Apache Kafka

and the Rise of Stream Processing

Guozhang Wang

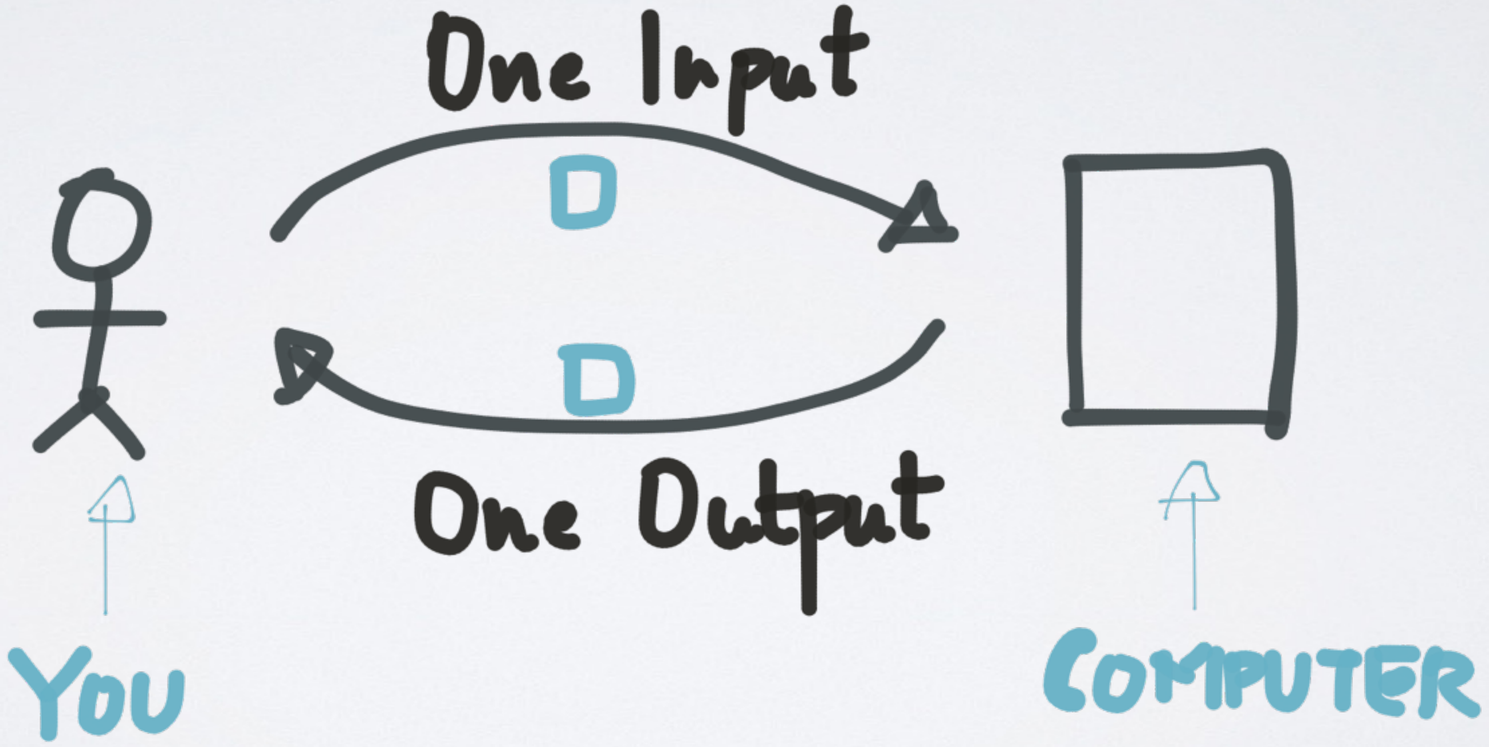
QCon, April 17, 2017

*What is **NOT** Stream Processing?*

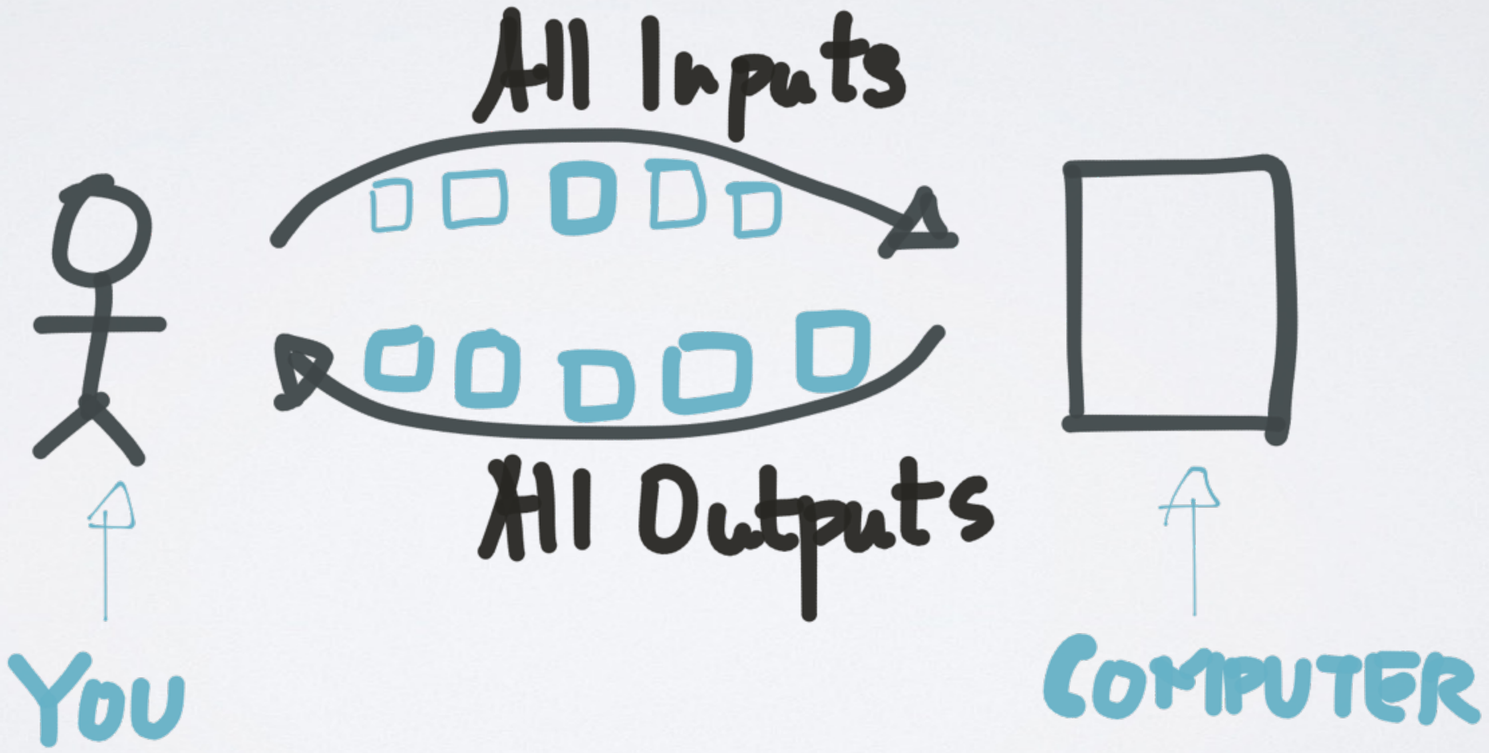
Stream Processing isn't (necessarily)

- *Transient, approximate, lossy…*
- *.. that you must have batch processing as safety net*

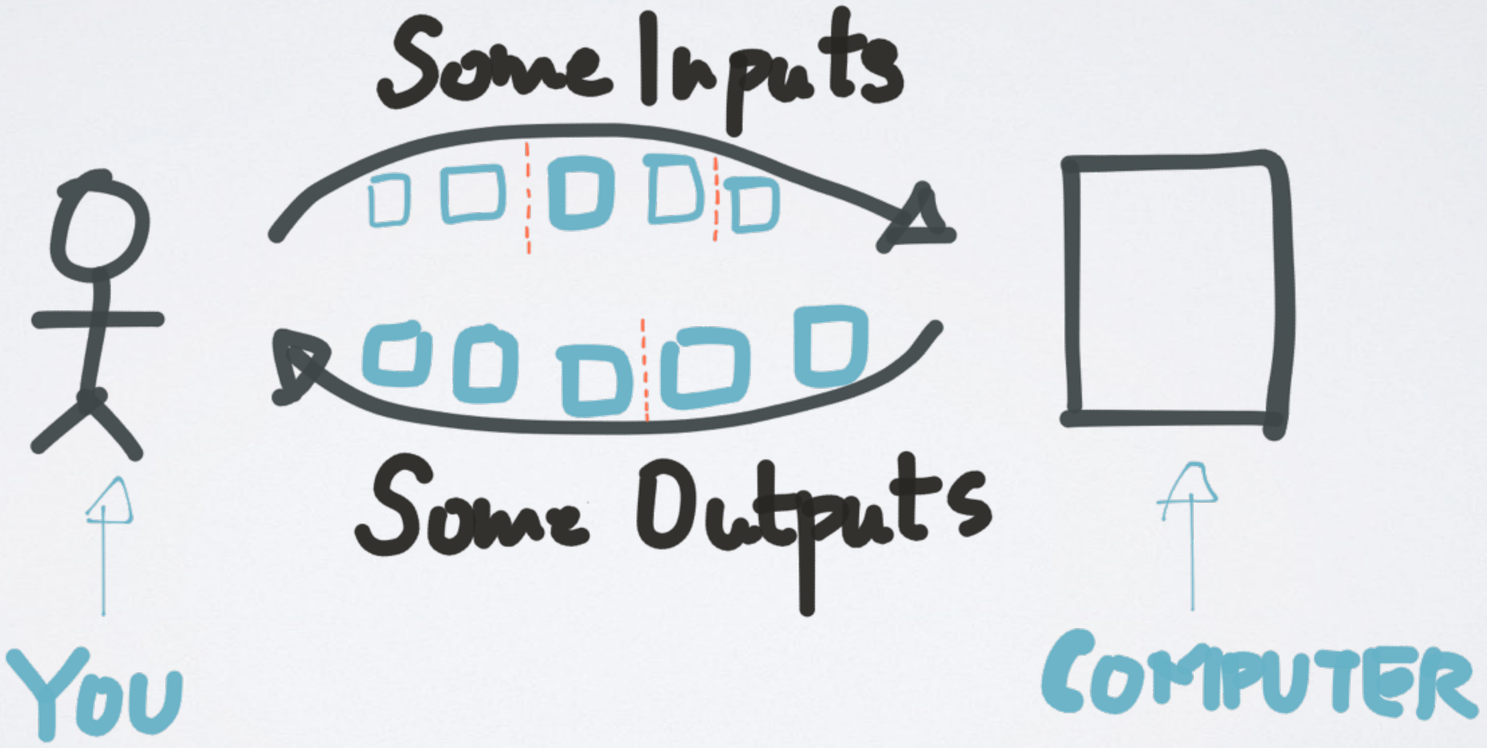
Request / Response



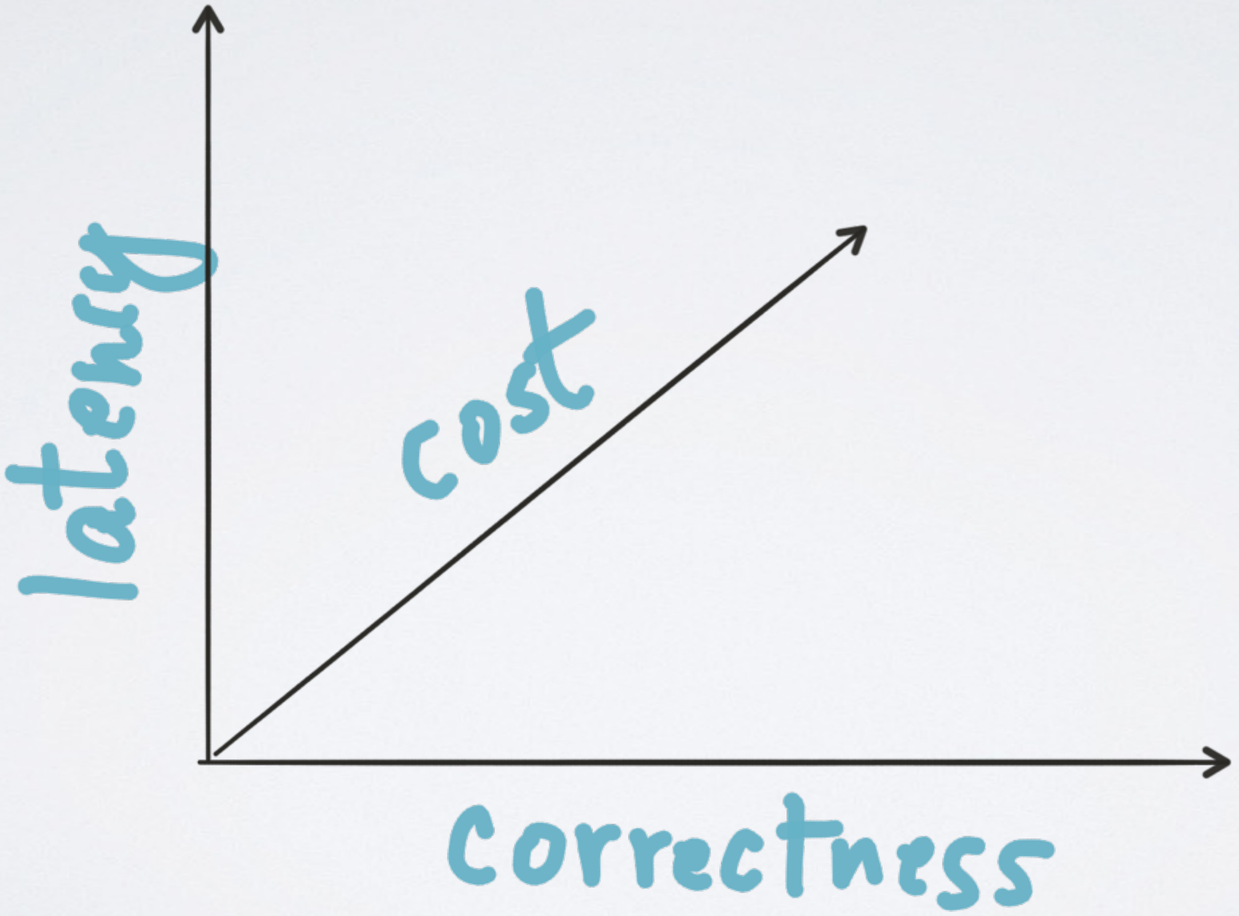
Batch



Stream Processing



TRADE OFFS



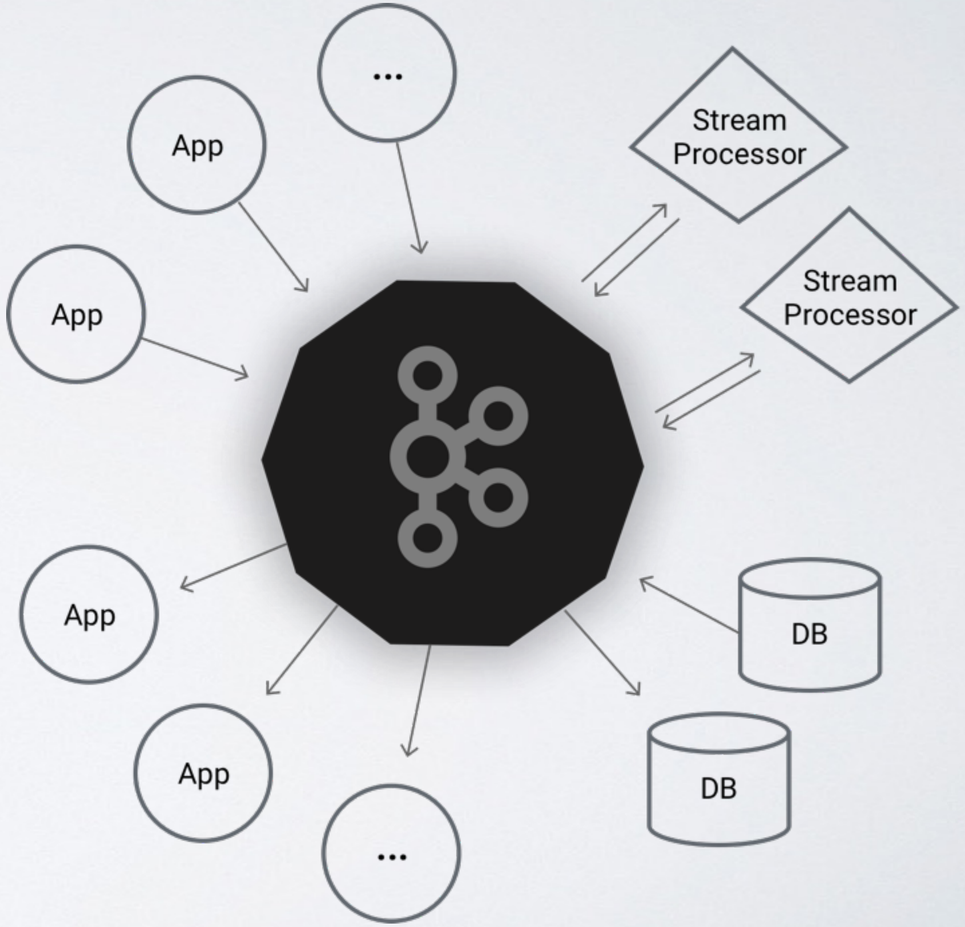
Stream Processing

- *A different programming paradigm*
  - *.. that brings computation to **unbounded** data*
  - *.. with tradeoffs between latency / cost / correctness*

