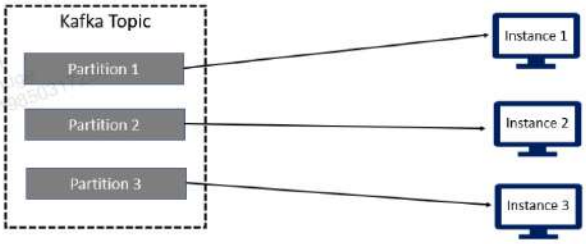

**Apache Kafka**

Kafka Streams is built on top of Kafka Client API. Streams API leverages the native capabilities of Apache Kafka to offer data parallelism, distributed coordination, and fault tolerance.




To understand Kafka Stream's parallelism, we must have a clear understanding of the following concepts.

1. Multi threading in Kafka Streams.
2. Multiple instances of Streams Application.
3. Streams Topology
4. Streams Task

Let's try to understand these ideas independently and then we will tie them together using an example.

**1. Multi threading in Kafka streams-Vertical Scaling**

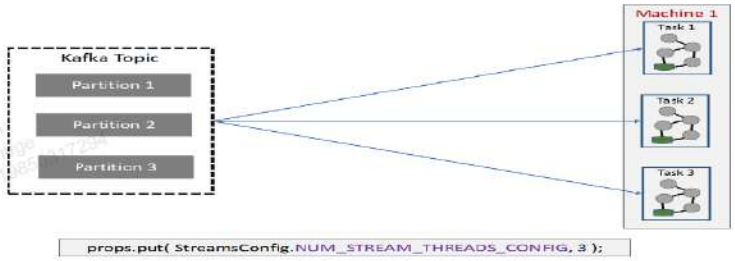
A typical Streams application runs as a single-threaded application by default. However, Kafka Streams allows the user to configure the number of threads that the library can use to parallelize processing within an application instance.

When running multiple threads, each thread executes one or more Stream Tasks (explained below) within the same application instance. Configuring the number of threads is as simple as setting a property as shown below.

```
props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 3);
```




**Apache Kafka**




```
props.put( StreamsConfig.NUM_STREAM_THREADS_CONFIG, 3 );
```

While configuring multiple threads, it is worth noting that there is no shared state amongst the threads. Hence, no inter-thread coordination is necessary. This makes it very simple to run Stream Tasks in parallel across the application threads. All we need is to set the configuration.

**2. Multiple instances of the stream application-Horizontal Scaling**

We can easily configure multiple threads for our application to increase the degree of parallelism. However, all those threads would be running within a single application. **The number of threads that we can start is limited by the available resources on a single machine.**

We can overcome this limitation by launching multiple instances of the same application on a different computer. For example, we can start one application with five threads on one machine and start another instance with three threads on a different machine.

In that case, we will be able to achieve eight Streams Task running in parallel as shown below.


Here Scaling our stream, processing an application with Kafka Streams is easy. We simply need to start additional threads or instances or a combination of both, and Kafka Streams takes care of creating Streams Tasks that run in parallel.


**Apache Kafka**

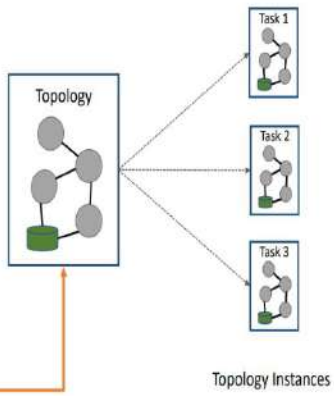
What is a Stream Task and How work is shared by parallel running Stream Tasks?

In Order to understand the Stream task, first we need to know about the Topology.

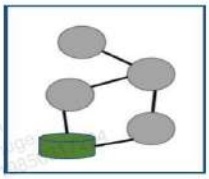
3. **Streams Topology:**

When we write a Kafka Streams application, we define the computational logic using processor APIs which gets bundled into a Topology as shown below:

```
StreamsBuilder builder = new StreamsBuilder();

KStream<String, PosInvoice> KS0 = builder.stream(AppConfigs.posTopicName,
    Consumed.with(AppSerdes.String(), AppSerdes.PosInvoice()));

KS0.filter((k, v) ->
    v.getDeliveryType().equalsIgnoreCase(
        AppConfigs.DELIVERY_TYPE_HOME_DELIVERY))
    .to(AppConfigs.shipmentTopicName,
        Produced.with(AppSerdes.String(), AppSerdes.PosInvoice()));

KS0.filter((k, v) ->
    v.getCustomerType().equalsIgnoreCase(
        AppConfigs.CUSTOMER_TYPE_PRIME))
    .mapValues(RecordBuilder::getNotification)
    .to(AppConfigs.notificationTopic,
        Produced.with(AppSerdes.String(), AppSerdes.Notification()));

KS0.mapValues(RecordBuilder::getMaskedInvoice)
    .flatMapValues(RecordBuilder::getHadoopRecords)
    .to(AppConfigs.hadoopTopic,
        Produced.with(AppSerdes.String(), AppSerdes.HadoopRecord()));

Topology topology = builder.build();
```




