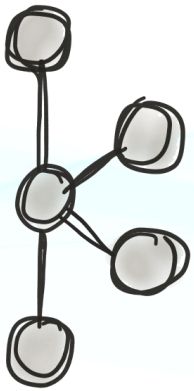


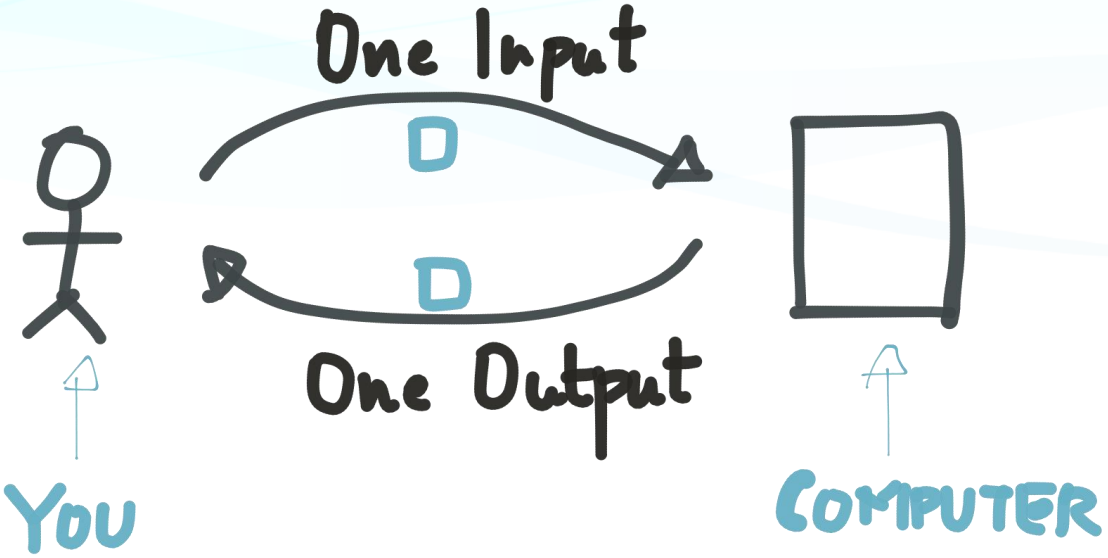
Demystifying Stream Processing with Apache Kafka