*Why **Kafka** in Stream Processing?*

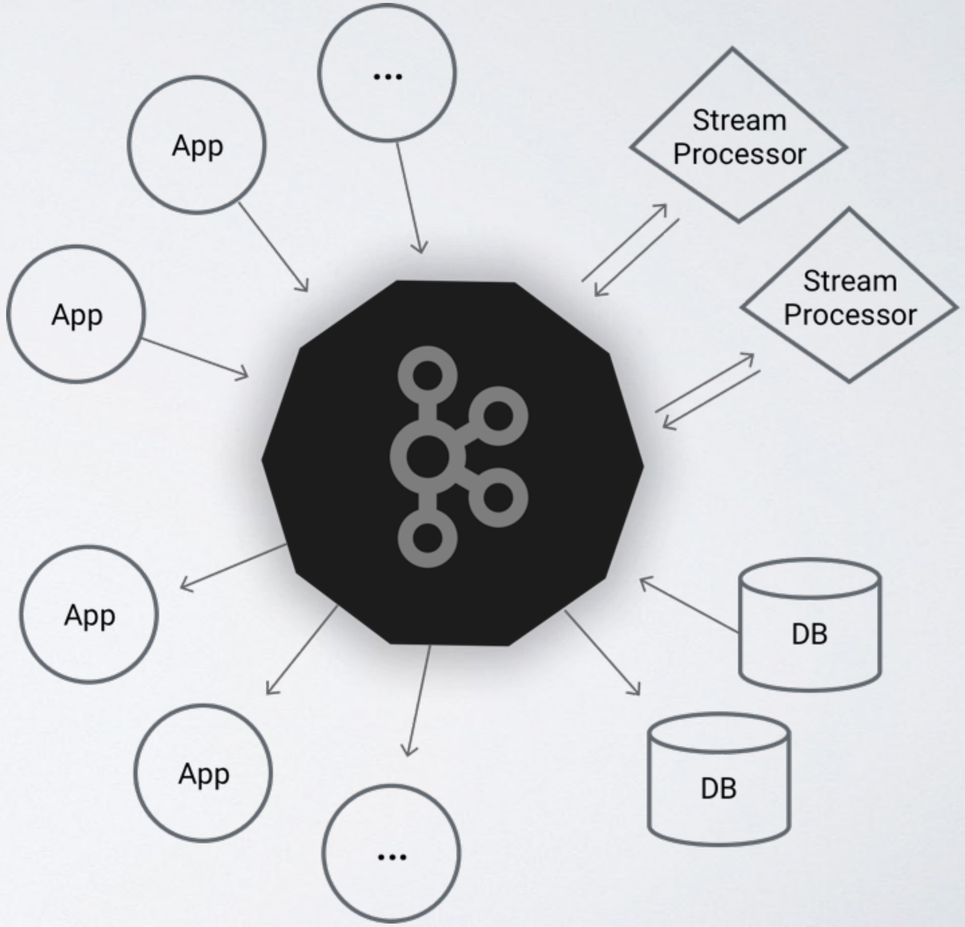
Kafka: Streaming Platform

- *Persistent Buffering*
- *Logical Ordering*
- *Scalable "source-of-truth"*



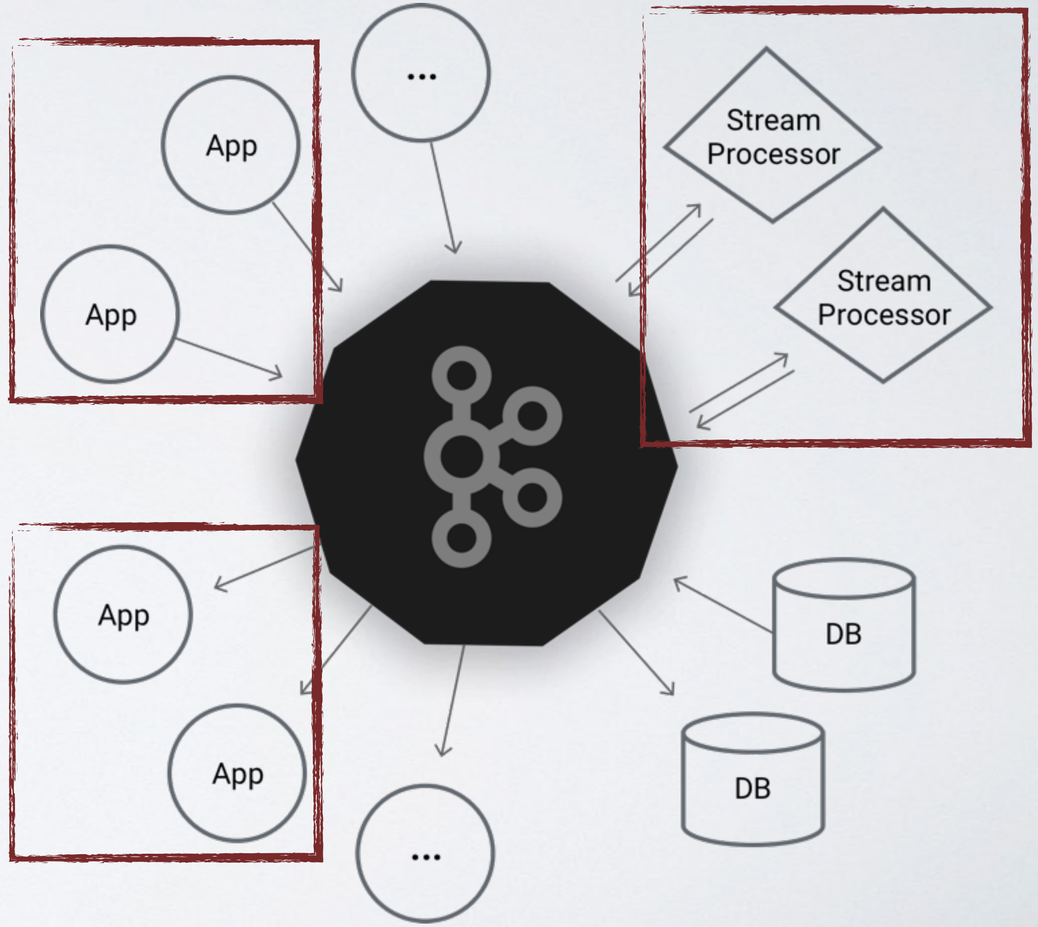
Kafka: Streaming Platform

- *Publish / Subscribe*
  - *Move data around as online streams*
- *Logical Ordering*
- *Store*
- *Scalable “source-of-truth”*
  - *“Source-of-truth” continuous data*
- *Process*
  - *React / process data in real-time*



Kafka: Streaming Platform

- *Publish / Subscribe*
  - *Move data around as online streams*
- *Store*
  - *"Source-of-truth" continuous data*
- *Process*
  - *React / process data in real-time*



Stream Processing with Kafka

Stream Processing with Kafka

- *Option I: Do It Yourself !*

Stream Processing with Kafka

- *Option I: Do It Yourself !*

```
while (isRunning) {
    // read some messages from Kafka
    inputMessages = consumer.poll();

    // do some processing…

    // send output messages back to Kafka
    producer.send(outputMessages);
}
```

DIY Stream Processing is *Hard*

- *Ordering*
- *Partitioning & Scalability*
- *Fault tolerance*
- *State Management*
- *Time, Window & Out-of-order Data*
- *Re-processing*

Stream Processing with Kafka

- *Option I: Do It Yourself !*

- *Option II: full-fledged stream processing system*
  - *Storm, Spark, Flink, Samza, ..*


ONE DOES NOT SIMPLY PROCESS DATA
WITHOUT A CLUSTER
imgflip.com

MapReduce Heritage?

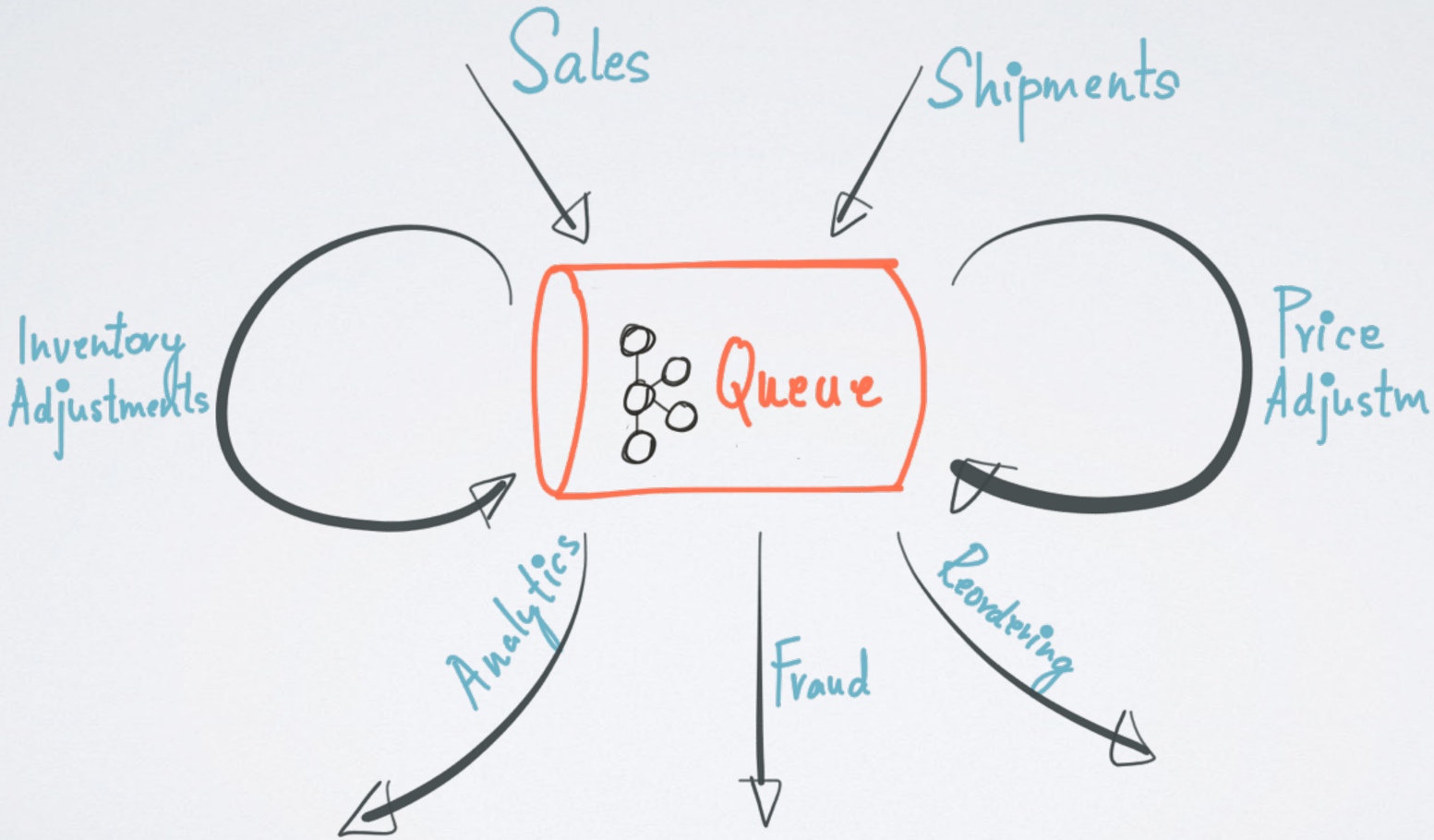
- *Config Management*
- *Resource Management*
- *Deployment*
- *etc..*

MapReduce Heritage?

- *Config Management*
- *etc..*

Can I just use my own?!

Example: Async. Micro-Services



Stream Processing with Kafka

- *Option I: Do It Yourself !*
- *Option II: full-fledged stream processing system*
- *Option III: lightweight stream processing* ***library***

Kafka Streams (0.10+)

- *New client library besides producer and consumer*
- *Powerful yet easy-to-use*
  - *Event-at-a-time, Stateful*
  - *Windowing with out-of-order handling*
  - *Highly scalable, distributed,* fault tolerant
  - *and more..*

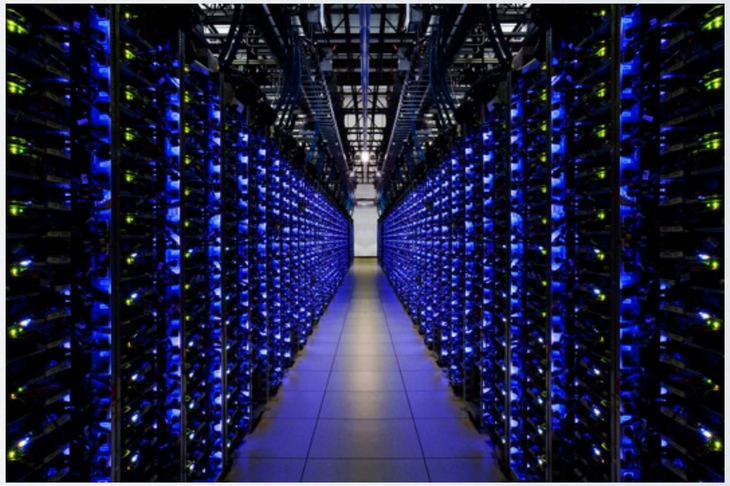
Anywhere, anytime

Ok.

Ok.

Ok.

Ok.



