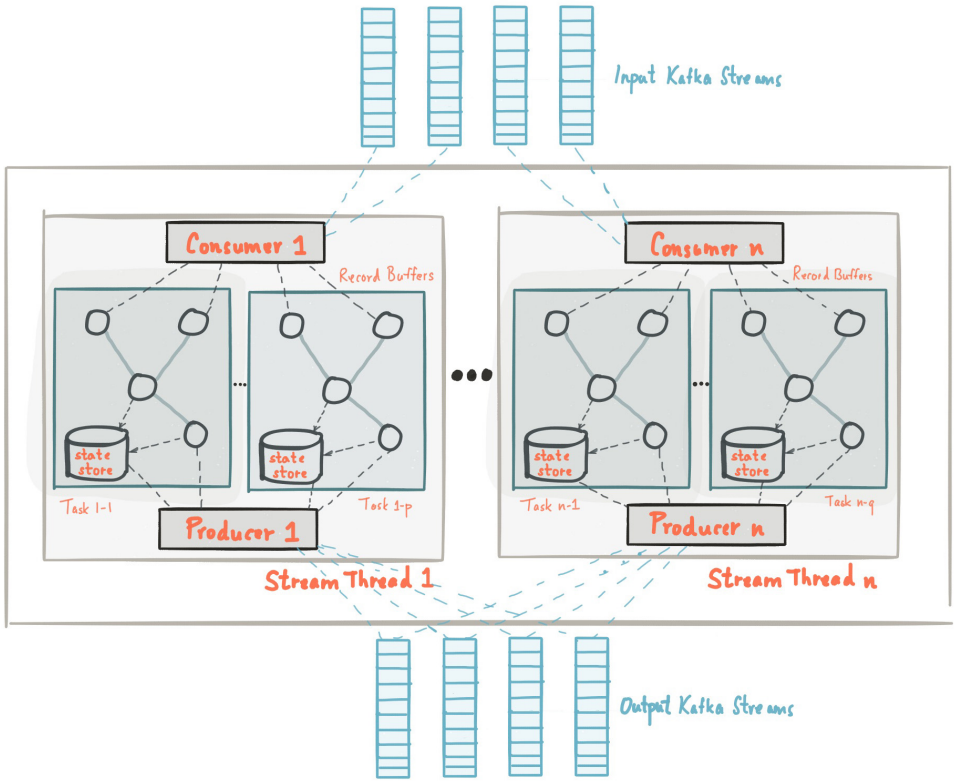
ARCHITECTURE

Kafka Streams simplifies application development by building on the Kafka producer and consumer libraries and leveraging the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity. In this section, we describe how Kafka Streams works underneath the covers.

The picture below shows the anatomy of an application that uses the Kafka Streams library. Let's walk through some details.



Stream Partitions and Tasks

The messaging layer of Kafka partitions data for storing and transporting it. Kafka Streams partitions data for processing it. In both cases, this partitioning is what enables data locality, elasticity, scalability, high performance, and fault tolerance. Kafka Streams uses the concepts of **partitions** and **tasks** as logical units of its parallelism model based on Kafka topic partitions. There are close links between Kafka Streams and Kafka in the context of parallelism:

- Each **stream partition** is a totally ordered sequence of data records and maps to a Kafka **topic**

partition.

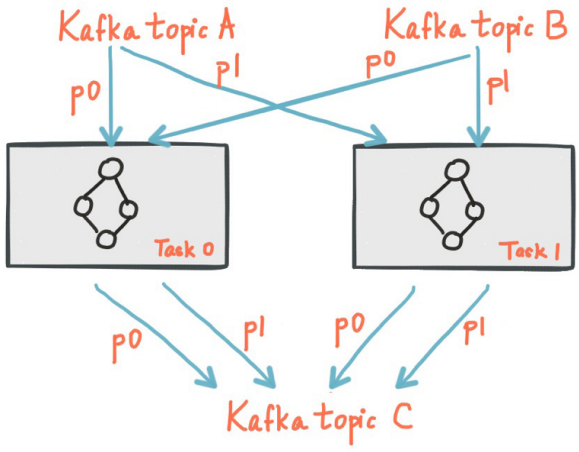
- A **data record** in the stream maps to a Kafka **message** from that topic.
- The **keys** of data records determine the partitioning of data in both Kafka and Kafka Streams, i.e., how data is routed to specific partitions within topics.

An application's processor topology is scaled by breaking it into multiple tasks. More specifically, Kafka Streams creates a fixed number of tasks based on the input stream partitions for the application, with each task assigned a list of partitions from the input streams (i.e., Kafka topics). The assignment of partitions to tasks never changes so that each task is a fixed unit of parallelism of the application. Tasks can then instantiate their own processor topology based on the assigned partitions; they also maintain a buffer for each of its assigned partitions and process messages one-at-a-time from these record buffers. As a result stream tasks can be processed independently and in parallel without manual intervention.

Slightly simplified, the maximum parallelism at which your application may run is bounded by the maximum number of stream tasks, which itself is determined by maximum number of partitions of the input topic(s) the application is reading from. For example, if your input topic has 5 partitions, then you can run up to 5 applications instances. These instances will collaboratively process the topic's data. If you run a larger number of app instances than partitions of the input topic, the "excess" app instances will launch but remain idle; however, if one of the busy instances goes down, one of the idle instances will resume the former's work.