confluent

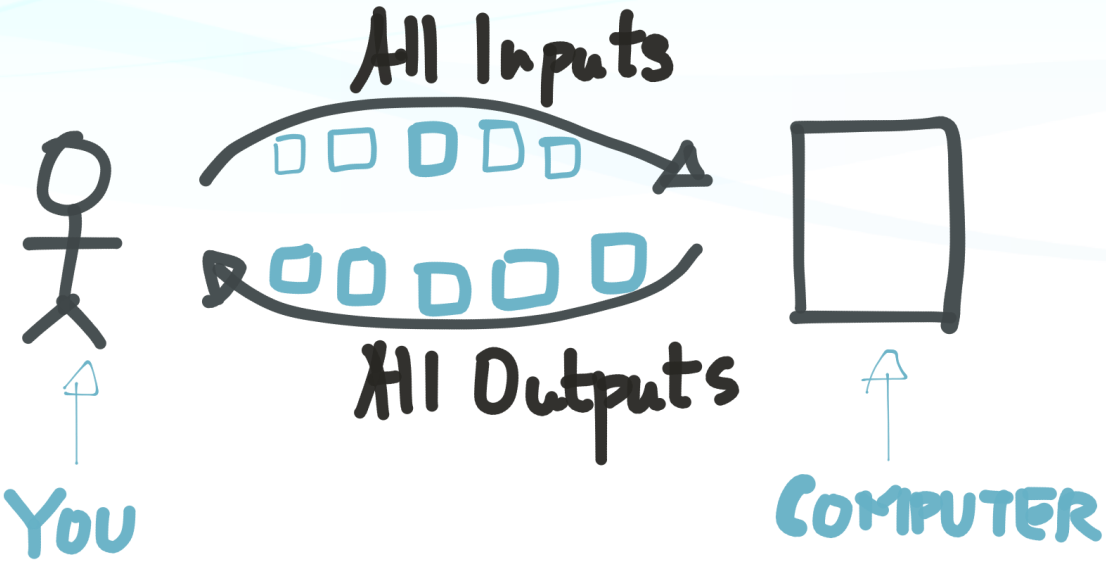
3 Paradigms for Programming

1. request/response
2. batch
3. stream processing

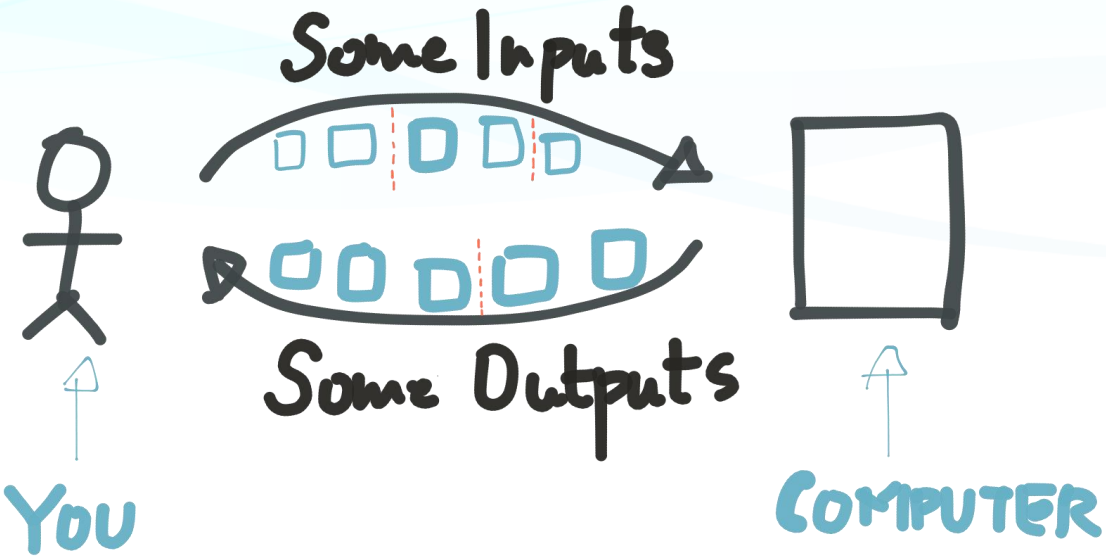
Request/Response



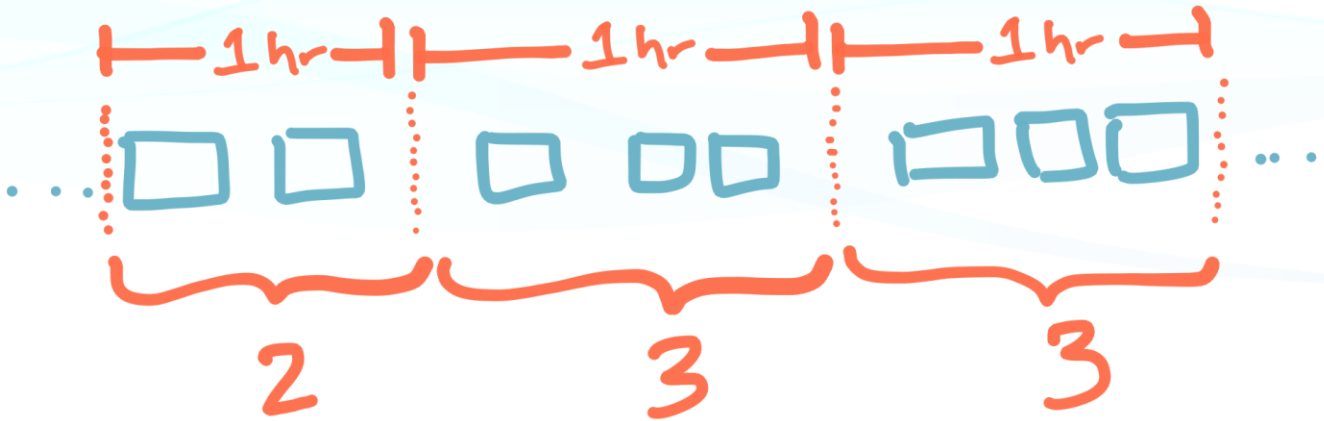
Batch



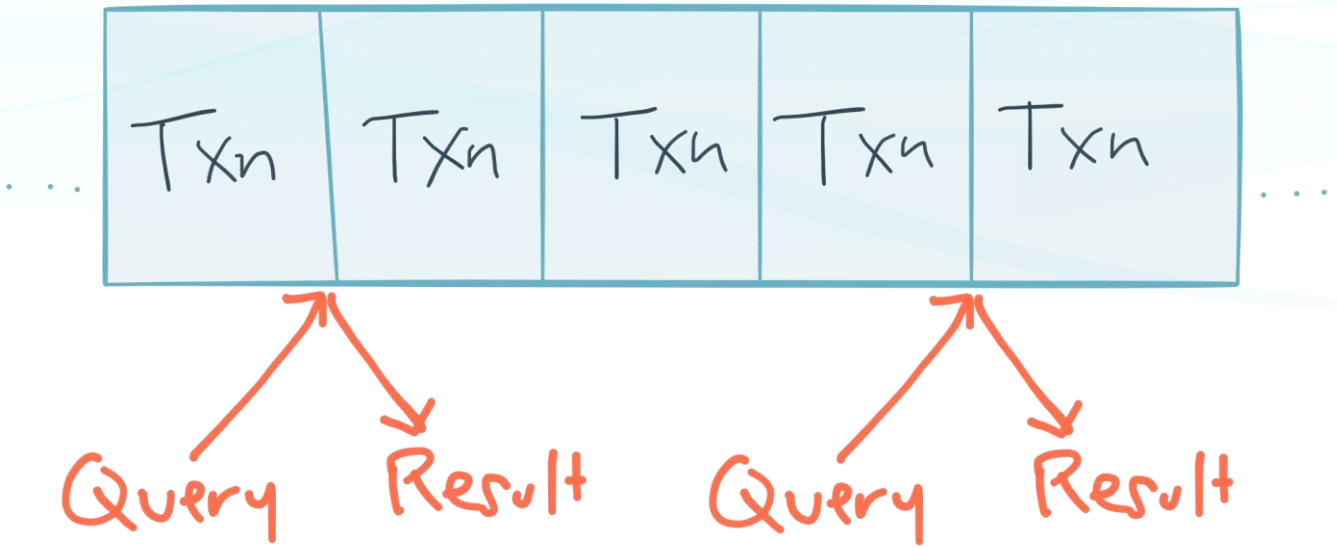
Stream Processing



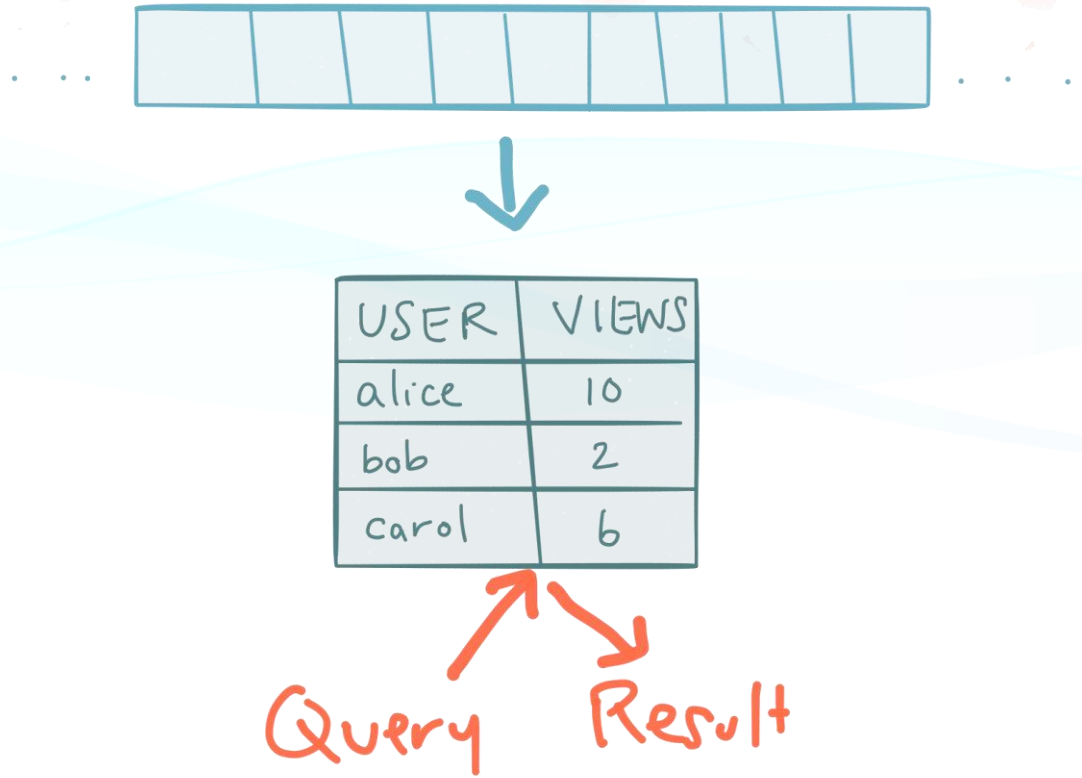
Stream Processing



Databases as Streams



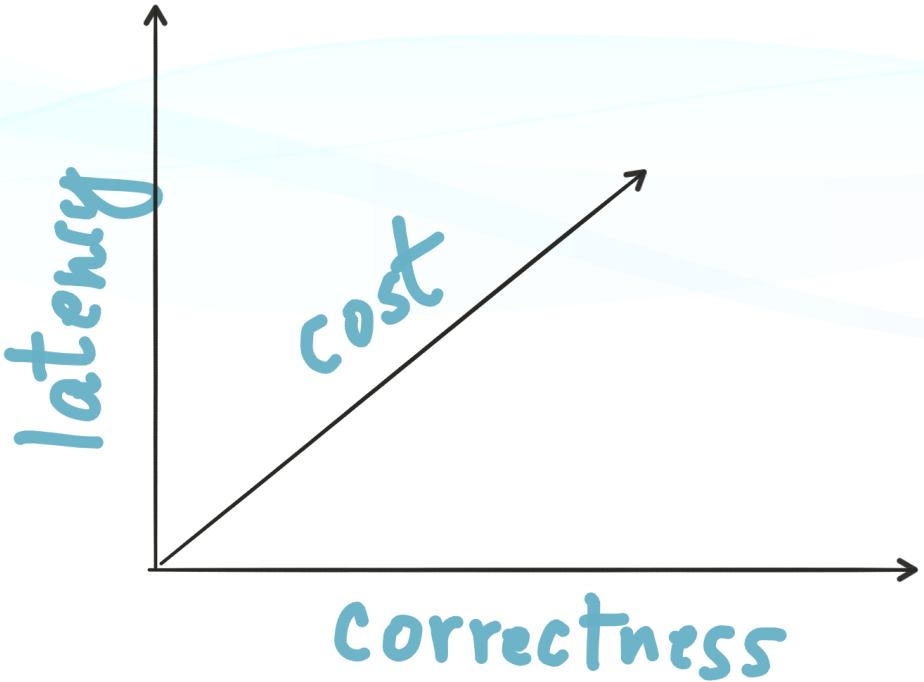
Stream Processing Microservices



Stream Processing isn't (necessarily)

- Transient
- Approximate
- Lossy

Trade Offs



stream processing

=

f("what happened")

events

Approach #1

DIY!

stream processing

DIY Stream Processing

Problem: Stream Processing has some hard parts

- partitioning & scalability
- state
- windowing & time
- semantics & fault tolerance
- reprocessing

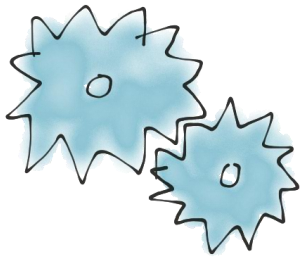
Approach #2

Use a stream processing framework

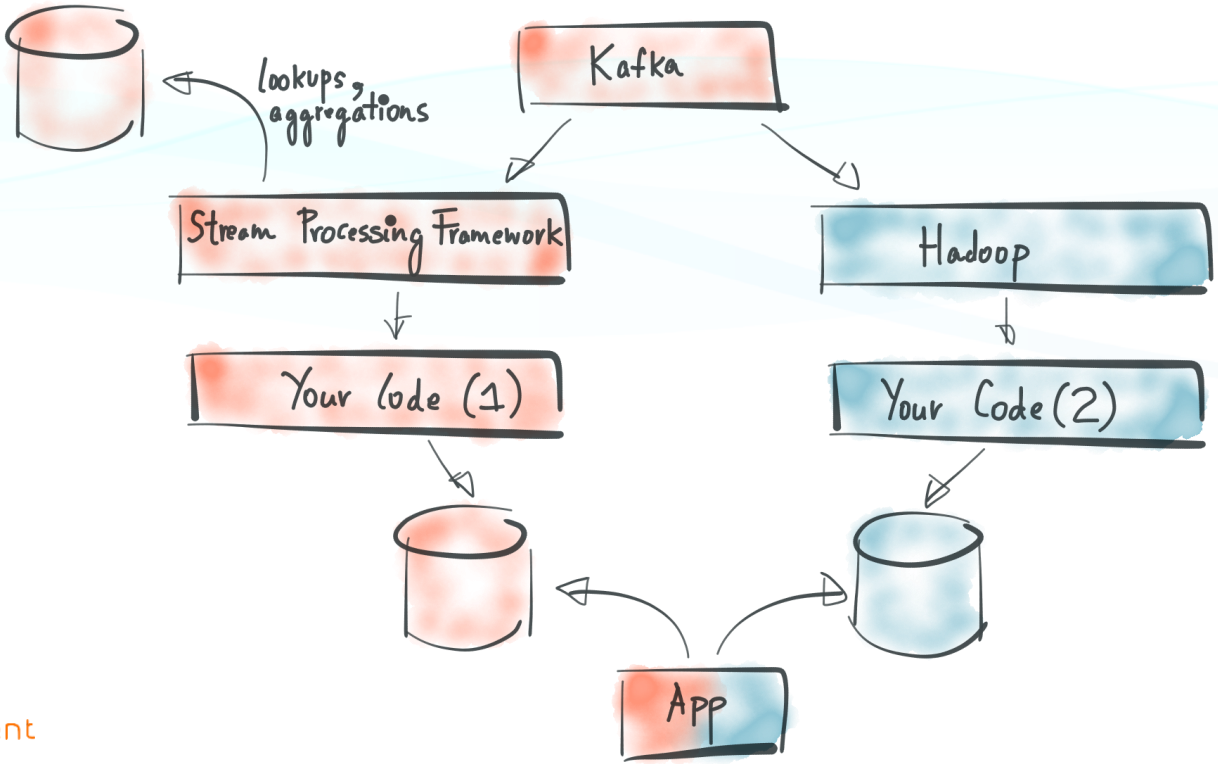
- Spark
- Storm
- Samza
- Flink

et al

Problem : Lots Of moving parts !