Anywhere, anytime

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
  <version>0.10.0.0</version>
</dependency>
```

Anywhere, anytime

Simple *is* ***Beautiful***

Kafka Streams DSL

```
public static void main(String[] args) {
    // specify the processing topology by first reading in a stream from a topic
    KStream<String, String> words = builder.stream(”topic1”);

    // count the words in this stream as an aggregated table
    KTable<String, Long> counts = words.countByKey(”Counts”);

    // write the result table to a new topic
    counts.to(”topic2”);

    // create a stream processing instance and start running it
    KafkaStreams streams = new KafkaStreams(builder, config);
    streams.start();
}
```

Kafka Streams DSL

```
public static void main(String[] args) {
    // specify the processing topology by first reading in a stream from a topic
    KStream<String, String> words = builder.stream("topic1");

    // count the words in this stream as an aggregated table
    KTable<String, Long> counts = words.countByKey("Counts");

    // write the result table to a new topic
    counts.to("topic2");

    // create a stream processing instance and start running it
    KafkaStreams streams = new KafkaStreams(builder, config);
    streams.start();
}
```

Kafka Streams DSL

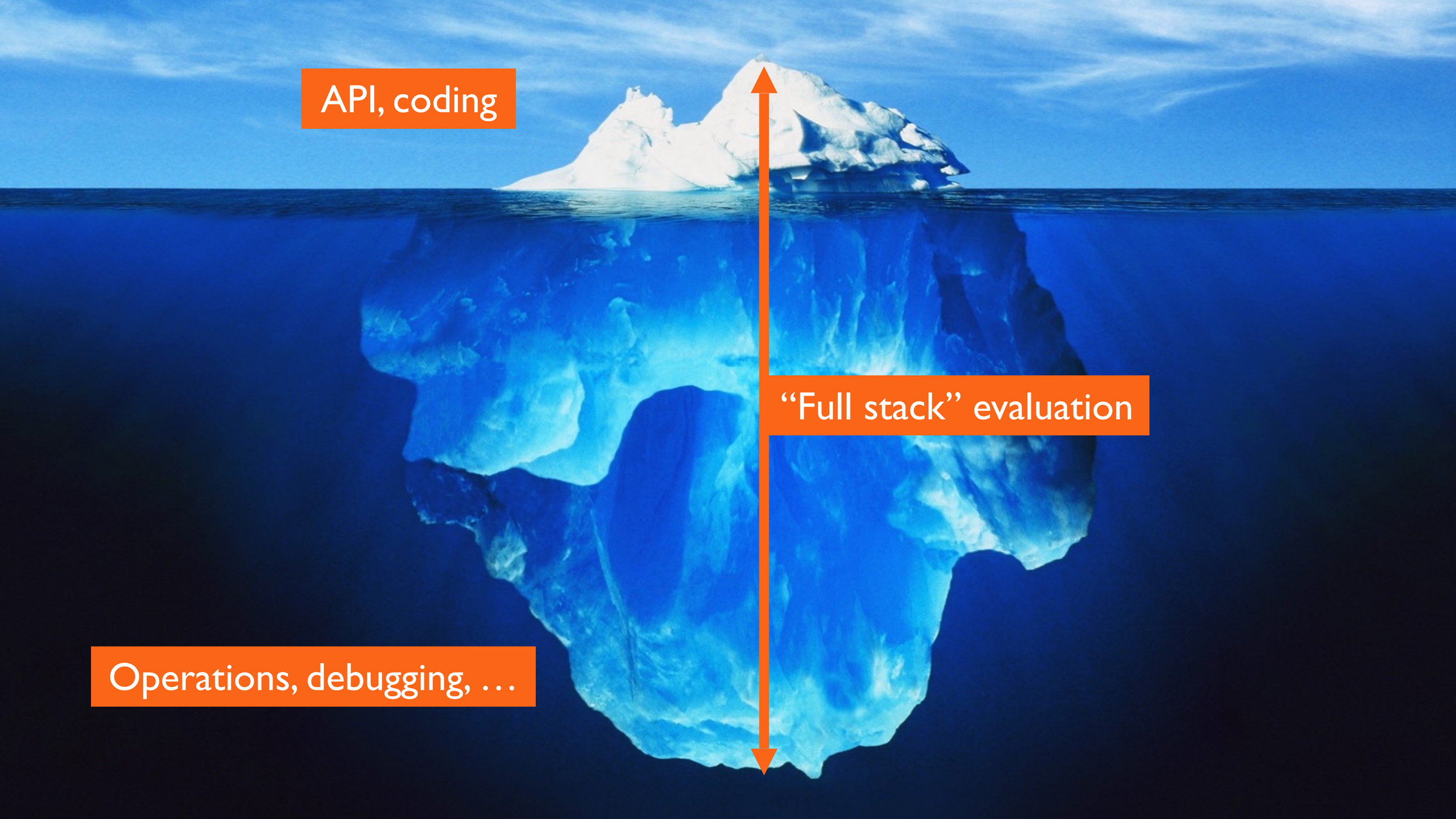
```
public static void main(String[] args) {
    // specify the processing topology by first reading in a stream from a topic
    KStream<String, String> words = builder.stream("topic1");

    // count the words in this stream as an aggregated table
    KTable<String, Long> counts = words.countByKey("Counts");

    // write the result table to a new topic
    counts.to("topic2");

    // create a stream processing instance and start running it
    KafkaStreams streams = new KafkaStreams(builder, config);
    streams.start();
}
```

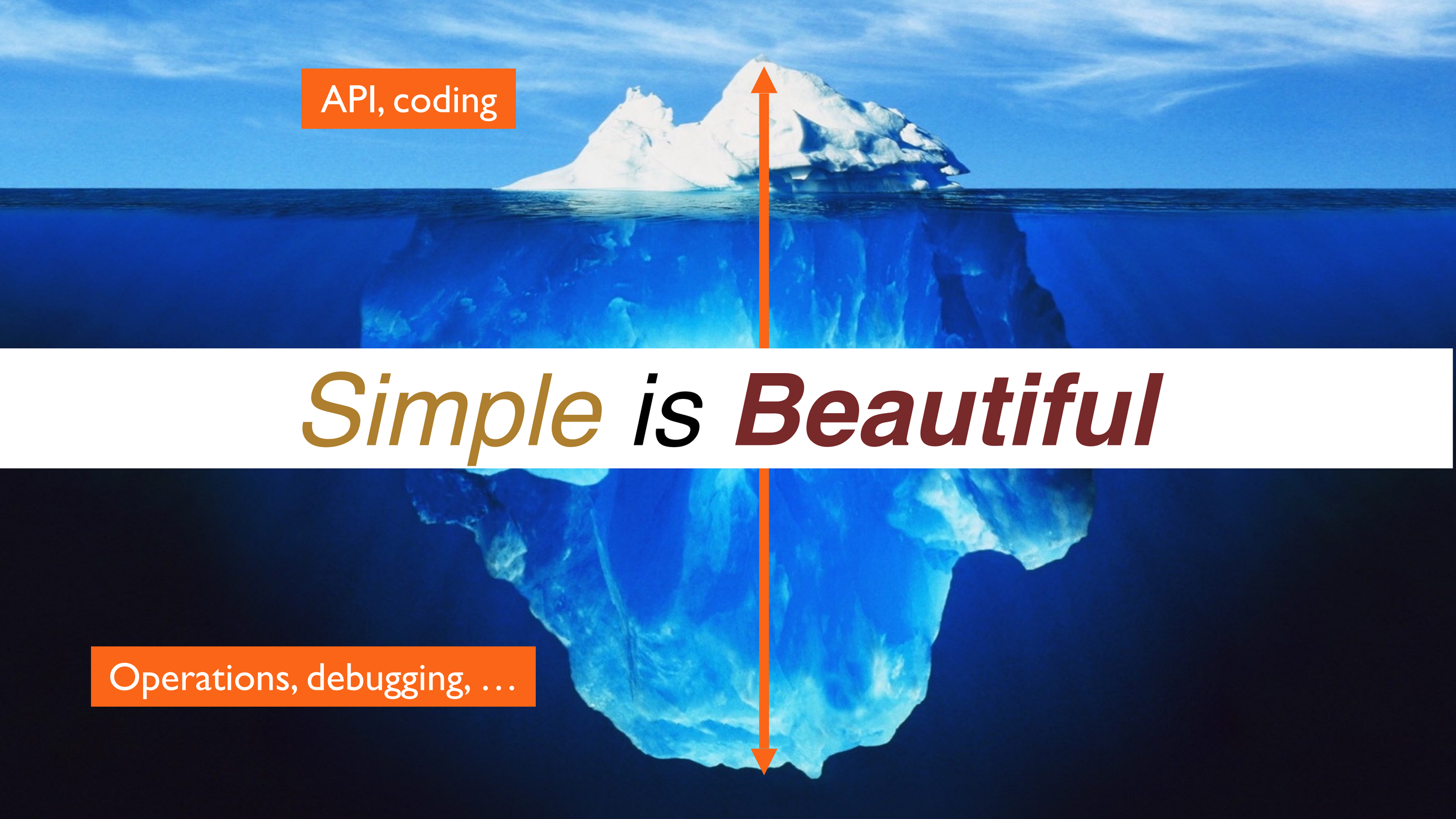
Kafka Streams DSL

```
public static void main(String[] args) {
    // specify the processing topology by first reading in a stream from a topic
    KStream<String, String> words = builder.stream("topic1");

    // count the words in this stream as an aggregated table
    KTable<String, Long> counts = words.countByKey("Counts");

    // write the result table to a new topic
    counts.to("topic2");

    // create a stream processing instance and start running it
    KafkaStreams streams = new KafkaStreams(builder, config);
    streams.start();
}
```

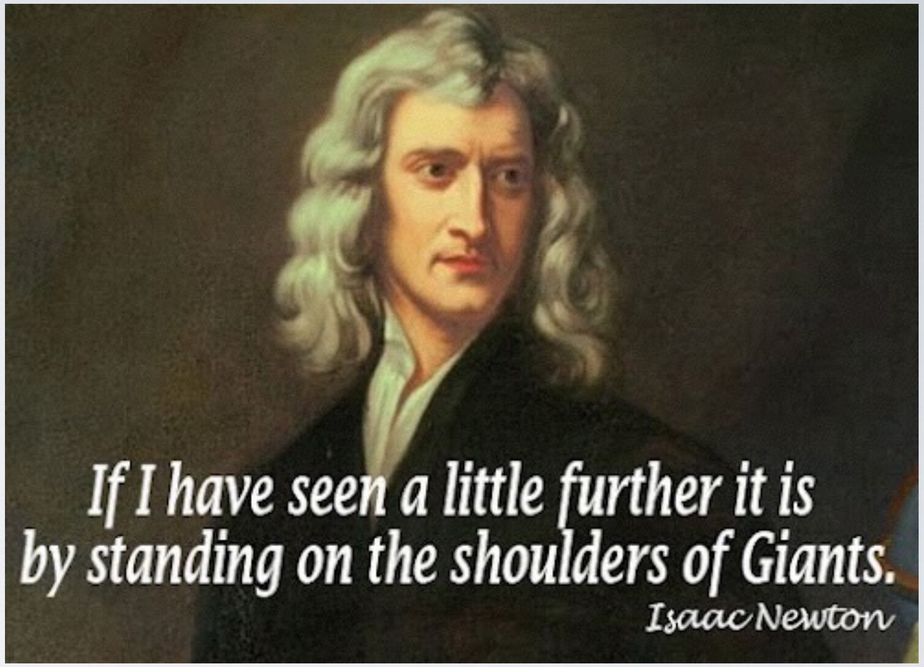
Kafka Streams DSL

```
public static void main(String[] args) {
    // specify the processing topology by first reading in a stream from a topic
    KStream<String, String> words = builder.stream(”topic1”);

    // count the words in this stream as an aggregated table
    KTable<String, Long> counts = words.countByKey(”Counts”);

    // write the result table to a new topic
    counts.to(”topic2”);

