

- JSON is sent as `String`

```
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.apache.kafka.common.serialization.StringSerializer;
import com.google.gson.Gson;

Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName()); // <-- *1*

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

// create a customer object
Customer customer = new Customer(1, "Homer", "homer@simposon.com")

String customerJSON = gson.toJson(customer); // <--  *2*
// {"id": 1, "name": "Homer",  "email": "homer@simpson.com" }

ProducerRecord <Int, String> record = new ProducerRecord <> ("topic1", customer.id, customerJSON);

producer.send (record);
```

Using JSON in Consumer

```java
import java.util.Properties;
import java.util.Collections;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.apache.kafka.common.serialization.StringDeserializer;
import com.google.gson.Gson;

Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());  // *1*

Consumer <Int, String> consumer = new Consumer <> (props);
consumer.subscribe(Collections.singletonList("topic1"));

final ConsumerRecords <Int, String> records = consumer.poll(100);

for (ConsumerRecord<Int, String> record : records) {
    try {
        int customerId = record.key();
        String customerJSON = record.value();  // *2*

        // use GSON to deserialize JSON object into Customer object
        Customer customer = gson.fromJson(customerJSON, Customer.class);  //  *3*
        // process 'customer' object

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Avro

- Avro is a data serialization format
- It is language neutral; Avro has APIs for Java, Python ..etc)
- Data is stored in binary format, schema is stored in JSON format
- A key feature of Avro is robust support for data schemas that change over time — often called **schema evolution**
- avro.apache.org

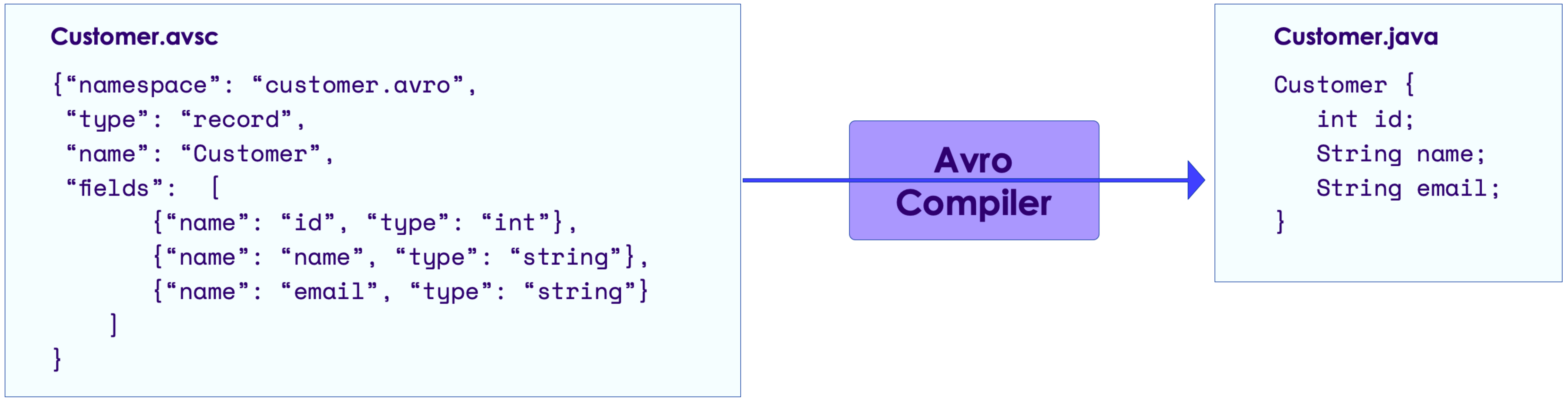
Avro Format

- Avro schema is defined as JSON file. Here is `Customer.avsc` file

```
{"namespace": "customer.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id",    "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "email", "type": "string"}
 ]
}
```

- Compile the avro schema into Java class. This will generate `Customer.java`

```
$   java -jar /path/to/avro-tools-1.10.2.jar compile schema
                Customer.avsc \  # <--- schema file
                src/java/main  #  <--- destination folder for generated Java class file
```



```
{"namespace": "customer.avro",
 "type": "record",
 "name": "Customer",
 "fields":  [
      {"name": "id", "type": "int"},
      {"name": "name", "type": "string"},
      {"name": "email", "type": "string"}
    ]
}
```

```
Customer {
   int id;
   String name;
   String email;
}
```

Avro in Kafka - Producer

```
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.IntegerSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializer;

Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
// Avro serializer
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());

KafkaProducer <Int, Customer> producer = new KafkaProducer<>(props);

// create a customer object
Customer customer = Customer.newBuilder()
                            .setId(1)
                            .setName("Homer Simpson")
                            .setEmail("homer@simpson.com").
                            .build();

ProducerRecord <Int, Customer> record = new ProducerRecord <> ("topic1", customer.id, customer);

producer.send (record);
```

Avro in Kafka - Consumer

```
import java.util.Properties;
import java.util.Collections;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.Consumer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;

Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "Kafka-Avro-Consumer-1");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                IntDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                KafkaAvroDeserializer.class.getName());

// Use Specific Record or else you get Avro GenericRecord.
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");

Consumer <Int, Customer> consumer = new Consumer <> (props);
consumer.subscribe(Collections.singletonList("topic1"));

final ConsumerRecords <Int, Customer> records = consumer.poll(100);

records.forEach (record -> {

    Customer customerRecord = record.value();  // AvroRecord!

    System.out.printf("%s %d %d %s \n", record.topic(),
                      record.partition(), record.offset(), customerRecord);
});
```

Handling Schema Evolution

- Let's say we have messages in the following format
- Version 1

Id	Type	Success
12345	Click	YES

- Version 2

Id	Type	Success	Message
12345	Click	YES	Page not found

- Q: How will the consumer process this?

Avro Schema

- Version 1

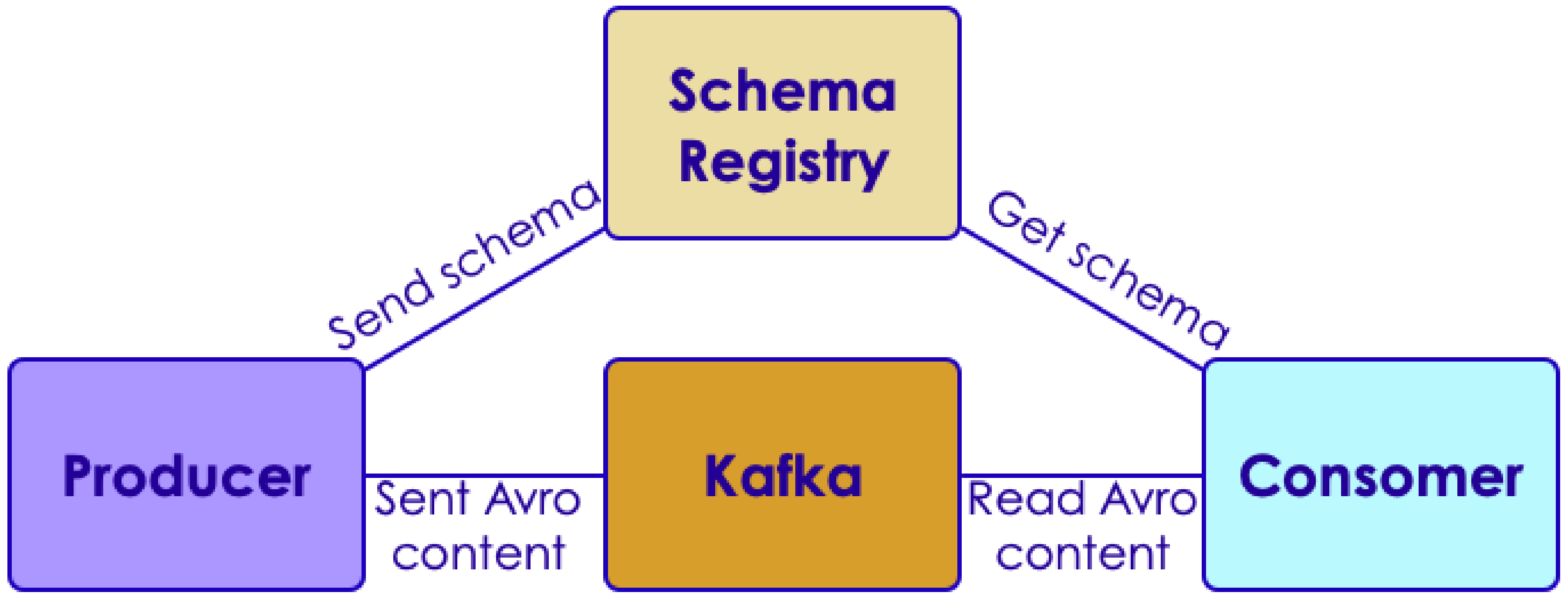
```
{"namespace": "com.example.videos",
  "type": "record",
  "name": "Event",
  "fields": [
     {"name": "id", "type": "int"},
     {"name": "type",  "type": "string"},
     {"name": "success",  "type": "string"}
]
}
```

- Version 2