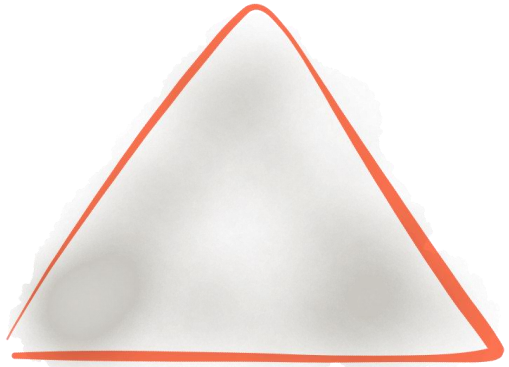
Example Architecture



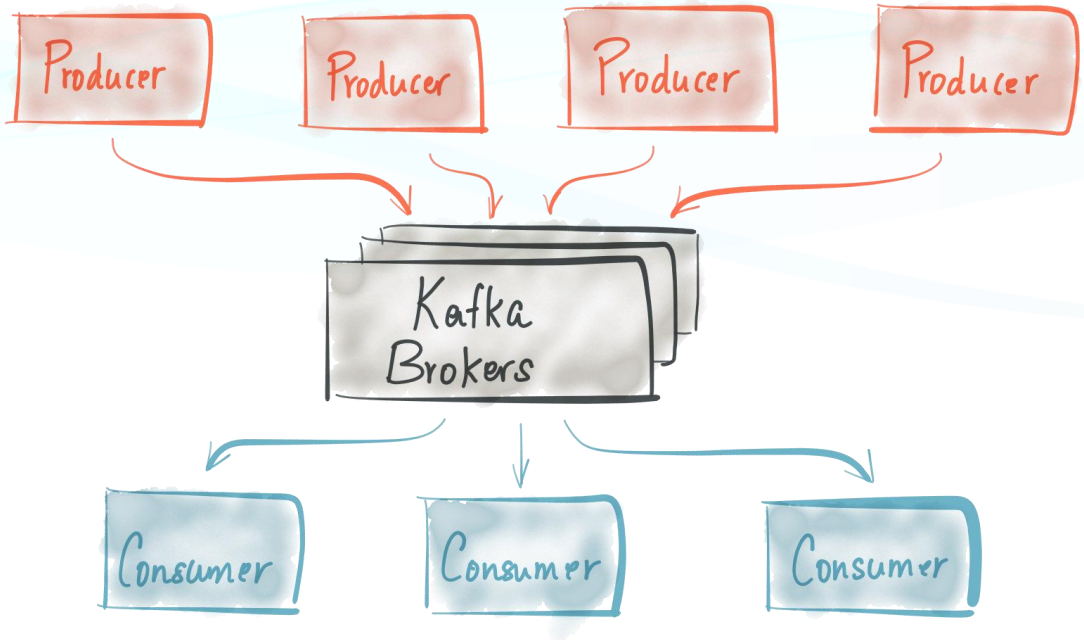
The Hard Part Of Distributed Systems

Coding < Debugging < Operations

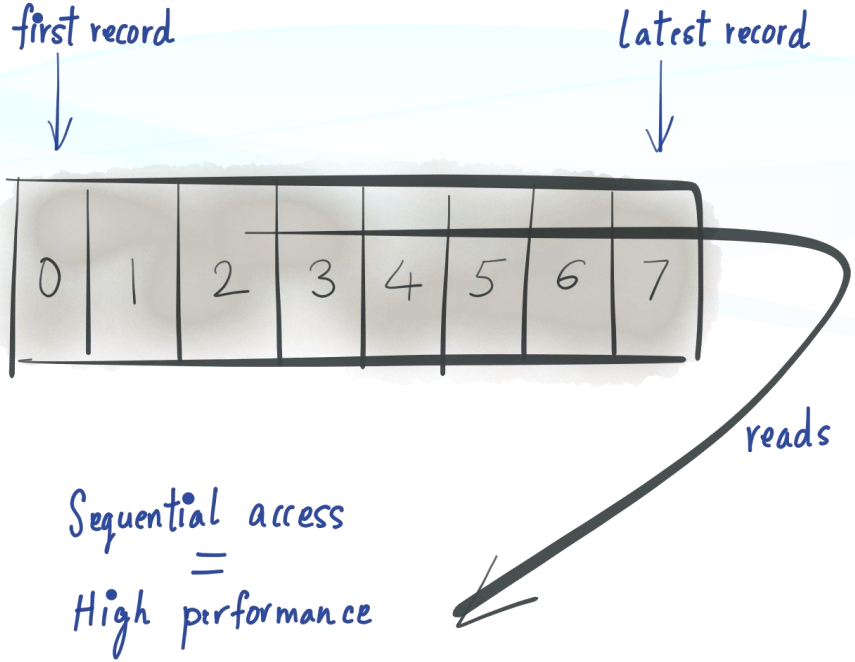
Kafka Provides primitives



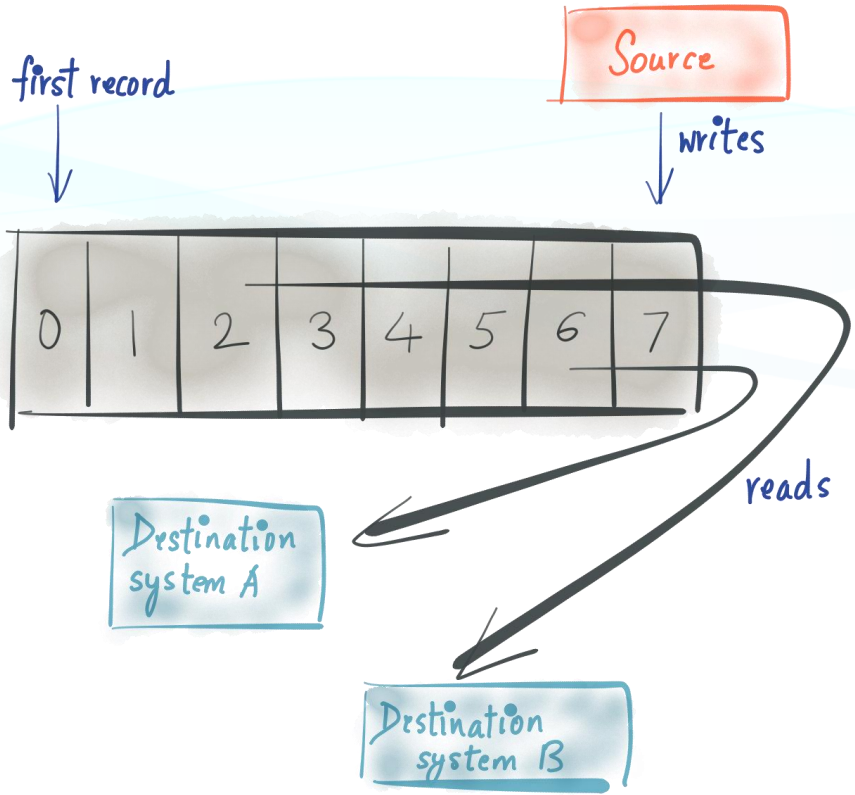
Apache Kafka : 10,000 ft view

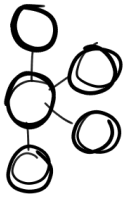


Kafka Key Idea : Log

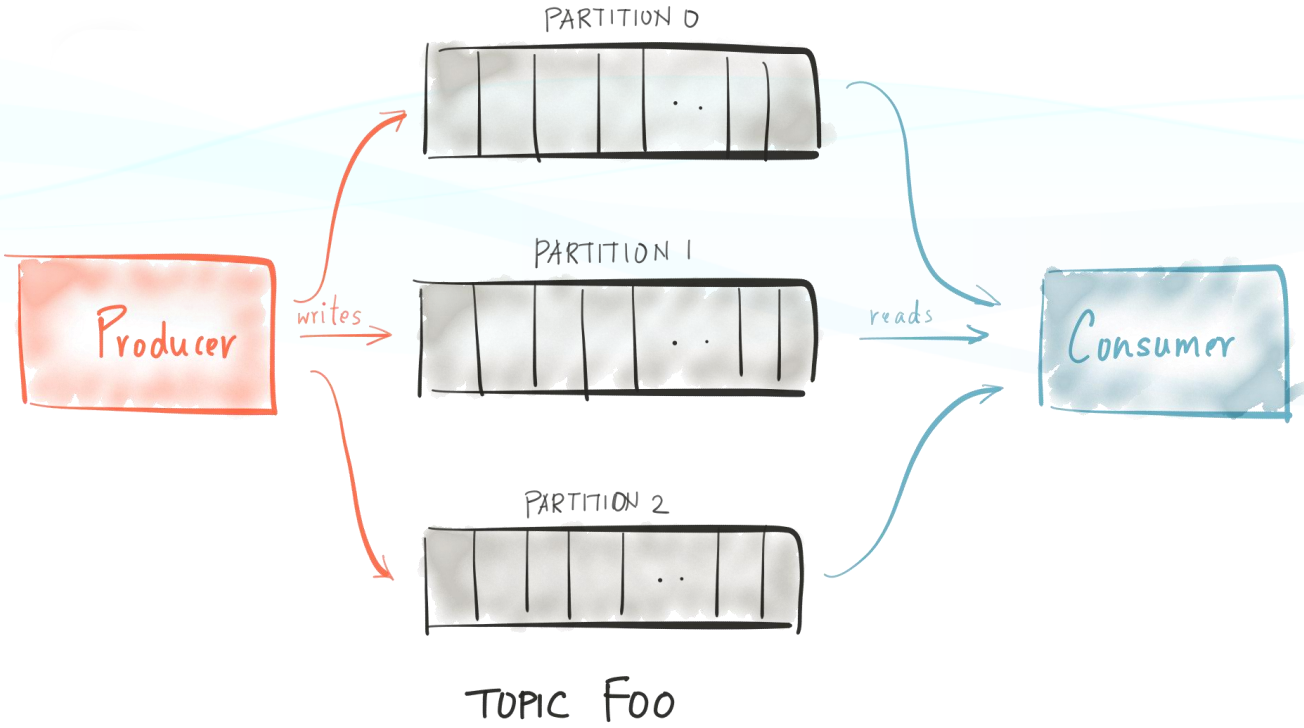


Logs & Pub-Sub

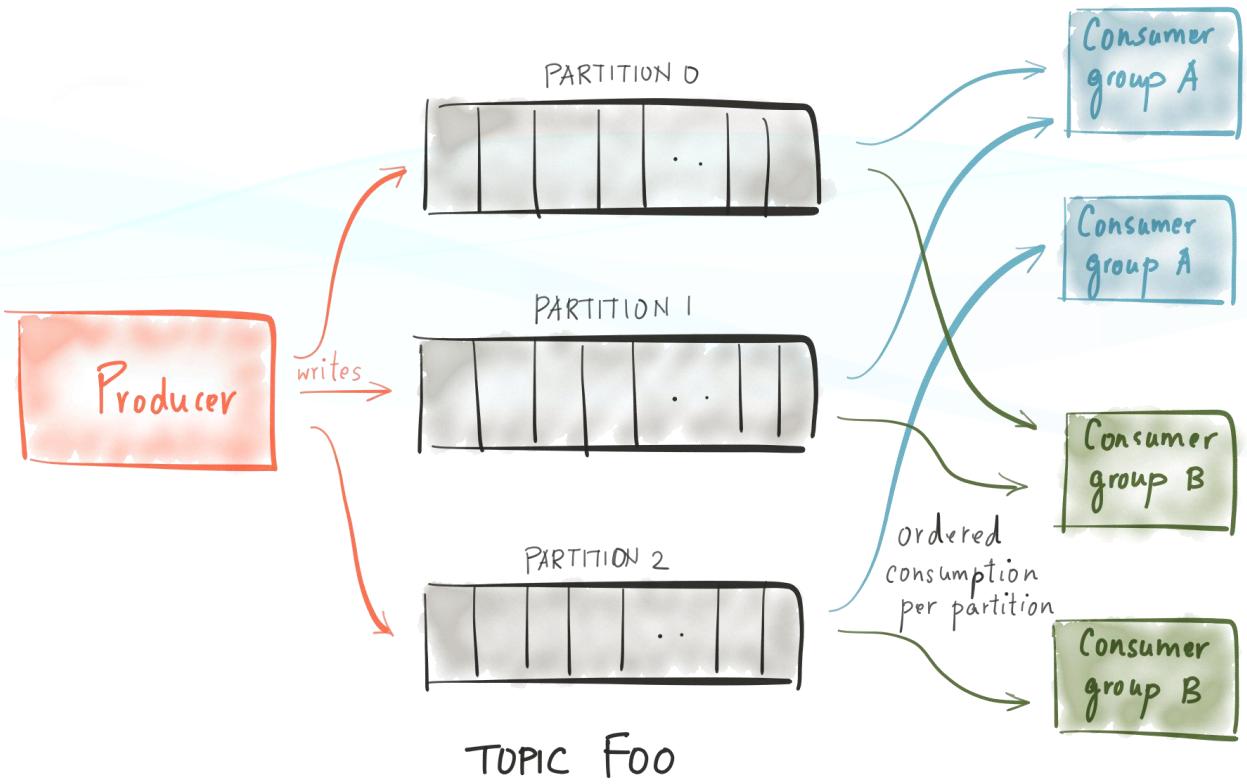

first record
Source
writes
0
1
2
3
4
5
6
7
reads
Destination system A
Destination system B



