

Kafka Streams

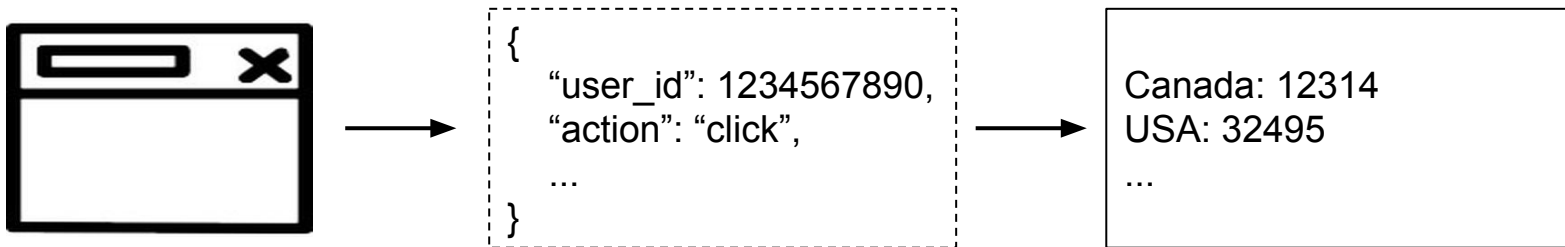
The easiest way to start with stream processing

demonware

Stream processing

Modern streams of data

- Web and mobile analytics (clicks, page views, etc.)
- IoT sensors
- Metrics, logs and telemetry
- ...



Data processing styles

- **Batch processing:**
 - Slow, expensive, not very flexible, etc.
 - Mostly for reporting
 - Way to go, historically
- **Stream processing:**
 - Balance between latency and throughput, easy to redeploy
 - Near-realtime features
 - Last 5-10 years

Stream processing operations (“Streaming ETL”)

- Validation, transformation, enrichment, deduplication, ...
- Aggregations
- Joins
- Windowing
- Integrations and Storage

Stream processing challenges

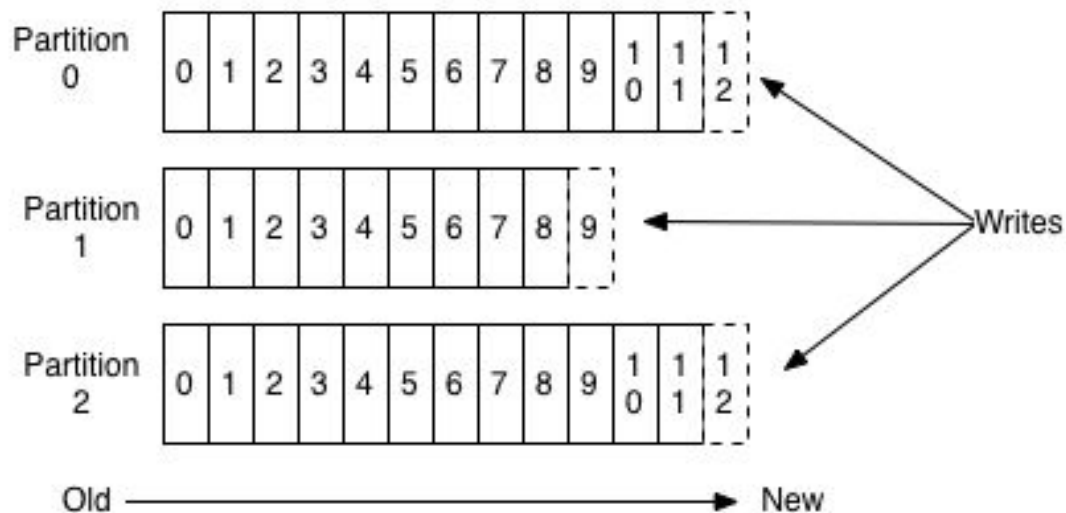
- Delivery guarantees
- Latency / Throughput
- Fault tolerance
- Backpressure
- Event-time vs. Processing-time