    // create a stream processing instance and start running it
    KafkaStreams streams = new KafkaStreams(builder, config);
    streams.start();
}
```


API, coding
“Full stack” evaluation
Operations, debugging, …

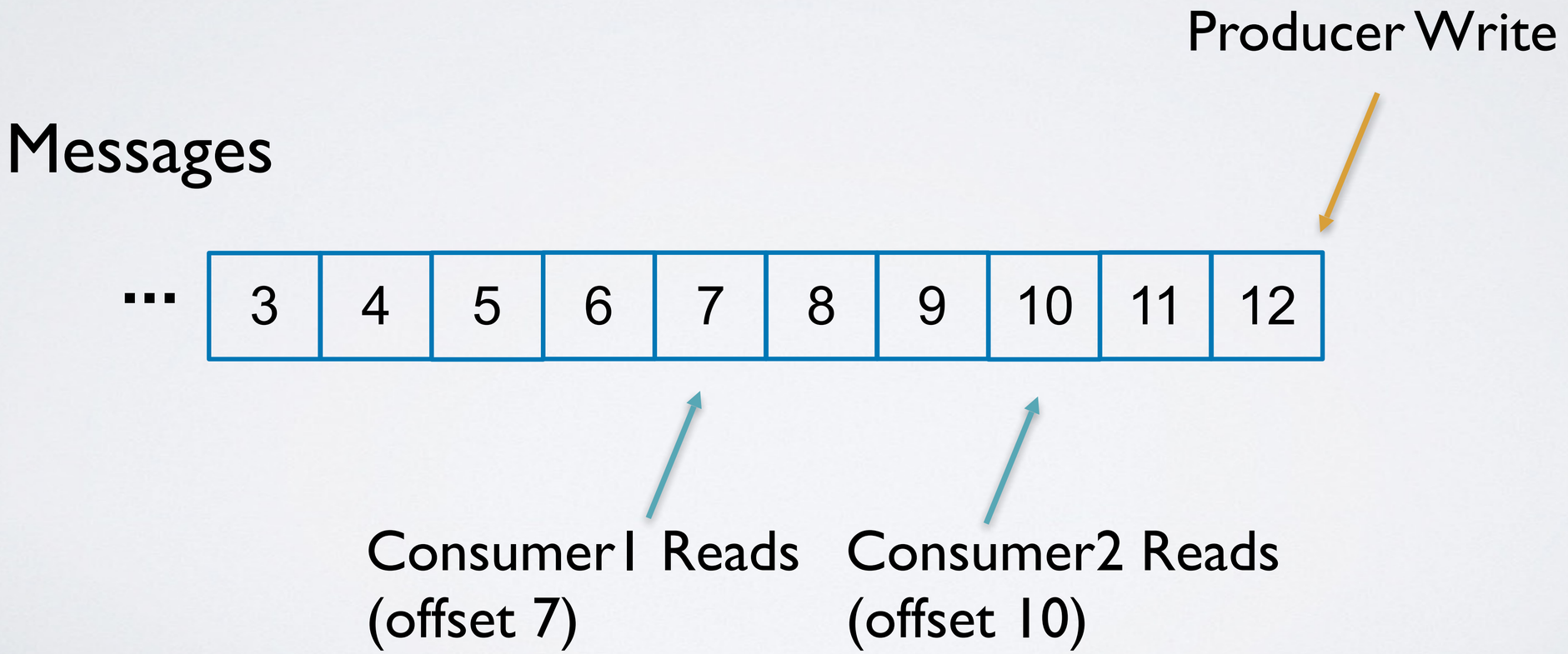

API, coding
Simple is Beautiful
Operations, debugging, …

Key Idea:

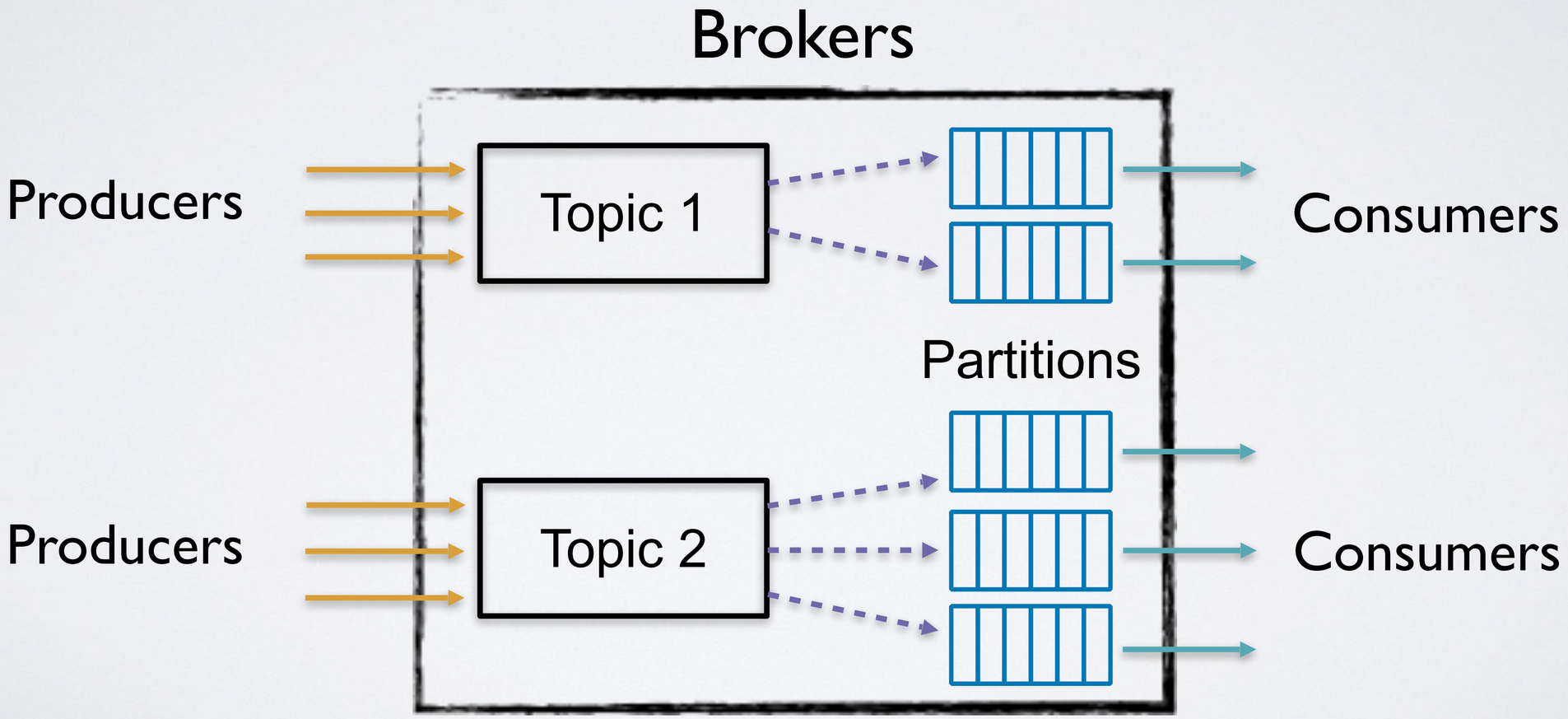


Outsource hard problems to Kafka!

Kafka Concepts: the *Log*

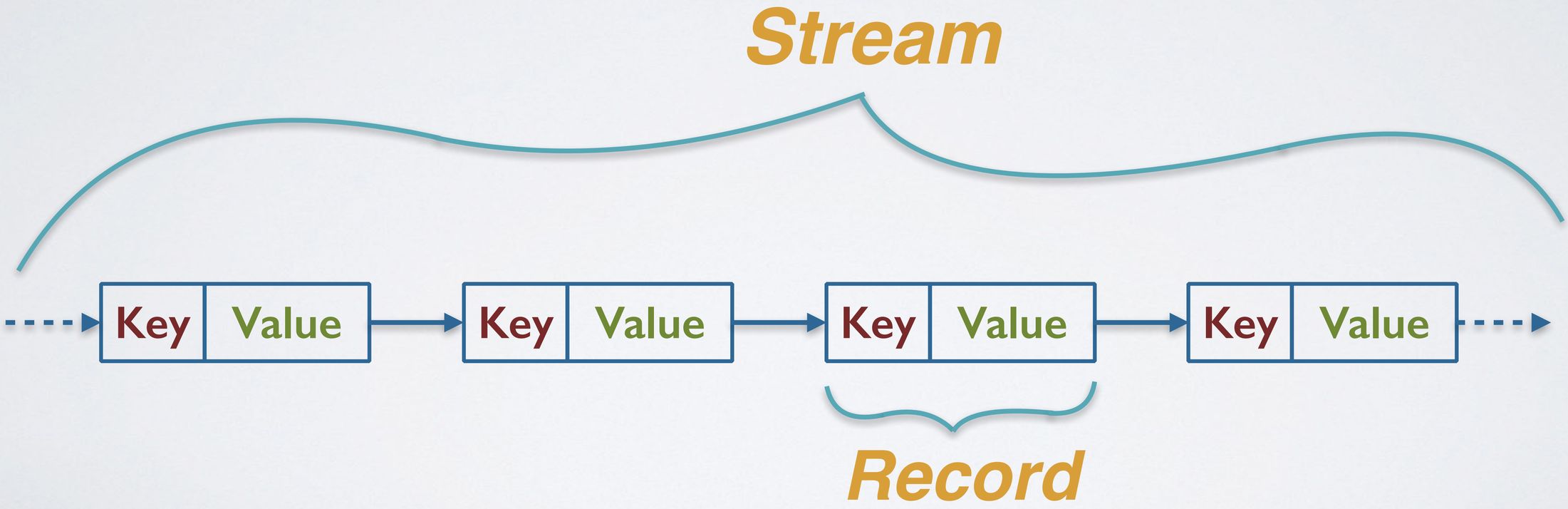


Kafka Concepts: the *Log*



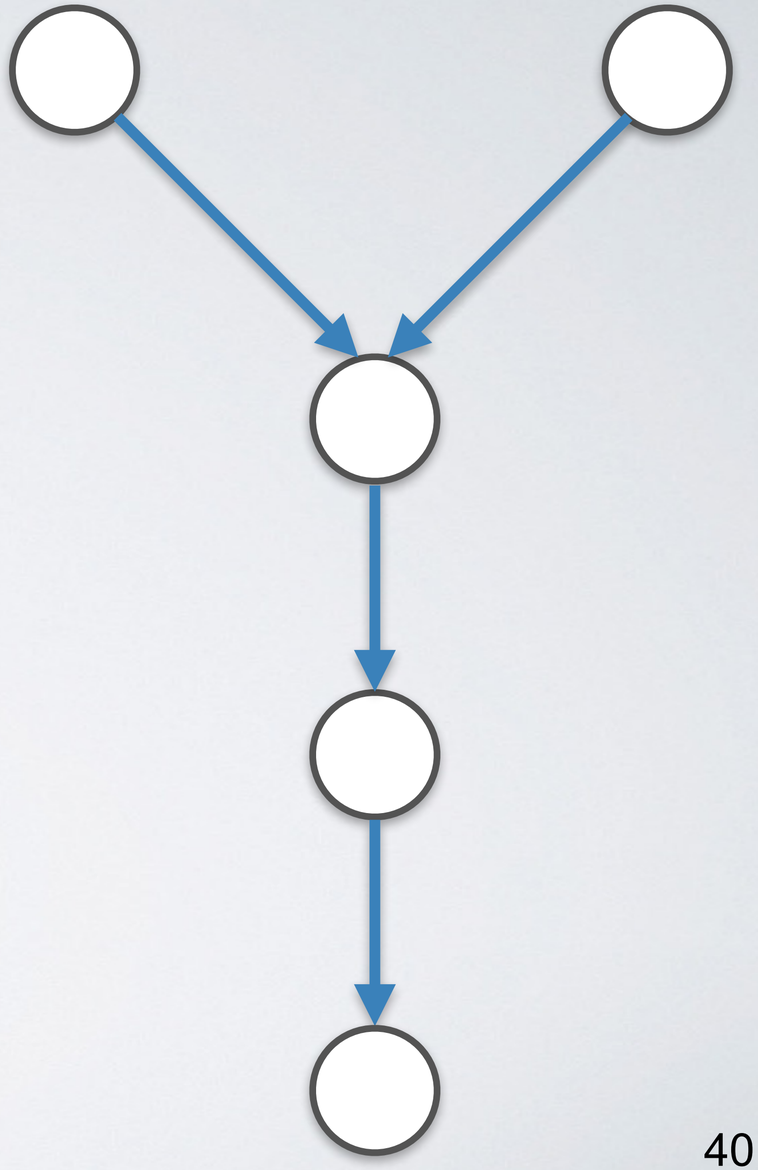
Kafka Streams: **Key Concepts**

Stream and Records

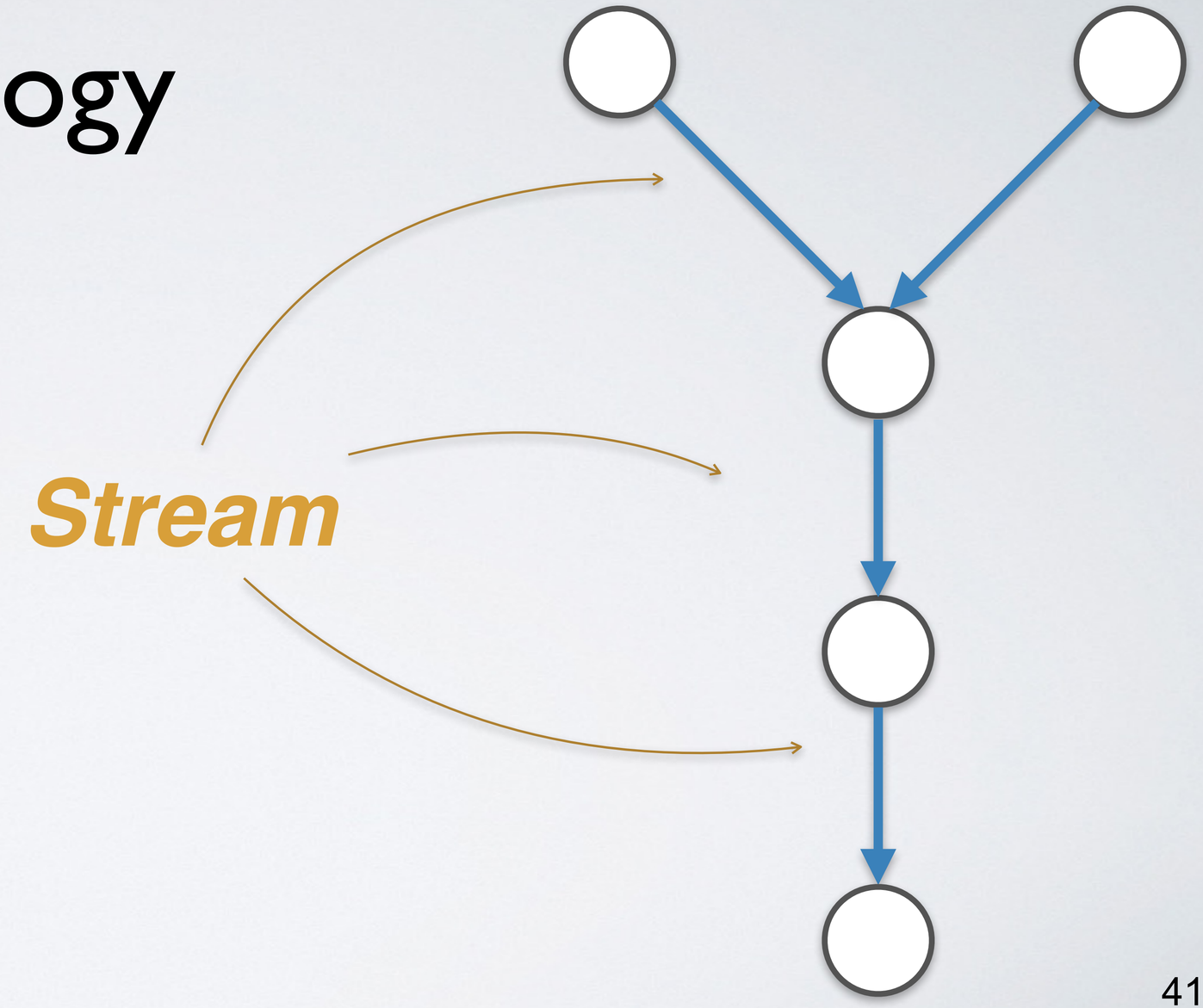


Processor Topology

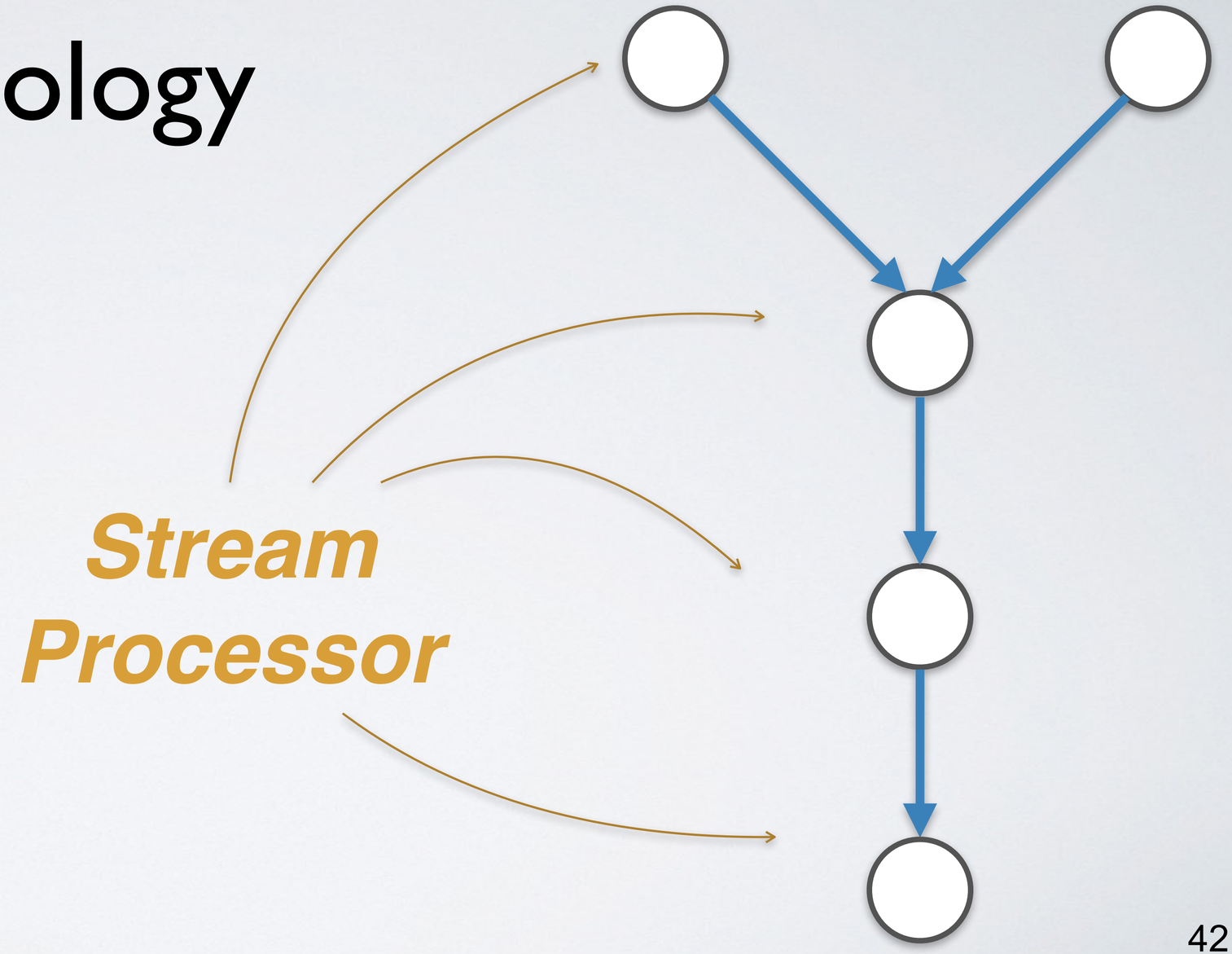
```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic3”);
```



Processor Topology

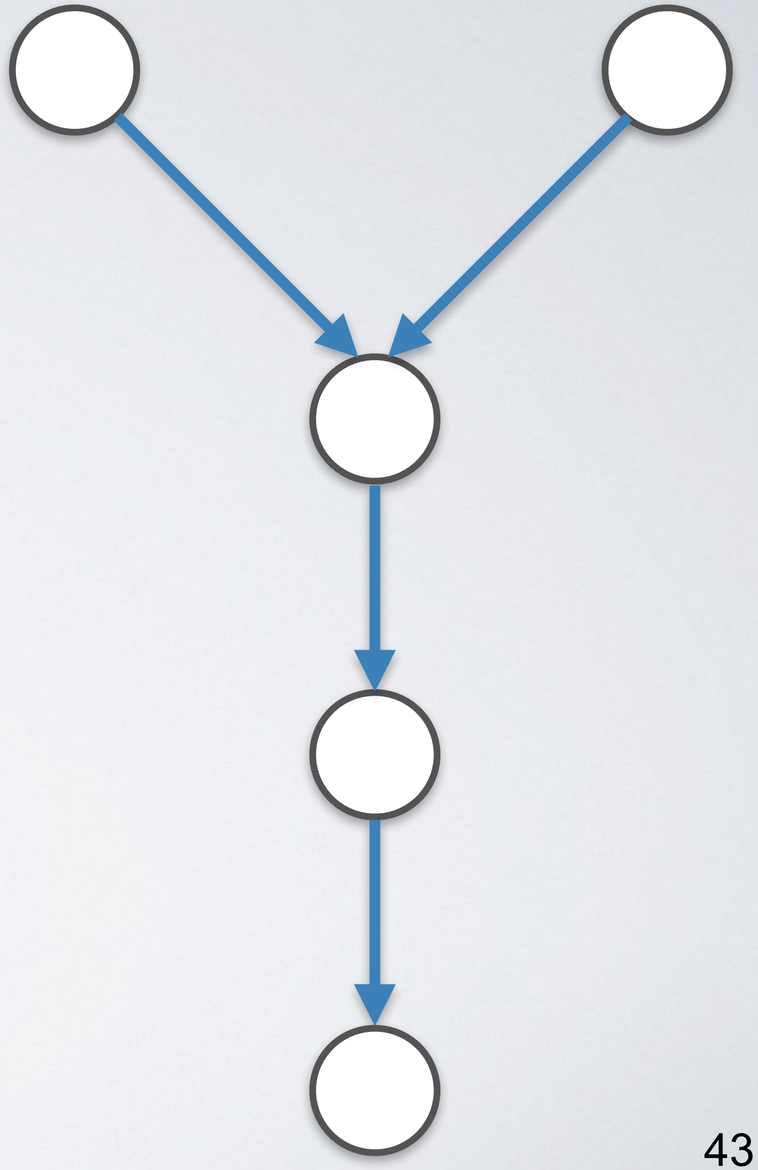


Processor Topology



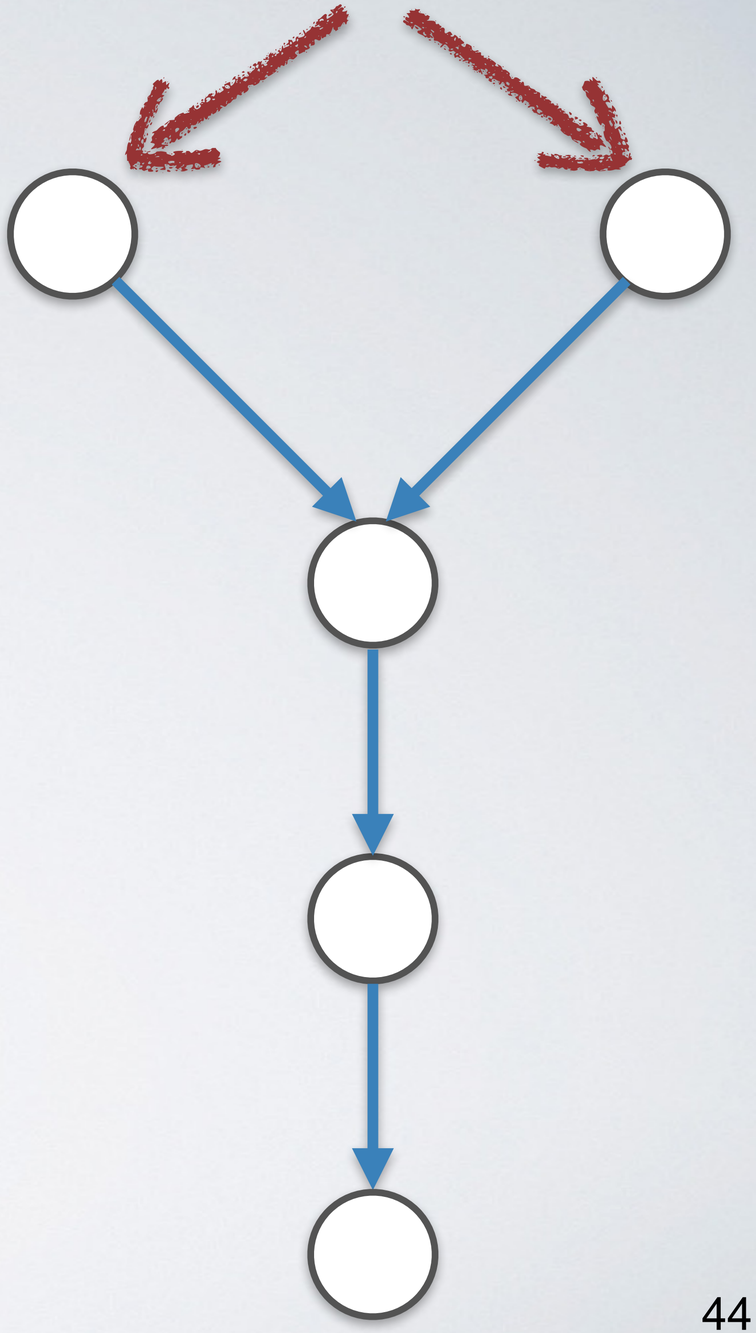
Processor Topology

