Kafka Streams: Architecture

Rajesh Pasham

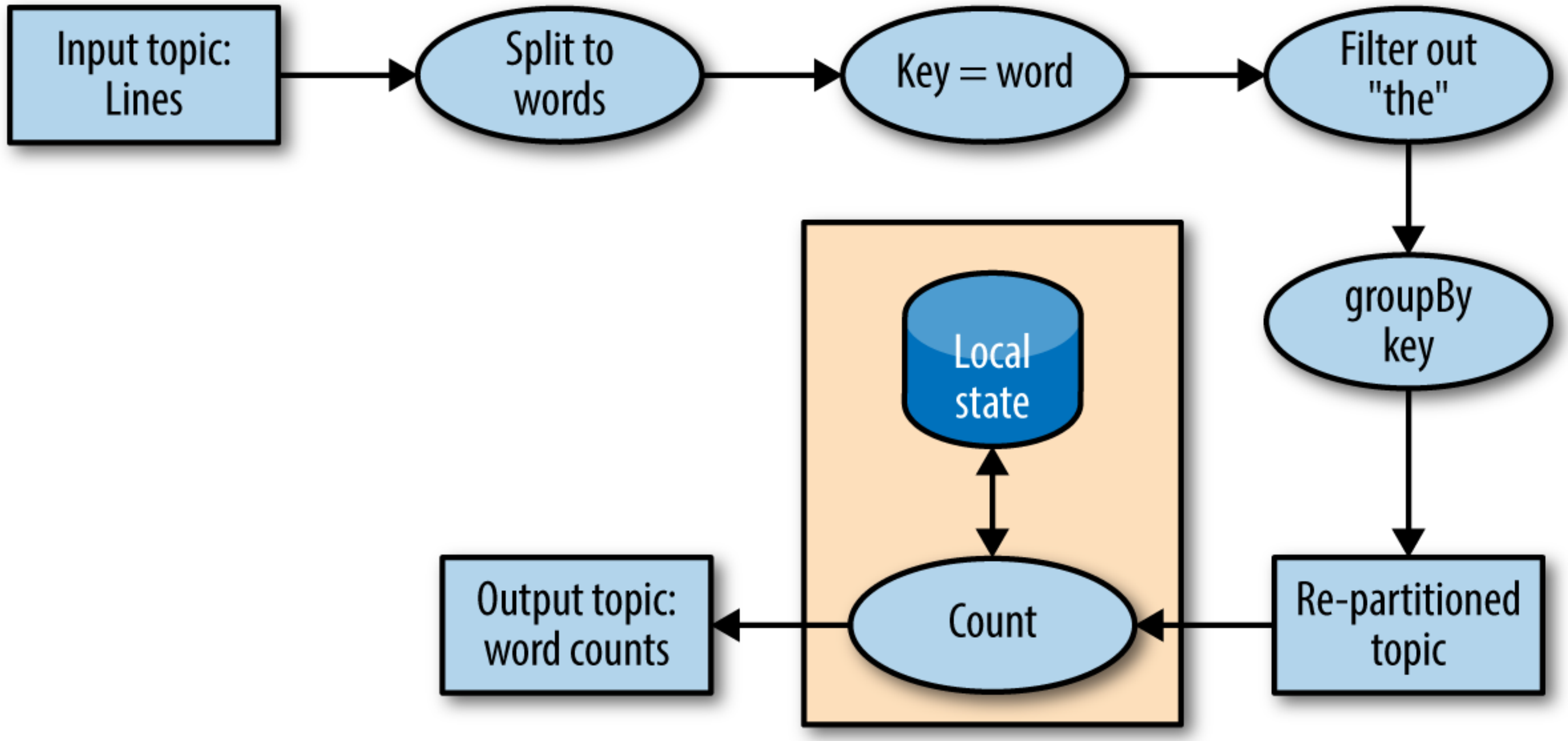
Kafka Streams: Architecture Overview

- To understand better how Kafka’s Streams library actually works and scales, we need to peek under the covers and understand some of the design principles behind the API.

Building a Topology

- Every streams application implements and executes at least one *topology*.
- Topology is a set of operations and transitions that every event moves through from input to output

Kafka Streams: Architecture Overview



Topology for the word-count stream processing example

Kafka Streams: Architecture Overview

- Even a simple app has a nontrivial topology.
- The topology is made up of processors - those are the nodes in the topology graph (represented by circles in our diagram).
- Most processors implement an operation of the data - filter, map, aggregate, etc.
- There are also source processors, which consume data from a topic and pass it on, and sink processors, which take data from earlier processors and produce it to a topic.
- A topology always starts with one or more source processors and finishes with one or more sink processors.