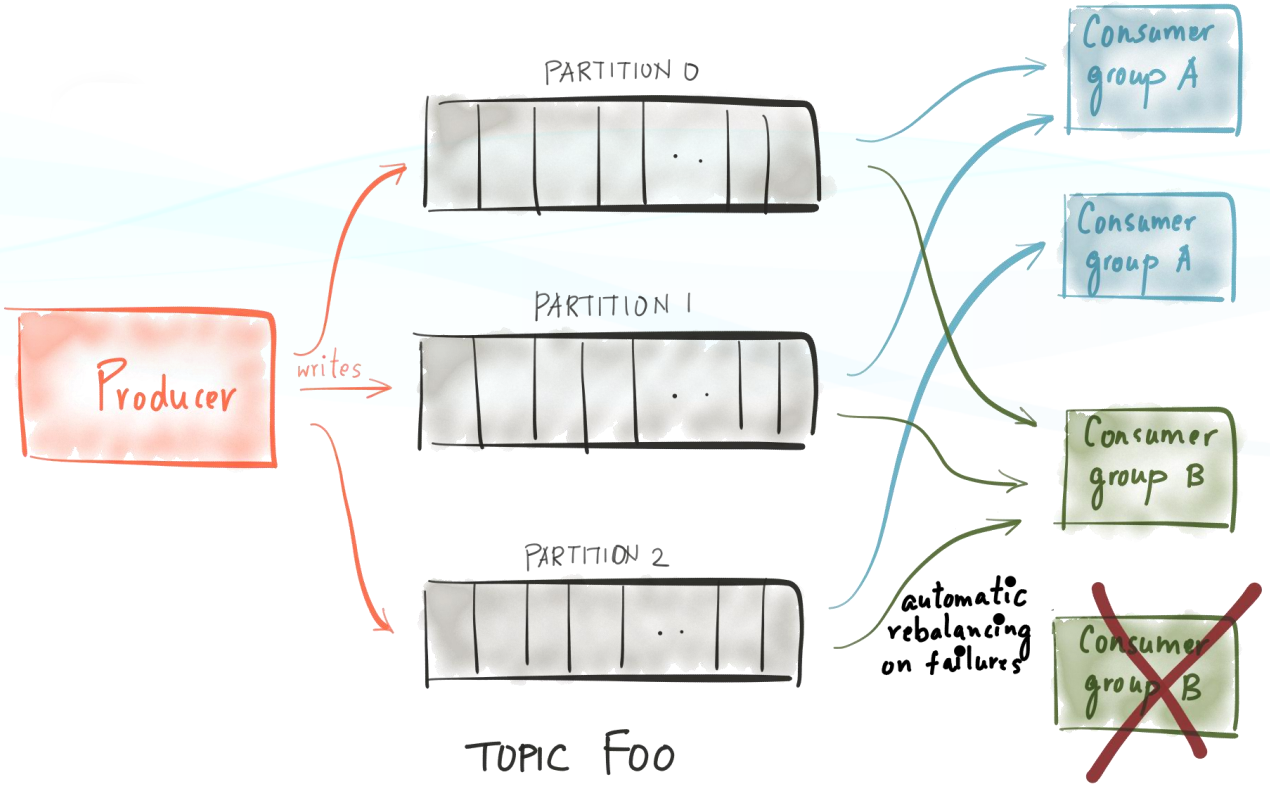
Topic ≈ Partitioned Log



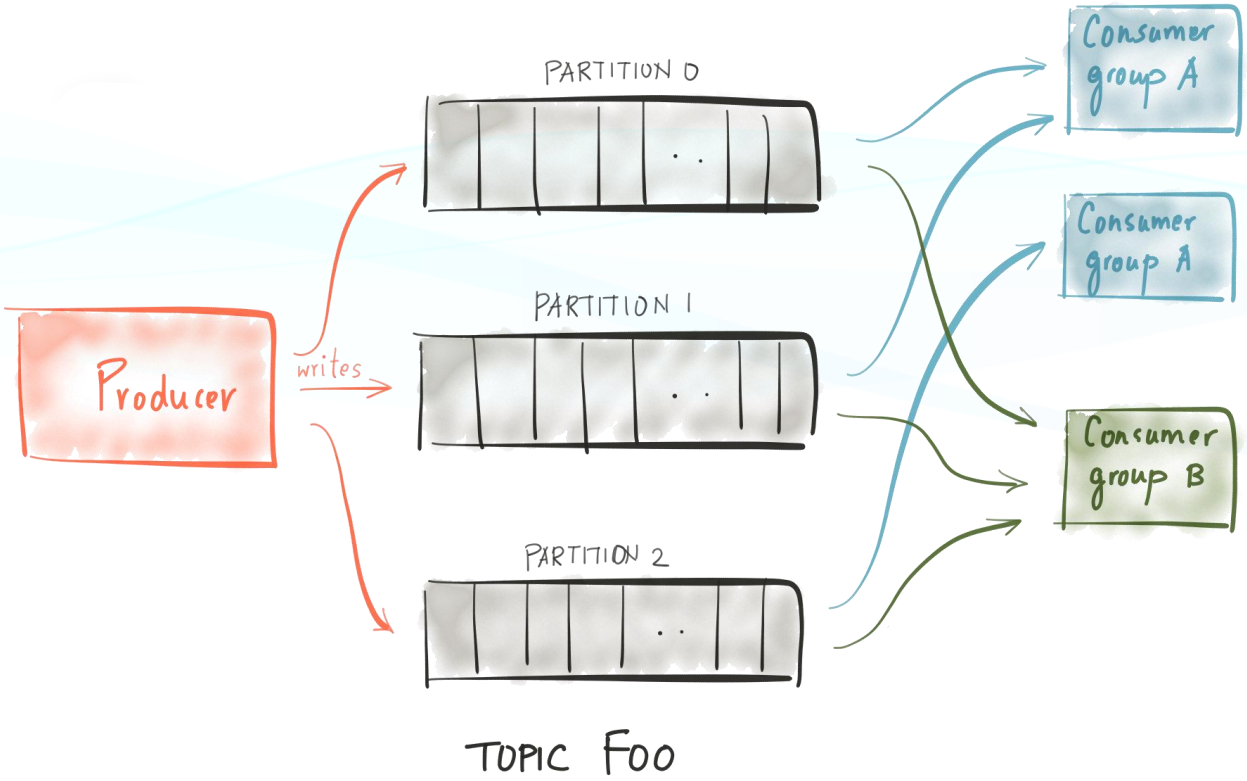
Scalable Consumption

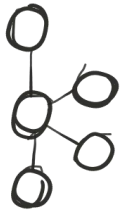


Fault Tolerant Consumption

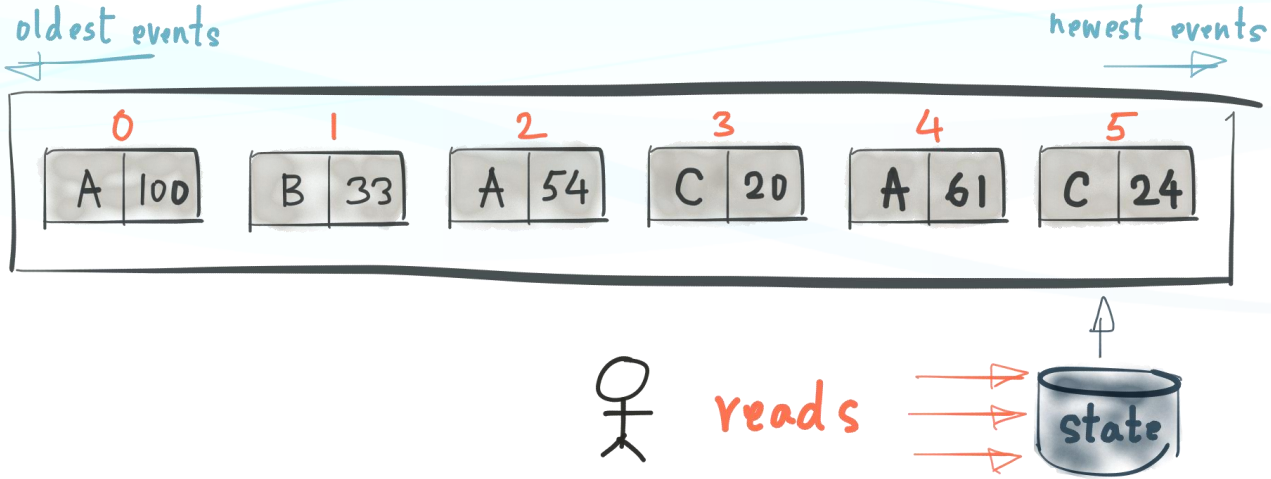


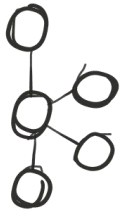
Fault Tolerant Consumption



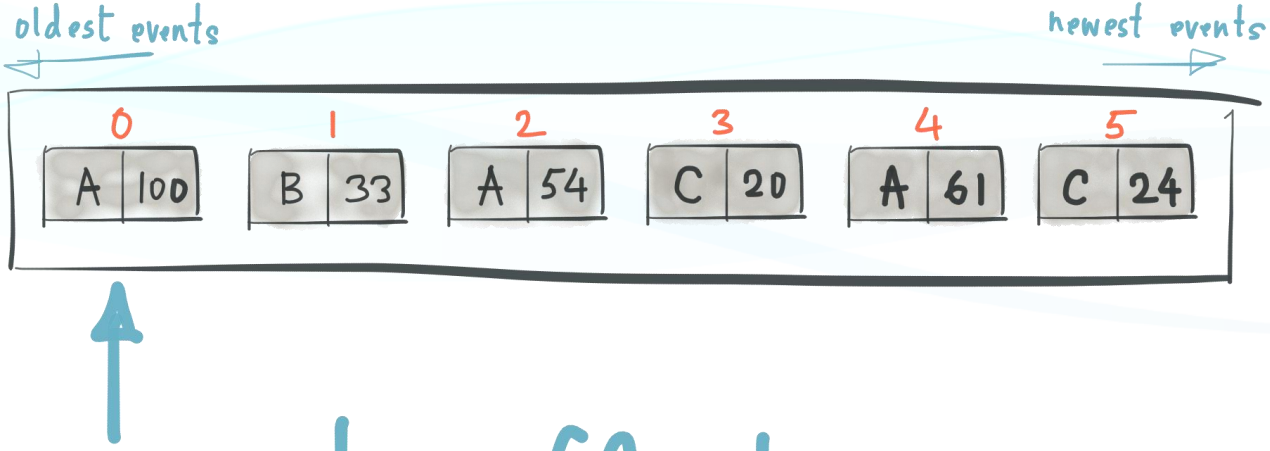


REPROCESSING

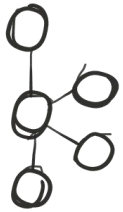




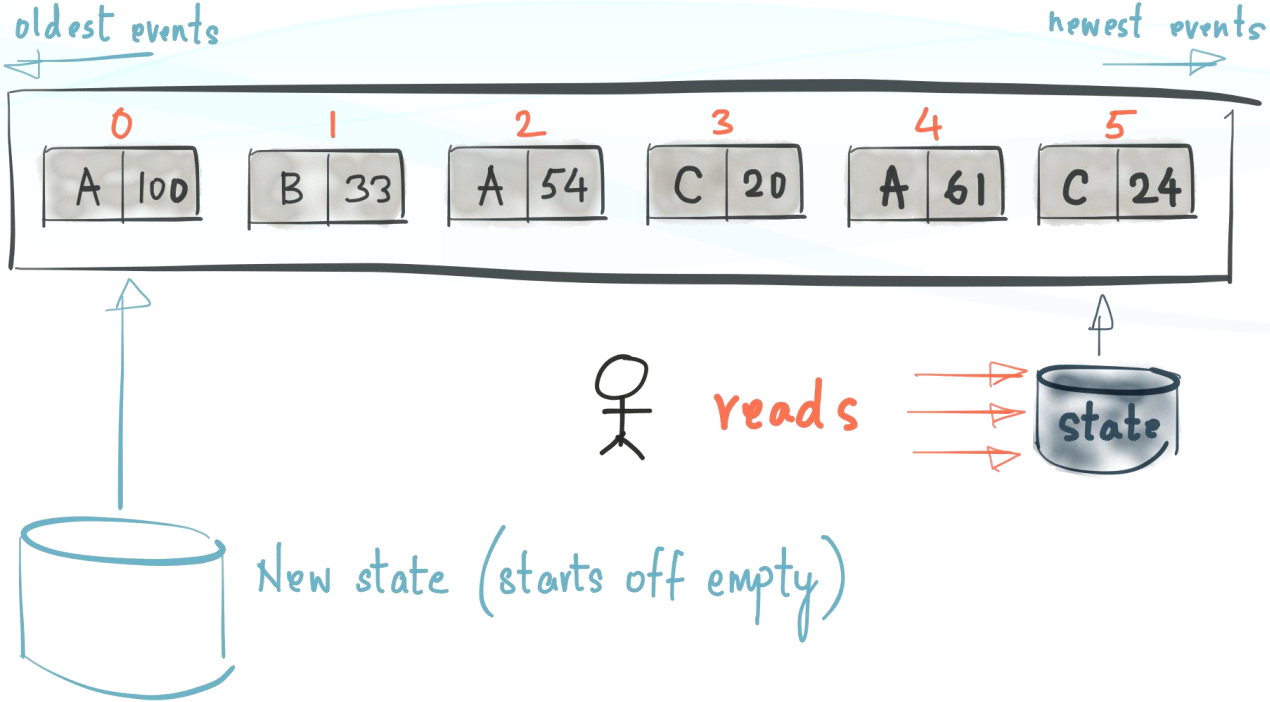
REPROCESSING

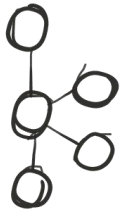


set offset = 0

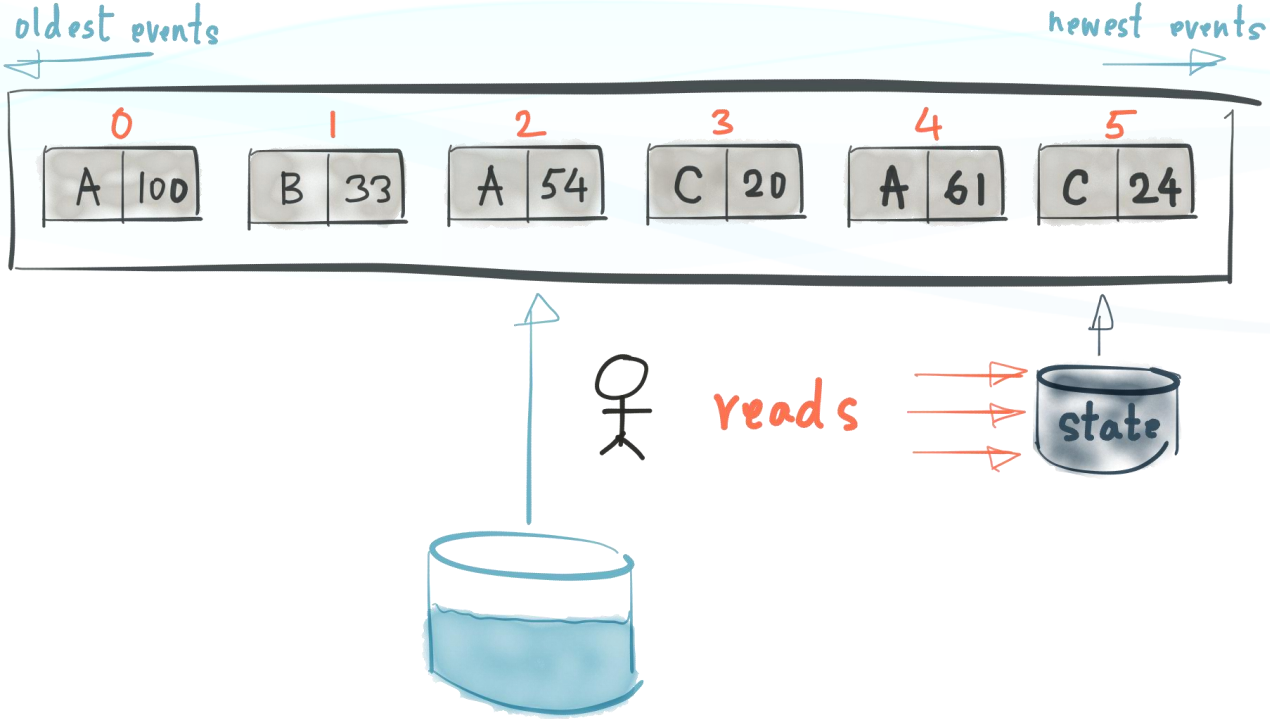


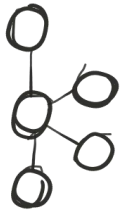
REPROCESSING



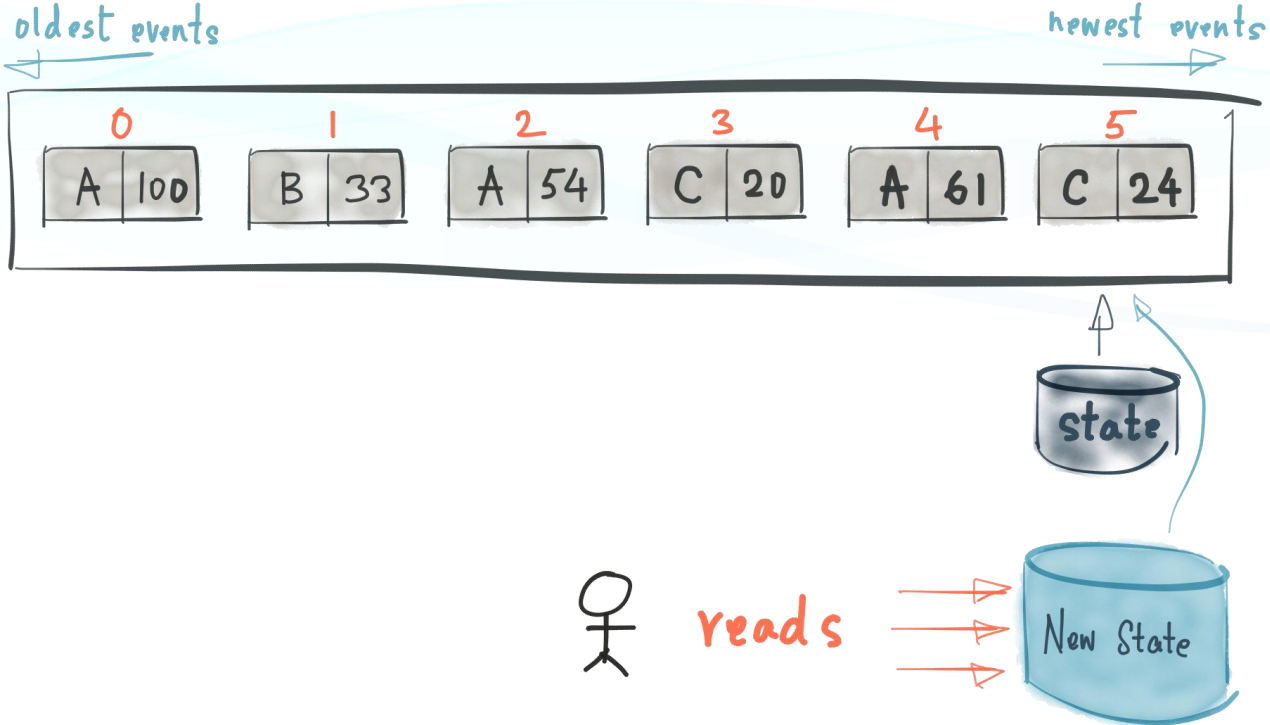


REPROCESSING

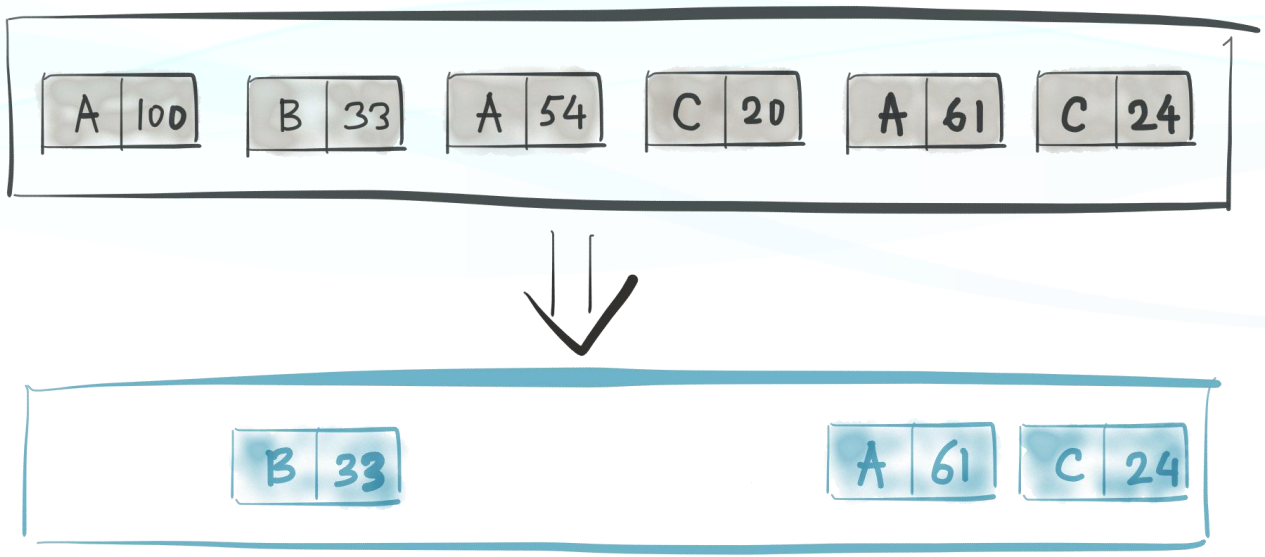




REPROCESSING

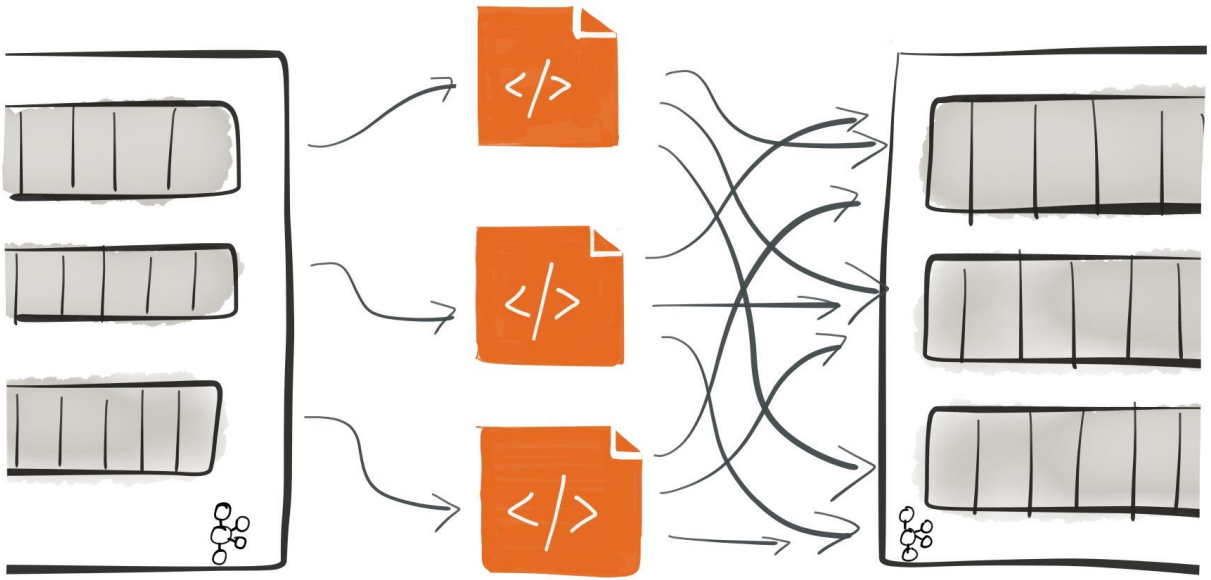


Log Compaction



Logs unify Batch & Stream Processing

STREAM PROCESSING



CONSUMER + CUSTOM + PRODUCER CODE

Kafka Streams

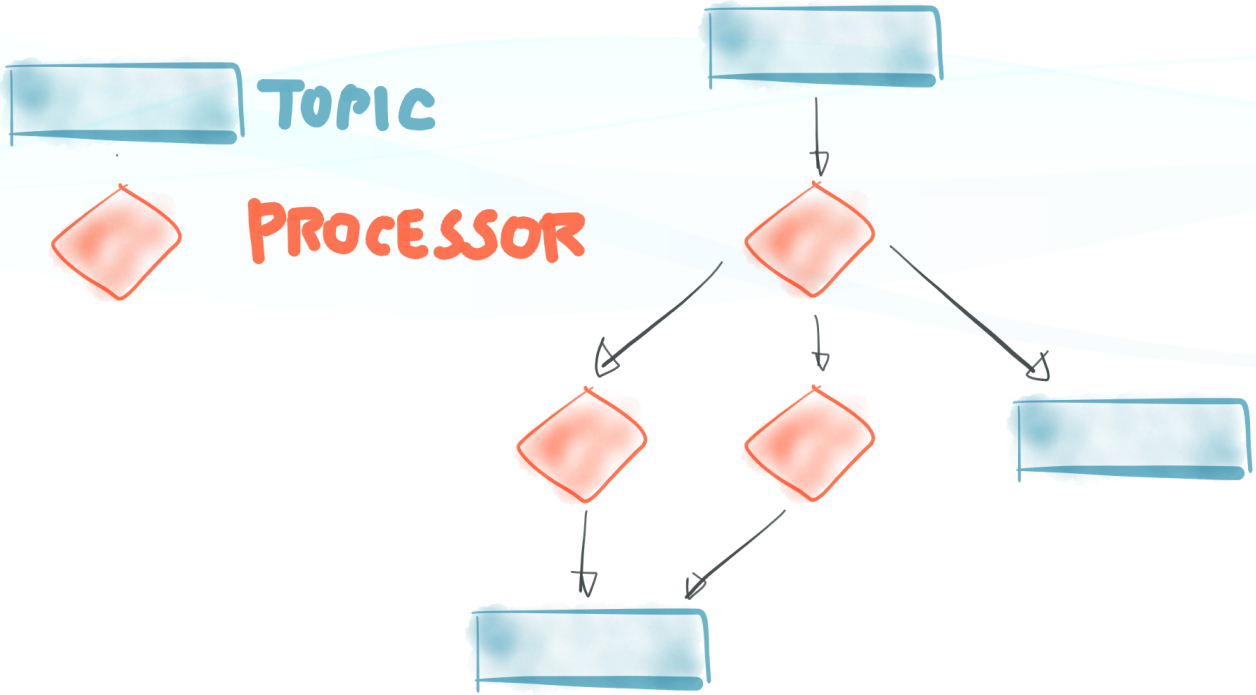
