

Kafka Notes 1

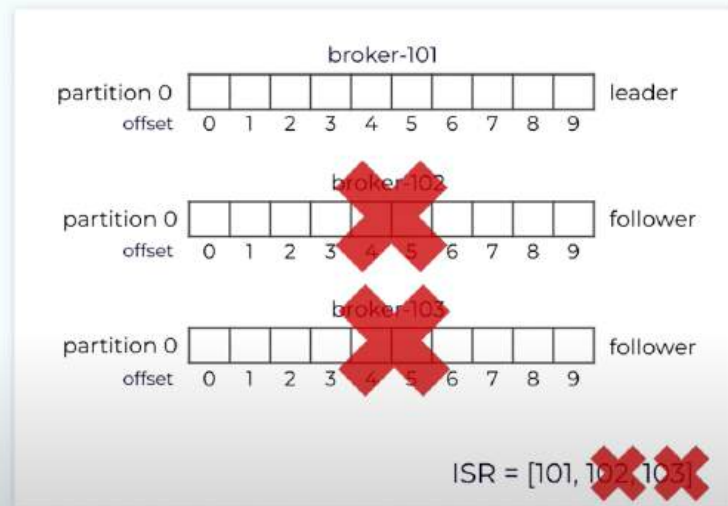
3 ways to send a msg to Kafka:

1. Fire and forget (we may lose some portion of the messages)
2. Synchronous send (use this when the method is critical and can't afford to lose anything, producer will wait until gets the record metadata/exception)
3. Asynchronous send - gives better throughput than synchronous.
 - We pass a callback object while sending a record.
 - we keep sending msgs without waiting for result, and later we handle exception in the callback function.
 - `max.in.flight.requests.per.connection` -> controls no. of msgs that can be sent without receiving a result.

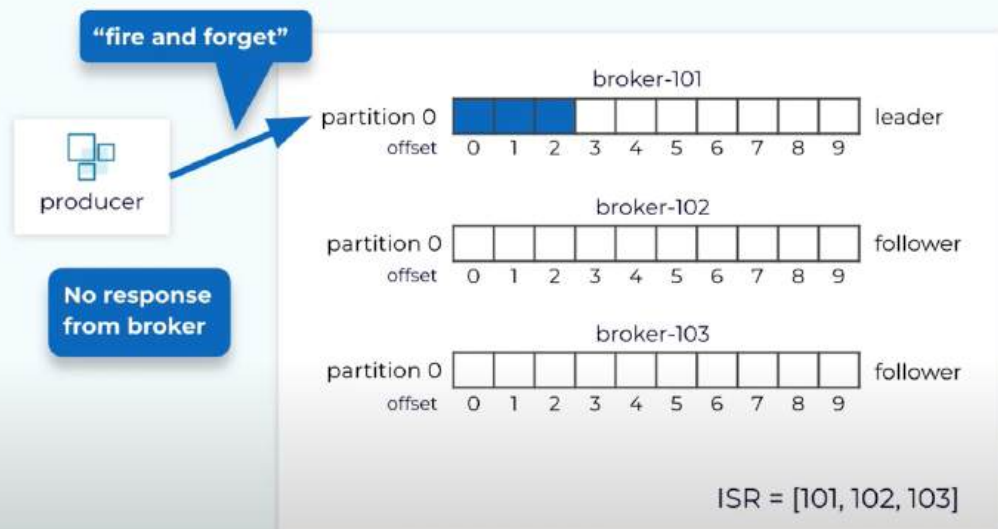
```
        producer.send(record, new MyProducerCallback());
        System.out.println("AsynchronousProducer call completed");
        producer.close();
    }
}

class MyProducerCallback implements Callback{

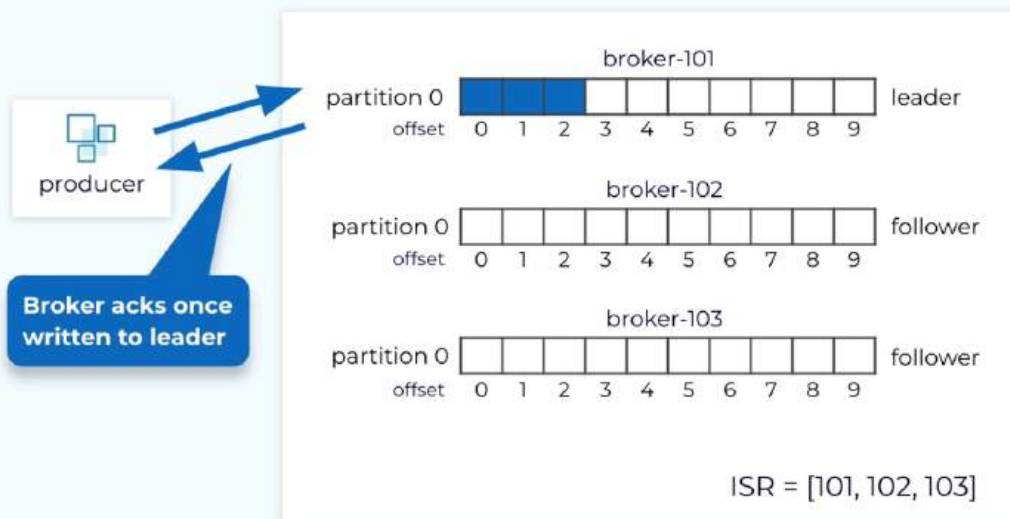
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null)
            System.out.println("AsynchronousProducer failed with an exception")
        else
            System.out.println("AsynchronousProducer call Success:");
    }
}
```



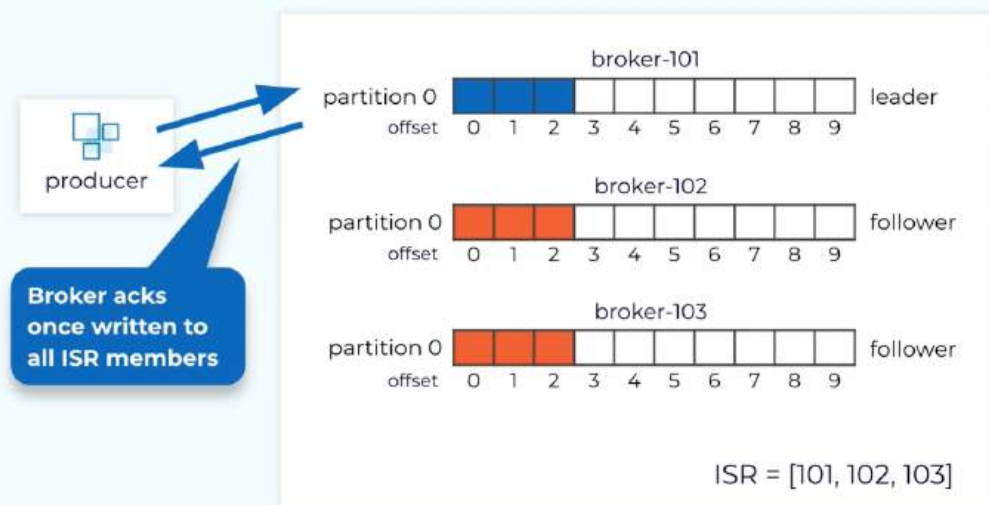
Producer acks=0



Producer acks=1

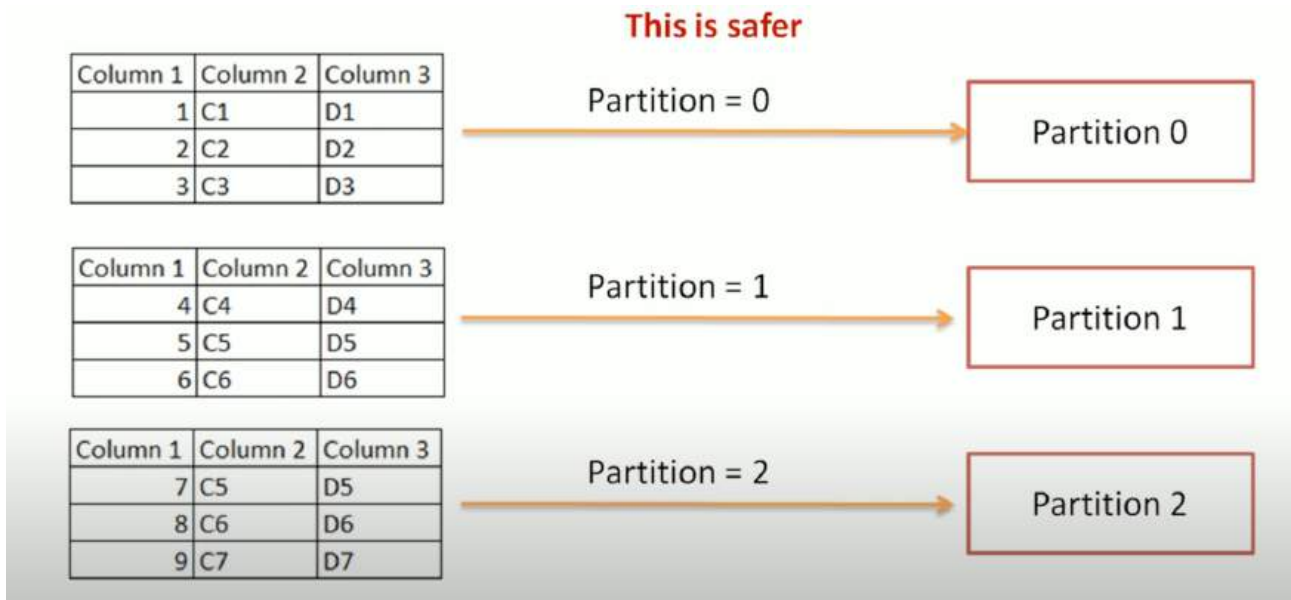


Producer acks=all

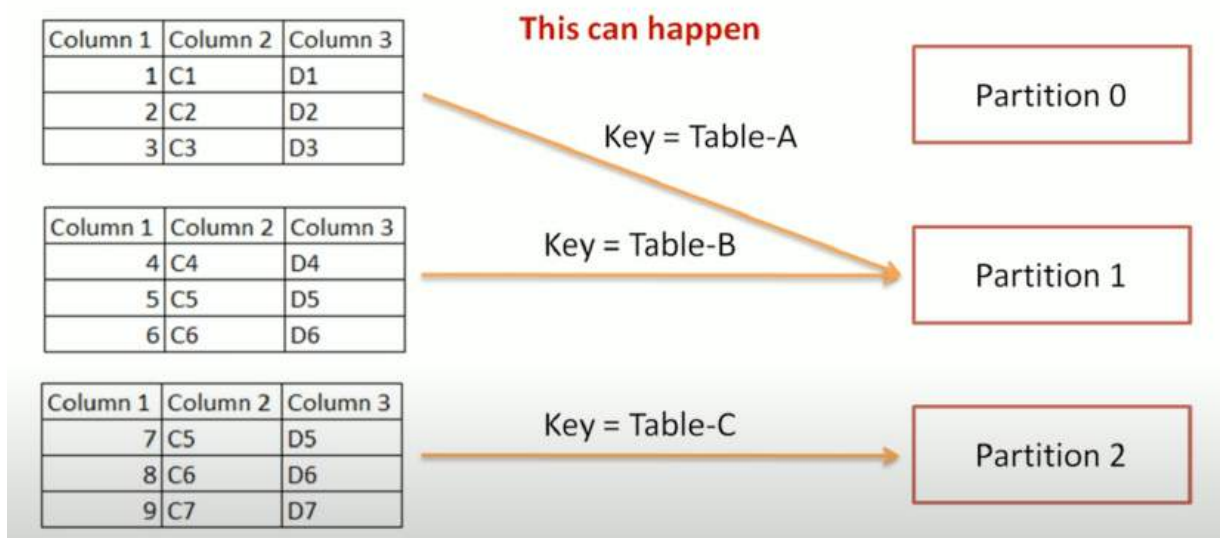


Partitioner:

- if a partition is specified in the record, use it.

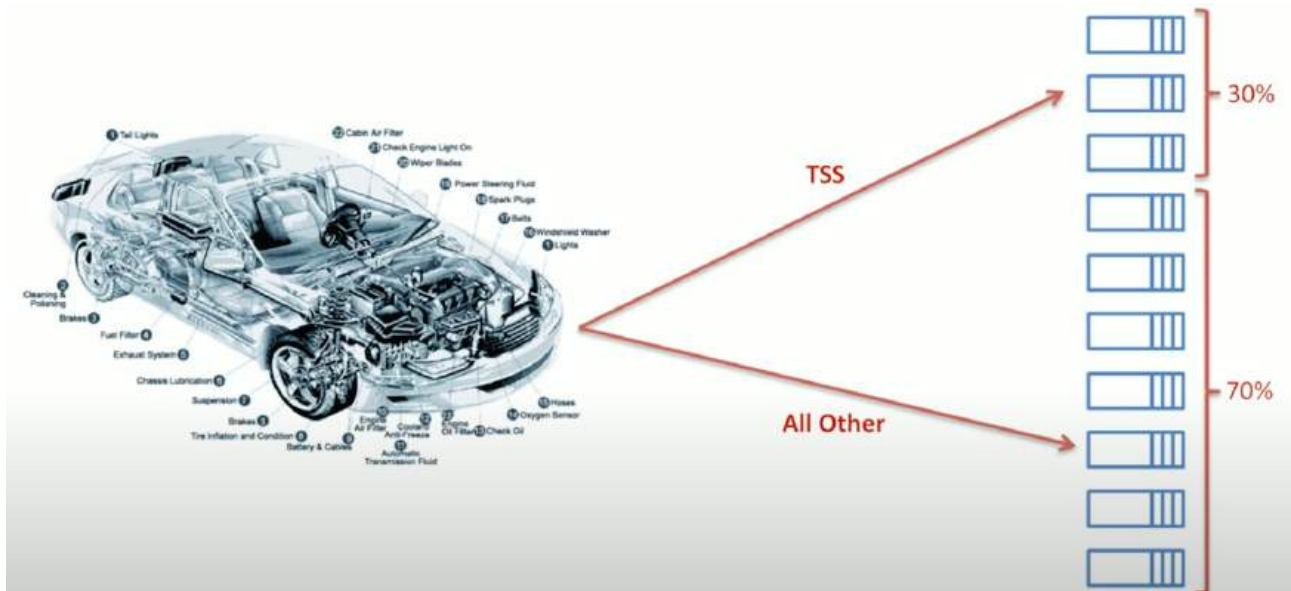


- if no partition is specified but a key is present choose a partition based on a hash of key
 - if no. of partition increases then we start receiving different hash key.



- if no partition or key is present choose a partition in a round-robin fashion.

- Custom partitioner:



```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092,localhost:9093");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("partitioner.class", "SensorPartitioner");
props.put("speed.sensor.name", "TSS");
```

```
public class SensorPartitioner implements Partitioner {

    private String speedSensorName;

    public void configure(Map<String, ?> configs) {
        speedSensorName = configs.get("speed.sensor.name").toString();
    }

    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {

        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        int sp = (int) Math.abs(numPartitions * 0.3);
        int p = 0;

        if ( (keyBytes == null) || (!(key instanceof String)) )
            throw new InvalidRecordException("All messages must have sensor name as key");

        if ( ((String)key).equals(speedSensorName) )
            p = Utils.toPositive(Utils.murmur2(valueBytes)) % sp;
        else
            p = Utils.toPositive(Utils.murmur2(keyBytes)) % (numPartitions - sp) + sp ;

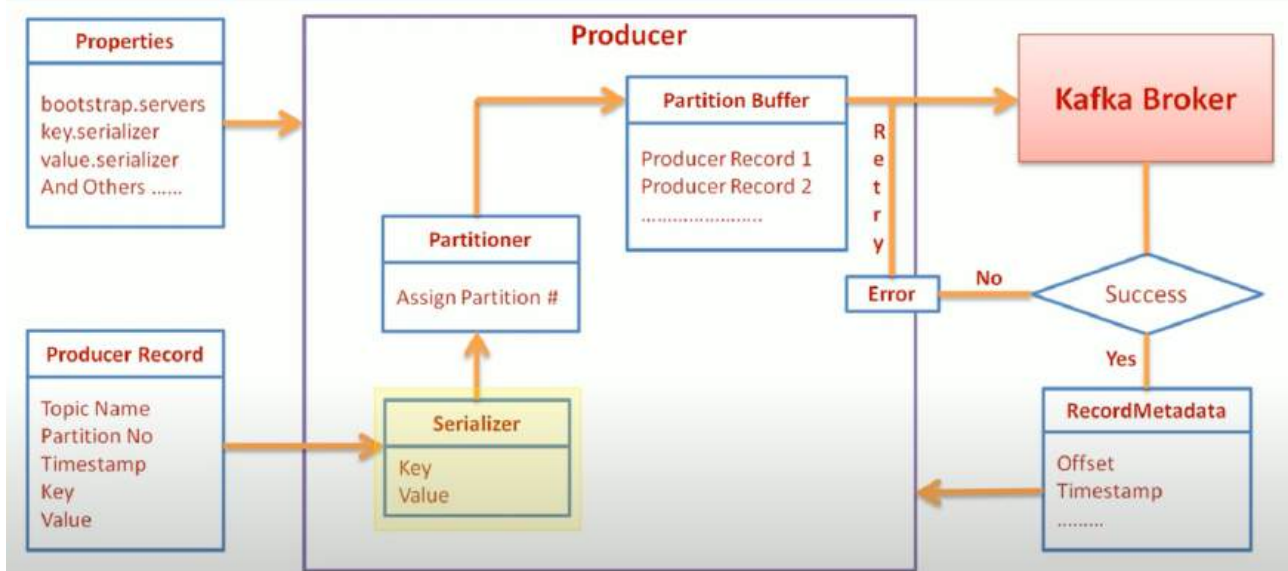
        System.out.println("Key = " + (String)key + " Partition = " + p );
        return p;
    }

    public void close() {}
}
```

- Here when the key is equal to speedSensorName then we are hashing using valueBytes else with keyBytes Bcz when the key is equal to speedSensorName, if we hash using the key then every time the record will go to the same partition. But we want to divide them in 3 partitions.

Serializer

- we can send different type of data to broker.