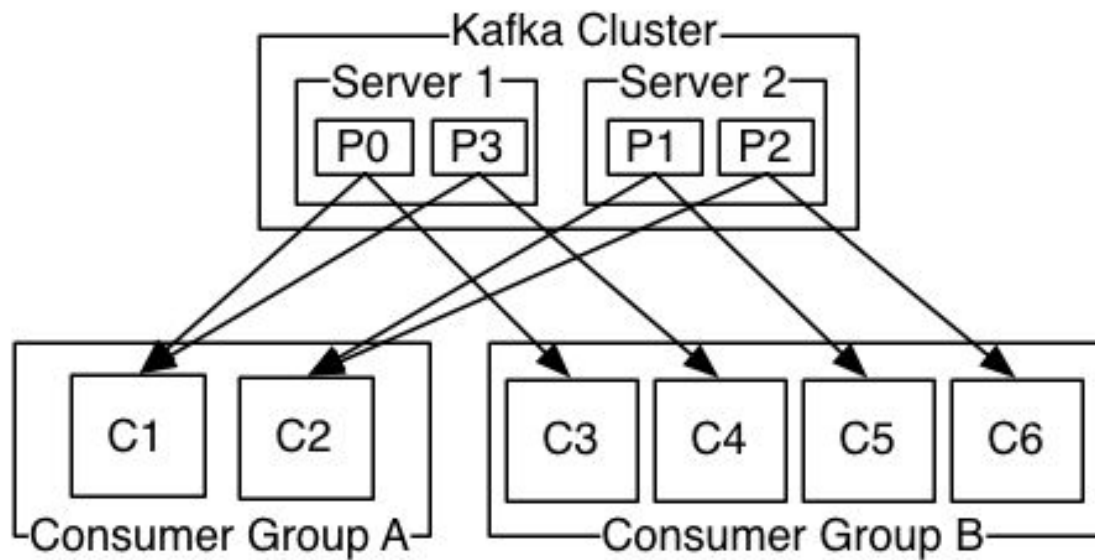
Stream processing techniques

- One-message-at-a-time OR Micro-batches
- State management (in-memory, on-disk, replicated)