```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic3”);
```



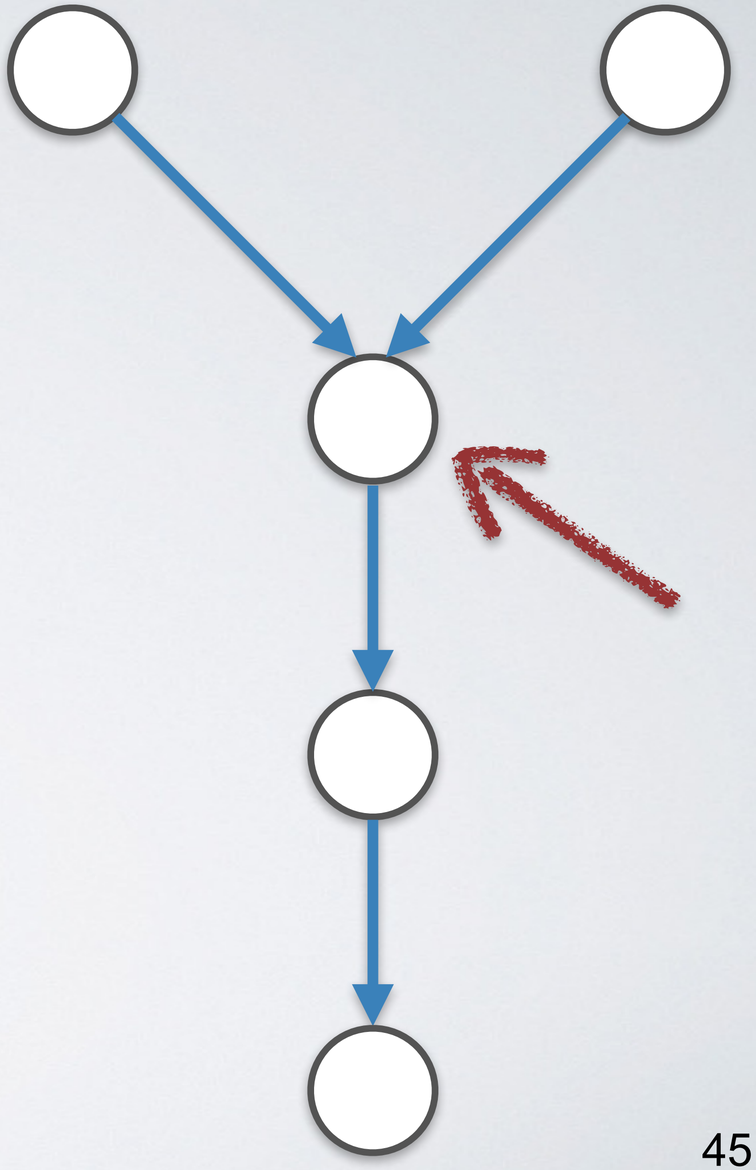
Processor Topology

```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic3”);
```



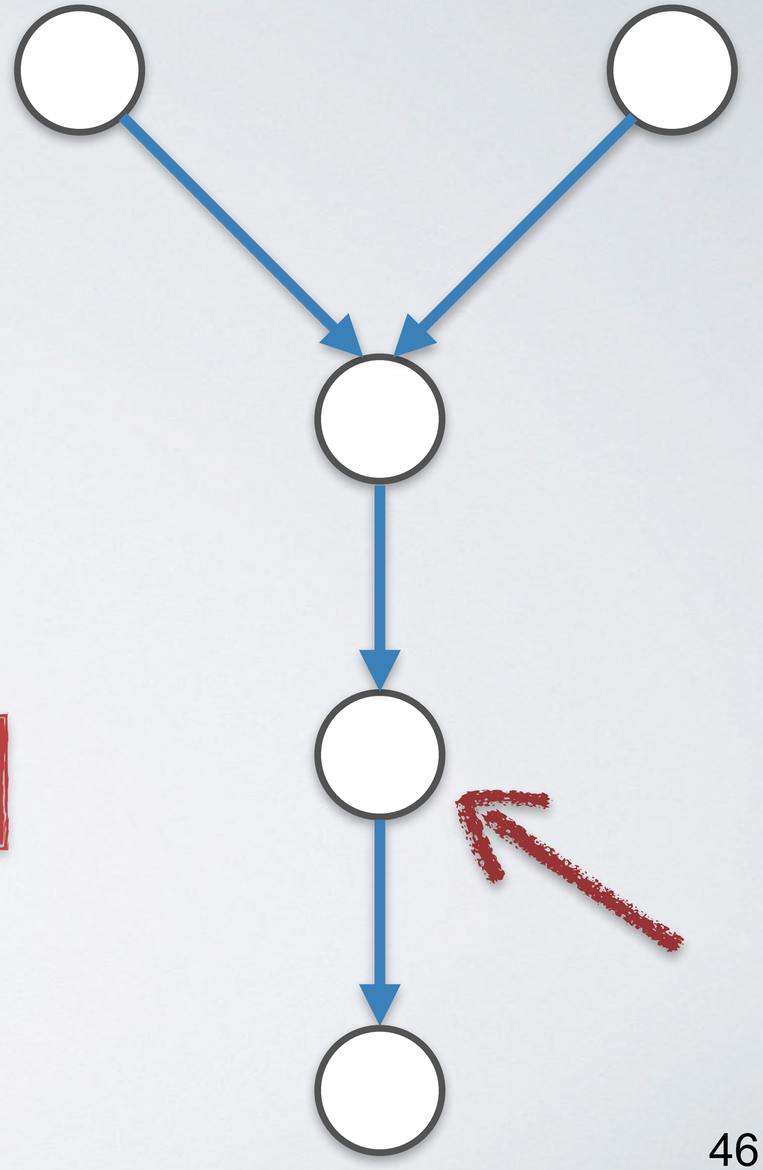
Processor Topology

```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic3”);
```



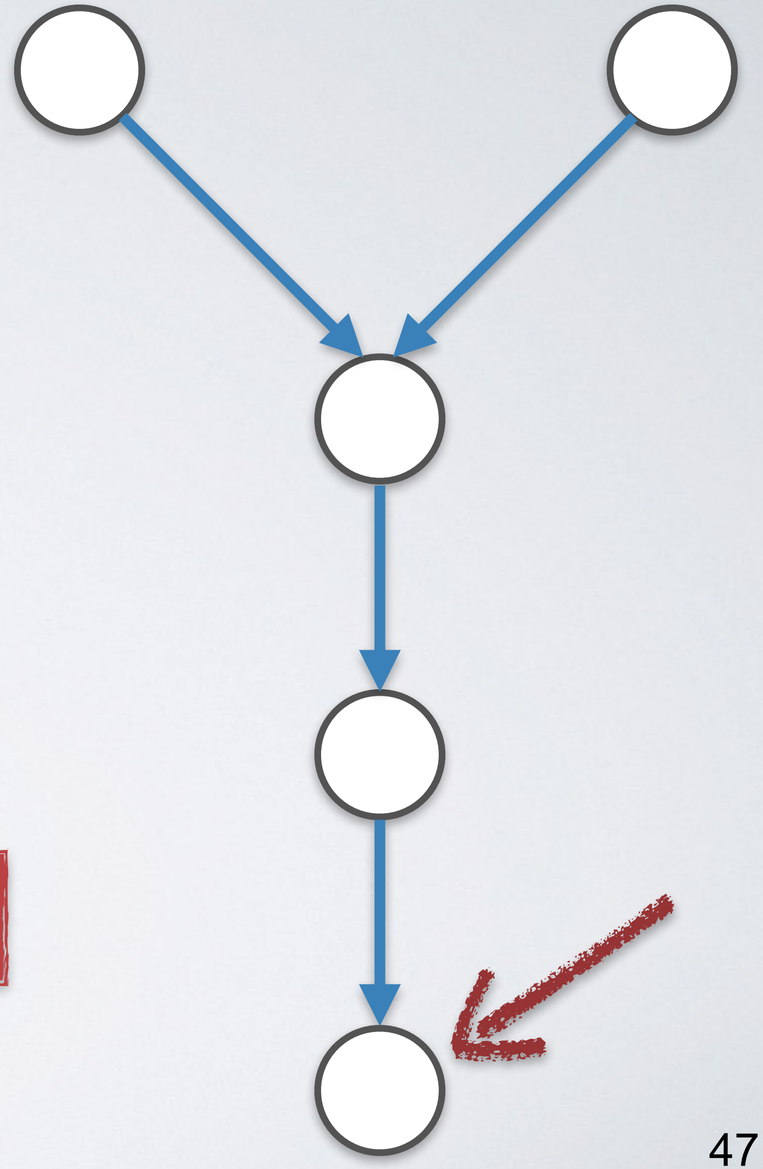
Processor Topology

```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic3”);
```

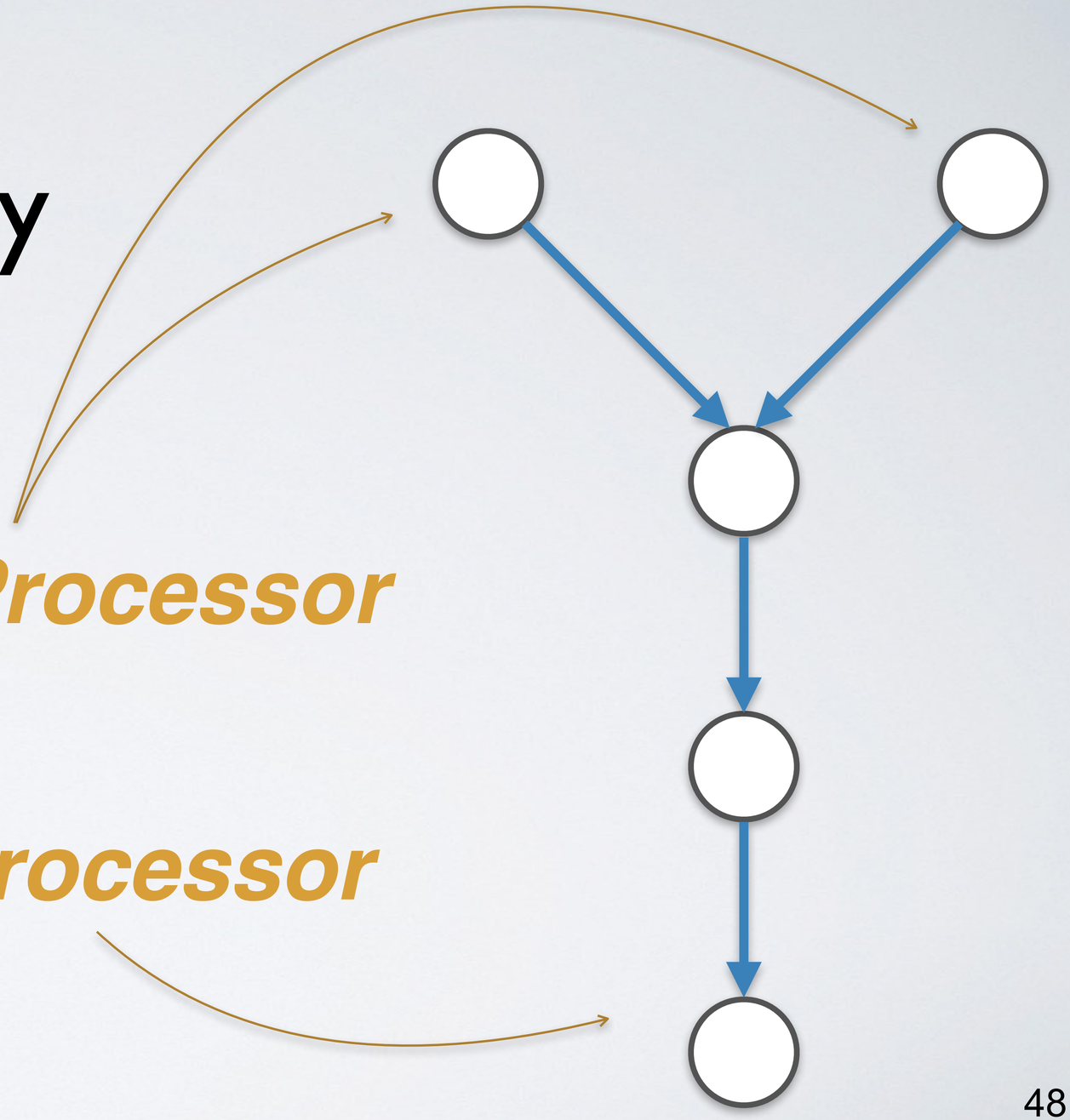


Processor Topology