- custom serializer and deserializer:

```

51 lines (40 sloc) | 1.65 KB
1 import org.apache.kafka.common.serialization.Serializer;
2 import org.apache.kafka.common.errors.SerializationException;
3 import java.io.UnsupportedEncodingException;
4 import java.util.Map;
5 import java.nio.ByteBuffer;
6
7 public class SupplierSerializer implements Serializer<Supplier> {
8     private String encoding = "UTF8";
9
10    @Override
11    public void configure(Map<String, ?> configs, boolean isKey) {
12        // nothing to configure
13    }
14
15    @Override
16    public byte[] serialize(String topic, Supplier data) {
17
18        int sizeOfName;
19        int sizeOfDate;
20        byte[] serializedName;
21        byte[] serializedDate;
22
23        try {
24            if (data == null)
25                return null;
26
27            serializedName = data.getName().getBytes(encoding);
28            sizeOfName = serializedName.length;
29            serializedDate = data.getStartDate().toString().getBytes(encoding);
30            sizeOfDate = serializedDate.length;
31
32            ByteBuffer buf = ByteBuffer.allocate(4+4+sizeOfName+4+sizeOfDate);
33            buf.putInt(data.getID());
34            buf.putInt(sizeOfName);
35            buf.put(serializedName);
36            buf.putInt(sizeOfDate);
37            buf.put(serializedDate);
38
39            return buf.array();
40
41        } catch (Exception e) {
42            throw new SerializationException("Error when serializing Supplier to byte[]");
43        }
44    }
45
46    @Override
47    public void close() {
48        // nothing to do
49    }
50 }

```



```

public Supplier deserialize(String topic, byte[] data) {

    try {
        if (data == null){
            System.out.println("Null recieved at deserialize");
            return null;
        }

        ByteBuffer buf = ByteBuffer.wrap(data);
        int id = buf.getInt();

        int sizeOfName = buf.getInt();
        byte[] nameBytes = new byte[sizeOfName];
        buf.get(nameBytes);
        String deserializedName = new String(nameBytes, encoding);

        int sizeOfDate = buf.getInt();
        byte[] dateBytes = new byte[sizeOfDate];
        buf.get(dateBytes);
        String dateString = new String(dateBytes,encoding);

        DateFormat df = new SimpleDateFormat("EEE MMM dd HH:mm:ss Z yyyy");

        return new Supplier(id,deserializedName,df.parse(dateString));
    }
}

```

```

public class SupplierConsumer{

    public static void main(String[] args) throws Exception{

        String topicName = "SupplierTopic";
        String groupName = "SupplierTopicGroup";

        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092,localhost:9093");
        props.put("group.id", groupName);
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "SupplierDeserializer");

        KafkaConsumer<String, Supplier> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList(topicName));

        while (true){
            ConsumerRecords<String, Supplier> records = consumer.poll(100);
            for (ConsumerRecord<String, Supplier> record : records){
                System.out.println("Supplier id= " + String.valueOf(record.value().getID()) + " Supplier Name = " + record
                );
            }
        }
    }
}

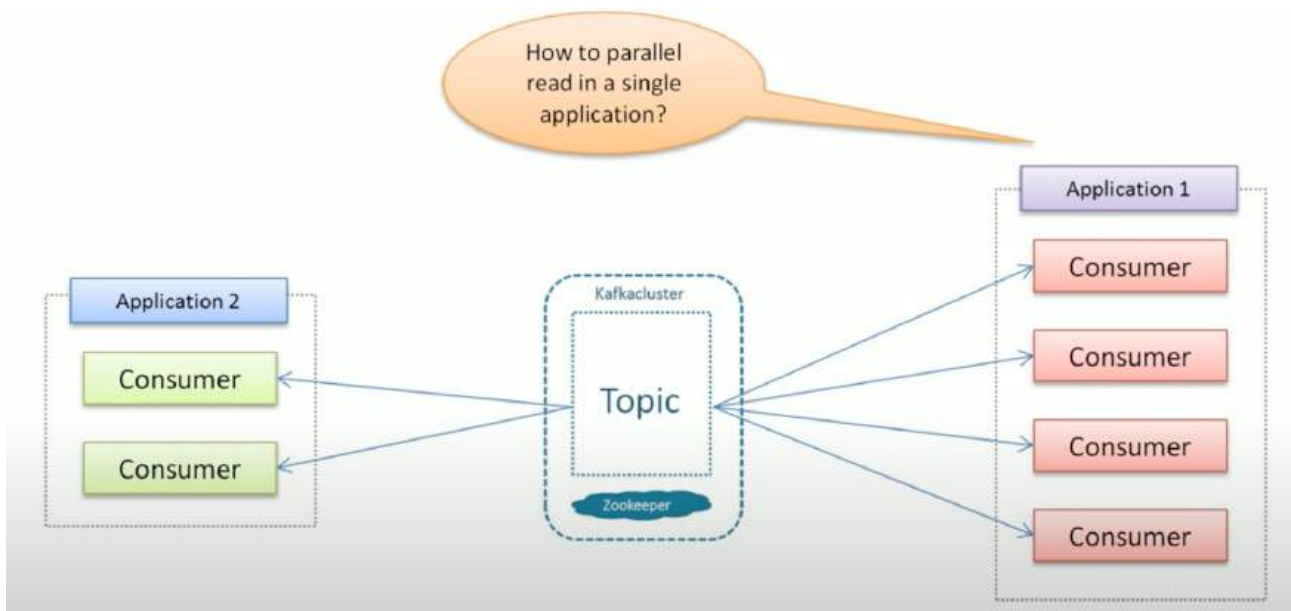
```

Producer Configs

- buffer.memory (Default: 32 MB)
 - total amount of memory (in bytes) available to the producer for buffering messages that are waiting to be sent to the broker.
- compression.type (Default: no compression)

- for compressing the data in the msg sent by the producer.
- reduces the network bandwidth usage and storage on Kafka brokers at the cost of additional CPU usage for compression and decompression.
- gzip: Gzip compression (high compression ratio, slower)
- snappy compression (low compression ratio, faster)
- lz4 compression (fast)
- zstd: Zstandard compression (high compression ratio and fast)
- batch.size (Default: 16 KB)
- linger.ms (Default: 0 Immediate send)
- client.id (Default: empty string)
 - unique identifier for the kafka producer client.
 - used to identify producer in broker logs and metrics
 - `client.id=my-producer-client`
- max.request.size (Default: 1 MB)
 - If a batch exceeds this size, producer will either split the batch or fail the send operation.
- In asynchronous send, if batch 1 fails to be sent to the broker, and batch 2 succeeds.
- Then batch 1 will retry and batch 2 is already sent.
- So, batch 1 will be sent later to the broker and order of delivery will be lost.
- (we are sending both the batches to the same partition)

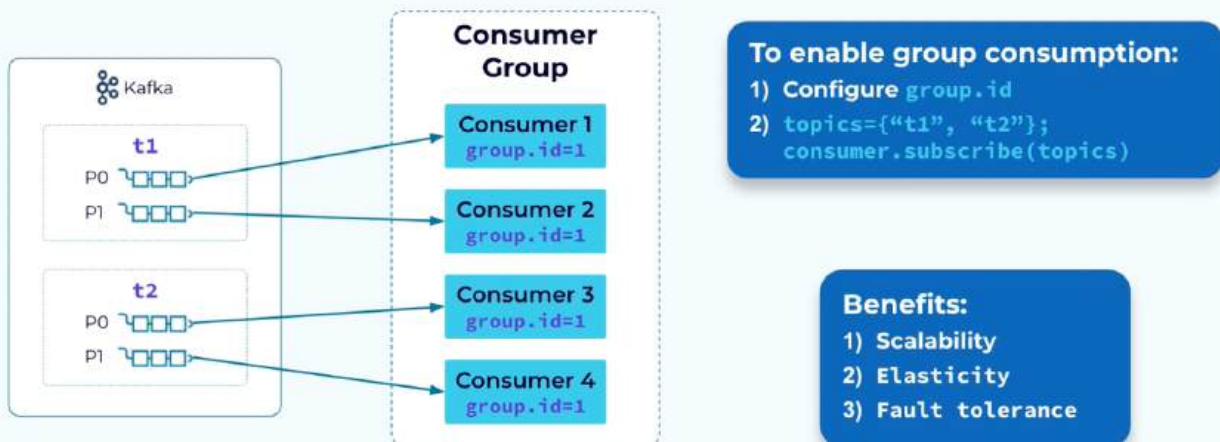
Consumer Group:



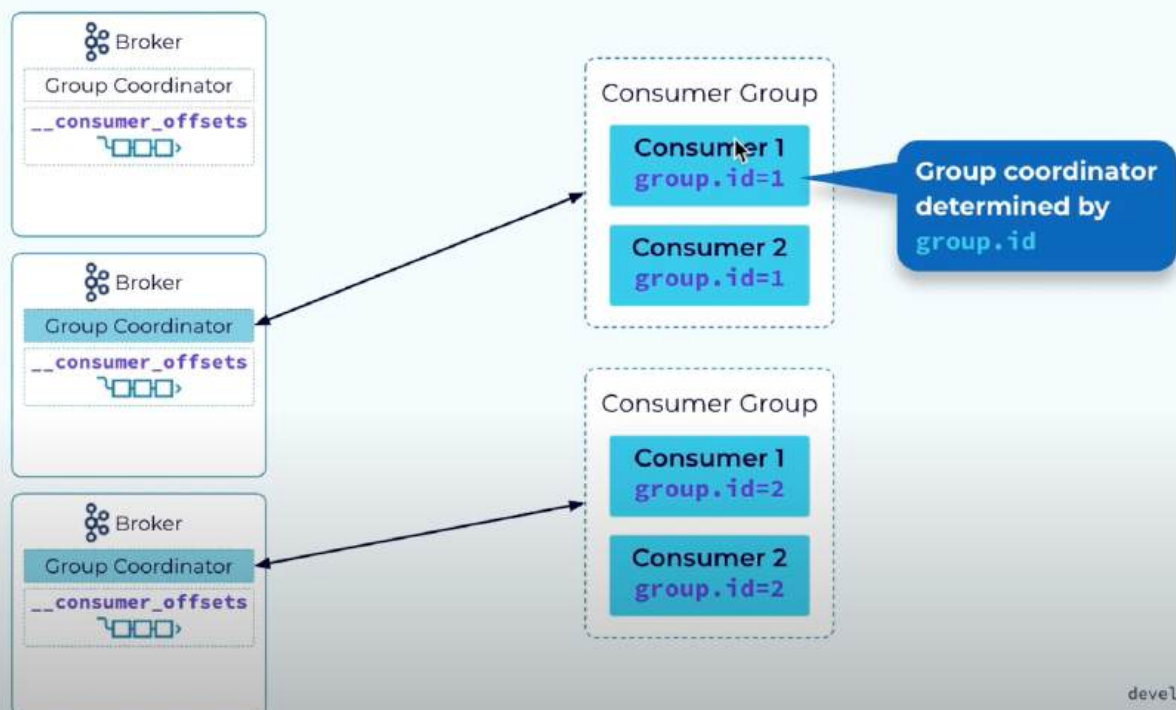
- used to read and process data in parallel.
- partitions are never shared among members of same group at the same time.
- Group coordinator maintains a list of active consumers.

- Group leader executes a rebalance activity.
- Rebalance initiated when the list of consumers is modified.
- Automatically manages partition assignment
 - 1. Range
 - 2. Round robin
- detects entry/exit/failure of a consumer and perform partition rebalancing activity.