- library
- 2 interfaces
  1. processor API
  2. kstream DSL

Libraries
vs
Frameworks

Processor API

```
/**
 * A processor of messages.
 *
 * @param <K> the type of keys
 * @param <V> the type of values
 */
public interface Processor<K, V> {

    void init(ProcessorContext context);

    void process(K key, V value);

    void punctuate(long timestamp);

    void close();
}
```

DAGS



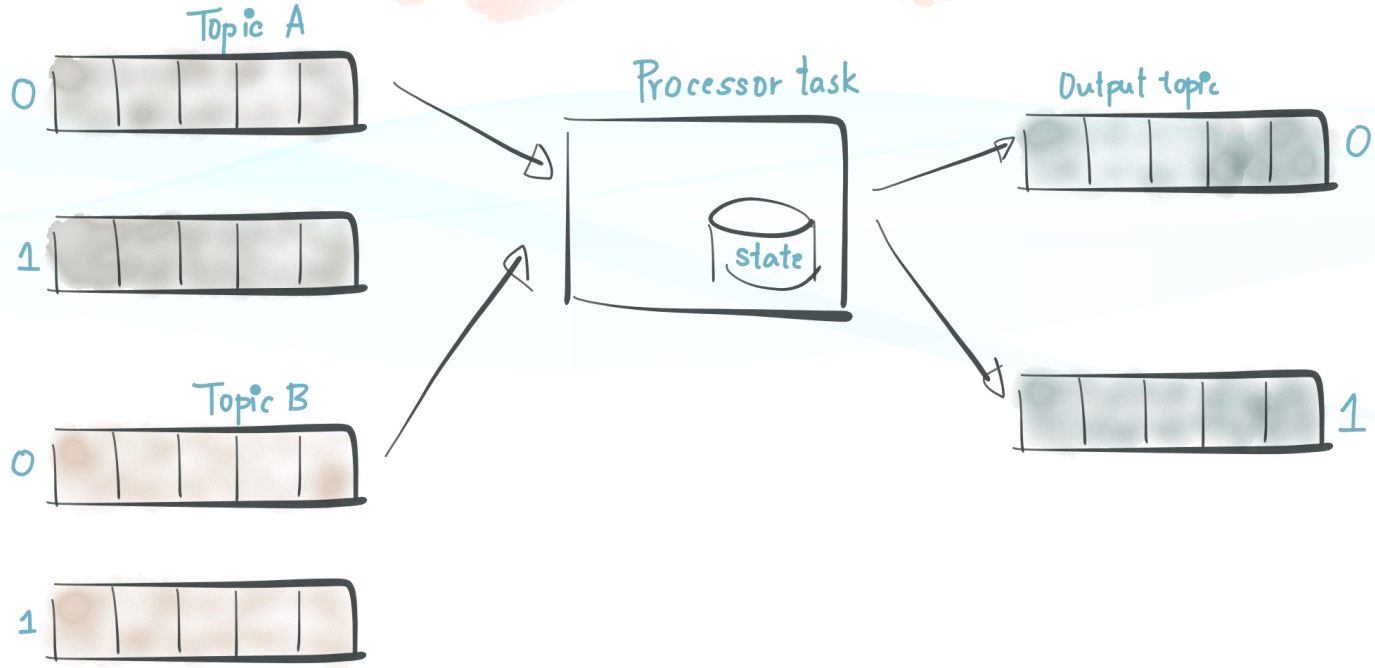
Processor API

```
public static void main(String[] args) throws Exception {
    Properties props = new Properties();
    props.put(StreamingConfig.CLIENT_ID_CONFIG, "Example-Processor-Job");
    props.put(StreamingConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(StreamingConfig.TIMESTAMP_EXTRACTOR_CLASS_CONFIG, WallclockTimestampExtractor.class);
    StreamingConfig config = new StreamingConfig(props);

    TopologyBuilder builder = new TopologyBuilder();
    builder.addSource("SOURCE", new StringDeserializer(), new StringDeserializer(), "topic-source");