```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic3”);
```



Processor Topology



Source Processor

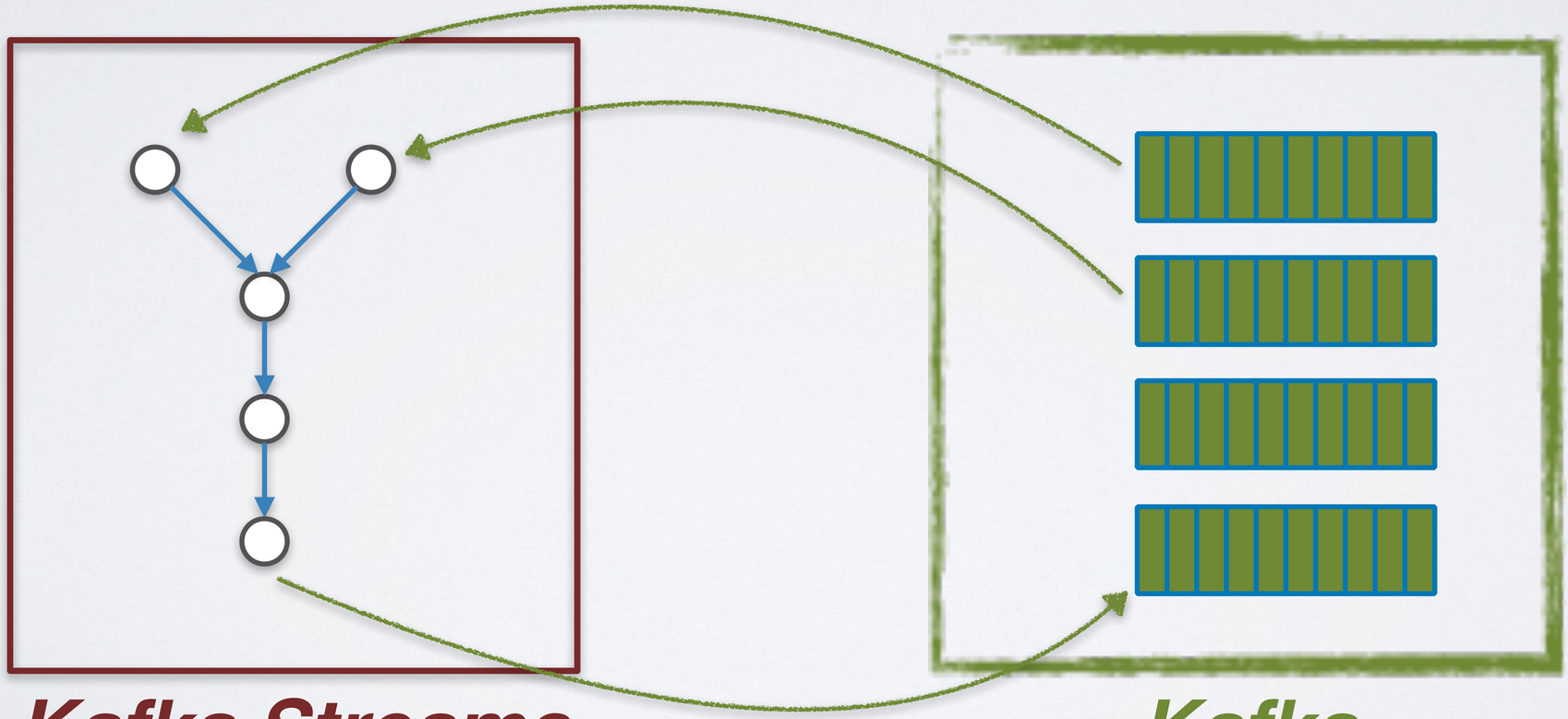
KStream<..> stream1 = builder.stream(

KStream<..> stream2 = builder.stream(

Sink Processor

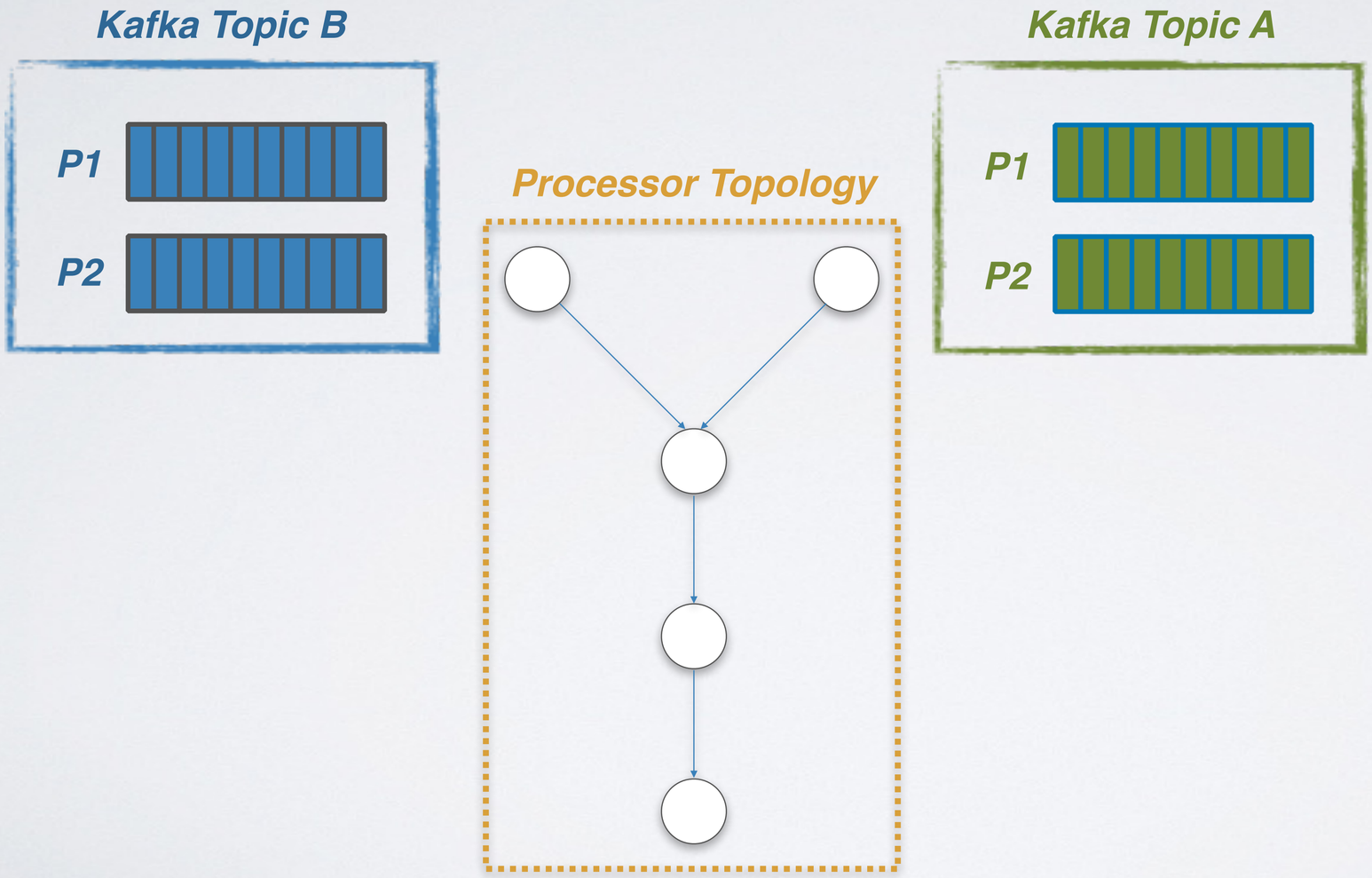
aggregated.to(

Processor Topology

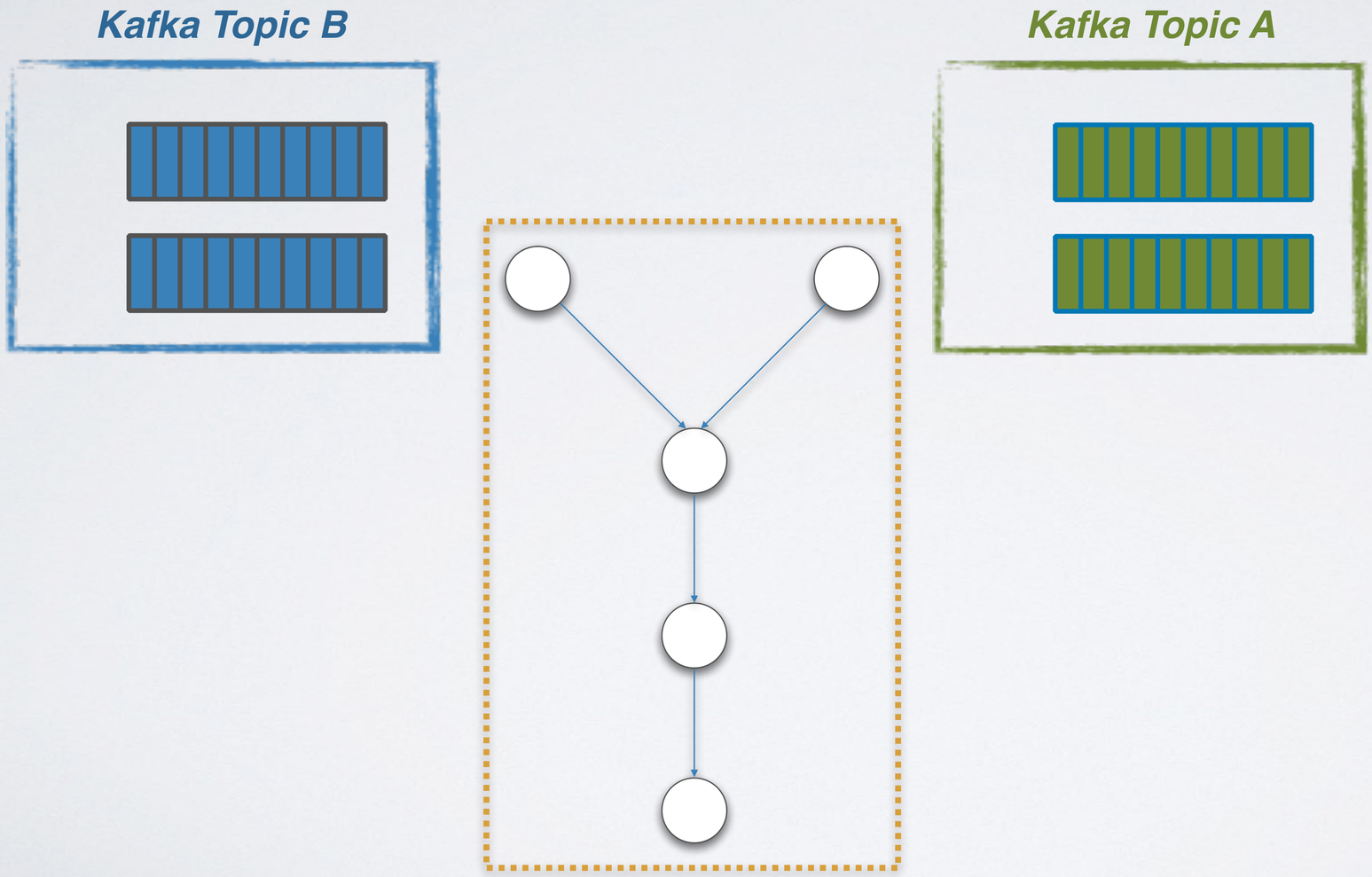


Stream Partitions and Tasks

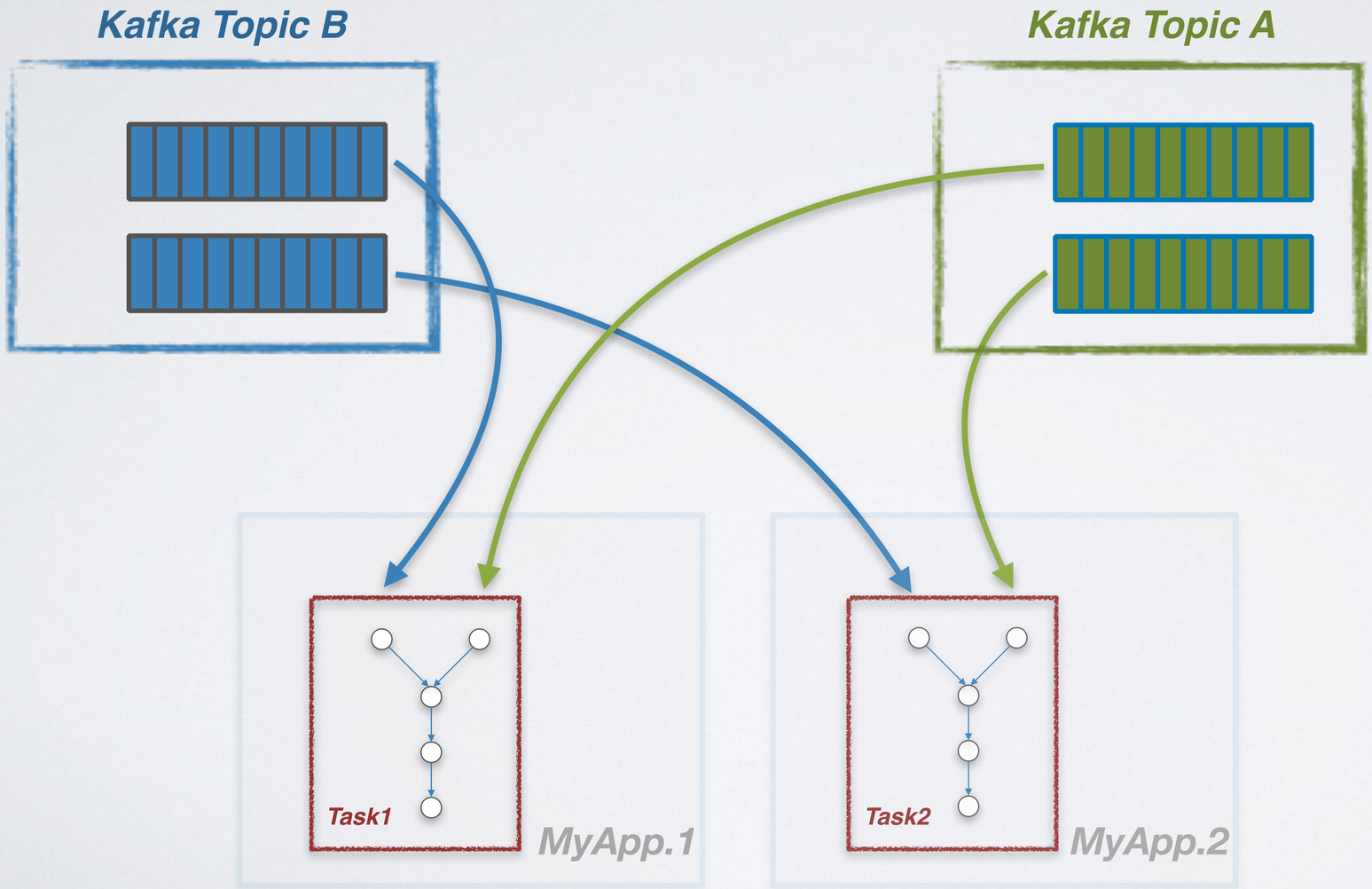
Stream Partitions and Tasks



Stream Partitions and Tasks



Stream Threads



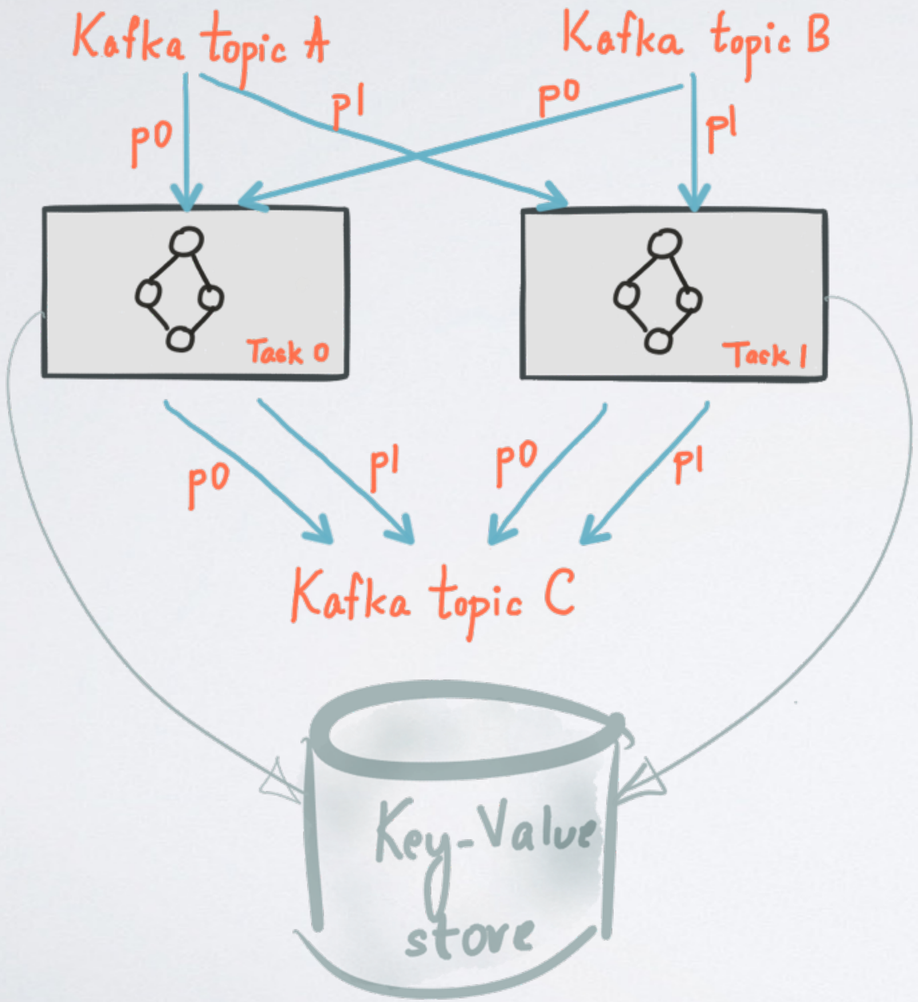
States in Stream Processing

- *filter*
- *map*
- *join*
- *aggregate*

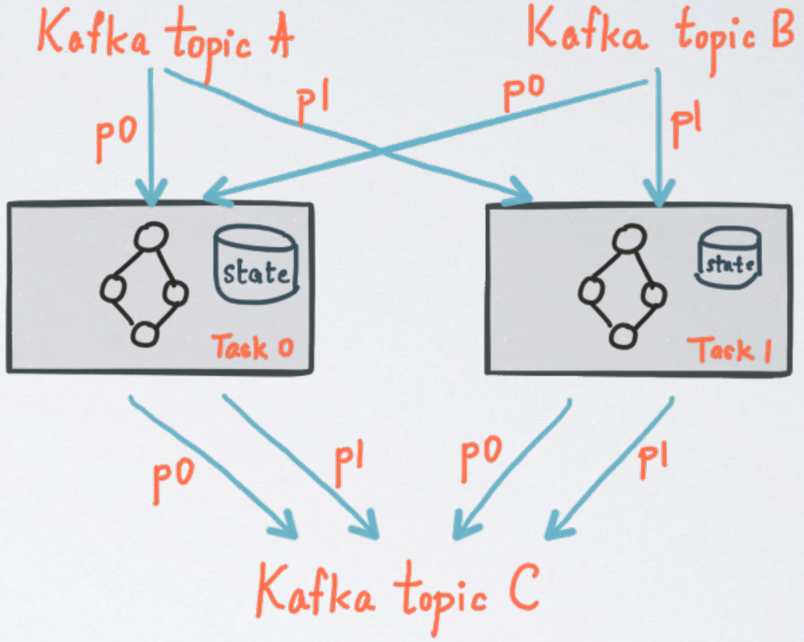
Stateless

Stateful

Remote State



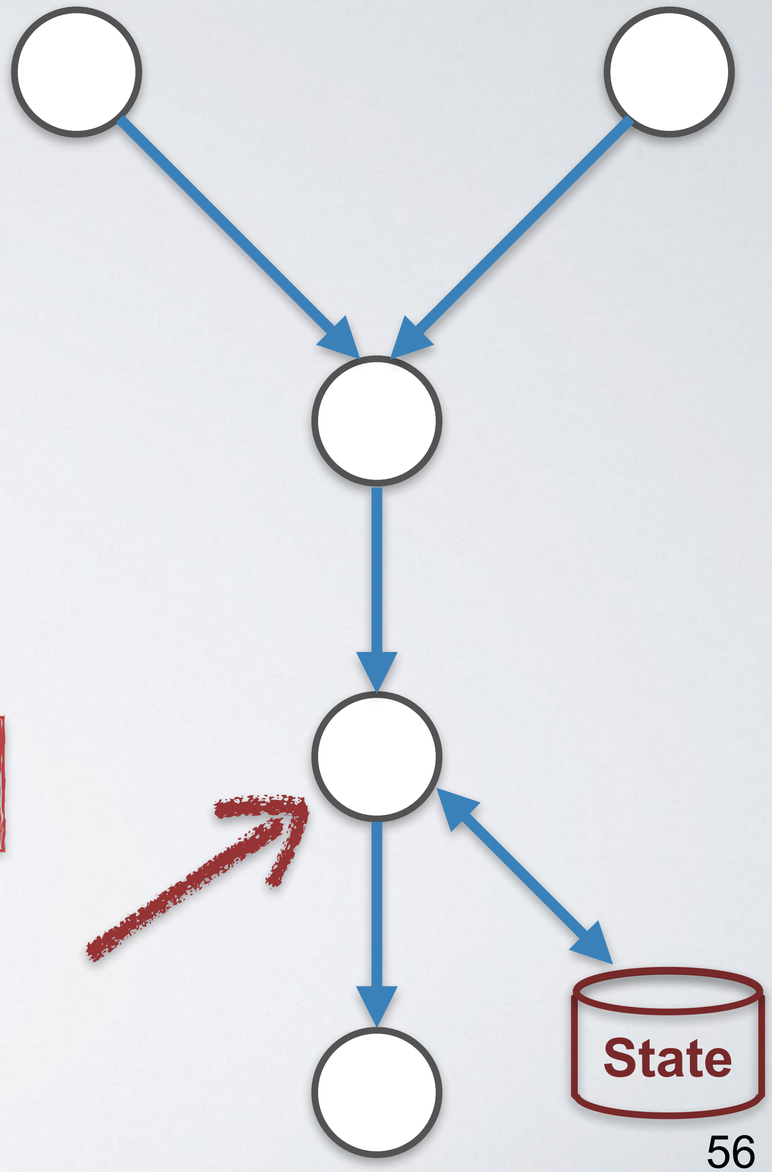
Local State



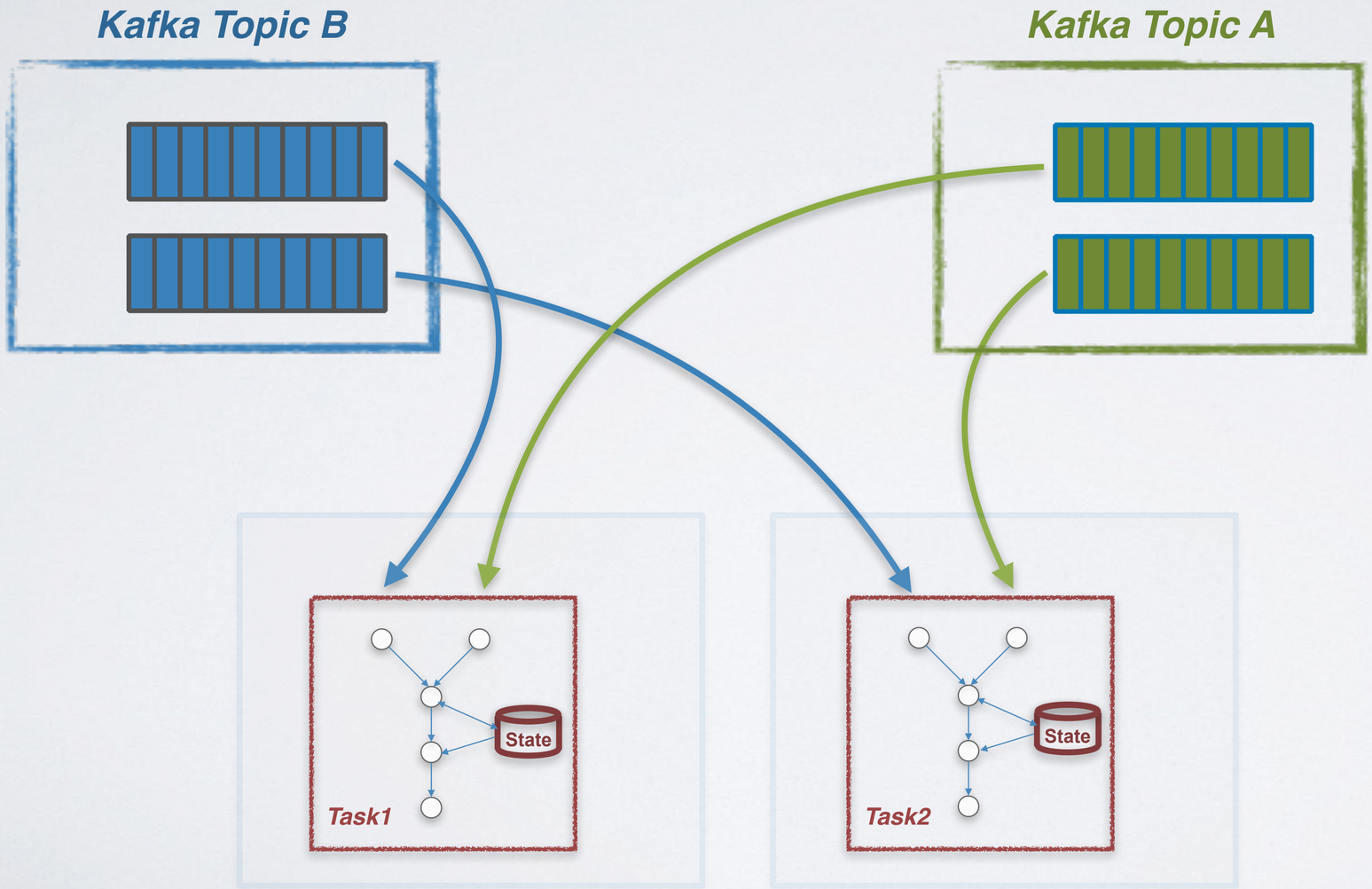
- faster
- better isolation
- flexible

States in Stream Processing