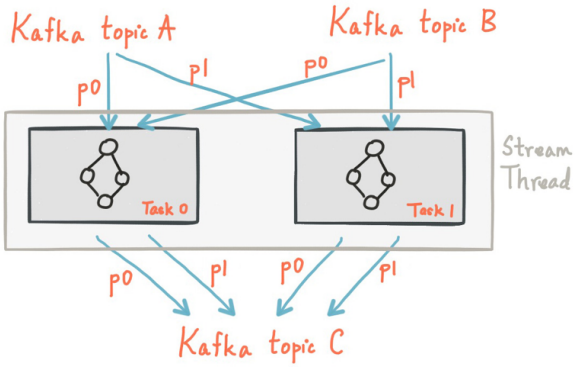
It is important to understand that Kafka Streams is not a resource manager, but a library that "runs" anywhere its stream processing application runs. Multiple instances of the application are executed either on the same machine, or spread across multiple machines and tasks can be distributed automatically by the library to those running application instances. The assignment of partitions to tasks never changes; if an application instance fails, all its assigned tasks will be automatically restarted on other instances and continue to consume from the same stream partitions.

The following diagram shows two tasks each assigned with one partition of the input streams.



Threading Model

Kafka Streams allows the user to configure the number of **threads** that the library can use to parallelize processing within an application instance. Each thread can execute one or more tasks with their processor topologies independently. For example, the following diagram shows one stream thread running two stream tasks.



Starting more stream threads or more instances of the application merely amounts to replicating the topology and having it process a different subset of Kafka partitions, effectively parallelizing processing. It is worth noting that there is no shared state amongst the threads, so no inter-thread coordination is necessary. This makes it very simple to run topologies in parallel across the application instances and threads. The assignment of Kafka topic partitions amongst the various stream threads is transparently handled by Kafka Streams leveraging Kafka's coordination functionality.

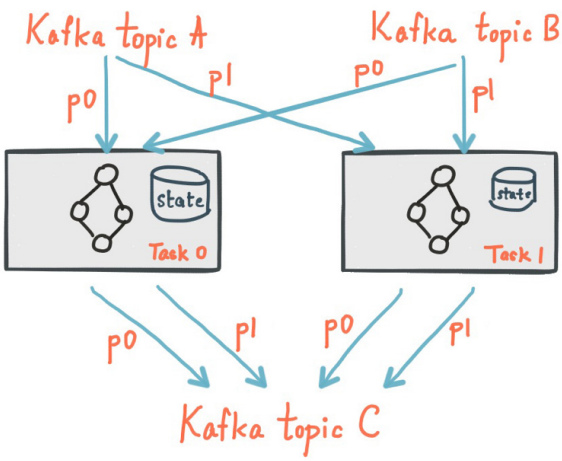
As we described above, scaling your stream processing application with Kafka Streams is easy: you merely need to start additional instances of your application, and Kafka Streams takes care of distributing partitions amongst tasks that run in the application instances. You can start as many threads of the application as there are input Kafka topic partitions so that, across all running instances of an application, every thread (or rather, the tasks it runs) has at least one input partition to process.

Local State Stores

Kafka Streams provides so-called **state stores**, which can be used by stream processing applications to store and query data, which is an important capability when implementing stateful operations. The Kafka Streams DSL, for example, automatically creates and manages such state stores when you are calling stateful operators such as `join()` or `aggregate()`, or when you are windowing a stream.

Every stream task in a Kafka Streams application may embed one or more local state stores that can be accessed via APIs to store and query data required for processing. Kafka Streams offers fault-tolerance and automatic recovery for such local state stores.

The following diagram shows two stream tasks with their dedicated local state stores.



Fault Tolerance

Kafka Streams builds on fault-tolerance capabilities integrated natively within Kafka. Kafka partitions are highly available and replicated; so when stream data is persisted to Kafka it is available even if the application fails and needs to re-process it. Tasks in Kafka Streams leverage the fault-tolerance capability offered by the Kafka consumer client to handle failures. If a task runs on a machine that fails, Kafka Streams automatically restarts the task in one of the remaining running instances of the application.

In addition, Kafka Streams makes sure that the local state stores are robust to failures, too. For each state store, it maintains a replicated changelog Kafka topic in which it tracks any state updates. These changelog topics are partitioned as well so that each local state store instance, and hence the task accessing the store, has its own dedicated changelog topic partition. Log compaction is enabled on the changelog topics so that old data can be purged safely to prevent the topics from growing indefinitely. If tasks run on a machine that fails and are restarted on another machine, Kafka Streams guarantees to restore their associated state stores to the content before the failure by replaying the corresponding changelog topics prior to resuming the processing on the newly started tasks. As a result, failure handling is completely transparent to the end user.

Note that the cost of task (re)initialization typically depends primarily on the time for restoring the state by replaying the state stores' associated changelog topics. To minimize this restoration time, users can configure their applications to have **standby replicas** of local states (i.e. fully replicated copies of the state). When a task migration happens, Kafka Streams then attempts to assign a task to an application instance where such a standby replica already exists in order to minimize the task (re)initialization cost. See `num.standby.replicas` in the Kafka Streams Configs section.

« Previous

Next »