Kafka Streams leverages the Topology to parallelize the work. At runtime, the framework would create a set of logical stream tasks, each Streams Task would create an copy(instance) of the Topology object and execute it in parallel. What does that mean?



That means, starting more stream threads or more instances of the application are just a matter of instantiating the topology and having it process by an independent Streams Task.

4. **Streams Task:**

A Stream Task is nothing but just a logical unit of execution. When we start a new application identified by an APPLICATION_ID_CONFIG, Kafka Streams framework will create a fixed number of Streams Tasks.

```
public class KafkaStreamsConsumer {
    public static void main(String[] args) {
        Properties props = new Properties();

        props.put(StreamsConfig.APPLICATION_ID_CONFIG,AppConfig.APP_ID);
```

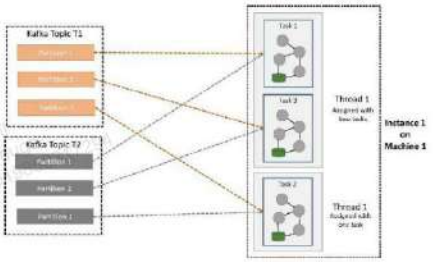
Imagine a kafka streams application that consumes from two topics T1 and T2 with each having 3 partitions. The framework will look at these details and it creates a fixed number of logical tasks.

***The number of tasks is equal to the number of partitions in the input topic of the application topology.***
In the case of multiple input topics, the generated number of tasks equals the largest number of partitions among the input topics.

In our example above the framework will create 3 tasks because the maximum number of partitions across the input topic T1 and T2 are 3. now these tasks will make their own copy of processor topology and becomes ready to apply the processing logic.



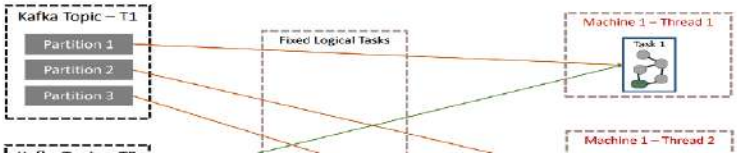

Apache Kafka

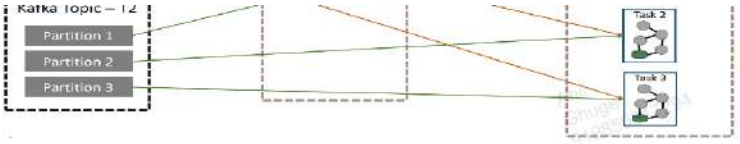



The next step is to assign the partitions to these tasks and Kafka would allocate the partitions evenly. That is one partition from each topic to each task. Finally every task will have two partitions to process one from each topic. Now, these tasks are ready to be assigned to application threads or instances.

If we now start the application with two threads, Kafka would assign one task to each thread, and the remaining third task will go to one of these threads because we do not have any other thread or instance. In this case, the task distribution is not even. The thread running two tasks might run slow. However, all tasks would be running, and ultimately all the data gets processed. This scenario is shown below:



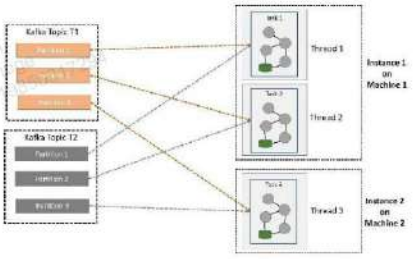






**Apache Kafka**

Now imagine we want to scale out this application. We decide to start another instance with a single thread on a different machine. A new thread T3 will be created, and one task would automatically migrate to the new thread as shown in Figure below:




When the task re-assignment occurs, task partitions and their corresponding local state stores will also migrate from the existing threads to the newly added thread. As a result, Kafka Streams has effectively re balanced the workload among instances of the application at the granularity of Kafka topic partitions.

**What if we wanted to add even more instances of the same application?**

We can do so until a certain point, which is when the number of running instances is equal to the number of stream tasks which is effectively the available input partitions to read from. Adding more instances beyond that point is an over-provisioning with idle instances.