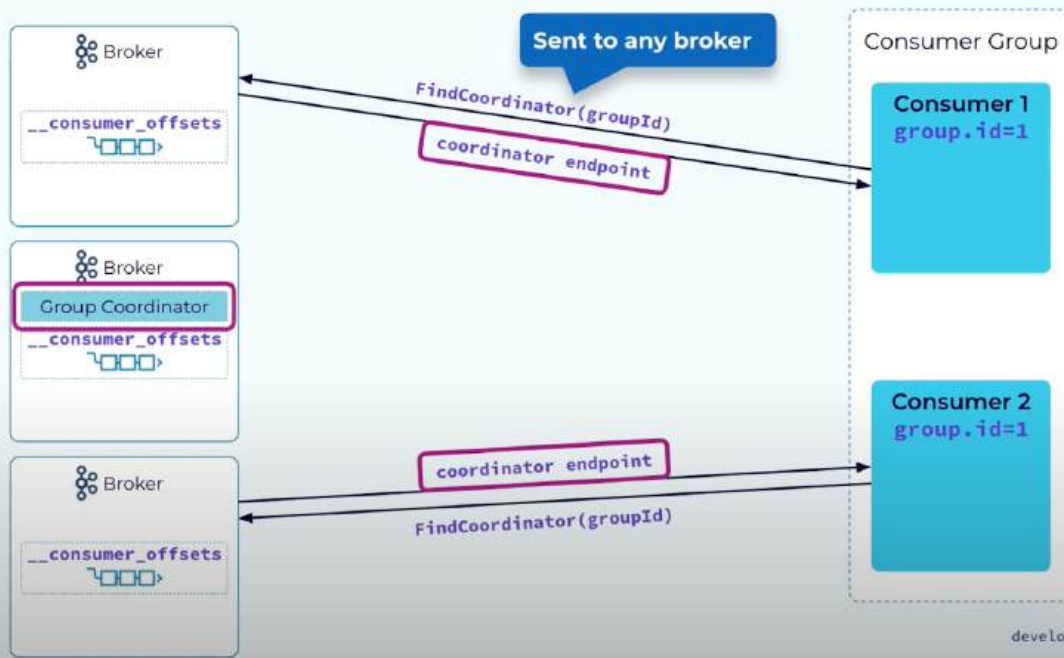
Kafka Consumer Group



Group Coordinator

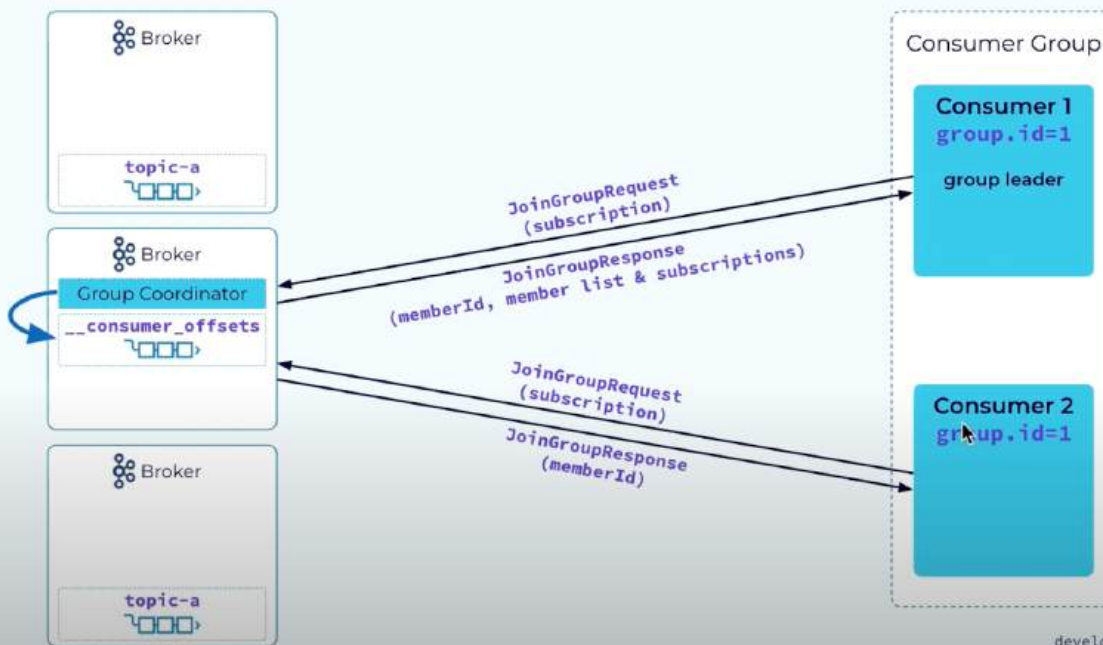


Group Startup: Step 1 - Find Group Coordinator



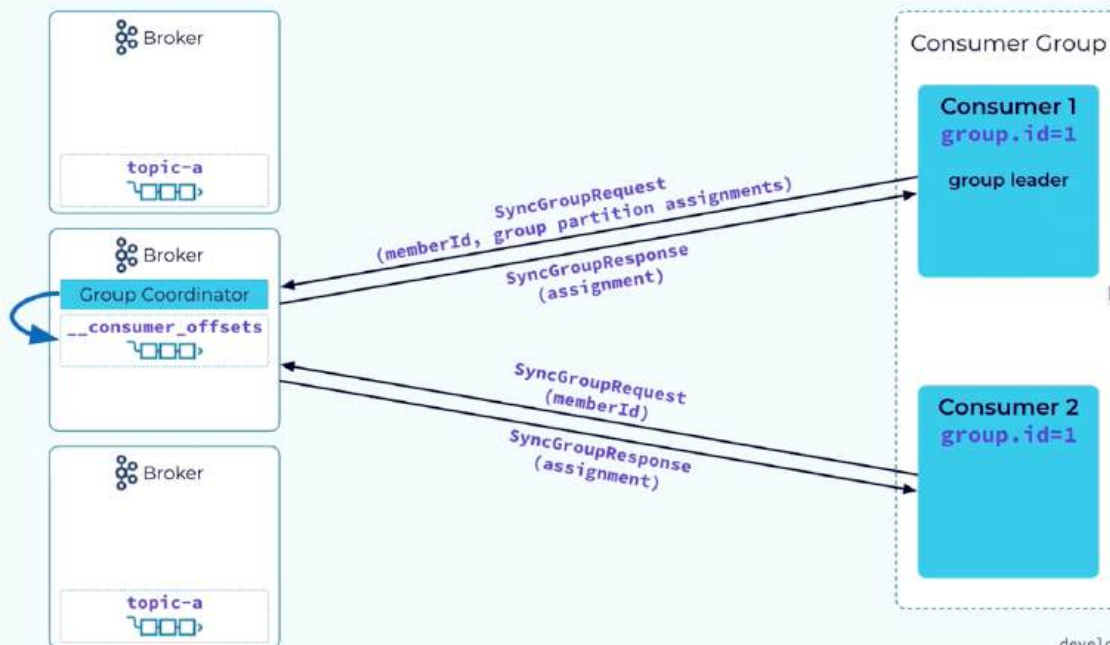
developer.confluent.io

Group Startup: Step 2 - Members Join

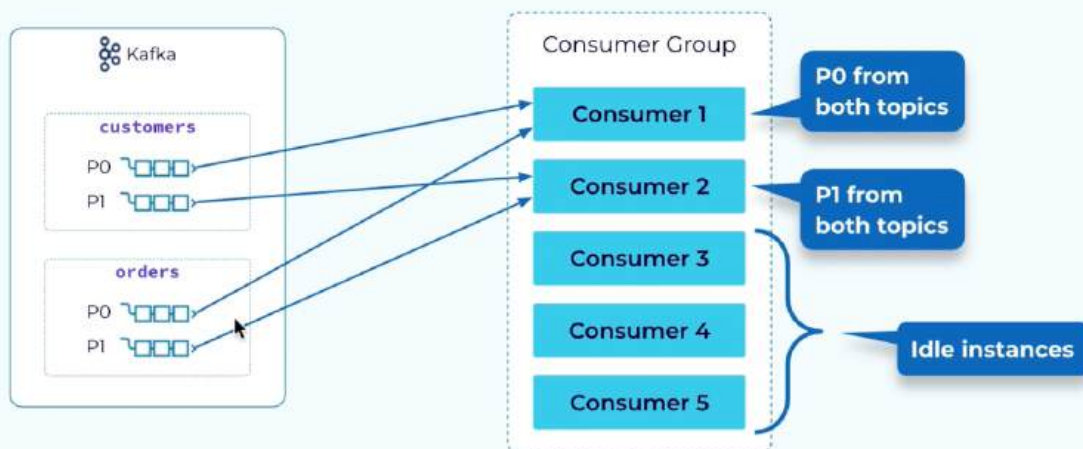


developer.confluent.io

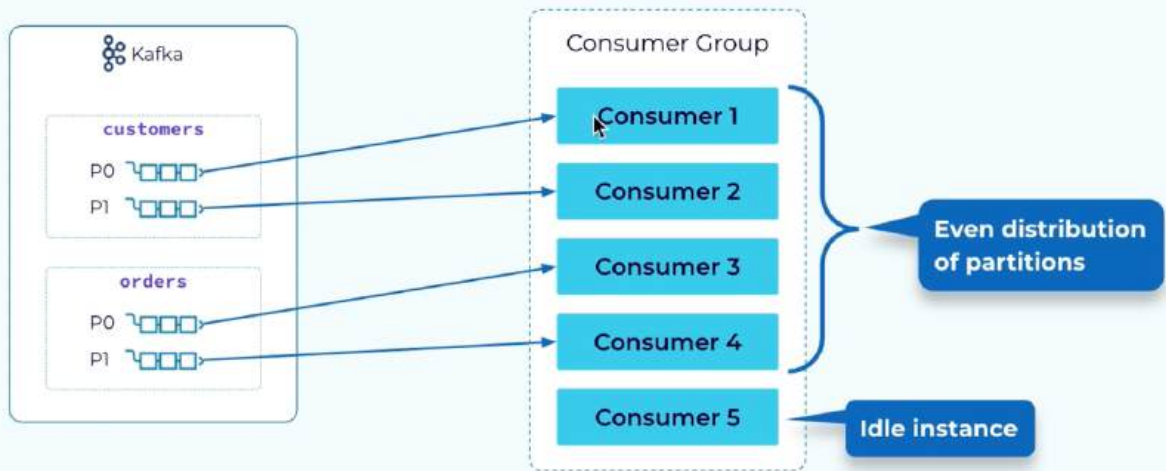
Group Startup: Step 3 - Partitions Assigned



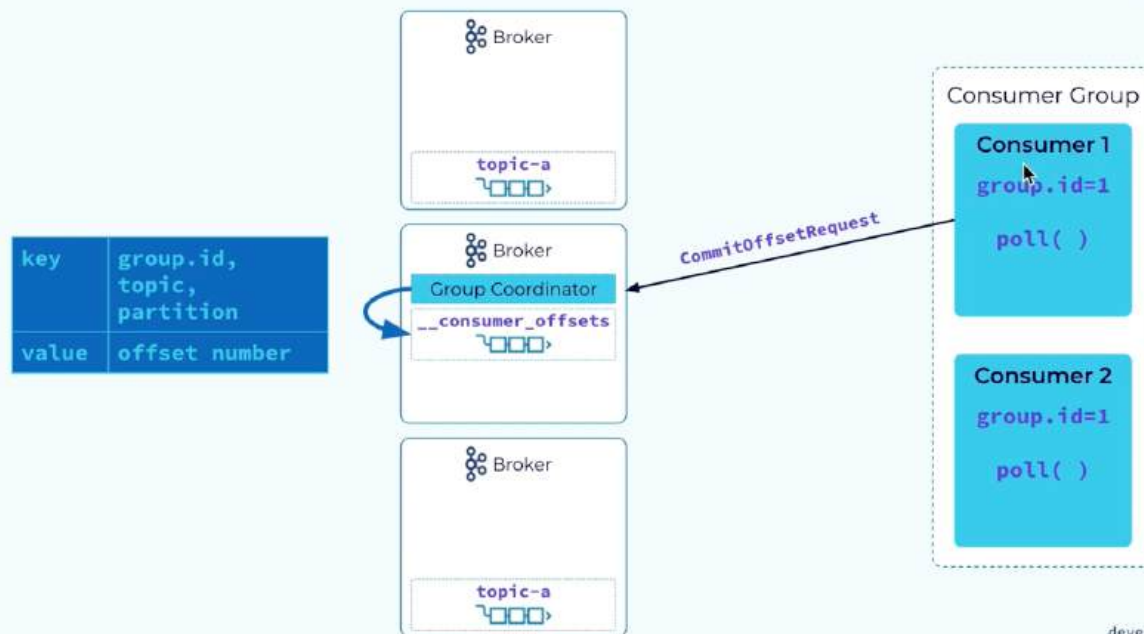
Range Partition Assignment Strategy



Round Robin and Sticky Partition Assignment Strategies

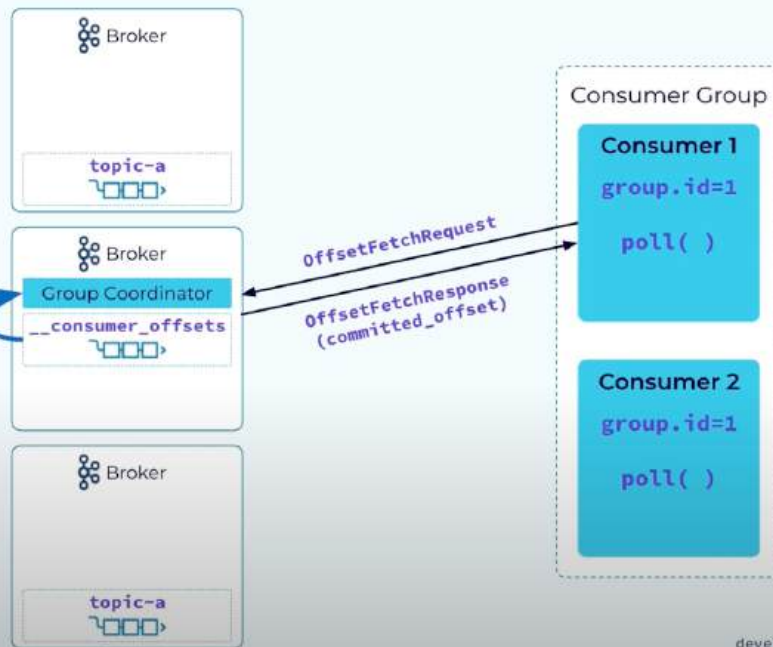


Tracking Partition Consumption



Determining Starting Offset to Consume

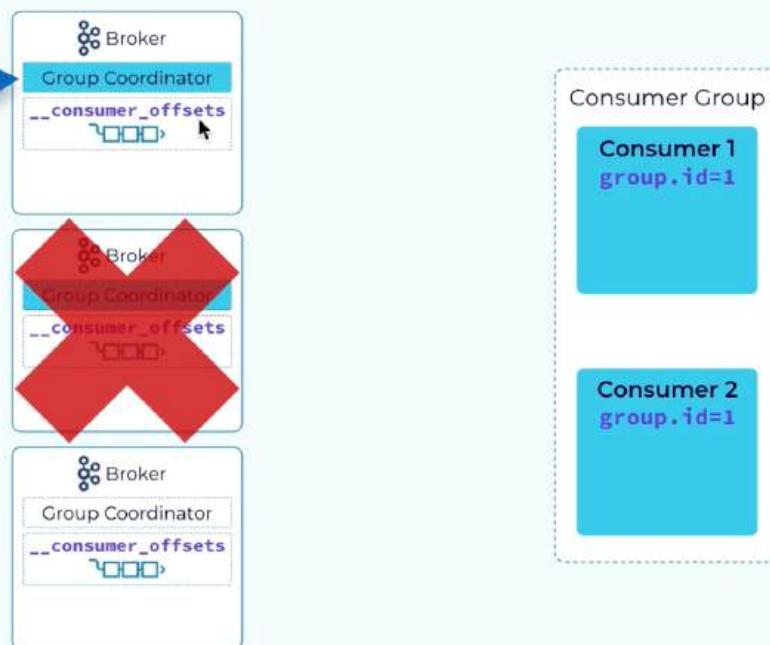
If no committed offset is available, `auto.offset.reset` value determines starting offset



developer.confluent.io

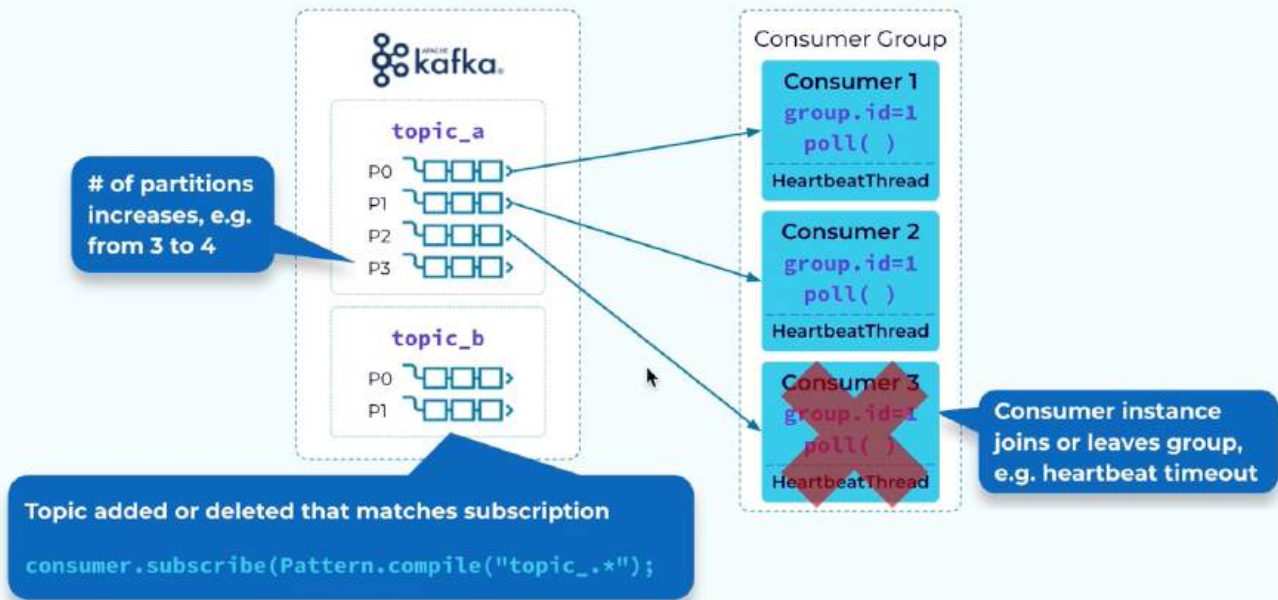
Group Coordinator Failover

Group coordinator fails over to `__consumer_offsets` new partition leader

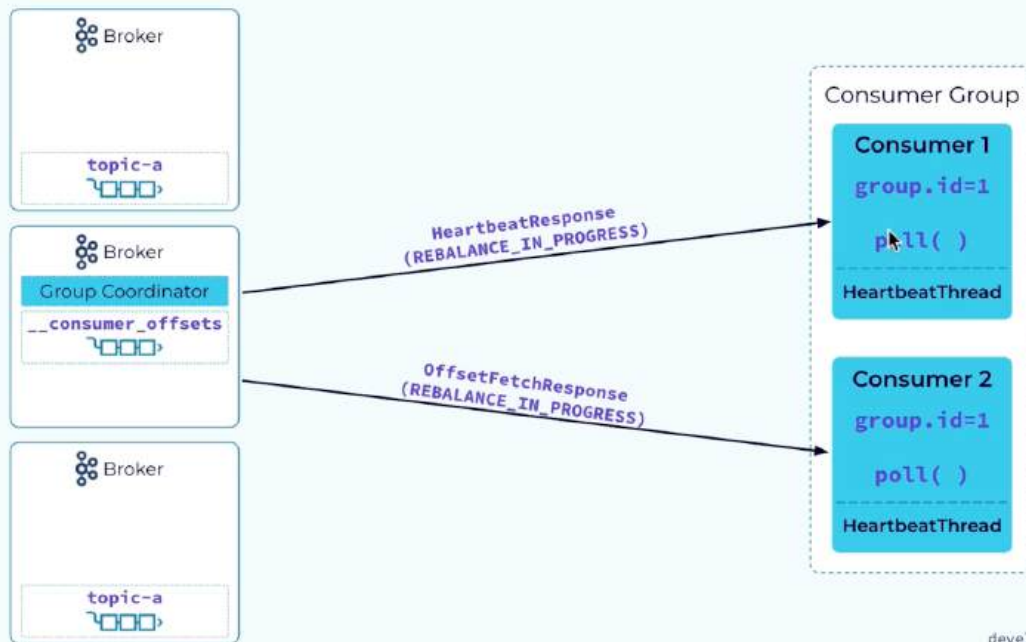


developer.confluent.io

Consumer Group Rebalance Triggers



Consumer Group Rebalance Notification



Offset:

- current offset: sent records to consumer
- committed offset: processed records by consumers
- Let's say,
enable.auto.commit = true &
auto.commit.interval.ms = 5000 msec
Consumer receives 10 messages so it increases offset to 10,
And it takes 4 secs to process these 10 msgs and make a new call,

but as the interval is 5 sec, the consumer will not commit the offset.

Now, rebalance is triggered at this moment. So, the partition goes to new owner

First 10 messages are processed but nothing is committed yet.

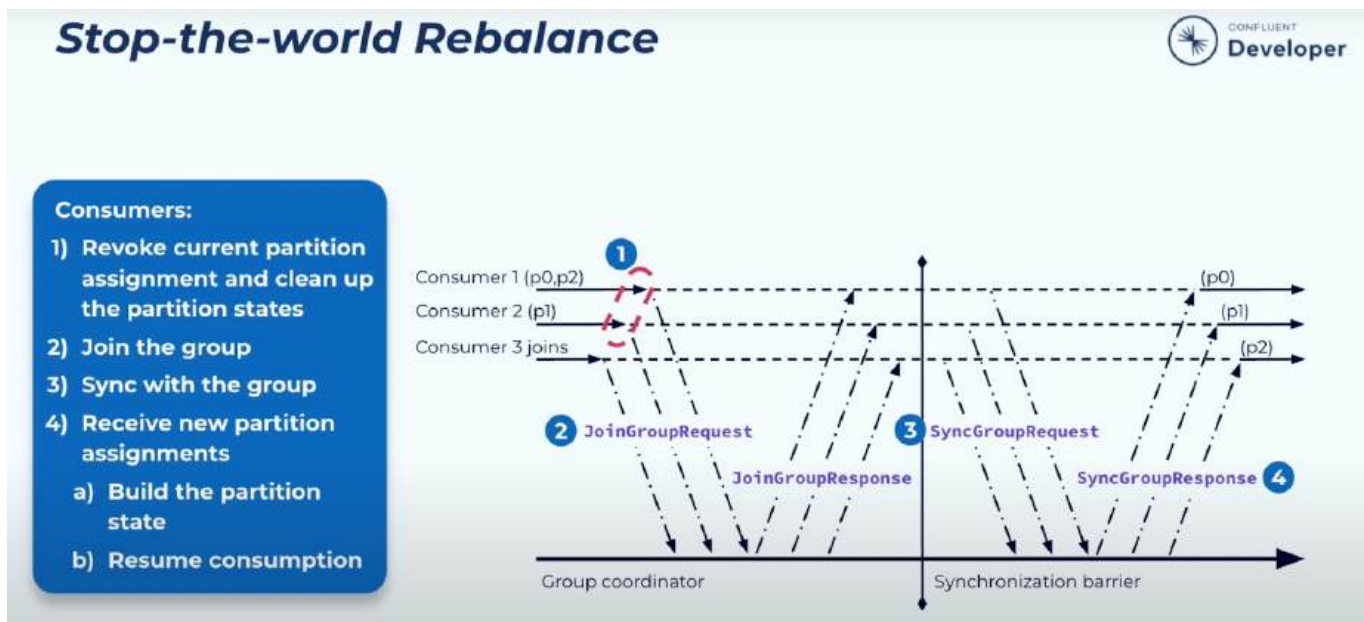
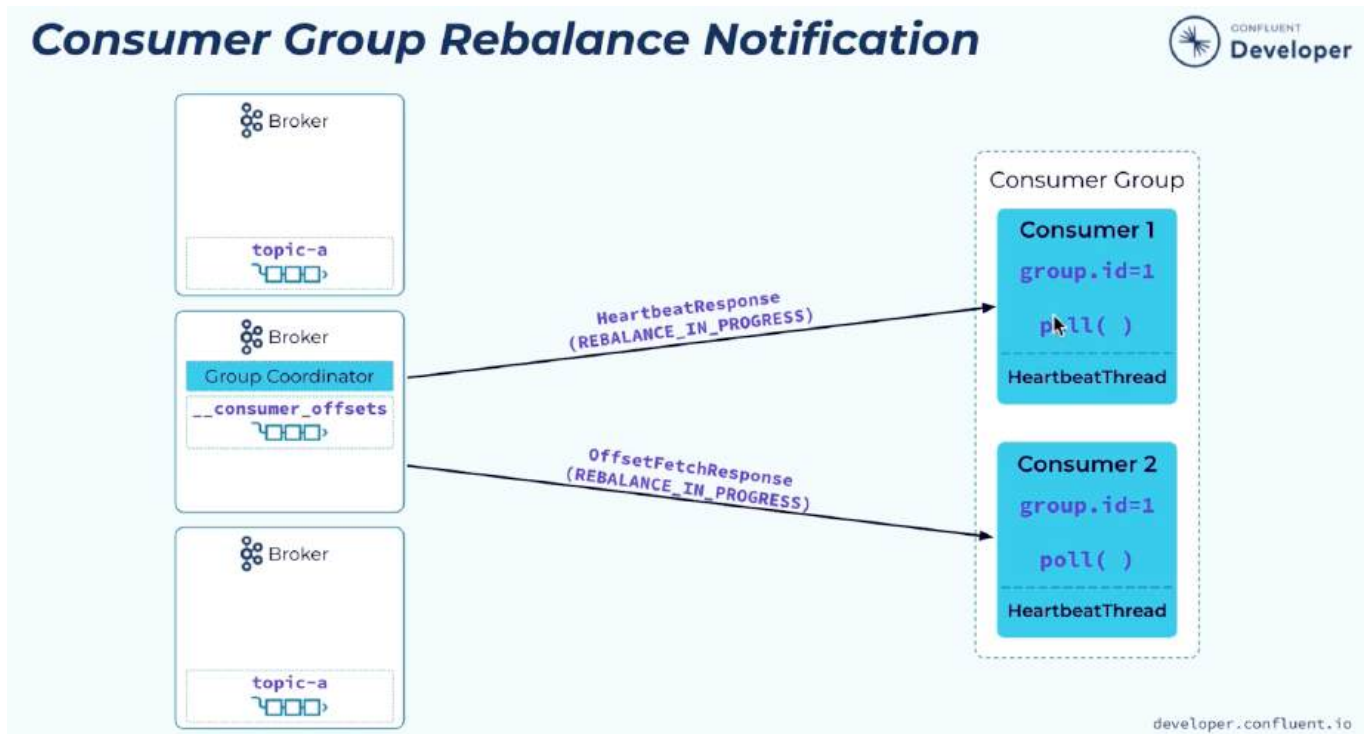
So, the new owner will start processing from the 1st message.