Kafka Streams: Architecture Overview

Scaling the Topology

- Kafka Streams scales by allowing multiple threads of executions within one instance of the application and by supporting load balancing between distributed instances of the application.
- You can run the Streams application on one machine with multiple threads or on multiple machines; in either case, all active threads in the application will balance the work involved in data processing.
- The Streams engine parallelizes execution of a topology by splitting it into tasks.

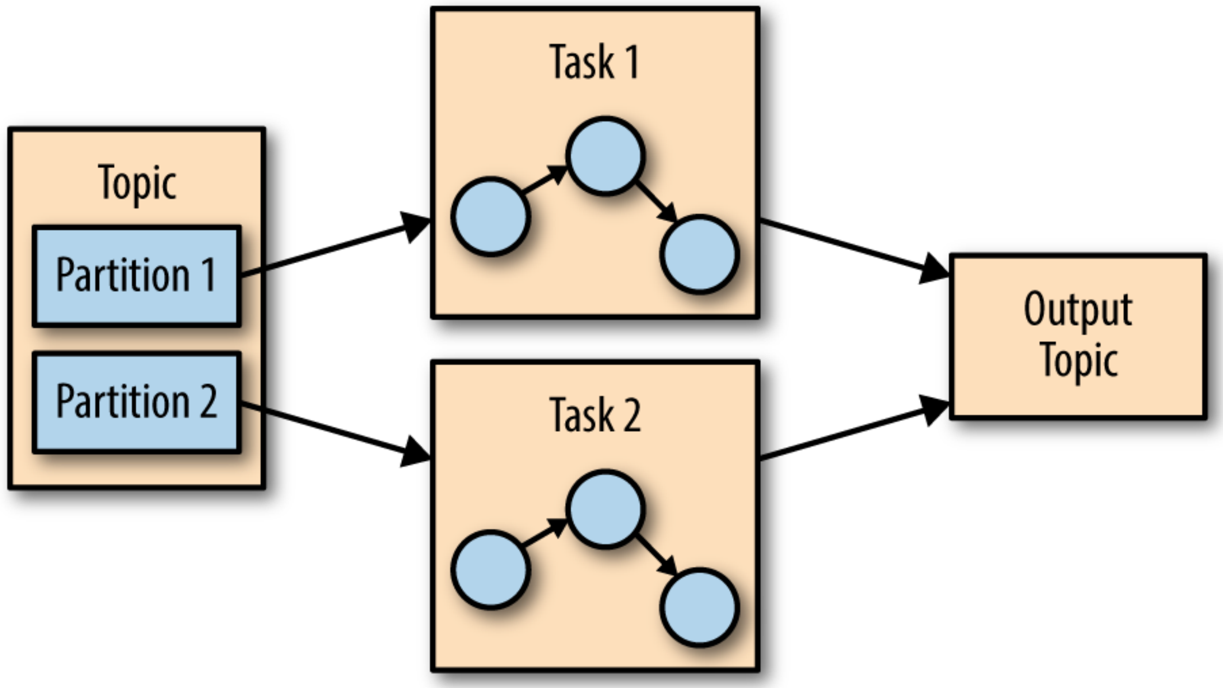
Kafka Streams: Architecture Overview

Scaling the Topology

- The number of tasks is determined by the Streams engine and depends on the number of partitions in the topics that the application processes.
- Each task is responsible for a subset of the partitions: the task will subscribe to those partitions and consume events from them.
- For every event it consumes, the task will execute all the processing steps that apply to this partition in order before eventually writing the result to the sink.
- Those tasks are the basic unit of parallelism in Kafka Streams, because each task can execute independently of others.

Kafka Streams: Architecture Overview

Scaling the Topology



Two tasks running the same topology—one for each partition in the input topic

Kafka Streams: Architecture Overview

Scaling the Topology

- The developer of the application can choose the number of threads each application instance will execute.
- If multiple threads are available, every thread will execute a subset of the tasks that the application creates.
- If multiple instances of the application are running on multiple servers, different tasks will execute for each thread on each server.
- This is the way streaming applications scale: you will have as many tasks as you have partitions in the topics you are processing.