Kafka

Anatomy of a Topic



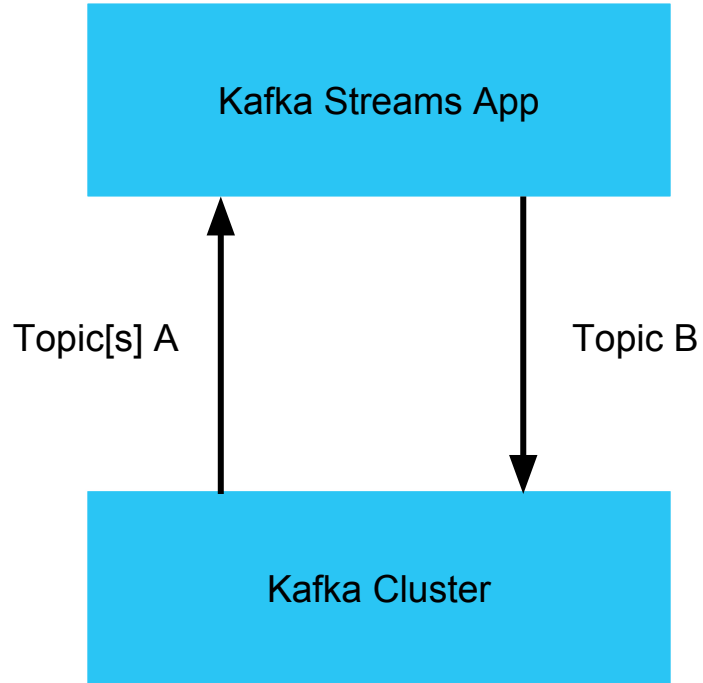


Kafka Streams

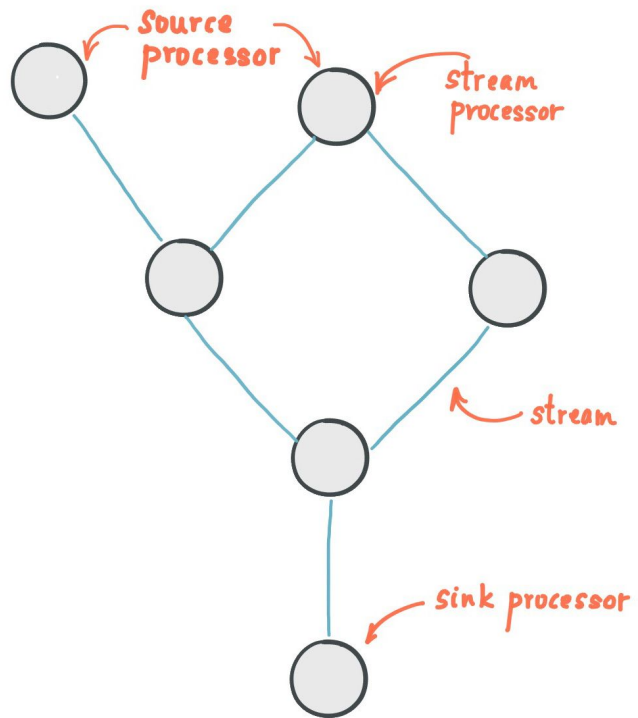
Kafka Streams

Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in Kafka clusters. It combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology.

- Available since Kafka 0.10 (May 2016)
- Java/Scala support
- Heavily relies on underlying Kafka cluster
- Need to integrate with external persistent systems? Use Kafka Connect



```
KStreamBuilder builder = new KStreamBuilder();  
  
KStream<byte[], String> textLines = builder.stream("TextLinesTopic");  
  
textLines  
    .mapValues((textLine) -> textLine.toUpperCase())  
    .to("UppercasedTextLinesTopic");
```