Solution:

Manual commit (`enable.auto.commit = false`)

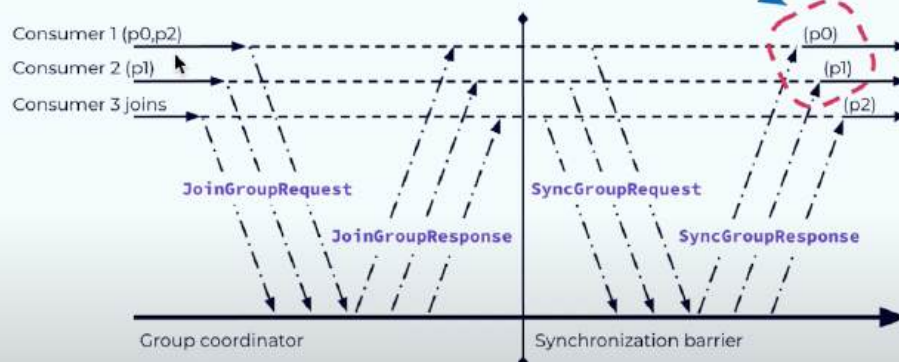
- sync commit
- async commit

Rebalance:



Stop-the-world Problem 1 - Rebuilding the State

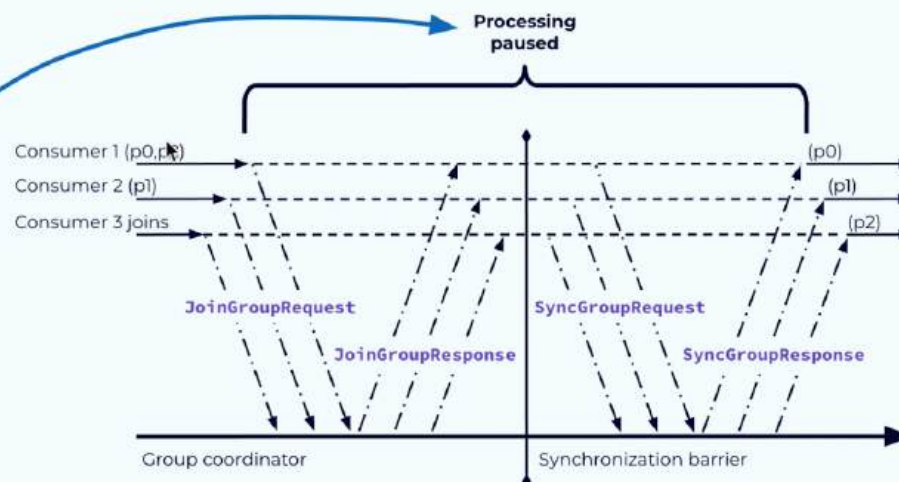
Since partitions p0 and p1 are assigned to the same consumer instance, rebuilding the state is unnecessary



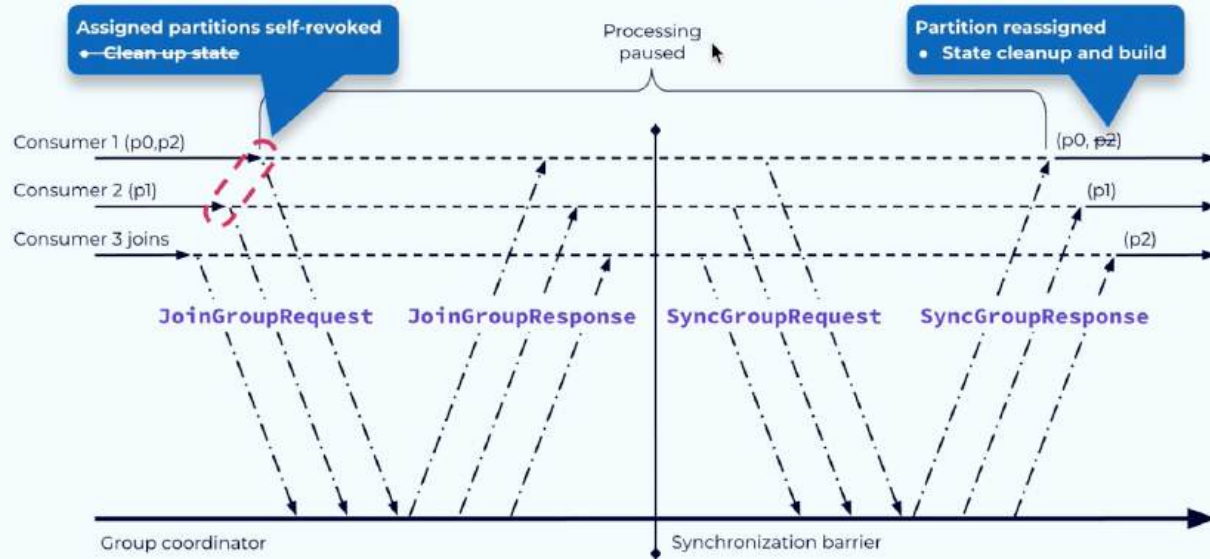
Stop-the-world Problem 2 - Paused Processing

Processing pauses for all subscribed partitions for the duration of the rebalance

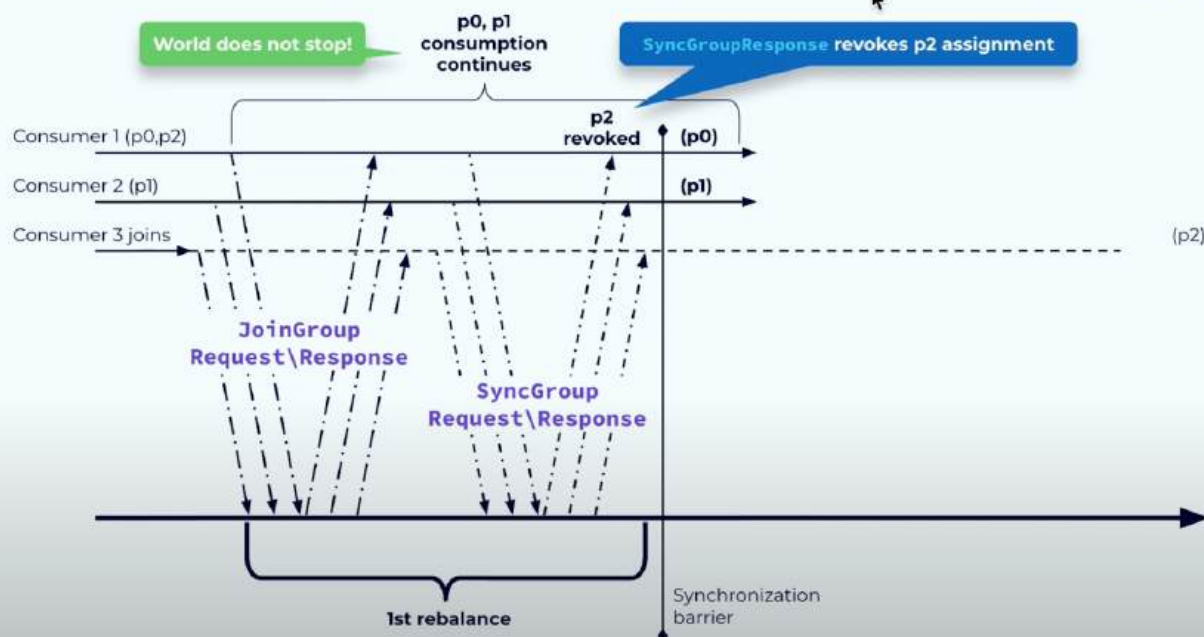
- The pausing for p0 and p1 is unnecessary



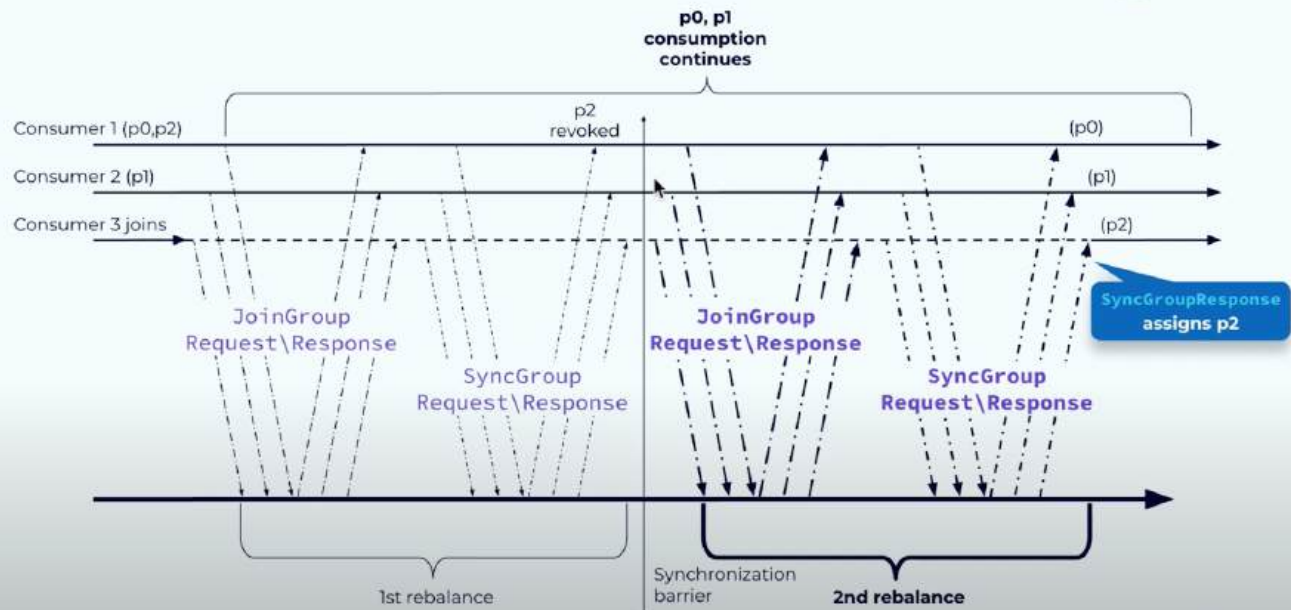
Avoid Needless State Rebuild with StickyAssignor



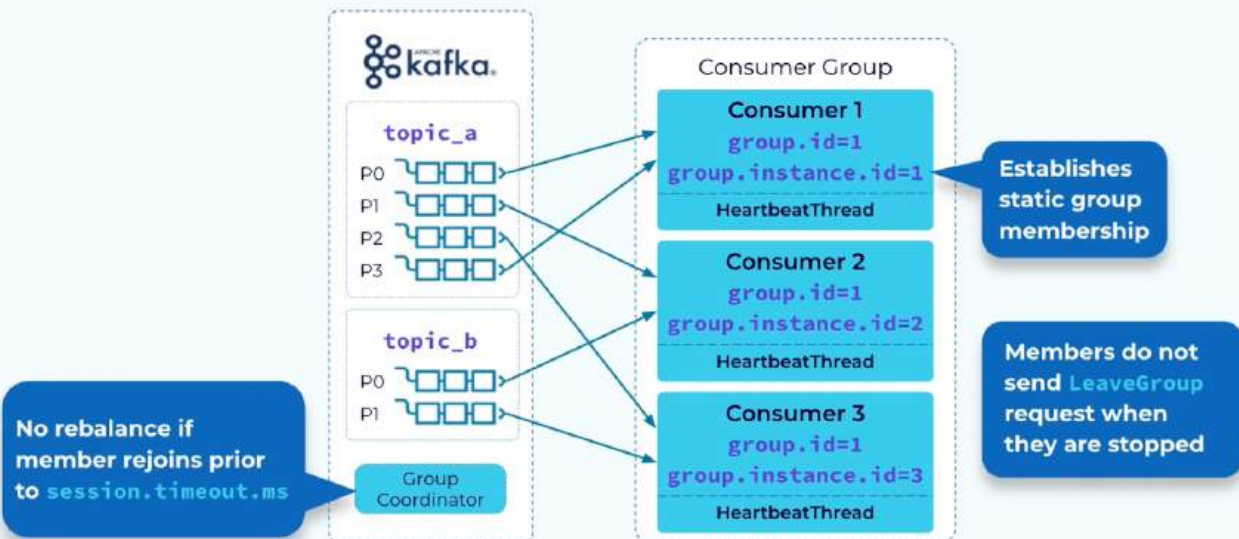
Avoid Processing Pause with CooperativeStickyAssignor

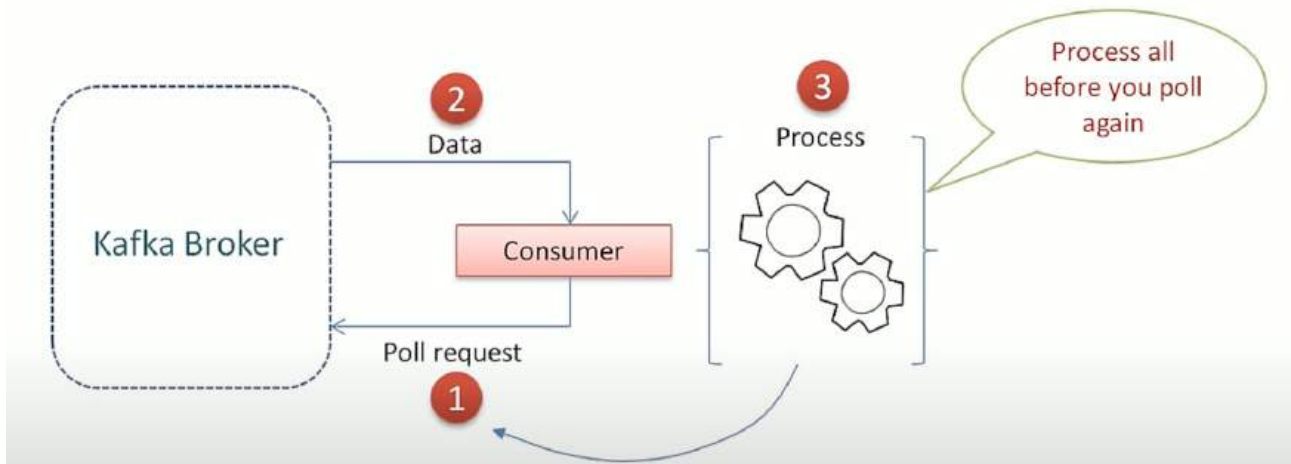


Avoid Processing Pause with CooperativeStickyAssignor



Avoid Rebalance with Static Group Membership





- delay in next poll, rebalance is triggered.

ConsumerRebalanceListener

1. onPartitionsRevoked
2. onPartitionsAssigned

ConsumerRebalanceListener

1. Maintain a list of offsets that are processed and ready to be committed.
2. Commit the offsets when Partitions are going away.

```
import java.util.*;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.*;

public class RandomConsumer{

    public static void main(String[] args) throws Exception{

        String topicName = "RandomProducerTopic";
        KafkaConsumer<String, String> consumer = null;

        String groupName = "RG";
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092,localhost:9093");
        props.put("group.id", groupName);
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("enable.auto.commit", "false");

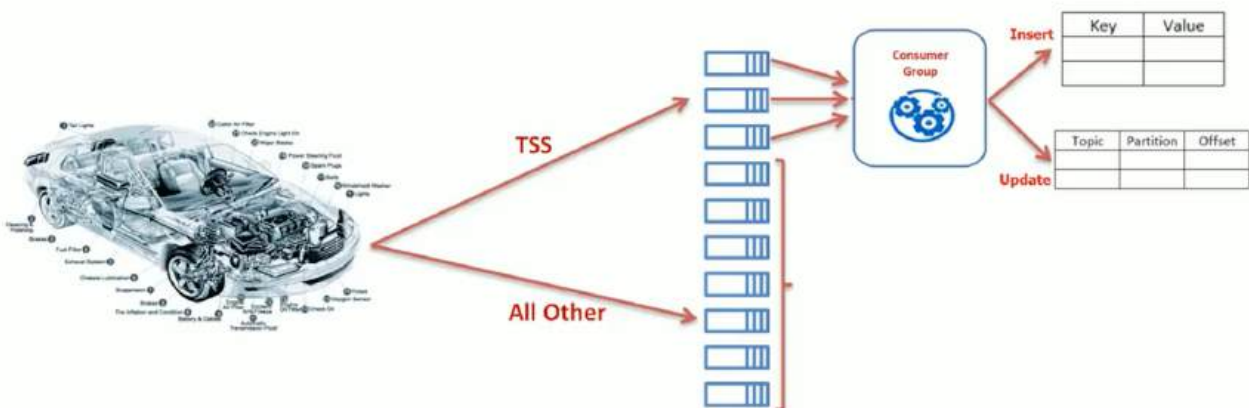
        consumer = new KafkaConsumer<>(props);
        RebalanceListener rebalanceListener = new RebalanceListener(consumer);

        consumer.subscribe(Arrays.asList(topicName), rebalanceListener);
        try{
            while (true){
                ConsumerRecords<String, String> records = consumer.poll(100);
                for (ConsumerRecord<String, String> record : records){
                    //System.out.println("Topic:" + record.topic() + " Partition:" + record.partition() + " Offset:" + record.offset());
                    // Do some processing and save it to Database
                    rebalanceListener.addOffset(record.topic(), record.partition(), record.offset());
                }
                //consumer.commitSync(rebalanceListener.getCurrentOffsets());
            }
        }catch(Exception ex){
            System.out.println("Exception:");
            ex.printStackTrace();
        }
        finally{
            consumer.close();
        }
    }
}
```

- The issue arises when consumer crashes after it has processed records but before it has committed the offsets.
- Upon restart (or after rebalance). the new consumer starts consuming from the last committed offset, which could be before the last successfully processed record. and leads to reprocessing some records.
- **Solution:**
 - 1. atomic offset with processing
 - 2. Idempotent Processing (Use unique identifier for records so that even if a msg is processed multiple times, the outcome remains consistent.)
 - 3. Kafka Transaction (Exactly-once semantics)
 - 4. Rebalance Listener Cleanup
 - this is typically helpful when consumer is about to be rebalanced and will lose ownership of a partition.
 - this can help in avoiding losing uncommitted offsets before partition is reassigned to another consumer.
 - can implement `onPartitionsRevoked` and `onPartitionAssigned` methods to commit offsets and clean up resources during a rebalance.