```
{"namespace": "com.example.videos",
  "type": "record",
  "name": "Event",
  "fields": [
     {"name": "id", "type": "int"},
     {"name": "type",  "type": "string"},
     {"name": "success", "type": "string"},
     {"name": "message", "type": "string"}  // <- new attribute
  ]}
```

Managing Multiple Schema Versions Using Schema Registry

- We can register multiple versions of Avro schema with **Schema Registry**
- The AVRO messages will have a **version** tag
- Then consumer can download the relevant schema and interpret the data accordingly.



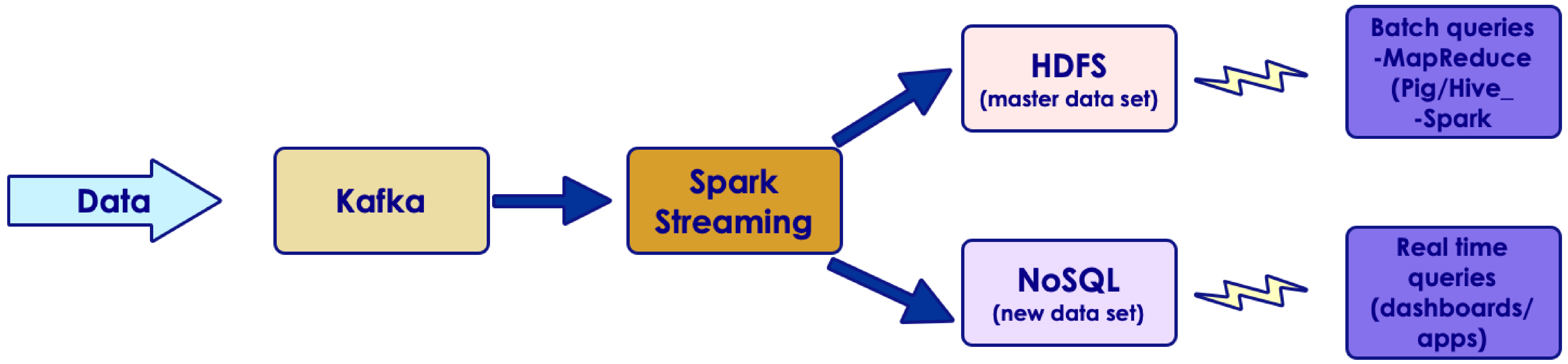
Lab : Using AVRO Schema (Intermediate Track)

- **Overview:**
  - Use AVRO schema in Kafka
- **Approximate Time:**
  - 30 mins
- **Instructions:**
  - Please follow **AVRO-1** lab
  - Please Note : This lab needs Confluent stack / schema registry
  - **Come back to this lab after installing Confluent stack**

Kafka & Spark

Lambda Streaming Architecture

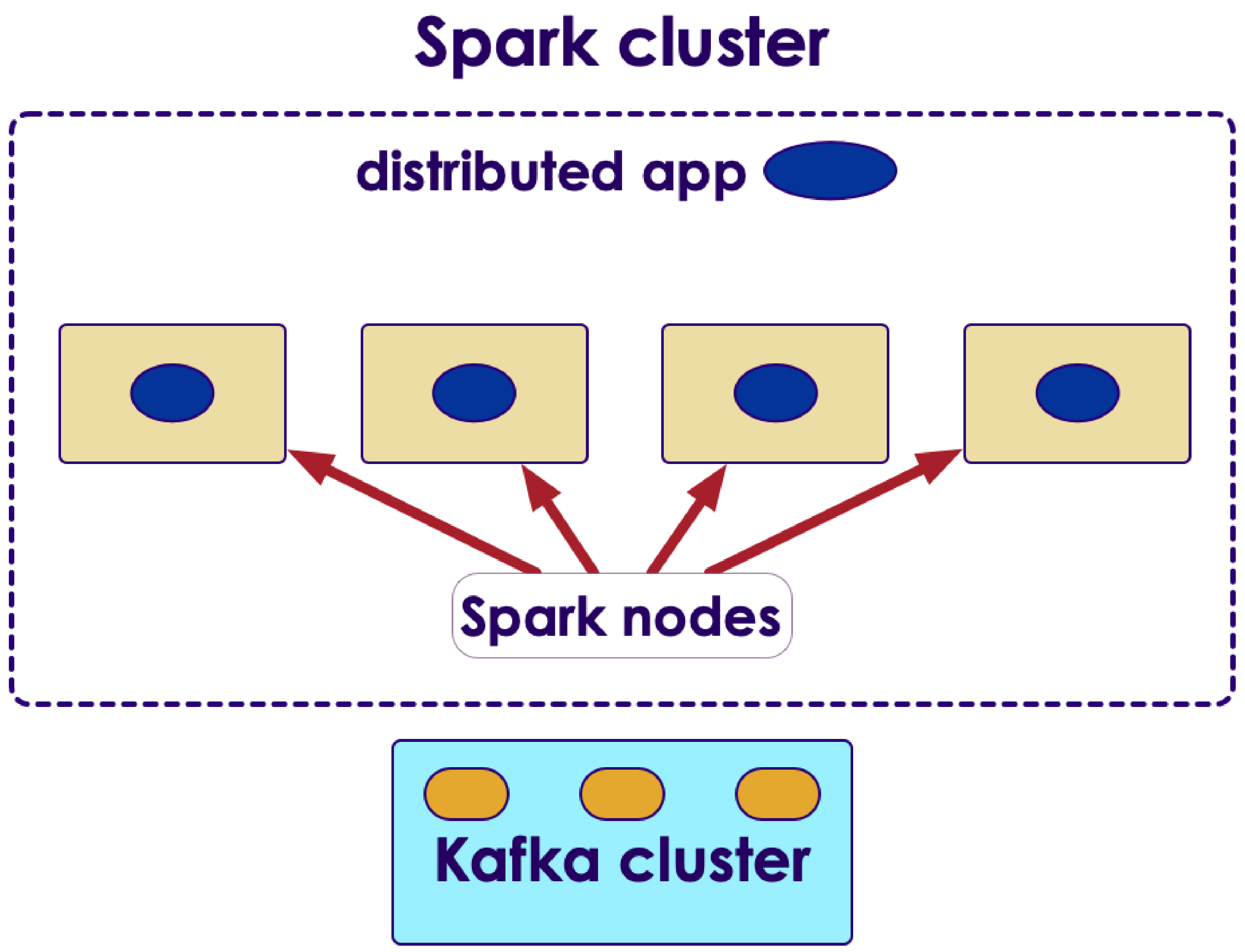
- Spark and Kafka are a popular pair in Lambda architecture


Data
Kafka
Spark Streaming
HDFS (master data set)
NoSQL (new data set)
Batch queries -MapReduce (Pig/Hive_ -Spark
Real time queries (dashboards/ apps)

- *Source: http://lambda-architecture.net/*

Spark and Kafka

- Spark is a distributed platform
  - It will run applications as multiple parallel tasks
  - Spark will also handle any failures (task failure, or node failure)
- Here we see two clusters - Spark cluster and Kafka cluster