    builder.addProcessor("PROCESS", new MyProcessorDef(), "SOURCE");
    builder.addSink("SINK", "topic-sink", new StringSerializer(), new IntegerSerializer(), "PROCESS");
    KafkaStreaming streaming = new KafkaStreaming(builder, config);
    streaming.start();
}
```

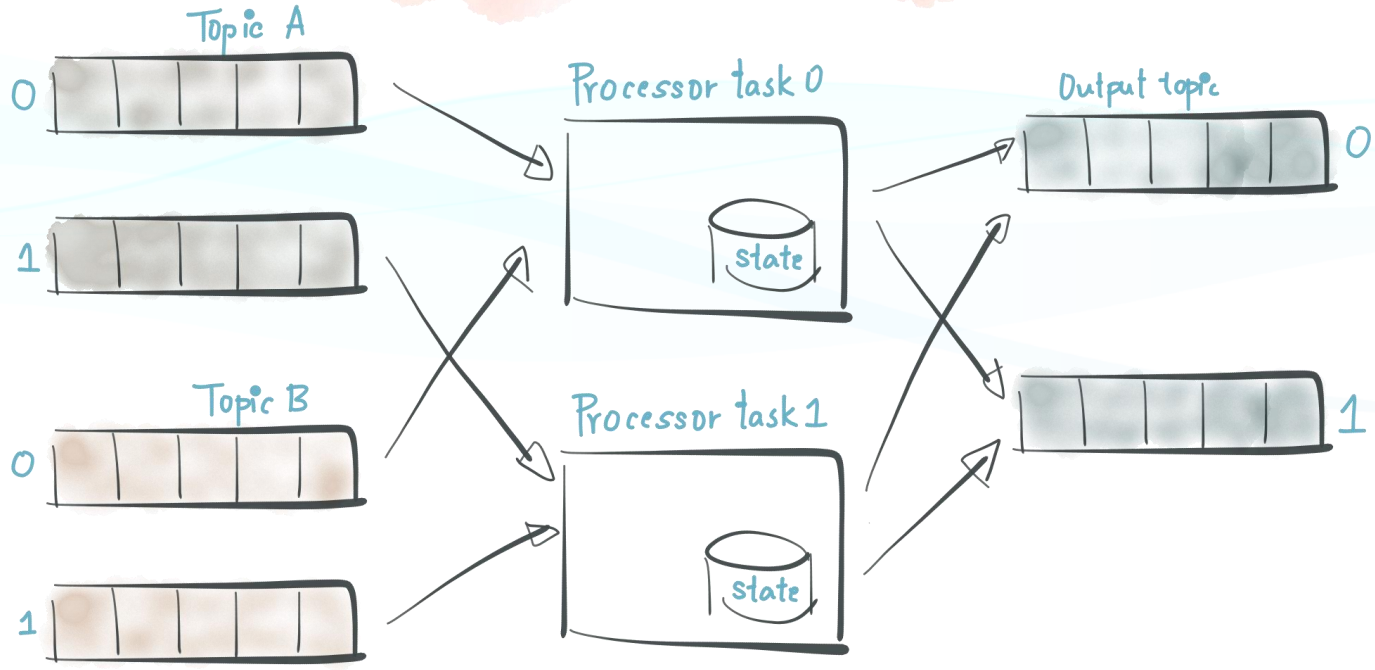
KSTREAMS

```
StreamingConfig config = new StreamingConfig(props);

// build the topology
KStreamBuilder builder = new KStreamBuilder();
KStream<String, String> stream1 = builder.from("topic1");

KStream<String, Integer> stream2 =
    stream1.map((key, value) -> new KeyValue<>(key, value.cost()))
           .aggregate(0, (oldval, newval) -> oldval + newval);

KStream<String, Integer>[] streams = stream2
    .filter((key, value) -> value > 10);

streams[0].sendTo("topic2");
streams[1].sendTo("topic3");

// start the process
KafkaStreaming kstream = new KafkaStreaming(builder, config);
kstream.start();
```