We need to take control of following

1. Partition assignment.
- ➔ 2. Committed Offset



- we will not send offset back to kafka, instead we will maintain another table for this.

1. Create a Database.
2. Create a table to insert TSS data.
3. Create a table to update TSS offsets.
- ➔ 4. Insert three rows for 3 TSS partitions.

```

consumer.seek(p0, getOffsetFromDB(p0));
consumer.seek(p1, getOffsetFromDB(p1));
consumer.seek(p2, getOffsetFromDB(p2));
System.out.println("New positions po=" + consumer.position(p0)
    + " p1=" + consumer.position(p1)
    + " p2=" + consumer.position(p2));

System.out.println("Start Fetching Now");
try{
    do{
        ConsumerRecords<String, String> records = consumer.poll(1000);
        System.out.println("Record polled " + records.count());
        rCount = records.count();
        for (ConsumerRecord<String, String> record : records){
            saveAndCommit(consumer,record);
        }
    }while (rCount > 0);
}catch(Exception e){
    System.out.println("Exception in main.");
}
finally{
    consumer.close();
}

```

```

private static void saveAndCommit(KafkaConsumer<String, String> c, ConsumerRecord<String, String> r){
    System.out.println("Topic=" + r.topic() + " Partition=" + r.partition() + " Offset=" + r.offset() + " Key=" + r.key() + " Value=" + r.value());
    try{
        Class.forName("com.mysql.jdbc.Driver");
        Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/test","root","pandey");
        con.setAutoCommit(false);

        String insertSQL = "insert into tss_data values(?,?)";
        PreparedStatement psInsert = con.prepareStatement(insertSQL);
        psInsert.setString(1,r.key());
        psInsert.setString(2,r.value());

        String updateSQL = "update tss_offsets set offset=? where topic_name=? and partition=?";
        PreparedStatement psUpdate = con.prepareStatement(updateSQL);
        psUpdate.setLong(1,r.offset()+1);
        psUpdate.setString(2,r.topic());
        psUpdate.setInt(3,r.partition());

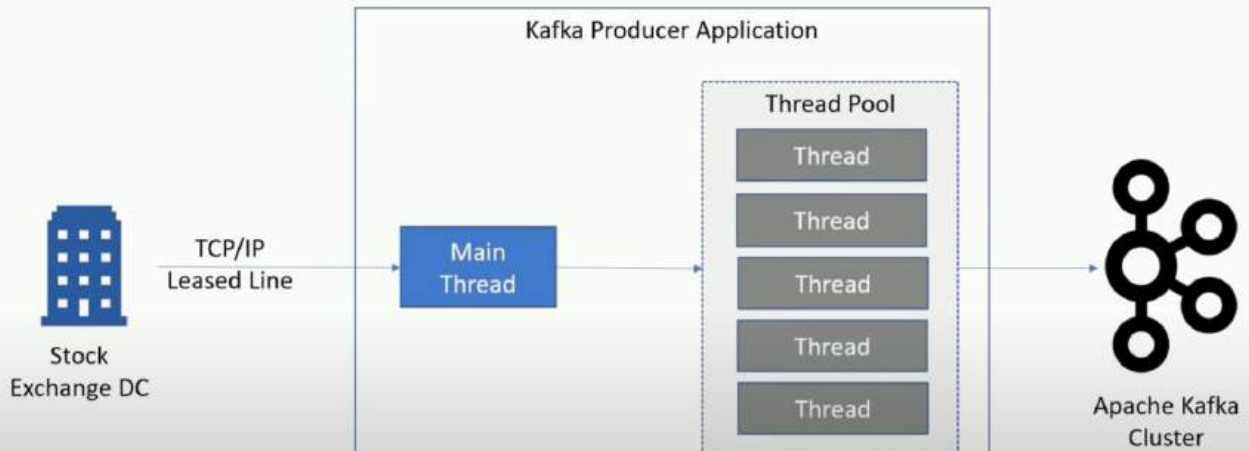
        psInsert.executeUpdate();
        psUpdate.executeUpdate();
        con.commit();
        con.close();
    }catch(Exception e){
        System.out.println("Exception in saveAndCommit");
    }
}

```

- It's exactly once processing scenario.
- Kafka efficiently provides us with at least once processing scenario.
- But for achieving exactly once, we need to do above things.

Scaling Kafka Producer:

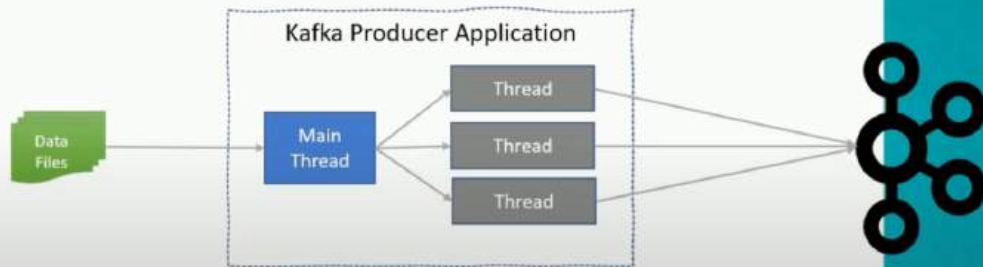
Scaling Kafka Producer



Scaling Kafka Producer

Problem Statement

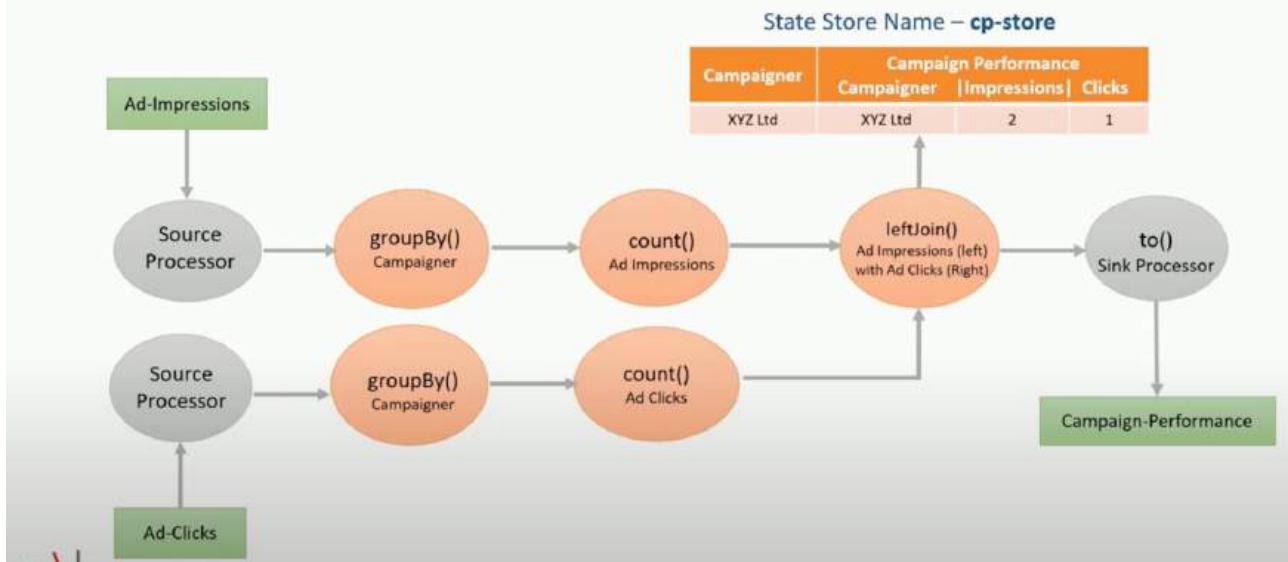
Create a multi-threaded Kafka Producer that sends data from a list of files to a Kafka topic such that independent thread streams each file.



- [Reference code](#)

Test a streaming processing Application:

Unit Testing



This example was an exercise for learners. Joins and Aggregates

Compute Following

1. Number of ad impressions.



```
100001:{"ImpressionID": "100001", "Campaigner": "ABC Ltd"}
100002:{"ImpressionID": "100002", "Campaigner": "XYZ Ltd"}
100003:{"ImpressionID": "100003", "Campaigner": "XYZ Ltd"}
100004:{"ImpressionID": "100004", "Campaigner": "XYZ Ltd"}
100005:{"ImpressionID": "100005", "Campaigner": "XYZ Ltd"}
100006:{"ImpressionID": "100006", "Campaigner": "ABC Ltd"}
100007:{"ImpressionID": "100007", "Campaigner": "ABC Ltd"}
100008:{"ImpressionID": "100008", "Campaigner": "ABC Ltd"}
100009:{"ImpressionID": "100009", "Campaigner": "ABC Ltd"}
100010:{"ImpressionID": "100010", "Campaigner": "ABC Ltd"}
```

Campaigner	Count
ABC Ltd	6
XYZ Ltd	4

2. Number of ad clicks.

```
100009:{"ImpressionID": "100009", "Campaigner": "ABC Ltd"}
100010:{"ImpressionID": "100010", "Campaigner": "ABC Ltd"}
100002:{"ImpressionID": "100002", "Campaigner": "XYZ Ltd"}
```

Campaigner	Count
ABC Ltd	2
XYZ Ltd	1

3. Compute CTR

Campaigner	Count
ABC Ltd	33.33%
XYZ Ltd	25.00%

Advert CTR – Test Cases

1. Test the impression flow from the source topic to the final output topic.

Test Steps	Test Data
Send one impression to the source topic (Ad- Impressions)	100001:["ImpressionID": "100001", "Campaigner": "ABC Ltd"]
Read the record from the final topic (Campaign Performance)	
Validate the Campaigner name	"ABC Ltd"
Validate impression count	1

2. Test the impression count increments correctly.

Test Steps	Test Data
Send second impression to the source topic (Ad- Impressions)	100002:["ImpressionID": "100002", "Campaigner": "ABC Ltd"]
Read the record from the final topic (Campaign Performance)	
Validate the Campaigner name	"ABC Ltd"
Validate impression count	2

3. Test the adClick count increments correctly

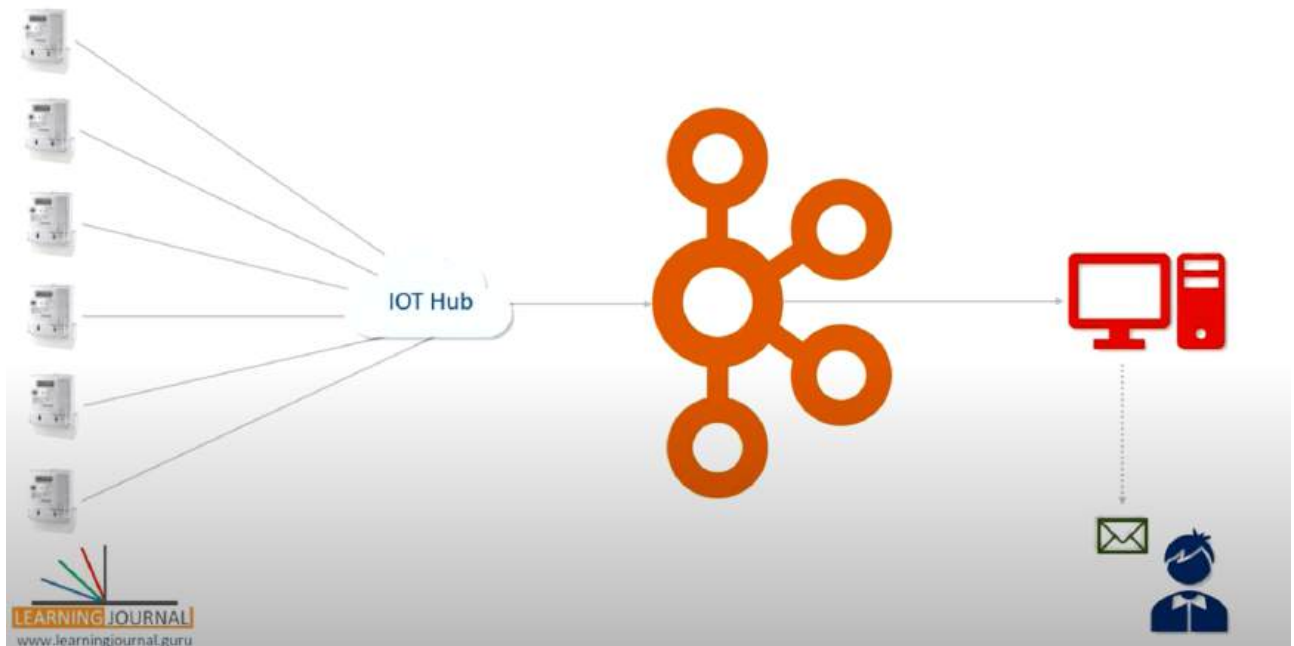
Test Steps	Test Data
Send one ad click event to the source topic (Ad- Clicks)	100001:["ImpressionID": "100001", "Campaigner": "ABC Ltd"]
Read the record from the final topic (Campaign Performance)	
Validate the Campaigner name	"ABC Ltd"
Validate impression count	2
Validate click count	1

4. Test the state store holds the correct state

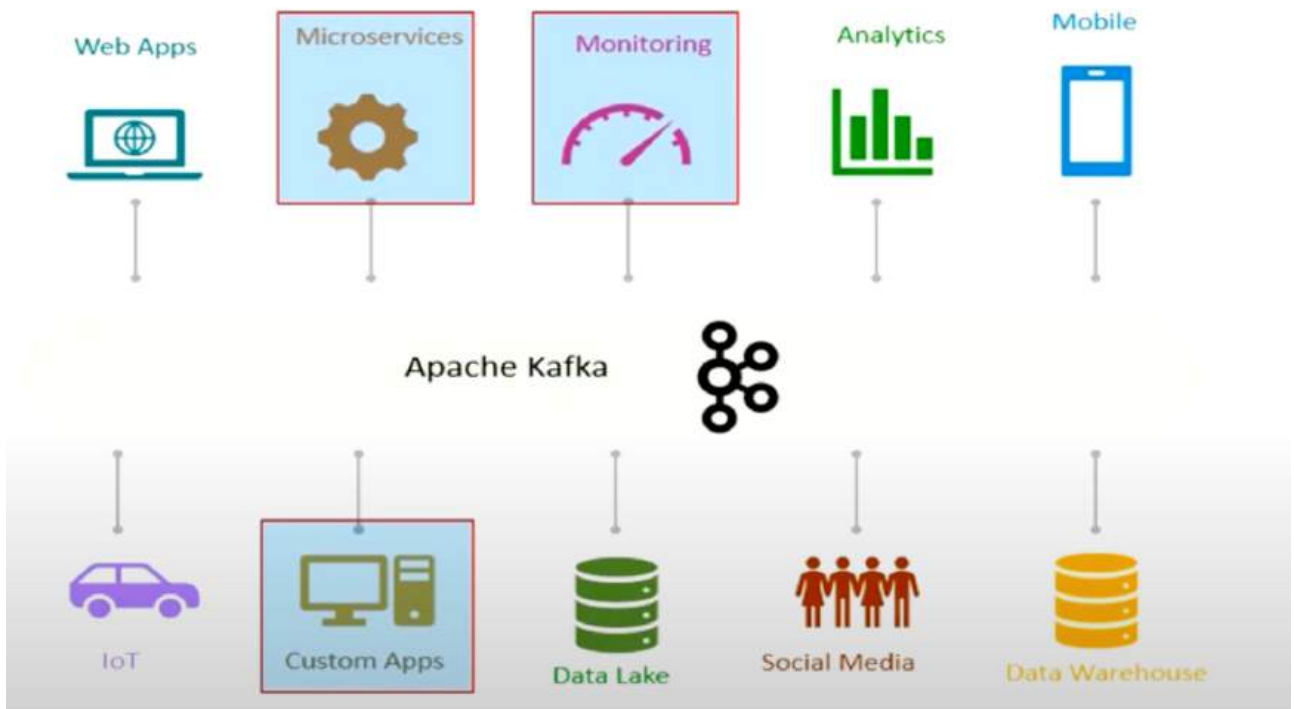
Test Steps	Test Data
Read the value from the state store for a key	Key="ABC Ltd"
Validate the Campaigner name	"ABC Ltd"
Validate impression count	2
Validate click count	1

What is Apache Kafka?

- highly scalable, and distributed platform for creating and processing streams in real-time.

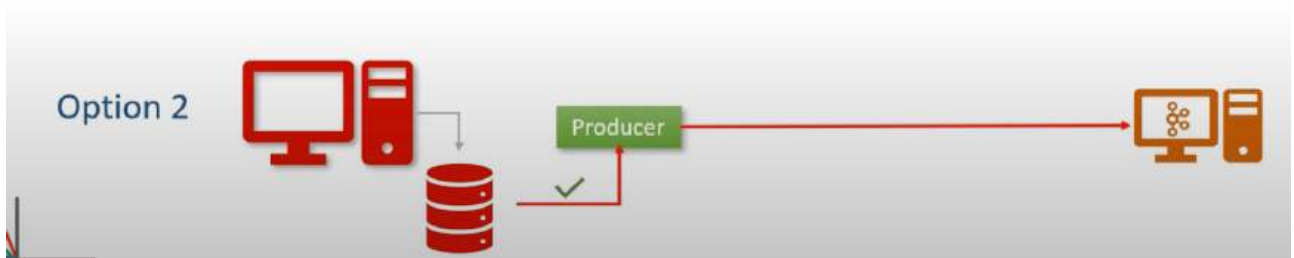


- Let's say , all the smart electricity metres send load related data to the kafka server.
- Which results in continuous streams of data.
- Bringing data from a smart metre to a Kafka server is called a real time stream.
- We have created an application which monitors electricity load in all the houses. As soon as it goes beyond the threshold, then the sms is sent to the owner.
- Which should be quick in a few minutes, there is no point in sending a message after an hour.
- Initially, It was designed to handle the data integration problem.