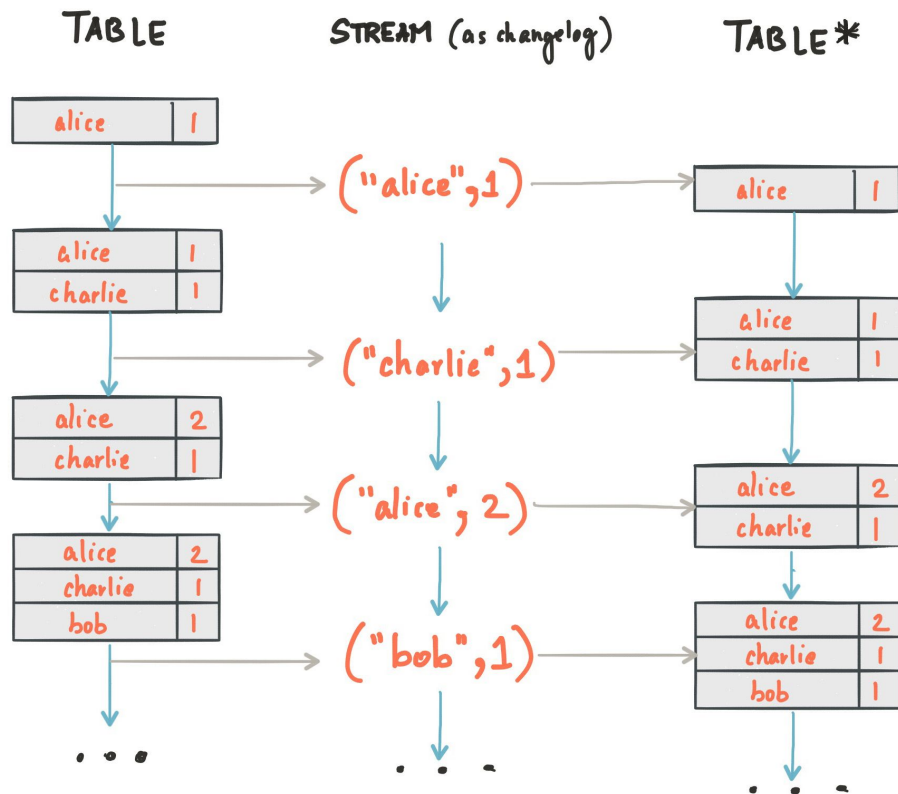
PROCESSOR TOPOLOGY

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> textLines = builder.stream("streams-plaintext-input");

KTable<String, Long> wordCounts = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    .groupBy((key, value) -> value)
    .count();

wordCounts.toStream().to("streams-wordcount-output");
```



Syntax

Streams DSL

- Declarative
- Functional
- Implicit state store management
- Stateless or stateful

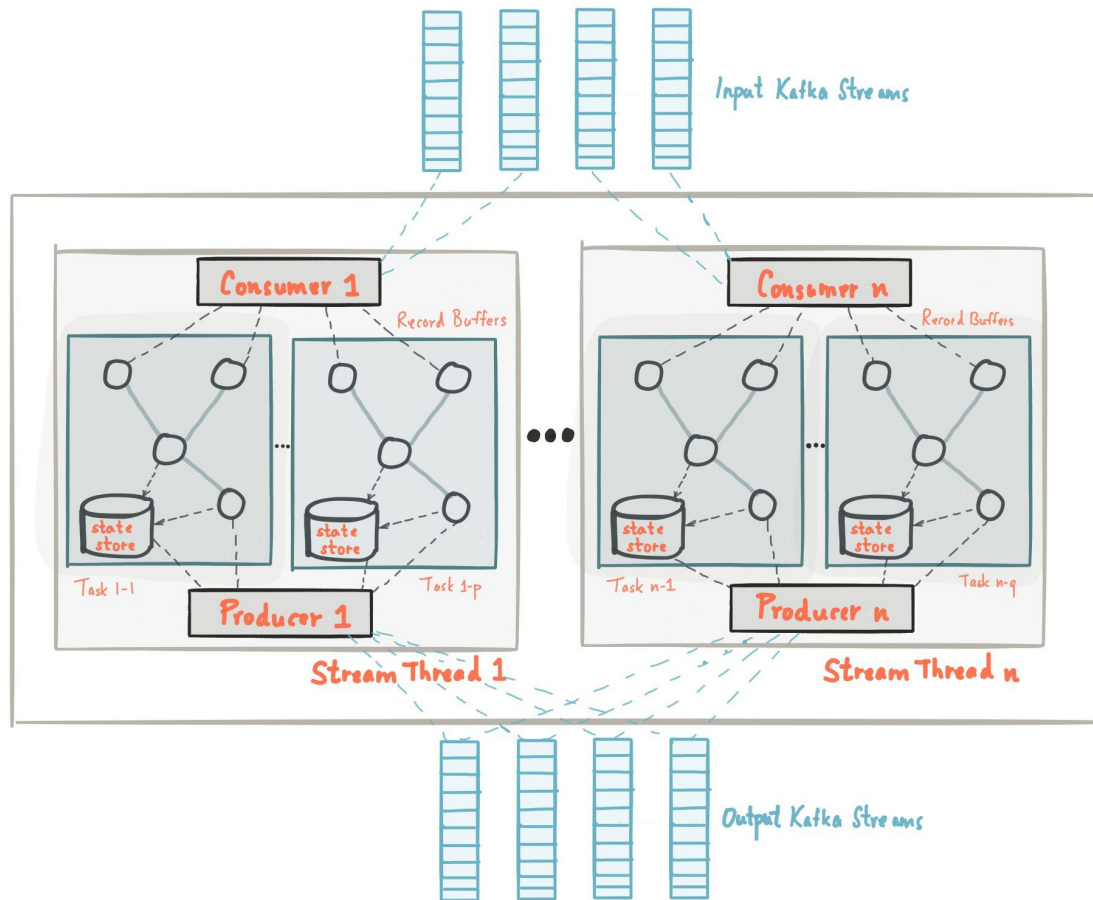
Low-level Processor API

- Imperative
- Explicit state store management
- Usually stateful

Kafka Streams doesn't
require YARN, Mesos,
Zookeeper, HDFS, etc.

Just a Kafka cluster*

*That you probably already have



Config example (HOCON)

