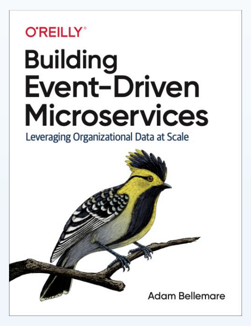
Adam Bellemare - Building Event-Driven Microservices

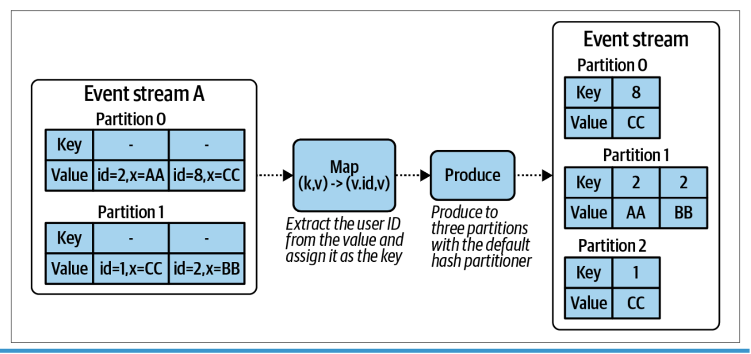

O'REILLY
Building Event-Driven Microservices
Leveraging Organizational Data at Scale
Adam Bellemare

5. Event-Driven Processing Basics

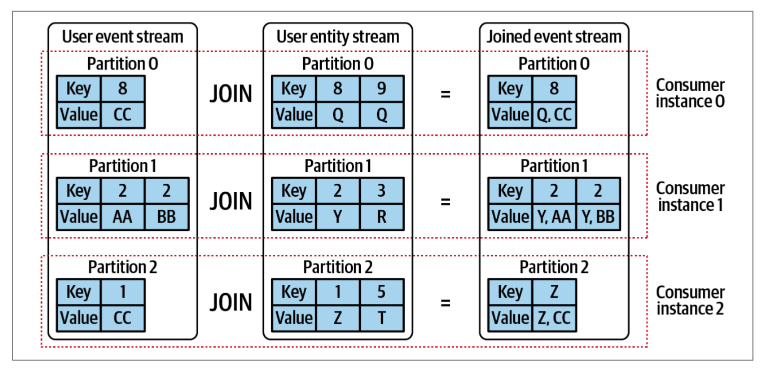
- Typical stream-sourced event-driven microservice pseudocode

```
Consumer consumerClient = new consumerClient(consumerGroupName, ...);
Producer producerClient = new producerClient(...);

while(true) {
    InputEvent event = consumerClient.pollOneEvent(inputEventStream);
    OutputEvent output = processEvent(event);
    producerClient.produceEventToStream(outputEventStream, output);

    //At-least-once processing.
    consumerClient.commitOffsets();
}
```

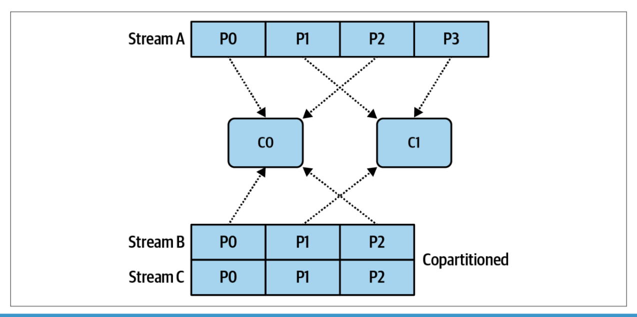
- Composing Stateless Topologies
  - Transformations
    - process a single event and emit zero or more output events
    - Types of transformation
      - Filter
      - Map
        - need to repartition if key was changed
      - MapValue
      - Custom transforms
  - Branching and Merging Streams
    - 2 stream branching scenarios
      - Routing based on particular properties (e.g., country, time zone, origin, product, or any number of features)
      - Emitting results to different output event streams (e.g. outputting events to a dead-letter stream)
    - Merging streams
      - rarely happens as microservices can consume many inputs
- Repartitioning Event Streams
  - **Repartitioning** is the act of producing a new event stream when one of the following happens
    - Different partition count
      - Increase an event stream's partition count
        - to increase downstream parallelism
        - or to match the number of partitions of another stream for copartitioning (covered later in this chapter)
    - Different event key
      - Change the event key to ensure that events with the same key are routed to the same partition
    - Different event partitioner
      - Change the logic used to select which partition an event will be written to
  - example



- Copartitioning Event Streams
  - **Copartitioning** is the repartition of an event stream into a new one with the same partition count and partition assignor logic as another stream
    - ⚠️ This is an important concept for stateful stream processing, as numerous stateful operations (such as streaming joins)
  - example



- Assigning Partitions to a Consumer Instance
  - ⚠️ Each microservice maintains its own unique consumer group represen-ting the collective offsets of its input event streams
  - Assigning Partitions with the Partition Assignor
    - A partition assignor ensures that partitions are distributed to the processing instances in a balanced and equitable manner
  - Assigning Copartitioned Partitions
    - The partition assignor is also responsible for ensuring that any copartitioning requirements are met
    - ⚠️ It is good practice to have the partition assignor implementation check to see that the event streams have an equal partition count and throw an exception on inequality.
  - Partition Assignment Strategies
    - The goal of a partition assignment algorithm is to ensure that partitions are evenly distributed across the consumer instances, assuming that the consumer instances are equal in processing capabilities
    - Strategies
      - Round-robin assignment



        - ⚠️ When the number of consumer instances changes the partition assignments should be reblanced
      - Static assignment
        - 👍 This option is most useful when large volumes of stateful data are materialized on any given instance, usually for internal state stores
      - Custom assignment
        - 👍 For example, assignment could be based on the current lag in the input event streams, ensuring an equal distribution of work across all of your consumer instances
- Recovering from Stateless Processing Instance Failures
  - ⚠️ the same as simply adding a new instance to a consumer group