```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic2”);
```

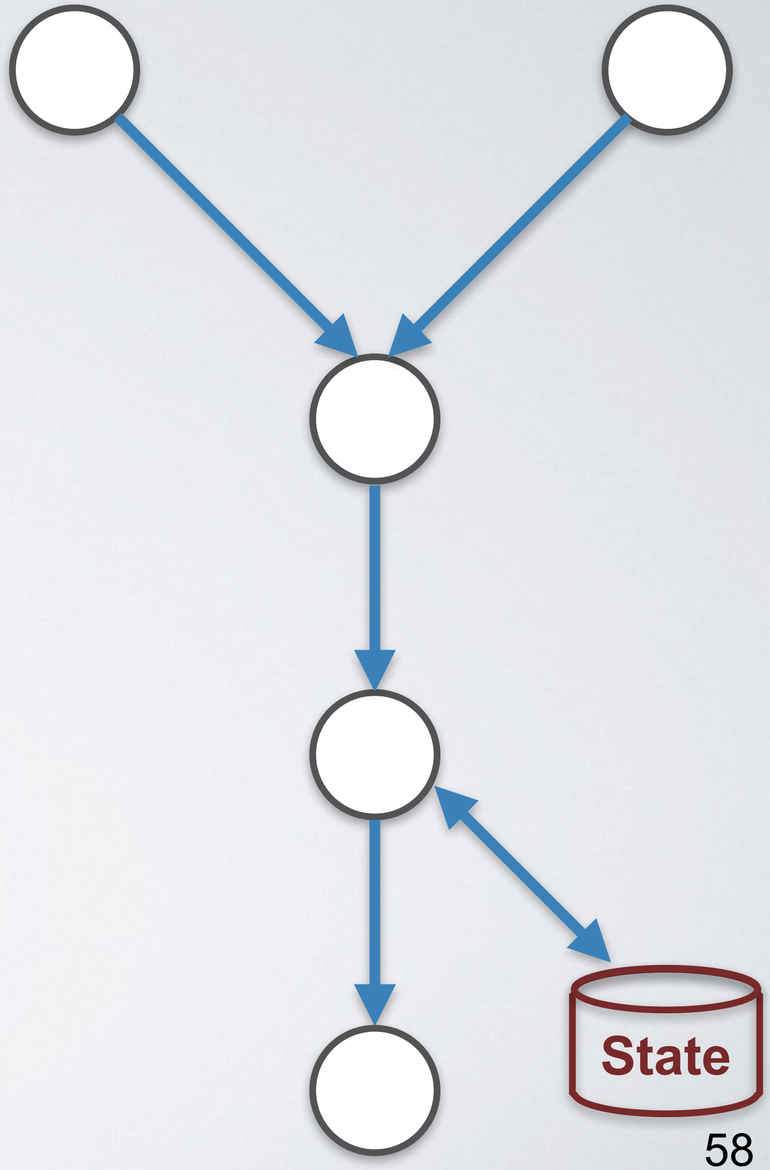


States in Stream Processing



Stream v.s. Table?

```
KStream<..> stream1 = builder.stream(”topic1”);

KStream<..> stream2 = builder.stream(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);