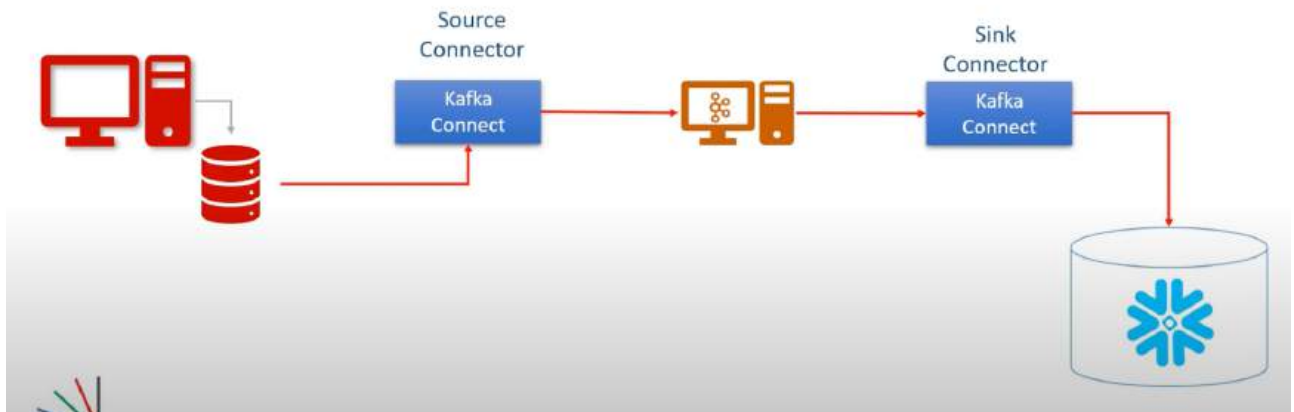


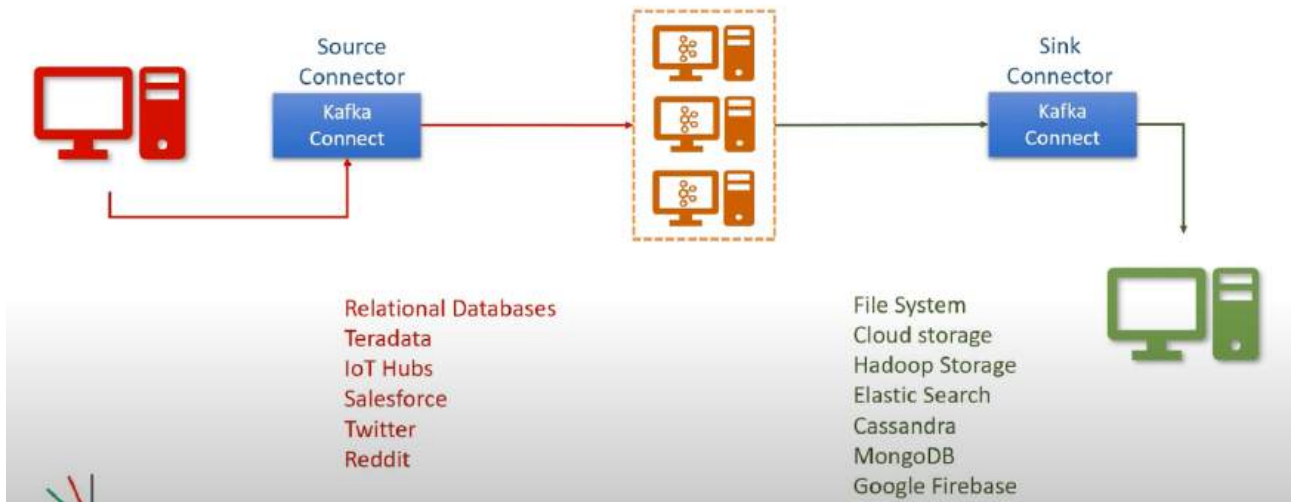
Kafka Connect:

- If we have access to the source code of invoicing app (red) and it is feasible to modify the code then we can create an embedded kafka producer in it and it becomes part of our source application.
- But what if we don't have access to the source code of the application.
 - We can connect the independent producer to source DB, it reads the data and sends it to kafka cluster.



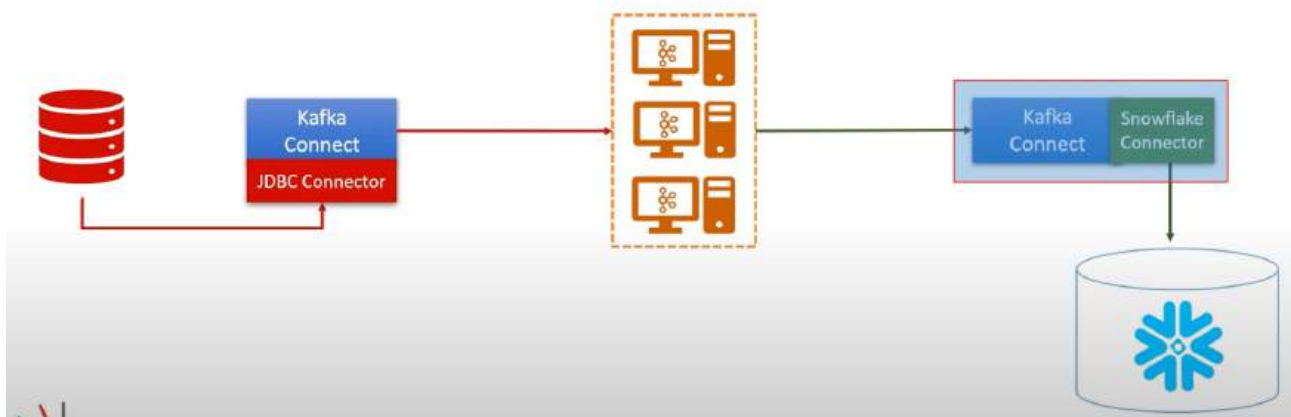
- That's what Kafka connect was designed for, We don't need to write any extra code.



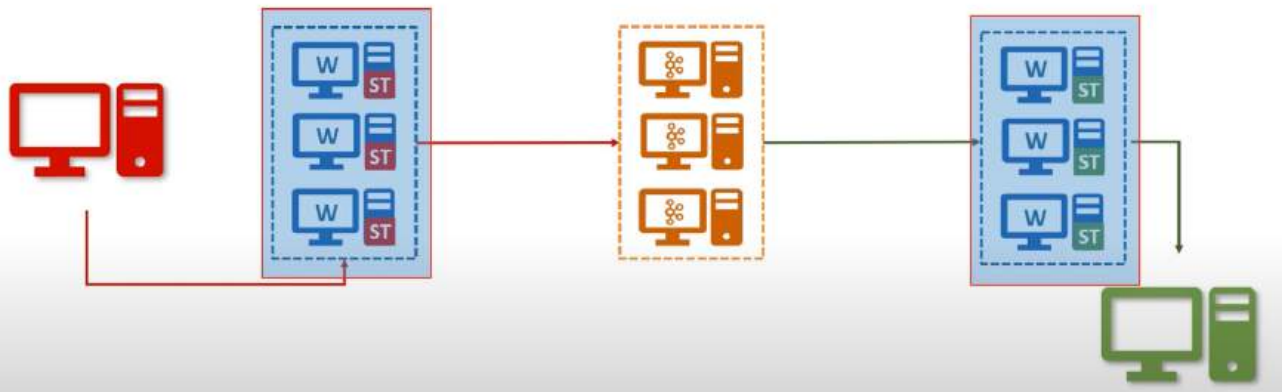


- **Kafka Connect framework:**

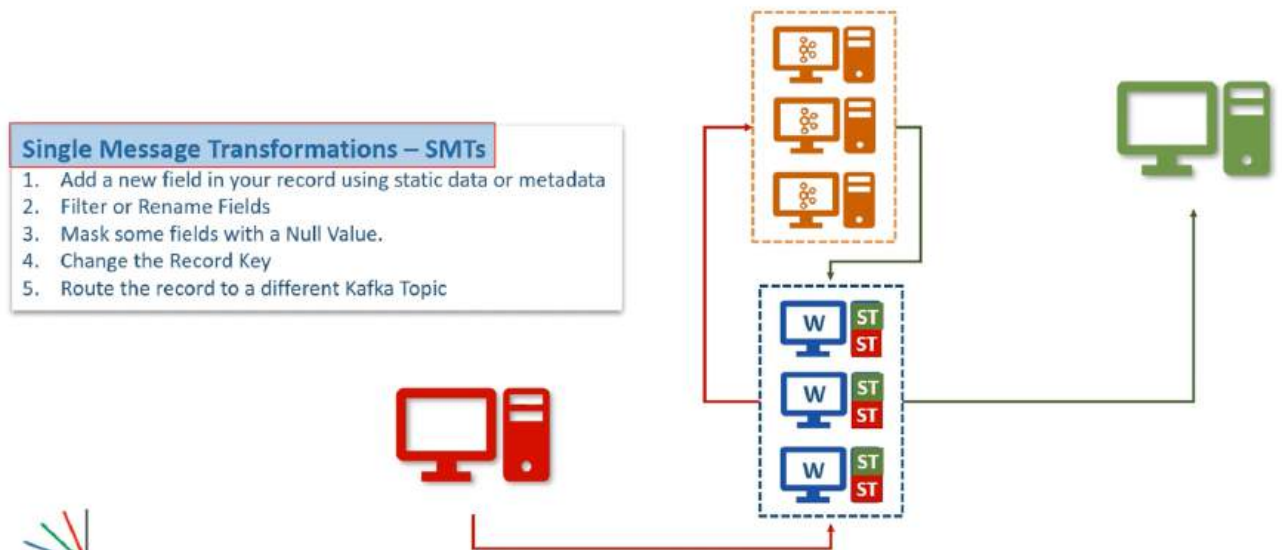
- 1. Source Connector:
 - (i) SourceConnector
 - (ii) SourceTask
- 2. Sink Connector:
 - (i) SinkConnector
 - SinkTask



Kafka Connect – Scalability?



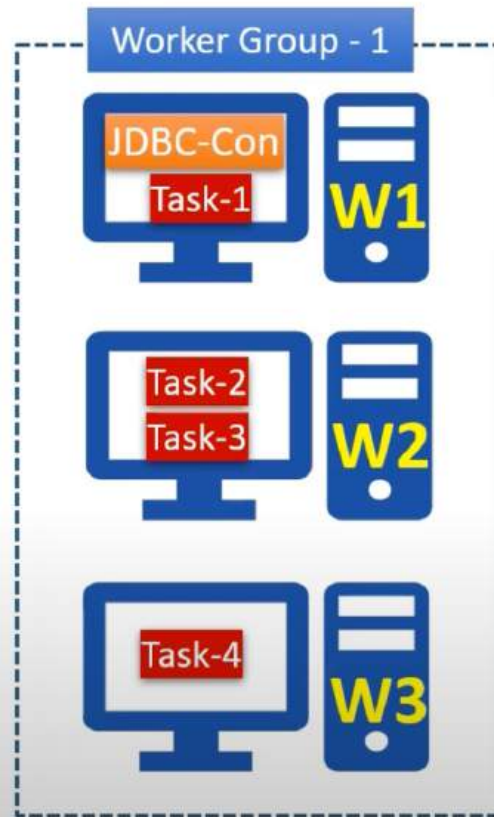
Kafka Connect – Transformations?



- Kafka Connect Architecture:

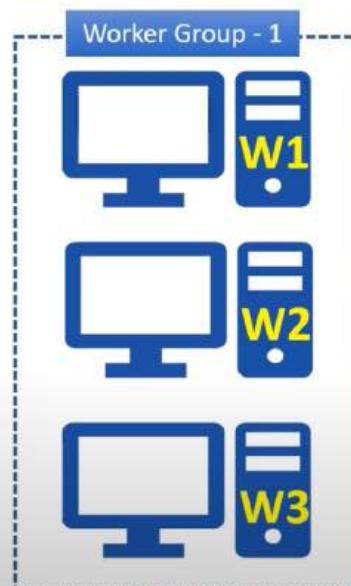
- 1. Worker
 - So, the workers give
 - 1. Readability
 - 2. High AVailability
 - 3. Scalability
 - 4. Load Balancing
- 2. Connector
- 3. Task

Kafka Connect – Architecture?



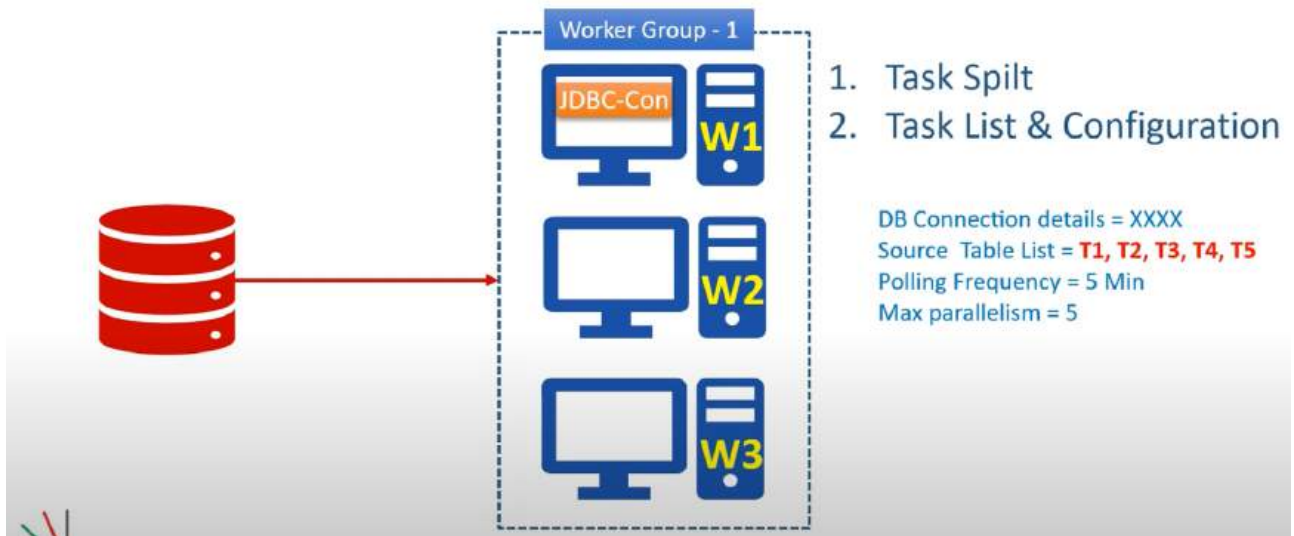
1. Fault Tolerant
2. Self Managed

Kafka Connect – Architecture?



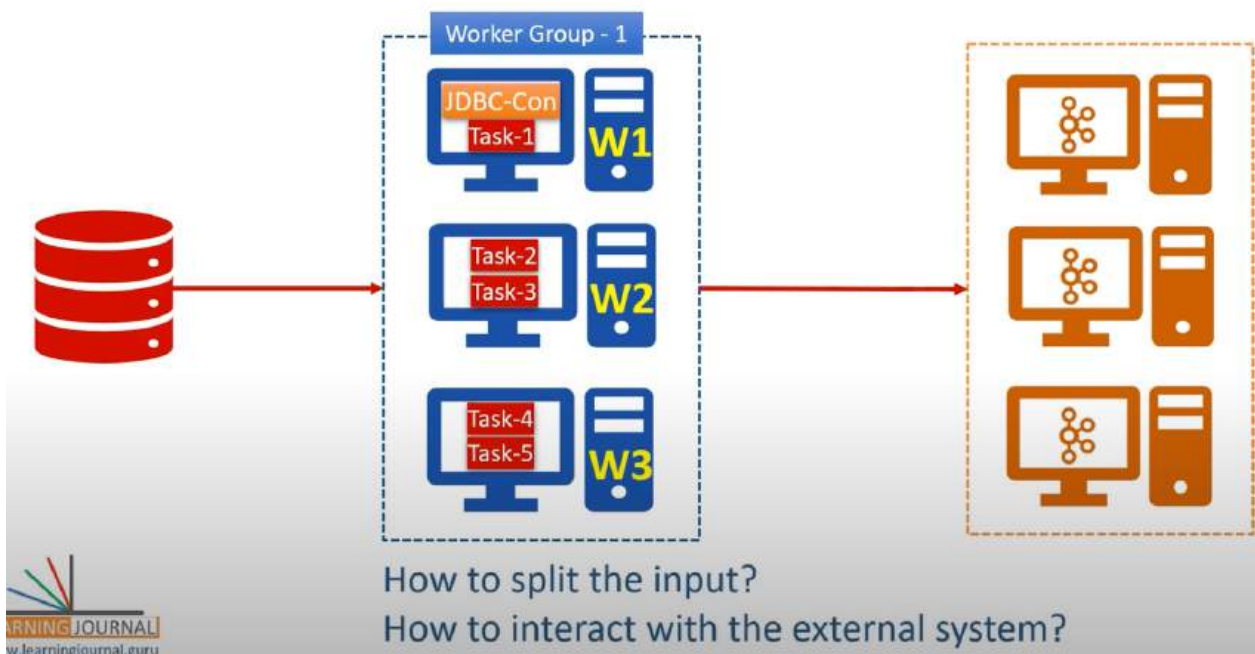
DB Connection details
Source Table List
Polling Frequency
Max parallelism

Kafka Connect – Architecture?



- Tasks are distributed across the workers, they're responsible for connecting to the source system, polling the data at regular intervals, collecting the records and handing over it to the worker. They don't send it to the kafka cluster.
- Task is only responsible to interact with the external system.

Kafka Connect – Architecture?