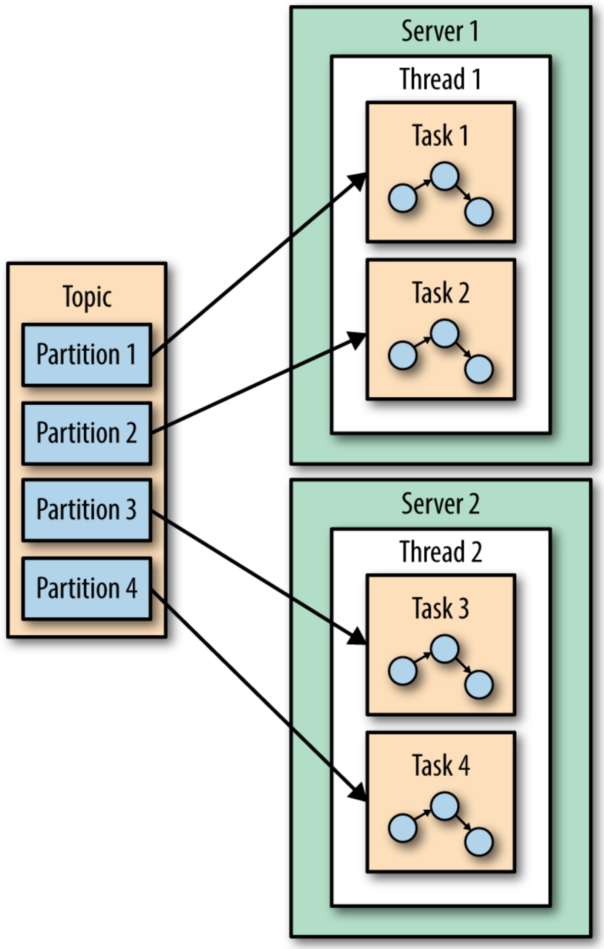
Kafka Streams: Architecture Overview

Scaling the Topology

- If you want to process faster, add more threads.
- If you run out of resources on the server, start another instance of the application on another server.
- Kafka will automatically coordinate work - it will assign each task its own subset of partitions and each task will independently process events from those partitions and maintain its own local state with relevant aggregates if the topology requires this.

Kafka Streams: Architecture Overview

Scaling the Topology



The stream processing tasks can run on multiple threads and multiple servers

Kafka Streams: Architecture Overview

Scaling the Topology

- Sometimes a processing step may require results from multiple partitions, which could create dependencies between tasks.
- For example, if we join two streams, as we did in the ClickStream example, we need data from a partition in each stream before we can emit a result.
- Kafka Streams handles this situation by assigning all the partitions needed for one join to the same task so that the task can consume from all the relevant partitions and perform the join independently.
- This is why Kafka Streams currently requires that all topics that participate in a join operation will have the same number of partitions and be partitioned based on the join key.

Kafka Streams: Architecture Overview

Scaling the Topology

- Another example of dependencies between tasks is when our application requires repartitioning.
- For instance, in the ClickStream example, all our events are keyed by the user ID.
- But what if we want to generate statistics per page? Or per zip code? We'll need to repartition the data by the zip code and run an aggregation of the data with the new partitions.
- If task 1 processes the data from partition 1 and reaches a processor that repartitions the data (groupBy operation), it will need to *shuffle*, which means sending them the events - send events to other tasks to process them.

Kafka Streams: Architecture Overview

Scaling the Topology

- Unlike other stream processor frameworks, Kafka Streams repartitions by writing the events to a new topic with new keys and partitions.
- Then another set of tasks reads events from the new topic and continues processing.
- The repartitioning steps break our topology into two subtopologies, each with its own tasks.
- The second set of tasks depends on the first, because it processes the results of the first subtopology.