Execution Model

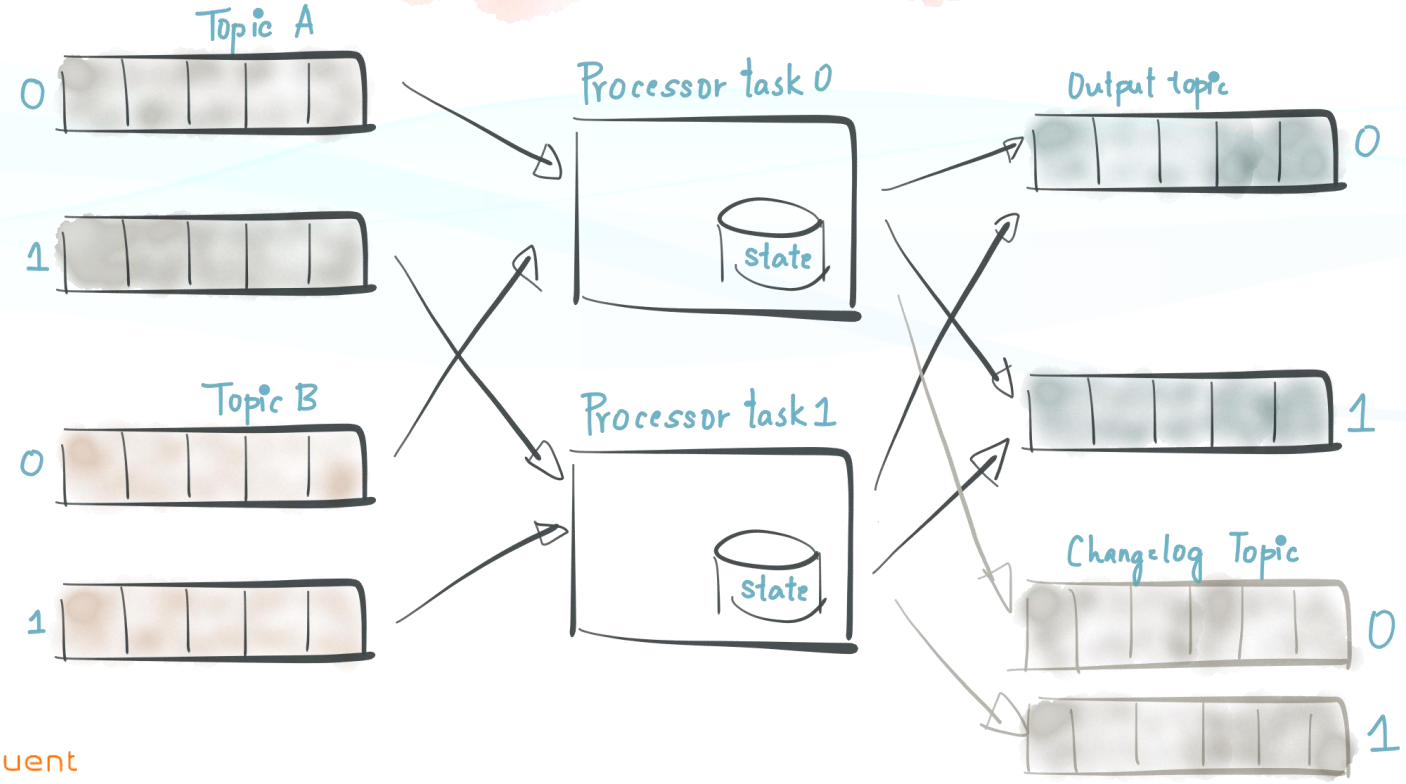


Execution Model

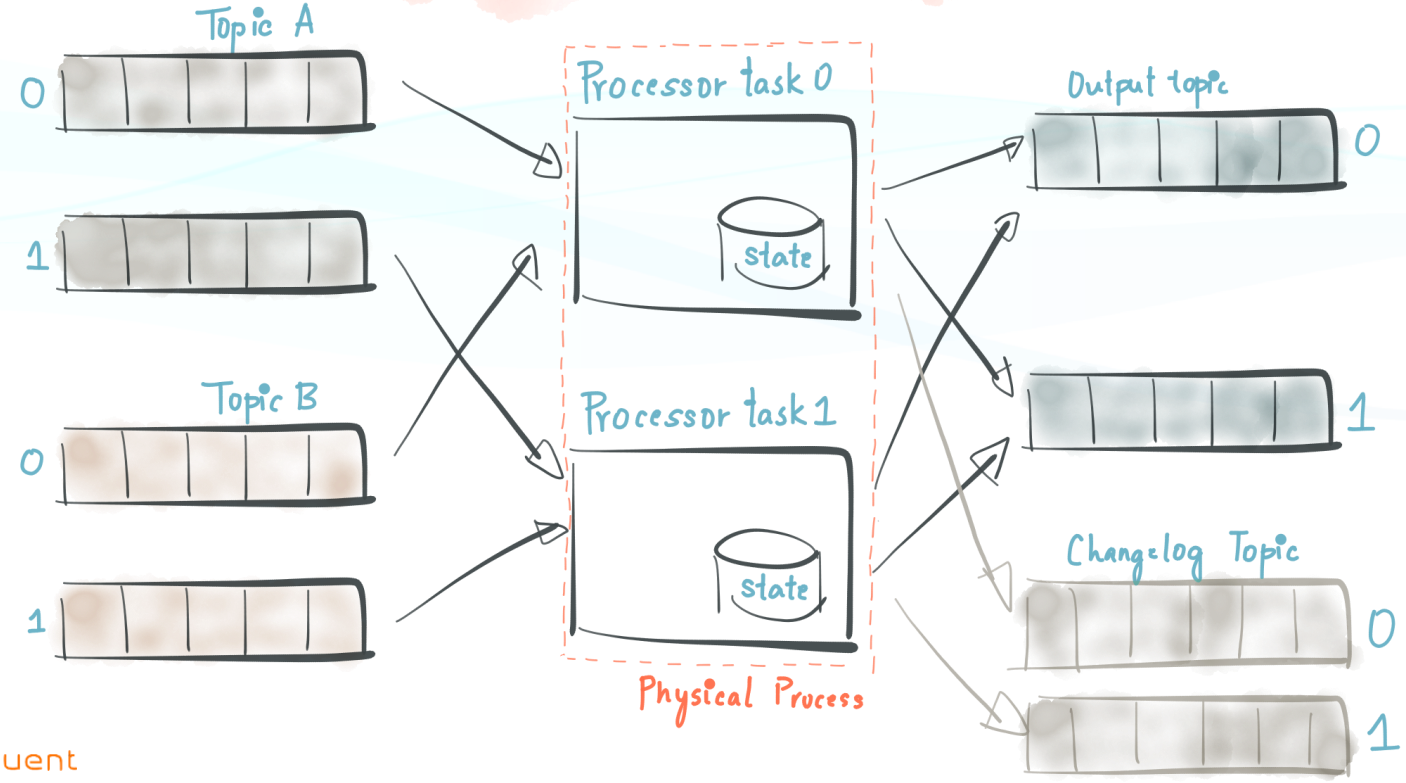


confluent

Execution Model

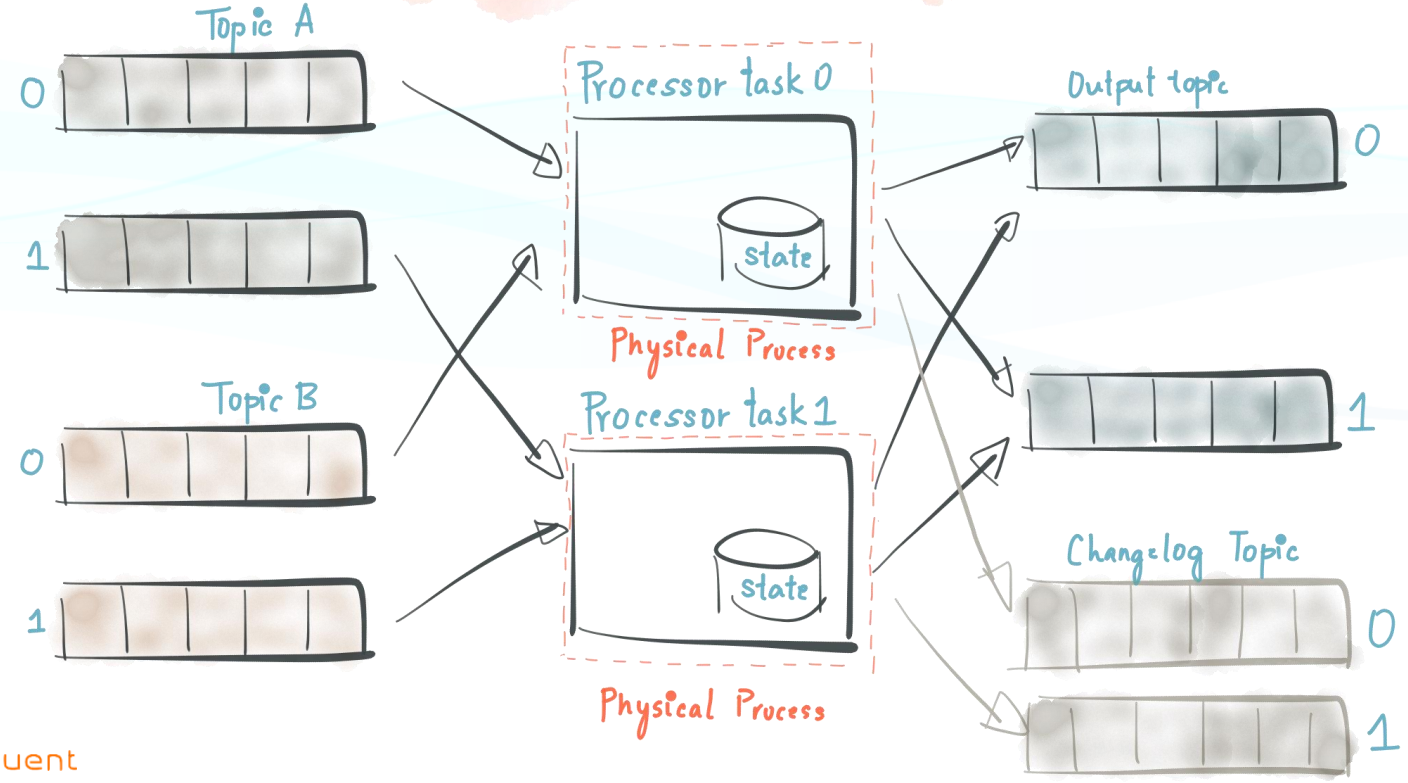


Execution Model



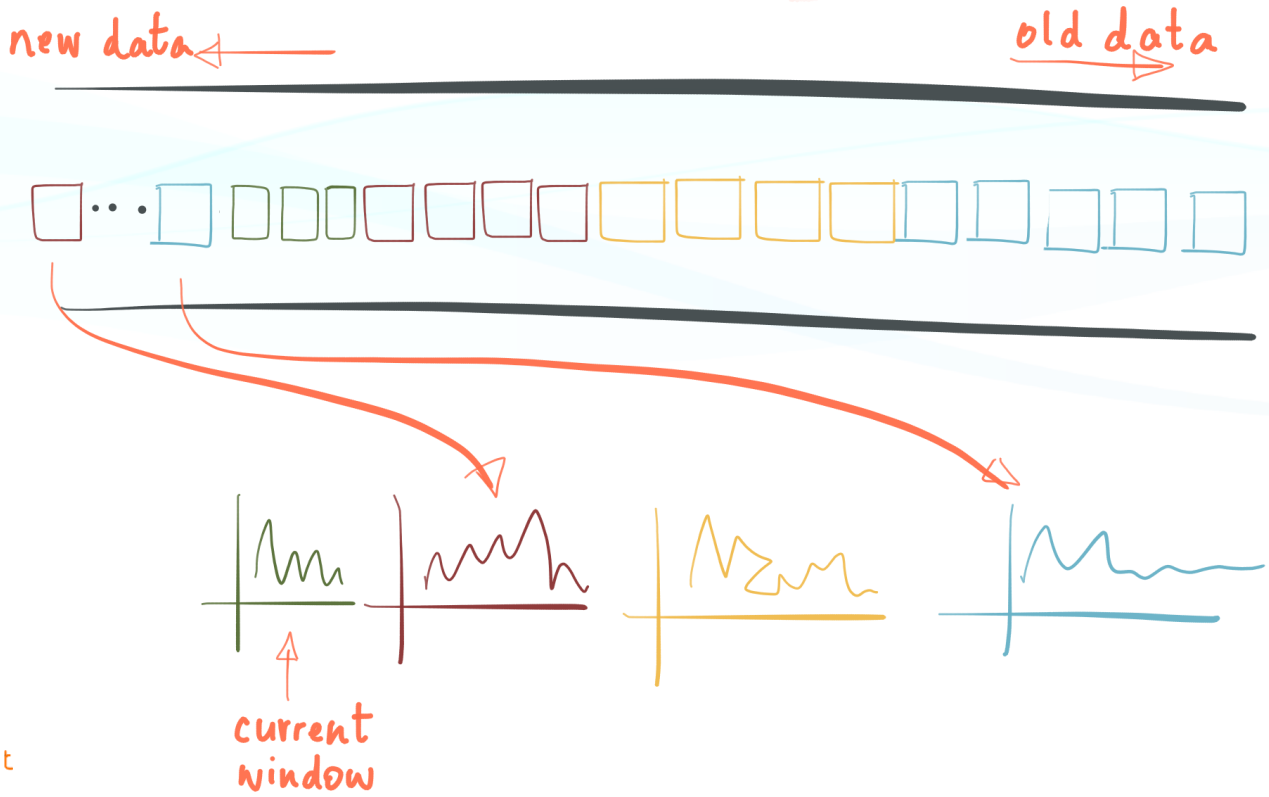
confluent

Execution Model

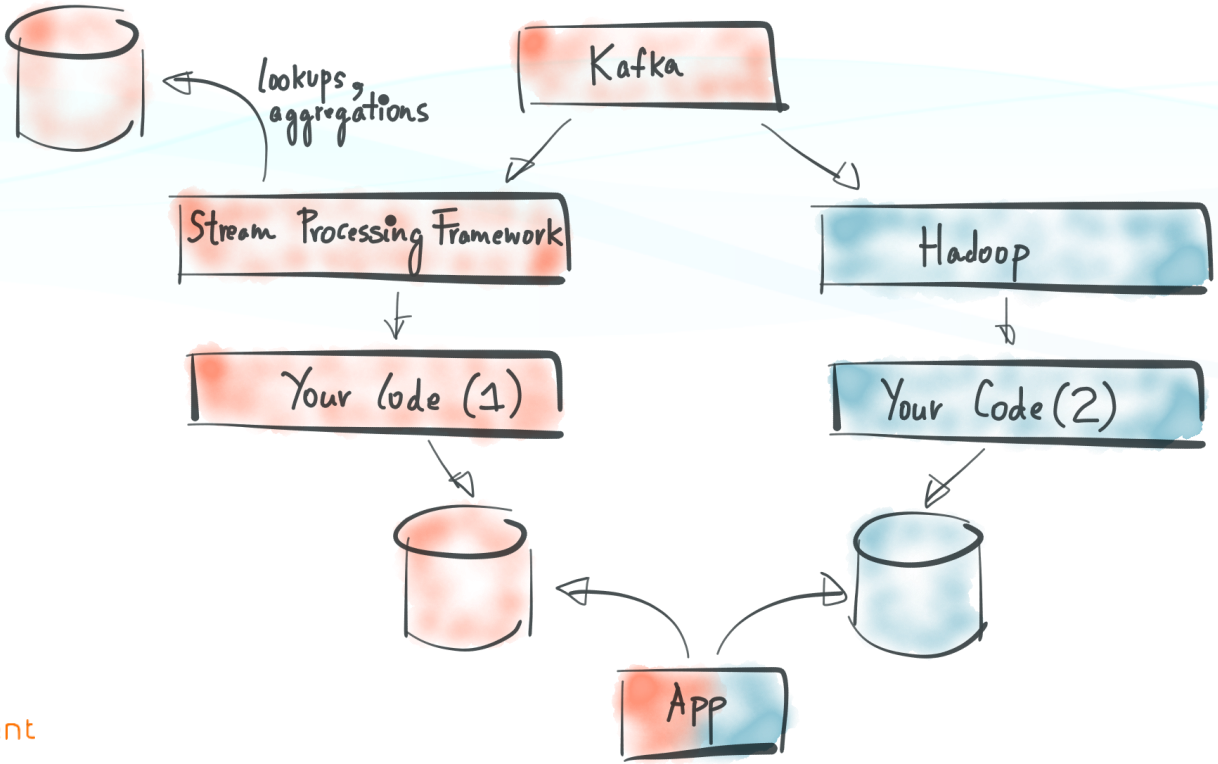


TIME

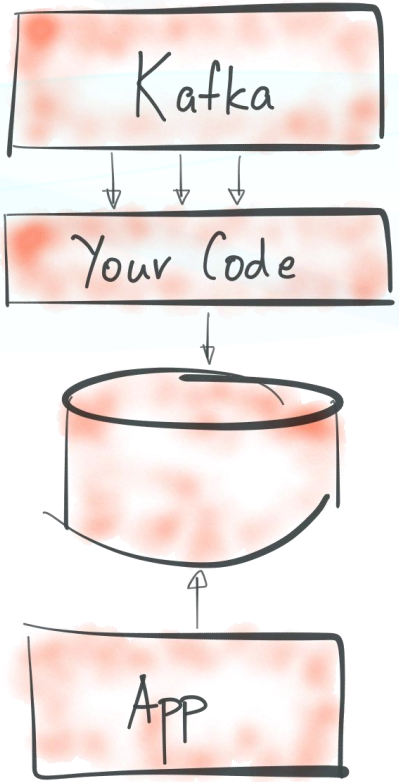