```
kafka.brokers = "broker1:9092,broker2:9092,broker3:9092,..."
```

```
kafka.topics = [  
  {from: "topic-a", to: "topic-b"},  
  {from: "topic-c", to: "topic-d"},  
  ...  
]
```

```
streams {  
  threads = 8  
  replication-factor = 3  
  producer {  
    acks = all  
  }  
}
```

```
consumer.auto.offset.reset = latest
```

Serializers/Deserializers

Every Kafka Streams application must provide SerDes (Serializer/Deserializer) for the data types of record keys and record values (e.g. `java.lang.String` or Avro objects) to materialize the data when necessary.

You can provide SerDes by using either of these methods:

- By setting default SerDes via a `StreamsConfig` instance.
- By specifying explicit SerDes when calling the appropriate API methods, thus overriding the defaults.

Serializers/Deserializers

```
Properties settings = new Properties();
settings.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
settings.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass().getName());
StreamsConfig config = new StreamsConfig(settings);
```

```
Serde<String> stringSerde = Serdes.String();
Serde<Long> longSerde = Serdes.Long();
KStream<String, Long> userCountByRegion = ...;
userCountByRegion.to("RegionCountsTopic", Produced.with(stringSerde, longSerde));
```

State stores

- Used for any stateful operation, implicitly or explicitly
- Backed by local RocksDB databases AND replicated changeset topic in Kafka
- Application's entire state is spread across the local state stores (following the same partitioning rules)
- Can be queried with standard API (even remotely)
- Support for key-value, window and custom stores

// writing

```
KStreamBuilder builder = ...;  
KStream<String, String> textLines = ...;
```

```
textLines  
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))  
    .groupBy((key, word) -> word, Serialized.with(stringSerde, stringSerde))  
    .count(Materialized.<String, String, KeyValueStore<Bytes, byte[]>as("CountsKeyValueStore"));
```

```
KafkaStreams streams = new KafkaStreams(builder, getSettings());  
streams.start();
```

// reading

```
ReadOnlyKeyValueStore<String, Long> keyValueStore = streams.store("CountsKeyValueStore",  
    QueryableStoreTypes.keyValueStore());
```

```
System.out.println("count for hello:" + keyValueStore.get("hello"));
```

```
KeyValueIterator<String, Long> range = keyValueStore.all();  
while (range.hasNext()) {  
    KeyValue<String, Long> next = range.next();  
    System.out.println("count for " + next.key + ": " + next.value);  
}
```

Aggregations

- Aggregate
- Reduce
- Count

All support windows.

Windowing

Window name	Behavior	Short description
Tumbling time window	Time-based	Fixed-size, non-overlapping, gap-less windows
Hopping time window	Time-based	Fixed-size, overlapping windows
Sliding time window	Time-based	Fixed-size, overlapping windows that work on differences between record timestamps
Session window	Session-based	Dynamically-sized, non-overlapping, data-driven windows

Windowing

```
KStream<String, GenericRecord> pageViews = ...;  
  