- As far as Kafka is concerned, Spark application will behave like a regular client (either Producer or Consumer)



Kafka Structured Streaming Example

```
// Subscribe to 1 topic
Dataset<Row> df = spark
                        .readStream()
                        .format("kafka")
                        .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
                        .option("subscribe", "topic1")
                        .load();

df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)");

// once a dataframe is established, we can query it
df.count();

filtered = df.filter (df['value'].contains("some-string-pattern"));
```

- References
    - Spark examples

Lab: Analyzing Clickstream Data

- We are going to be analyzing Clickstream data

```
{ "timestamp" :1451635200055,
"session":"session_57" ,
"domain":"twitter.com" ,
"cost":24,
"user":"user_31",
"campaign": "campaign_1",
"ip":"ip_64",
"action": "blocked" }
```

- Process the data and keep a running total of `'domain-count'`
    - Twitter.com: 10
    - Facebook.com: 12

Lab: Clickstream Lab

- **Overview:** Process clickstream data
- **Builds on previous labs:**
- **Approximate Time:** 30 - 40 mins
- **Instructions:**
  - Please follow lab 6
- **To Instructor:**

Review Questions

- How would you ensure that a message has been written durably in your Producer?
- True or False: Each Consumer reads from a single topic
- How does a Consumer retrieve messages?
- How does the Consumer do a `'clean shutdown'`?
  - Why is this important?
- How is compression enabled for messages?