Custom Partitioner

We have already learned about the importance of Kafka partitions and the default partitioner. Kafka topic partitions are a tool to support scalability by evenly distributing data across partitions and assigning partitions in consumer groups for parallel processing.

You can leverage this partitioning in a variety of ways. For example, if you have some experience in database tables, you already understand the importance of partitioning and bucketing your table into smaller chunks and group similar records on some criteria. The default partitioner allows you to distribute your data using the message key. However, in some scenarios, you may want to change the default behaviour and implement your own partitioning algorithm. Kafka producer API permits you to implement your custom partitioning strategy. You can apply custom bucketing or a partitioning approach by implementing your own partitioner class. Let's create an example to understand the mechanics of implementing custom partitioner.

Problem – Partitioned Producer

We want to create a simple producer example that does the following things.

1. Create a topic with exactly two partitions.
2. Create a producer that sends 1000 simple text messages to the given topic.
3. The message key should start from 1 and increment by one for each new message.
4. Create a custom partitioner that sends all messages with even key to the first partition and odd key to the second partition.

This example will help you understand the process of implementing custom partitioner. Once your data is partitioned into two groups, you can leverage Kafka consumer APIs to ensure that all odd messages are processed by one consumer and the even messages are processed by the other consumer. The idea is simple, but the same notion is leveraged heavily by the Kafka Streams API for aggregation and other features. When you want to calculate some aggregate for a key, you must ensure that all the records for the same key value are processed by the same consumer. Partitioning and repartitioning are at the core of achieving accurate aggregations.

Solution - Partitioned Producer

Code Listing below shows the odd/even partitioner implementation.

```
package guru.learningjournal.kafka.examples;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.errors.InvalidTopicException;
import org.apache.kafka.common.record.InvalidRecordException;

import java.util.Map;

public class OddEvenPartitioner implements Partitioner {

    @Override
    public void configure(Map<String, ?> map) {

    }

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

        if ((keyBytes == null) || (!(key instanceof Integer)))
            throw new InvalidRecordException(
                         "Topic Key must have a valid Integer value.");

        if (cluster.partitionsForTopic(topic).size() != 2)
            throw new InvalidTopicException(
                          "Topic must have exactly two partitions");

        return (Integer) key % 2;
    }

    @Override
    public void close() {

    }

}
```

Implementing a custom partitioner as simple as implementing `Partitioner` interface and overriding three methods as shown in *Code Listing.*

The `configure(Map<String, ?> config)` method is called once when the producer instantiates the `OddEvenPartitioner` object. You can use the configure method to read configuration key-value pairs and accordingly customize your serialization. These configurations can be set by the producer properties in the same way as we define producer configurations. In ourcase, we do not have anything to configure

Similarly, the `close()` method is called once when the producer destroys the `OddEvenPartitioner` instance. You can use this method to clean-up any resources if you have. In our example, we do not have anything to clean-up.

The real implementation happens in the `partition()` method. The partition method is invoked for each message record, and it gives you access to the topic name, key, value, and the cluster information. You can implement your partitioning algorithm and finally return an integer value representing the partition number where the message should be delivered. That is all we do in a partitioner, decide the partition number and return it. The algorithm for odd/even partitioner is straightforward. We divide the key by two and return the remainder.

Using Custom Partitioner

Using your custom partitioner is straightforward. All you need to do is to configure the PARTITIONER_CLASS_CONFIG value as shown in below *Code Listing.*

```
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
          OddEvenPartitioner.class);
```

Rest all should be same as a standard producer application. A complete code example as a working project is included in your course material.