KTable<Windowed<String>, Long> windowedPageViewCounts = pageViews  
    .groupByKey()  
    .windowedBy(TimeWindows.of(TimeUnit.MINUTES.toMillis(5)))  
    .count();
```

Time

- **Event-time:** “user”-defined time, generated by the application that uses Producer API
- **Processing-time:** Time when the record is being consumed (pretty much anytime)
- **Ingestion-time:** generated by the Kafka brokers, embedded in any message

Timestamp extractors can be used to achieve event-time semantics.

Joins

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN
KStream-to-KStream	Windowed	Supported	Supported	Supported
KTable-to-KTable	Non-windowed	Supported	Supported	Supported
KStream-to-KTable	Non-windowed	Supported	Supported	Not Supported
KStream-to-GlobalKTable	Non-windowed	Supported	Supported	Not Supported
KTable-to-GlobalKTable	N/A	Not Supported	Not Supported	Not Supported

Joins

```
KStream<String, Long> left = ...;  
KTable<String, Double> right = ...;  
KStream<String, String> joined = left.join(right,  
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue  
);
```

Example

```
builder.stream("play-events", Consumed.with(Serdes.String(), playEventSerde))  
  // group by key so we can count by session windows  
  .groupByKey(Serialized.with(Serdes.String(), playEventSerde))  
  // window by session  
  .windowedBy(SessionWindows.with(TimeUnit.MINUTES.toMillis(30)))  
  // count play events per session  
  .count(Materialized.<String, Long, SessionStore<Bytes, byte[]>>as("PlayEventsPerSession")  
    .withKeySerde(Serdes.String())  
    .withValueSerde(Serdes.Long()))  
  // convert to a stream so we can map the key to a string  
  .toStream()  
  // map key to a readable string  
  .map((key, value) -> new KeyValue<>(key.key() + "@" + key.window().start() + "->" +  
key.window().end(), value))  
  // write to play-events-per-session topic  
  .to("play-events-per-session", Produced.with(Serdes.String(), Serdes.Long()));
```

Example

jo@1484823406597->1484823406597 = 1
bill@1484823466597->1484823466597 = 1
sarah@1484823526597->1484823526597 = 1
jo@1484825207597->1484825207597 = 1
bill@1484823466597->1484825206597 = 2
sarah@1484827006597->1484827006597 = 1
jo@1484823406597->1484825207597 = 3
bill@1484828806597->1484828806597 = 1
sarah@1484827006597->1484827186597 = 2
...

Summary

Pros

- Rich Streams DSL provides very expressive language, low-level Processor API gives a lot of flexibility
- Seamless Kafka integration including exactly-once semantics
- No external dependencies
- Great fault-tolerance, “Cloud-ready” features, very easy to scale
- Stream/Table duality just makes sense!
- Built-in backpressure using Kafka Consumer API
- Easy to monitor with tons of metrics exposed over JMX

Cons

- Still very young framework (make sure to use 0.11+)
- Only support Kafka topics as sources and sinks (add external systems using Kafka Connect)
- Only support one Kafka cluster
- No true Batch API
- No ML functionality (but can easily integrate any JVM library)
- KSQL is in development preview
- Streams DSL can create [A LOT of internal topics](#)
- No Async IO support
- Scalability is limited (up to MAX number of partitions for all input topics)

So, should I start
using Kafka
Streams now?

Yes, if you're comfortable with
JVM and want to start using
stream processing

No, if you're happy with your
Spark/Flink environments

Maybe, if you just need to read & write to Kafka

Questions?

@sap1ens

Resources

- <https://docs.confluent.io/current/streams/index.html>
- <https://kafka.apache.org/documentation/streams/>
- <https://www.confluent.io/blog/watermarks-tables-event-time-dataflow-model/>
- <https://www.confluent.io/blog/enabling-exactly-kafka-streams/>
- <https://github.com/confluentinc/kafka-streams-examples>
- <http://mkuthan.github.io/blog/2017/11/02/kafka-streams-dsl-vs-processor-api/>
- <https://sap1ens.com/blog/2018/01/03/message-enrichment-with-kafka-streams/>