Kafka Solution Patterns:

Kafka Solution Patterns

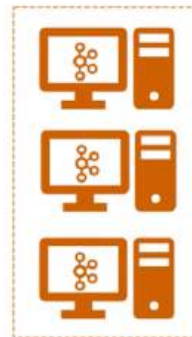


1. Data Integration Pattern
2. Microservice Architecture for Stream Processing
3. Real-time Data Warehousing

Kafka Solution Patterns

1. Enterprise Systems
2. Mobile/Web Apps
3. Cloud Systems
4. Microservices
5. Sensors
6. Log Entries
7. Click Streams
8. Transactions
9. Data Feeds
10. IOT Devices

1. Kafka Client APIs
2. Kafka Connect



Kafka Broker

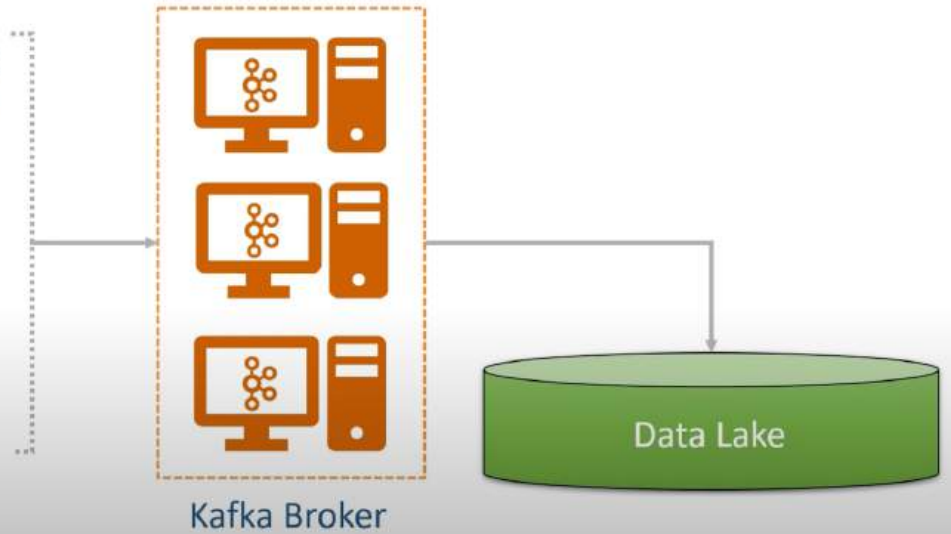
1. Decoupling of producers from the consumers.
2. Reliability and Fault Tolerance
3. Horizontal Scalability
4. Time sensitivity and performance
5. Extensibility



- if we are working with Bespoke/ inhouse application then we will use kafka client APIs and create raw producer and consumer APIs.
- if we are working COTS products then we will use Kafka connect.

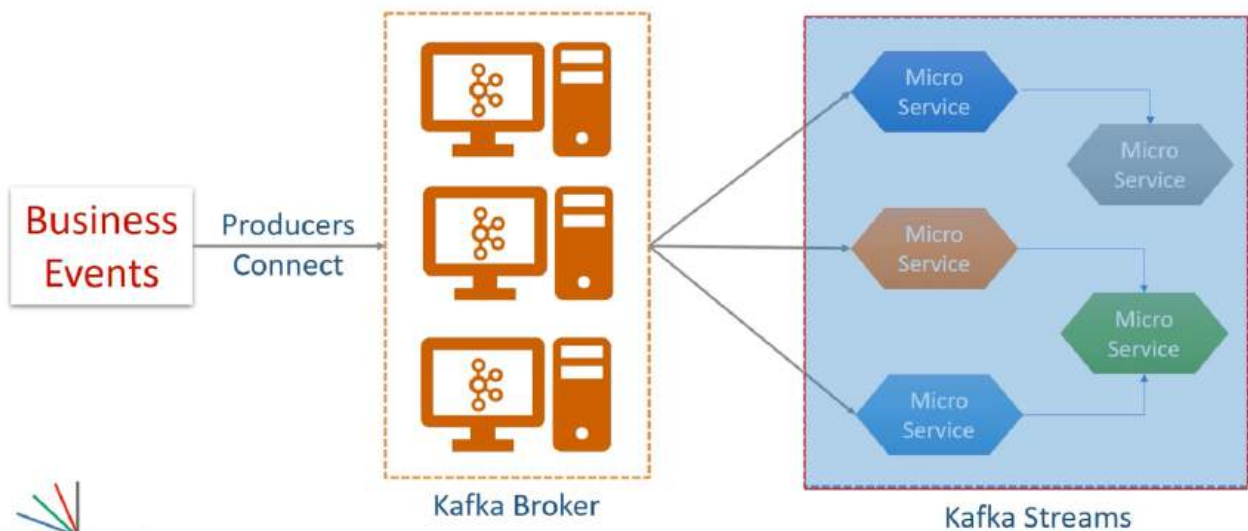
Kafka Solution Patterns

1. Enterprise Systems
2. Mobile/Web Apps
3. Cloud Systems
4. Microservices
5. Sensors
6. Log Entries
7. Click Streams
8. Transactions
9. Data Feeds
10. IOT Devices



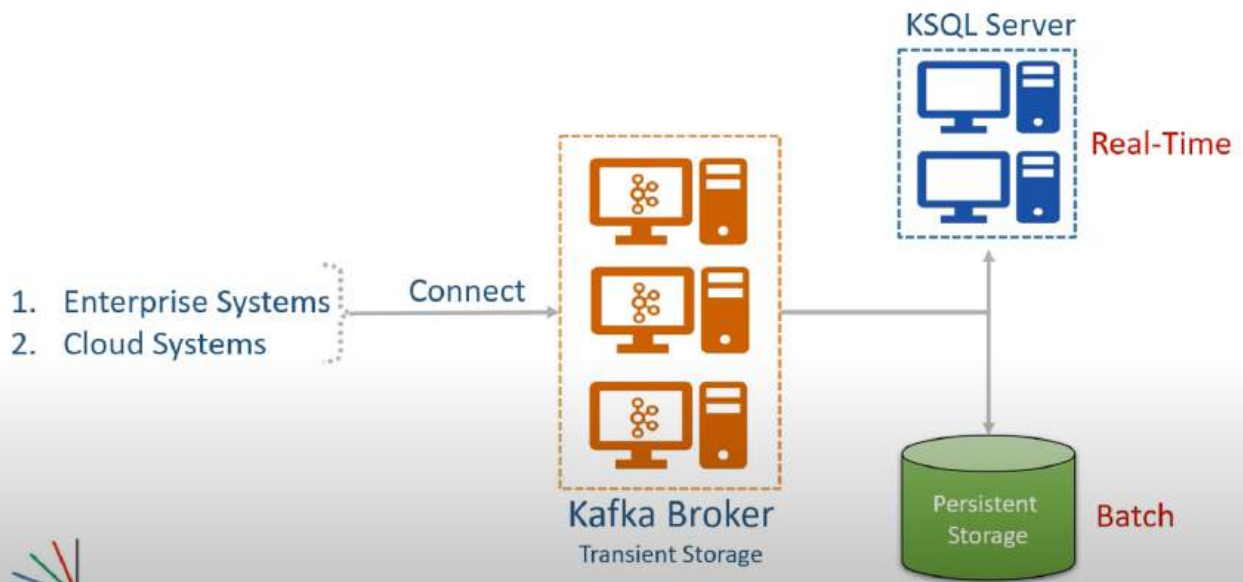
- Once the data is in Data Lake, we can use Apache Spark for the rest of the processing.

Kafka Solution Patterns

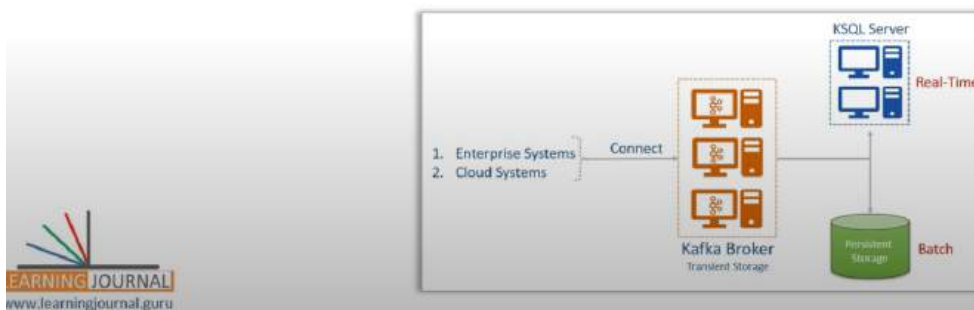
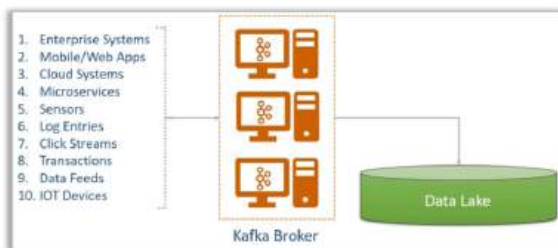


- Kafka connect is not used for microservices, here it will use raw/embedded kafka producer to share data with others.
- We'll use kafka streams for implementing business logic and achieving real time stream processing.
- Kafka consumers are not fit for that purpose and lack most of the capabilities for stream processing requirements.

Kafka Solution Patterns



Kafka Solution Patterns

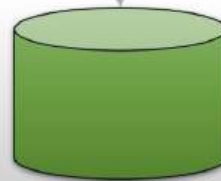


Kafka Streams:

- Unbounded - No definite starting or ending
- sequence of data in small packets (KB)
- eg. Sensors, Log entries, click streams, Transactions, Data Feeds

- 1. Sensors
- 2. Log Entries
- 3. Click Streams
- 4. Transactions
- 5. Data Feeds

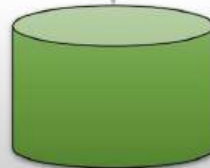
Query?



- Querying the data, and getting a specific answer to the question. Request and response approach, handled through SQS. This approach is all about asking a question one at a time, and getting an answer as quickly as possible.

- 1. Sensors
- 2. Log Entries
- 3. Click Streams
- 4. Transactions
- 5. Data Feeds

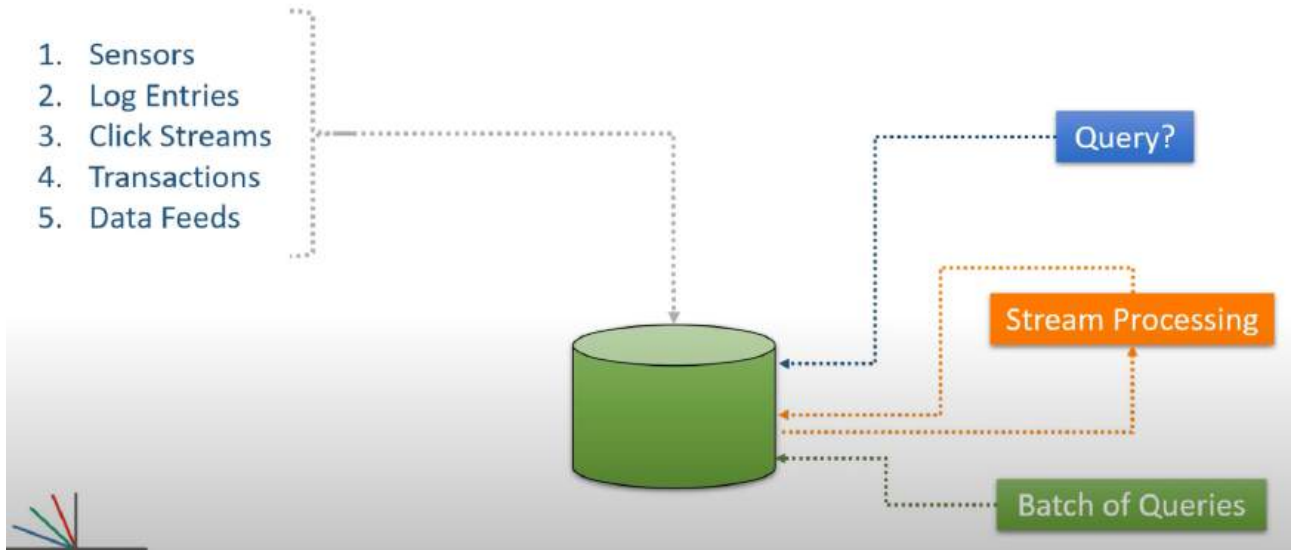
Query?



Batch of Queries

- Batch of Queries, one big job, and schedule a job to run at regular intervals. This approach is all about asking a bunch of questions at once and repeating every hour/day/...

1. Sensors
2. Log Entries
3. Click Streams
4. Transactions
5. Data Feeds



- In Stream Processing, We ask a question once and the system gives the most recent version of answer all the time.
- It is a continuous process and business data is available till the time.
- We can do continuous processing using DB and batch of queries also, but it becomes too difficult.



What is Kafka Streams?

A Java/Scala Library....

Input data must be in Kafka Topic....

You can embed Kafka Streams in your microservices....

Deploy anywhere (No Cluster Needed)...

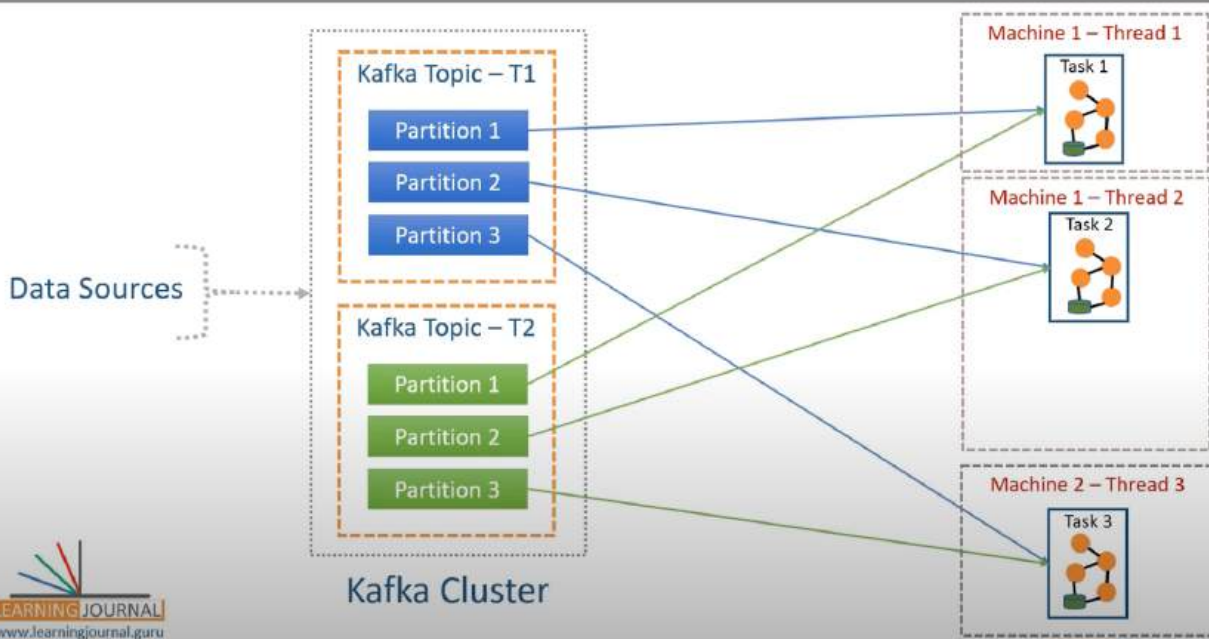
Out of the box parallel processing, scalability, and fault tolerance...



What Kafka Stream Offers?

1. Working with streams/tables and interoperating with them
2. Grouping and Continuously updating Aggregates
3. Join streams, tables, and a combination of both
4. Create and manage fault-tolerant, efficient local state stores
5. Flexible Windowing capability
6. Flexible Time Schematics - Event time, Processing time, Latecomers, High watermark, Exactly-once processing, etc
7. Interactive Query - Serving other microservices
8. Unit testing tools
9. Easy to use DLS and extensibility to create custom processors
10. Inherent fault tolerance and dynamic scalability
11. Deploy in containers and manage using Kubernetes

Kafka Streams - Introduction



- Kafka is a distributed streaming platform which is heavily used for real time data processing and building data pipelines and having an event driven architecture.

Defining Serdes

- Use Serializers and Deserializers (Serdes) to convert bytes of the record to a specific type
 - SERIALizer
 - DESerializer
- Keys Serdes can be independent from value Serdes
- There are many many built-in Serdes (e.g. Serdes.String, etc.)
- You can also define your own, as shown below (e.g. playEventSerde)

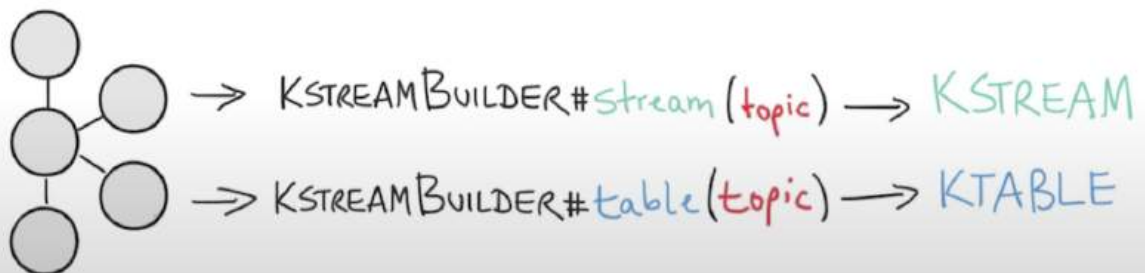
```
...
2 final Map<String, String> serdeConfig = Collections.singletonMap(AbstractKafkaAvroSerDeConfig
  .SCHEMA_REGISTRY_URL_CONFIG, SCHEMA_REGISTRY_URL);
3 final SpecificAvroSerde<PlayEvent> playEventSerde = new SpecificAvroSerde<>();
4
5 playEventSerde.configure(serdeConfig, false);
```

Creating the Streams

- Create a KStream object from one or more Kafka topics (e.g. play-events)

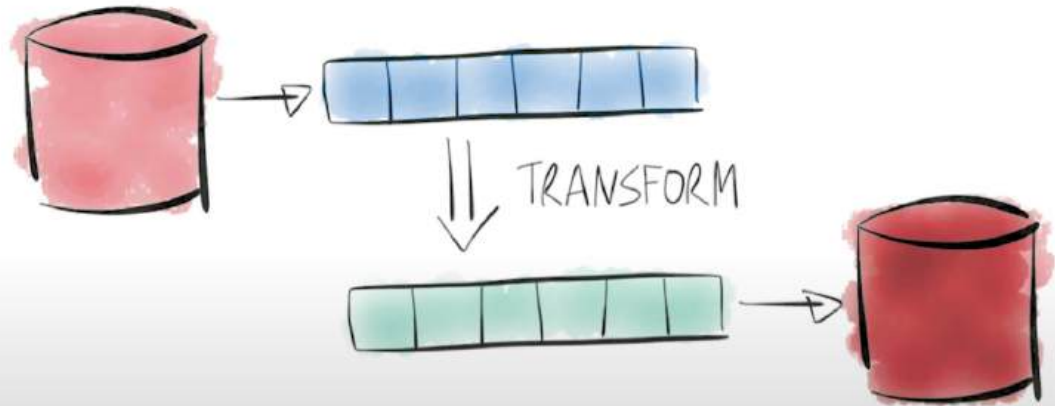
```
1 final KStreamBuilder builder = new KStreamBuilder();
2 final KStream<String, PlayEvent> playEvents =
3     builder.stream(Serdes.String(), playEventSerde, "play-events");
```