WINDOWING



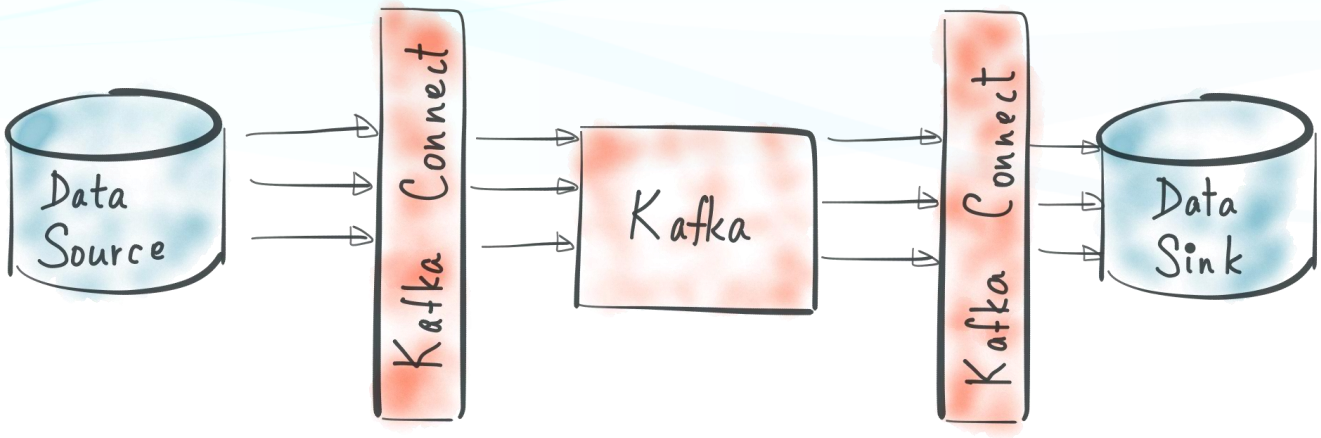
Example Architecture



KAFKA STREAMS



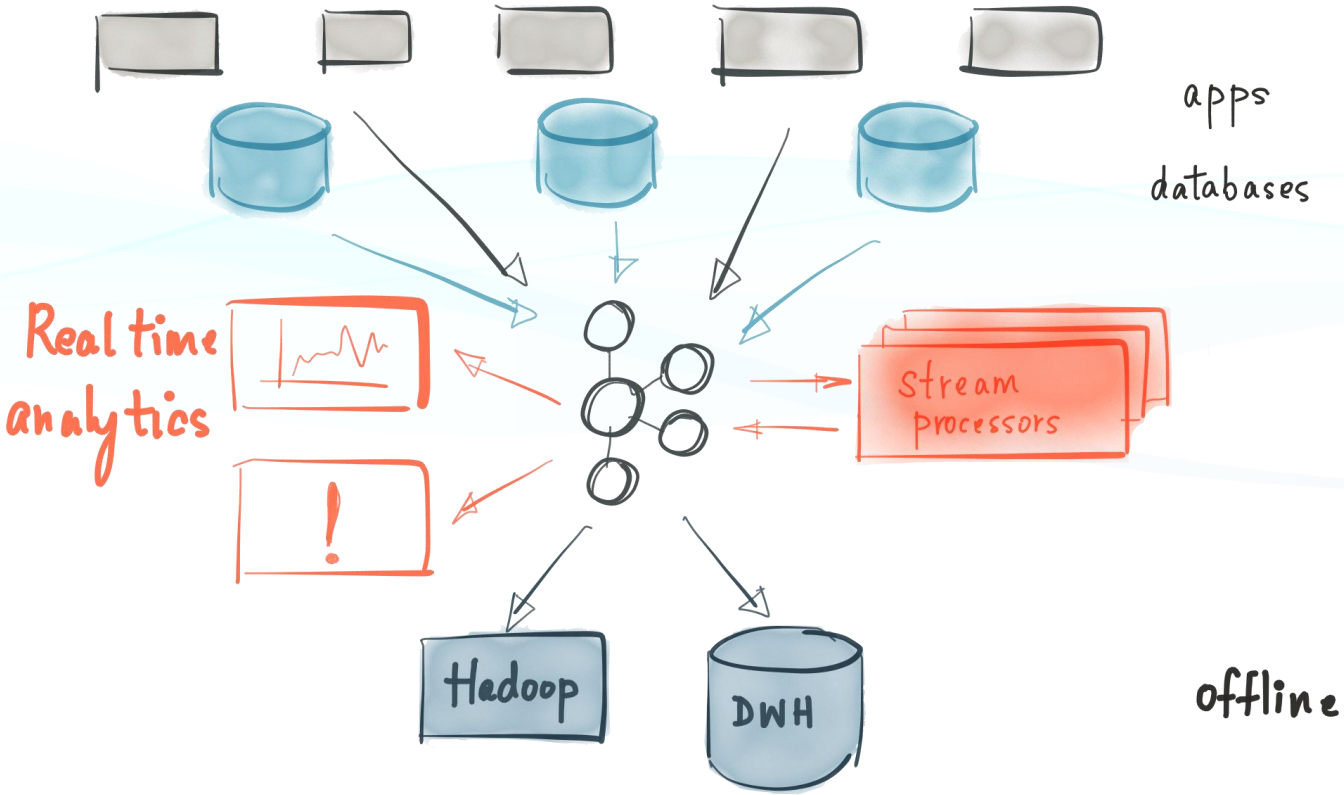
KAFKA CONNECT



Kafka Connect Does The Hard Work

1. Scale out
2. Fault tolerance
3. Central management

Stream Data Platform



THE END

@ewencp | confluent.io/blog

Download Apache Kafka
and Confluent Platform

www.confluent.io/download