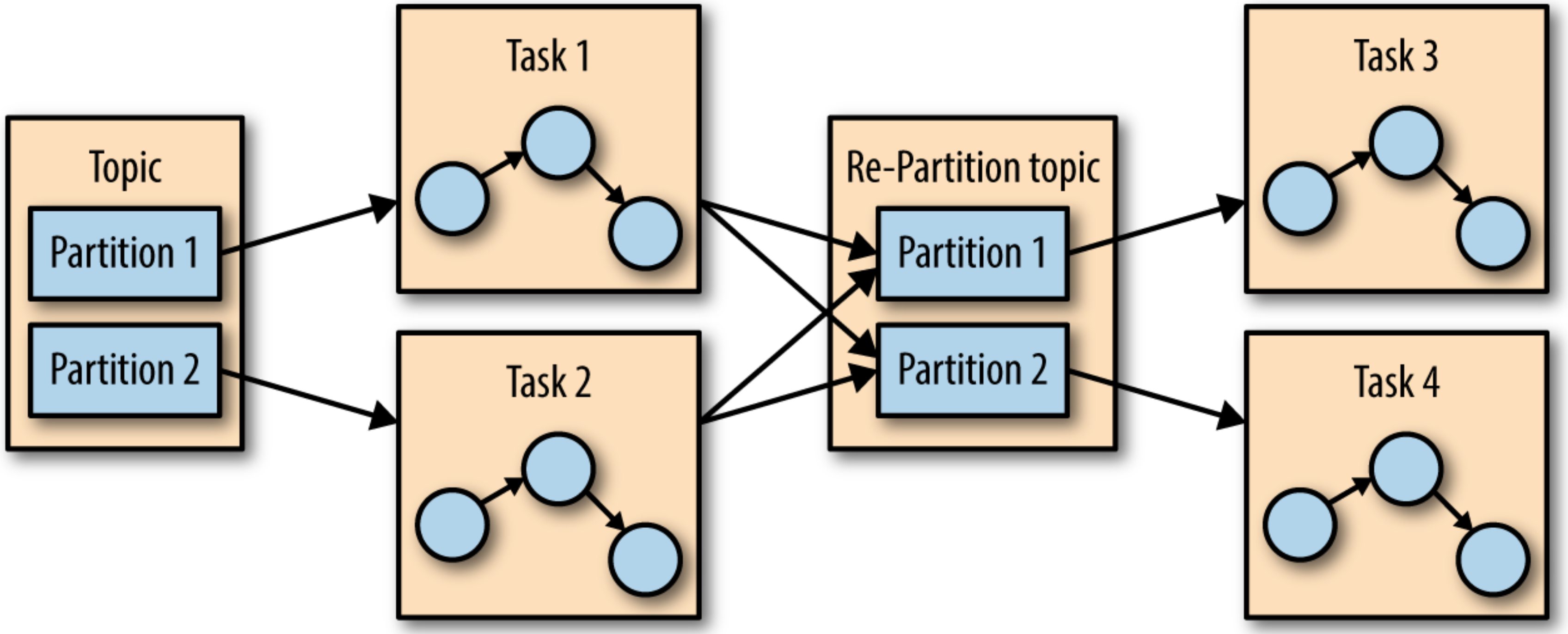
Kafka Streams: Architecture Overview

Scaling the Topology

- However, the first and second sets of tasks can still run independently and in parallel because the first set of tasks writes data into a topic at its own rate and the second set consumes from the topic and processes the events on its own.
- There is no communication and no shared resources between the tasks and they don’t need to run on the same threads or servers.
- This is one of the more useful things Kafka does - reduce dependencies between different parts of a pipeline.

Kafka Streams: Architecture Overview

Scaling the Topology



Two sets of tasks processing events with a topic for re-partitioning events between them

Kafka Streams: Architecture Overview

Surviving Failures

- The same model that allows us to scale our application also allows us to gracefully handle failures.
- First, Kafka is highly available, and therefore the data we persist to Kafka is also highly available.
- So if the application fails and needs to restart, it can look up its last position in the stream from Kafka and continue its processing from the last offset it committed before failing.
- Note that if the local state store is lost (e.g., because we needed to replace the server it was stored on), the streams application can always re-create it from the change log it stores in Kafka.

Kafka Streams: Architecture Overview

Surviving Failures

- Kafka Streams also leverages Kafka’s consumer coordination to provide high availability for tasks.
- If a task failed but there are threads or other instances of the streams application that are active, the task will restart on one of the available threads.
- This is similar to how consumer groups handle the failure of one of the consumers in the group by assigning partitions to one of the remaining consumers.

Stream Processing Use Cases

- Stream processing - or continuous processing - is useful in cases where you want your events to be processed in quick order rather than wait for hours until the next batch, but also where you are not expecting a response to arrive in milliseconds.
- Let’s look at a few real scenarios that can be solved with stream processing:
  - *Customer Service*
  - *Internet of Things*
  - Fraud Detection