- If you want a stream of records, use KStream
- If you want a changelog with only the latest value for a given key, use KTable



Transforming Data

STREAM + TABLE



"ALL PASSENGERS"

"PASSENGERS IN
TRANSIT TO EUROPE"

Stateless Transformation Example

- Transformation example below has two effects:
 - Filters records based on some specified criteria
 - Repartitions based on the new key and value

```
1 final KStream<Long, PlayEvent> playsBySongId =  
2   playEvents.filter((region, event) -> event.getDuration() >= MIN_CHARTABLE_DURATION)  
3   .map((key, value) -> KeyValue.pair(value.getSongId(), value));
```

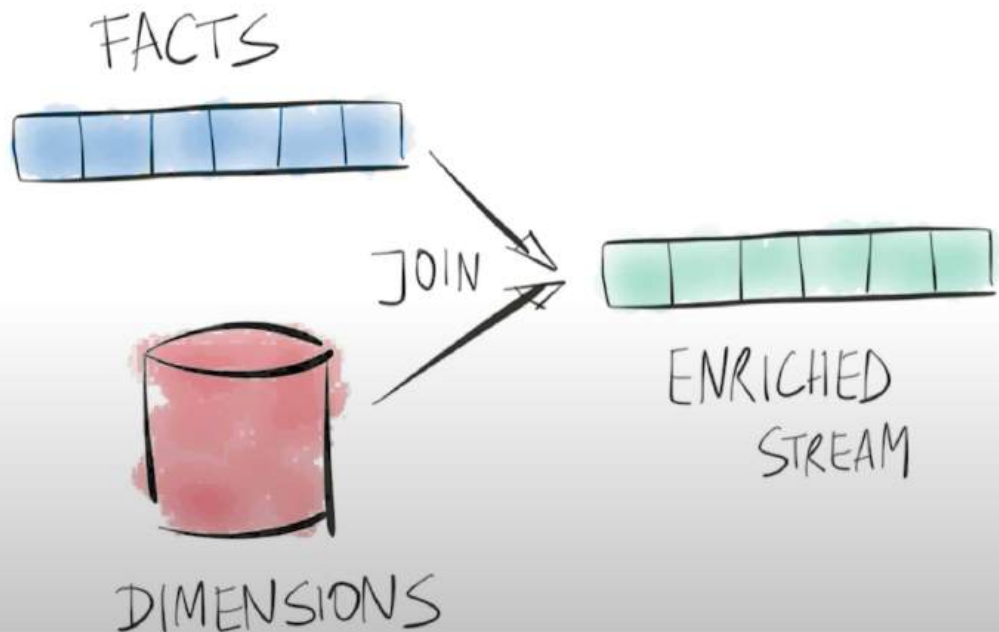
Join Example

- You can enrich records by merging data from different sources
- The example below does a primary key table lookup join
 - Effectively, where `playsBySongId.key == songTable.key`, use the table value `song`

```
1 final KStream<Long, Song> songPlays =  
2   playsBySongId.leftJoin(songTable, (playEvent, song) -> song, Serdes.Long(), playEventSerde);
```

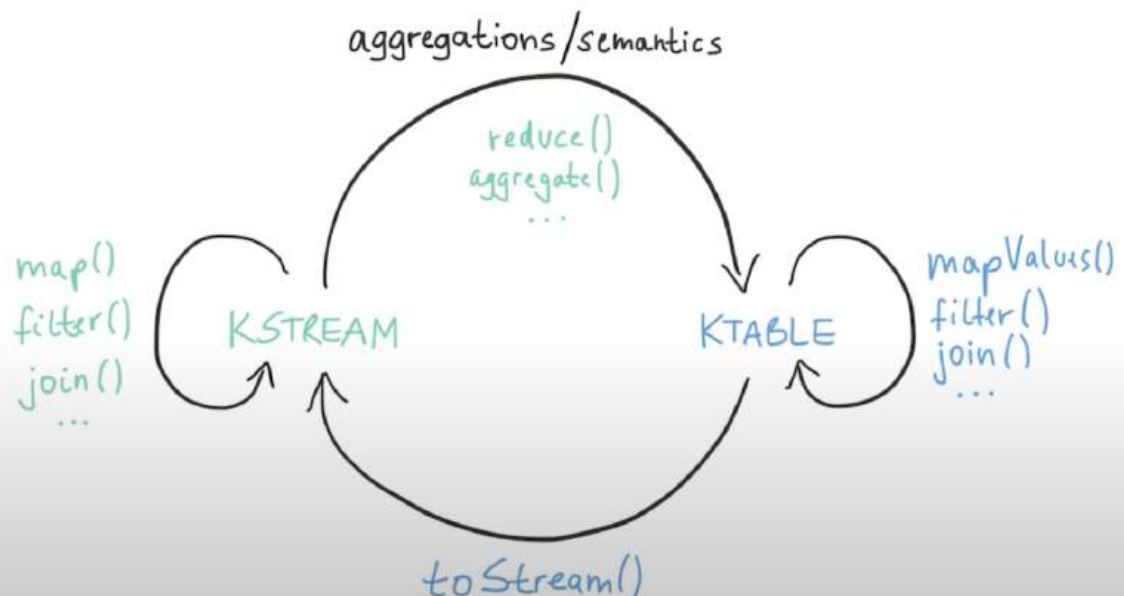
Merging Data

STREAM + TABLE



Stateful processing

- You can build stream processing topologies with mixed stateless and stateful processing



Count

- You may group data based on an existing key or repartition based on a new key
- Example below shows a count method
 - `groupBy`: re-partitions the data based a new key
 - `count`: count the number of occurrences of a song, requires the application to keep state
- Stateful operations can use state stores to store and query data

```
1 final KGroupedTable<Long, Long> groupedBySongId =  
2     songPlays.groupBy((songId, song) -> songId, Serdes.Long(), Serdes.Long());  
3 groupedBySongId.count("song-play-count"); ①
```

① Count results are backed by a state store called "song-play-count"

Reduce or Aggregate

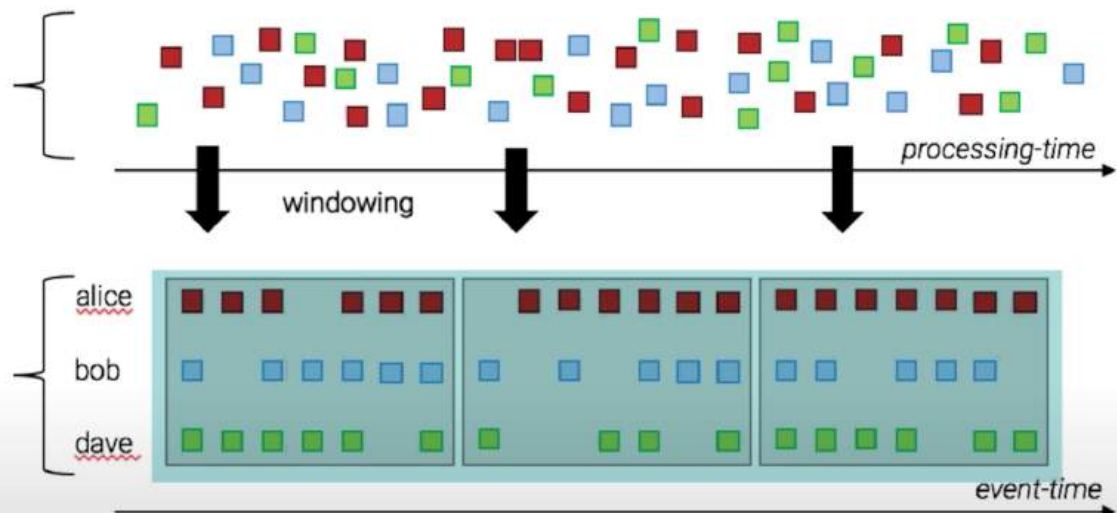
- You can combine current record values with previous record values
- You may use lambda expressions
 - In the example below, `TopFiveSongs` keeps track of the latest top 5 songs and it is automatically updated as new data comes in

```
1 final KTable<Song, Long> songPlayCounts =  
2     songPlaysKGroupedTable.aggregate(TopFiveSongs::new,  
3         (aggKey, value, aggregate) -> { aggregate.add(value); return aggregate; },  
4         (aggKey, value, aggregate) -> { aggregate.remove(value); return aggregate; }, ①  
5         topFiveSerde,  
6         "top-five-songs-by-genre"  
7     );
```

① `KGroupedTable` requires a subtractor (changelogs!), unlike `KGroupedStream`

Windowing

- You can *window* streams of data by time, i.e., group into buckets



- Compare implementation without and with windowing

```
1 groupedBySongId.count("song-play-count");  
2 groupedBySongId.count(TimeWindows.of(TimeUnit.MINUTES.toMillis(5)), "song-play-count-windowed");
```

Processing events with producer and consumer clients



```
public static void main(String[] args) {  
    try(Consumer<String, Widget> consumer = new KafkaConsumer<>(consumerProperties());  
        Producer<String, Widget> producer = new KafkaProducer<>(producerProperties())) {  
        consumer.subscribe(Collections.singletonList("widgets"));  
        while (true) {  
            ConsumerRecords<String, Widget> records = consumer.poll(Duration.ofSeconds(5));  
            for (ConsumerRecord<String, Widget> record : records) {  
                Widget widget = record.value();  
                if (widget.getColour().equals("red") {  
                    ProducerRecord<String, Widget> producerRecord = new ProducerRecord<>(  
                        "widgets-red", record.key(), widget);  
                    producer.send(producerRecord, (metadata, exception)-> {.....} );  
                }  
            }  
        }  
    }  
}
```


Processing events with Kafka Streams



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

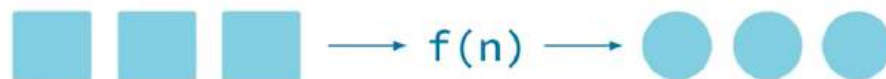
builder.stream("widgets", Consumed.with(stringSerde, widgetsSerde))
    .filter( (key, widget) -> widget.getColour.equals("red"))
    .to("widgets-red", Produced.with(stringSerde, widgetsSerde));
```

Defining A Stream



```
StreamBuilder builder = new StreamBuilder();
KStream<String, String> firstStream =
builder.stream(inputTopic, Consumed.with(Serdes.String(), Serdes.String()));
```

Mapping



```
mapValues(value -> value.substring(5))
map((key, value) -> ..)
```

Filtering




```
filter((key, value) -> Long.parseLong(value) > 1000)
```


Data Serialization - Serdes (Serializer/Deserializer)



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

```
KStream<String, MyObject> stream = builder.stream("topic",  
    Consumed.with(Serdes.String(), customObjectSerde)
```




Serdes used by Kafka Streams
for reading input bytes
into expected object types

Data Serialization - Serdes (Serializer/Deserializer)



```
KStream<String, CustomObject> modifiedStream =  
    stream.filter( (key, value) -> value.startsWith("ID5"))  
    .mapValues( value -> new CustomObject(value));
```

```
modifiedStream.to("output-topic", Produced.with(Serdes.String(), customObjectSerde);
```



Serdes used by Kafka Streams
for serializing objects into bytes

Data Serialization - Serdes (Serializer/Deserializer)



- Pre-existing serdes: String, Integer, Double, Long, Float, Bytes, ByteArray, ByteBuffer, UUID, and Void
- Additional Serdes available for working with Avro, Protobuf, and JSONSchema
 - Avro
 - SpecificAvroSerde
 - GenericAvroSerde
 - Protobuf
 - KafkaProtobufSerde
 - JSONSchema
 - KafkaJsonSchemaSerde

Joins - Types Available



- Stream-Stream
 - Inner - Only if both sides are available within the defined window is a joined result emitted
 - Outer - Both sides always produce an output record
 - Left-value + Right-value
 - Left-value + Null
 - Null + Right-value
 - Left-Outer - The left side always produces an output record
 - Left-value + Right-value
 - Left-value + Null

Joins - Example



```
KStream<String, String> leftStream = builder.stream("topic-A");
KStream<String, String> rightStream = builder.stream("topic-B");

ValueJoiner<String, String, String> valueJoiner = (leftValue, rightValue) -> {
    return leftValue + rightValue;
};

leftStream.join(rightStream,
                valueJoiner,
                JoinWindows.of(Duration.ofSeconds(10)));
```

Stateful Operations



- Stateless operations are great for operations where you don't need to remember
 - Filter - drop records that don't match a condition
- Other times you need to remember previous results
 - How many times has a particular customer logged in?
 - What's the total sum of tickets sold?
- For those situations where the previous state of an event is important, Kafka Streams offers stateful operations

Stateful Operations - Aggregation



```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> myStream = builder.stream("topic-A");

Aggregator<String, String, Long> characterCountAgg =
    (key, value, charCount) -> value.length() + charCount;
myStream.groupByKey().aggregate(() -> 0L,
                                characterCountAgg,
                                Materialized.with(Serdes.String(),
Serdes.Long()))
                                .toStream().to("output-topic");
```

•

Stateful Operations - Considerations



- In Kafka Streams stateful operations don't emit results immediately
- Internal caching buffer results
 - Factors controlling when cache emits records
 - Cache is full (10MB by default)
 - Commit interval (30 seconds)
 - To see all updates, set cache size to zero (also for debugging)

•

Windowing:

- is a technique to used to group events that occur within a specific time period or window, so that we can apply stateful operations (such as aggregation or joins) on those groups.
- the main goal is to control how records are grouped together for processing, enabling operations like sum, count, avg, and joins to be computed over a specific time period rather than on the entire system.
- windowing helps to make sense of the data by defining **time-based slices** over which processing occurs.
- For example, if you wanted to count events over the last **10 minutes**, you need a way to group events that happened within that time range. Windowing provides a mechanism for such operations.
- **1. Tumbling Window:**
 - fixed size, non-overlapping window.
 - Example:
 - **Window size:** 5 minutes

- Each 5-minute window is independent of the others.
- After 5 minutes, the window closes, and a new 5-minute window opens.
- used when we want to group records into distinct, non-overlapping time periods and process them separately.

• 2. Hopping Window:

- fixed size, that overlaps with adjacent windows.
- used when we want to group events over a fixed time period, but with some overlap between consecutive windows.
- we define a **window size** (like a tumbling window), and you also define a **hop size** — the amount of time the window moves forward after processing.
- Example:
 - **Window size:** 5 minutes
 - **Hop size:** 3 minutes
 - This means a new window will start every 3 minutes, and each window will cover the last 5 minutes.
- If you are calculating the sum of events over the last 5 minutes, but you want each window to overlap by 3 minutes:
 - **Window 1:** 10:00–10:05
 - **Window 2:** 10:03–10:08
 - **Window 3:** 10:06–10:11

• 3. Sliding Window:

- similar to hopping window, but with continuous movement and typically overlapping at smaller intervals.
- Difference between hopping and sliding windows is that in sliding window, the window doesn't "hop" to fixed intervals but can continuously slide with no gap.
- **Example:**
 - **Window size:** 5 minutes
 - **Slide interval:** 1 minute
 - This means that every minute, a new 5-minute window will start, and the windows will overlap almost entirely.
- used for real-time, continuous processing where you want each window to be updated very frequently, such as calculating rolling averages, trending values or continuous metrics.

• 4. Session Window:

- is a dynamic window where the window size is not fixed. Instead it is defined by inactivity gaps (i.e., a time period where no records arrive).

- A session window groups records based on the occurrence of events, and if a certain amount of time passes without any events, window closes.
- **Example:**
 - **Session gap:** 10 minutes
 - If there are 10 minutes of inactivity (no records), the session ends and a new session begins when a new event arrives.
- used when working with events that may have gaps between them but should still be grouped based on their proximity in time. Session windows are useful for activities like user sessions or tracking behavior over time with no constant event arrival.

Time Concepts:

- timestamps are a critical component in Kafka.
- kafka msg format has a dedicated timestamp field.
- **Event-Time:** A producer, including Kafka-streams library, automatically sets this timestamp field if user doesn't.
 - Timestamp is current wall-clock time of the producer env. when the event is created.
- **Ingestion-Time:** can configure Kafka broker to set this timestamp field when an event is appended to (stored in) the topic.
 - Timestamp is the current wall-clock time of the broker env.
- Timestamp of events drive the action in kafka streams.
- Earliest timestamp across all input partitions chosen first for processing.
- Kafka streams uses **TimestampExtractor** interface to get timestamp.
 - Default behavior is to use event timestamp (set by either event producer or kafka broker)
 - Timestamp set by producer (event-time) or broker (ingestion-time)
- If it's desired to use a timestamp embedded in the event "payload" (event key or event value), then provide a custom **TimestampExtractor** interface implementation.
- Time moves forward in kafka streams by these timestamps.
- how long a window remains open depends on timestamps only; it's completely detached from wall-clock time.
- Kafka streams has a concept of **Stream Time**.
- Any Input events with an event-time < stream-time are considered out-of-order.
 - For windowed operations, this means the event-timestamp is less than current stream-time, but is within the window time. (window size + grace period)
 - out-of-order records are accepted and processed.
- The grace period, a per-window setting, defines a cut-off for out-of-order events.

- Any out-of-order events that arrive after grace period are considered **too late**, and thus are ignored and not processed.
- The delay of an event is determined by (stream-time) - (event-timestamp)

Error Handling in Kafka Streams:

- 3 broad categories where errors may occur:
 - Entry: consuming records - network or deserialization errors
 - Processing of records: not in expected format.
 - Exit: producing records - network or serialization errors
- **Error Handling - Consuming:**
 - For errors consuming into Kafka Streams
 - **DeserializationExceptionHandler** interface:
 - Default configuration is **LogAndFailExceptionHandler**
 - Other option is to use **LogAndContinueExceptionHandler**
 - can provide a custom implementation and provide className via Kafka Stream configuration.
- **Error Handling - Processing:**
 - Exceptions from user logic bubble up and shut down the application.
 - Kafka stream provides **StreamUncaughtExceptionHandler**
 - Works for Exception not handled by kafka streams.
 - implementation u provide has 3 options:
 - Replace the thread.
 - Shutdown the individual streams instance
 - Shutdown all streams instances (with the same application id)
- **Error Handling - Producing:**
 - for errors producing from kafka streams
 - **ProductionExceptionHandler** interface
 - can respond with continue processing or fail
 - default configuration is **DefaultProductionExceptionHandler**
 - default option always returns fail.
 - other option: custom impl.
 - only applies to exception not handled by kafka streams **RecordTooLargeException**.