aggregated.to(”topic2”);
```



Tables ≈ ***Streams***

TABLES ≈ STREAMS

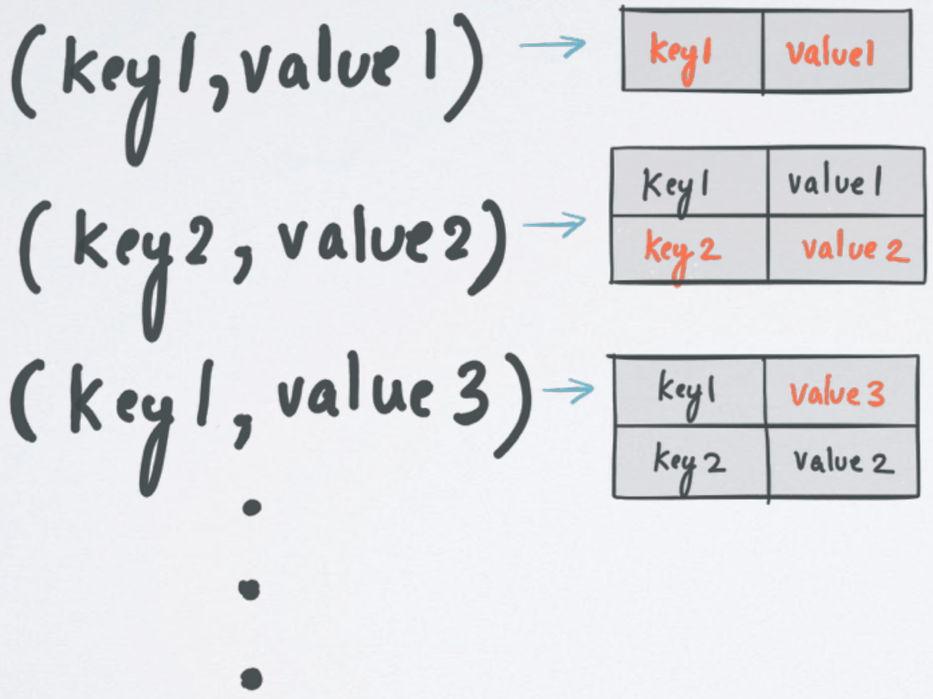
(key1, value1)

(key2, value2)

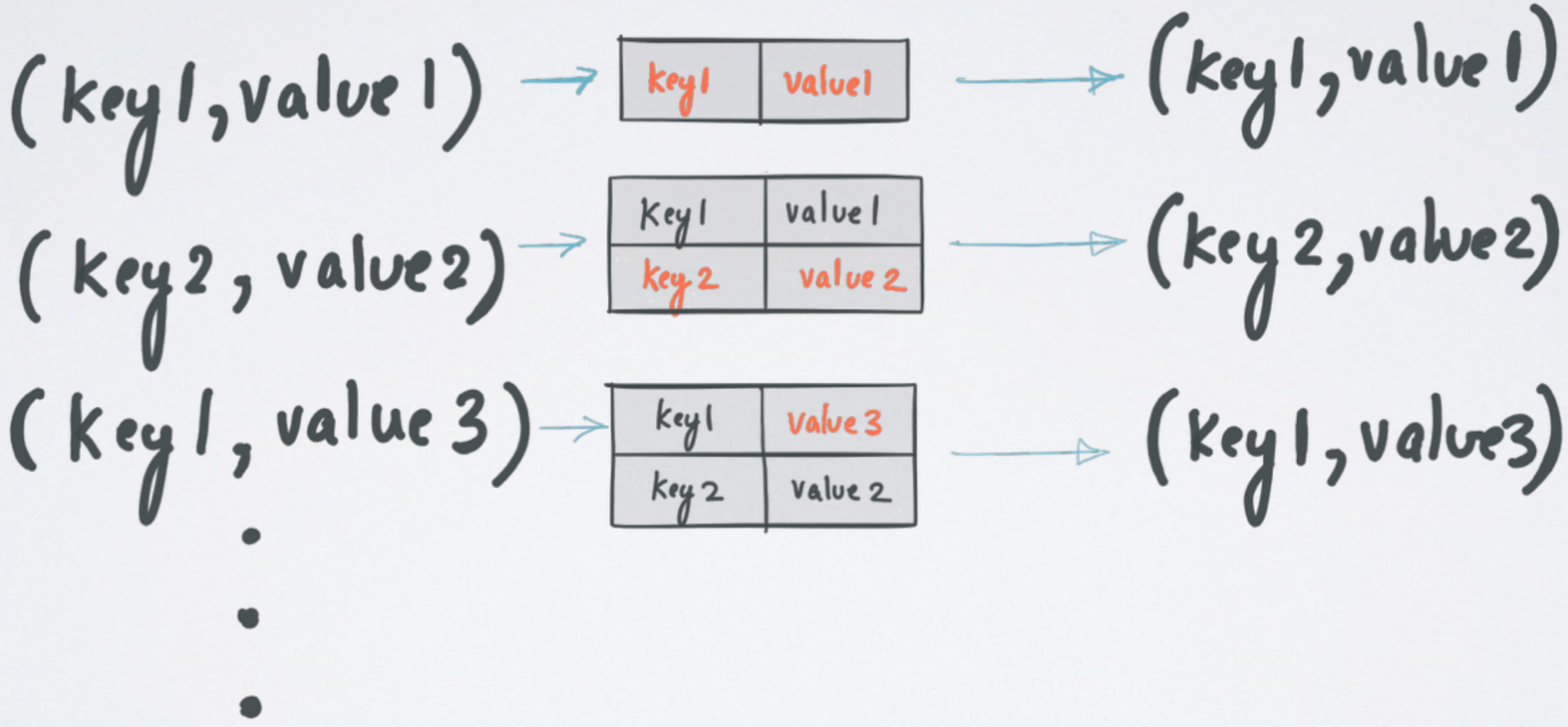
(key1, value3)

.
.
.

TABLES ≈ STREAMS



TABLES ≈ STREAMS



The Stream-Table Duality

- *A **stream** is a changelog of a **table***
- *A **table** is a materialized view at time of a **stream***
- *Example: change data capture (CDC) of databases*

KStream *= interprets data as record stream*

~ think: "append-only"

KTable *= data as changelog stream*

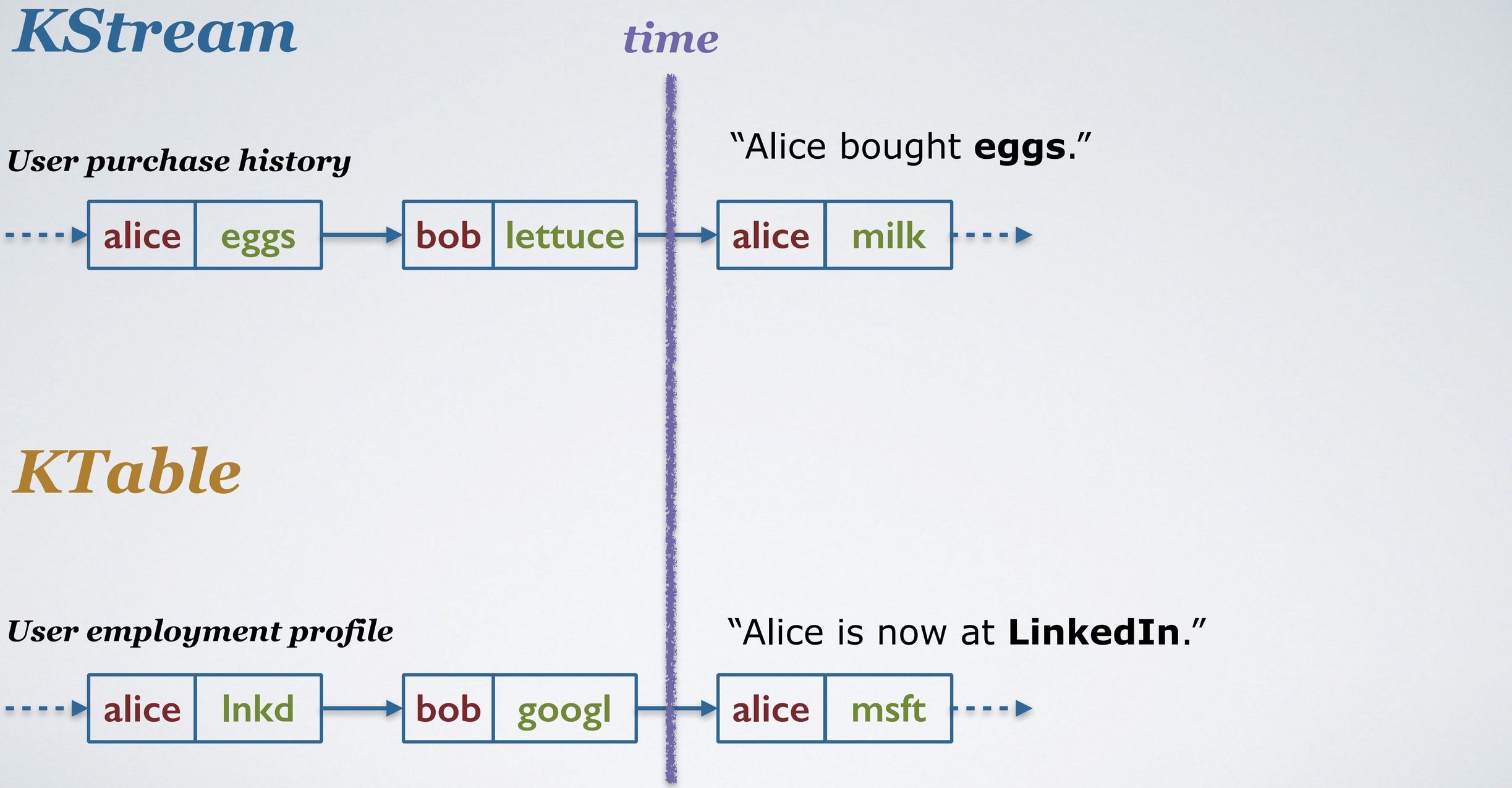
~ continuously updated materialized view

KStream

User purchase history

KTable

User employment profile


KStream
time
User purchase history
"Alice bought eggs."
alice
eggs
bob
lettuce
alice
milk
KTable
User employment profile
"Alice is now at LinkedIn."
alice
lnkd
bob
googl
alice
msft

KStream

time

User purchase history

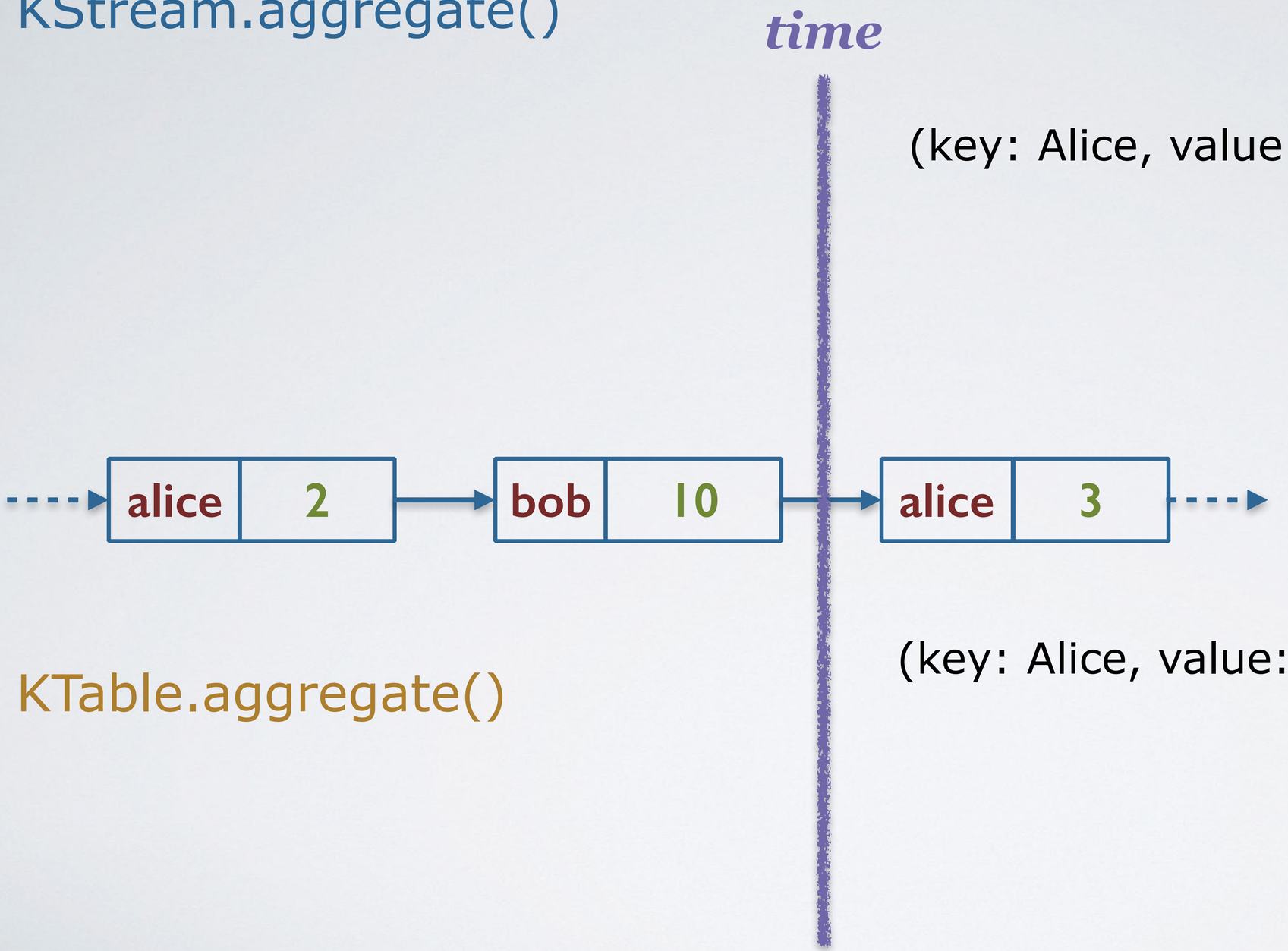
"Alice bought **eggs** and **milk**."

KTable

User employment profile

"Alice is now at ~~LinkedIn~~ **Microsoft**."

KStream.aggregate()

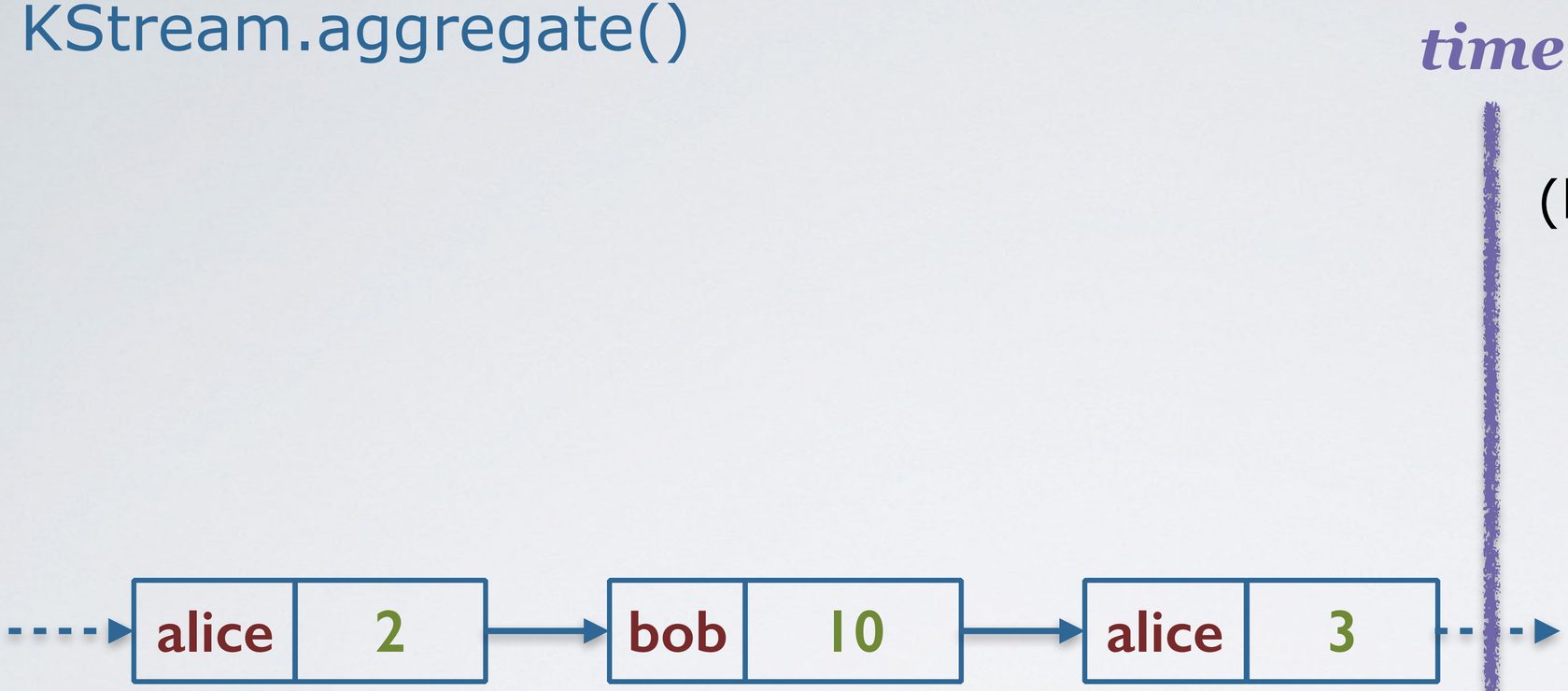


(key: Alice, value: 2)

KTable.aggregate()

(key: Alice, value: 2)

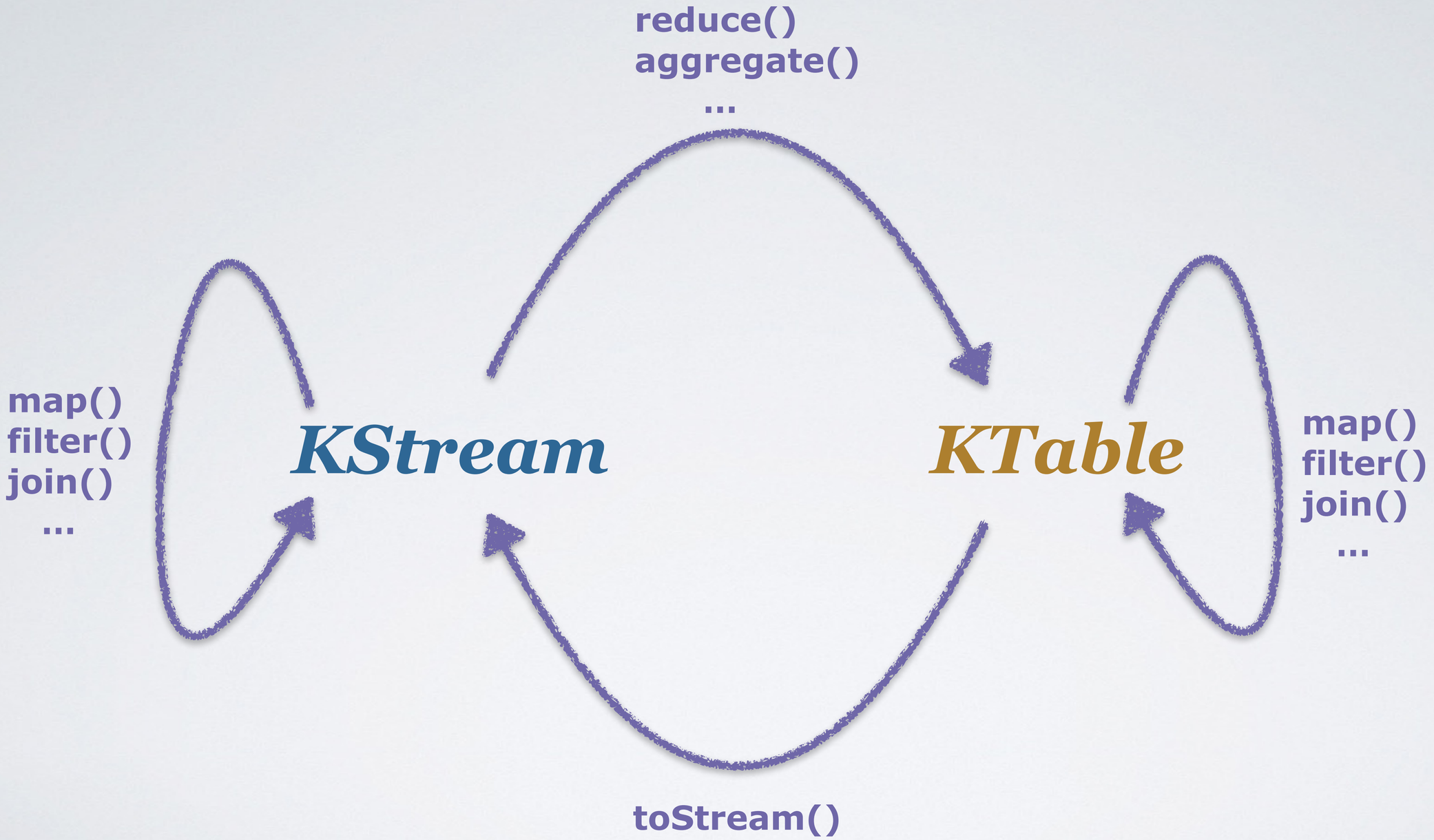
KStream.aggregate()



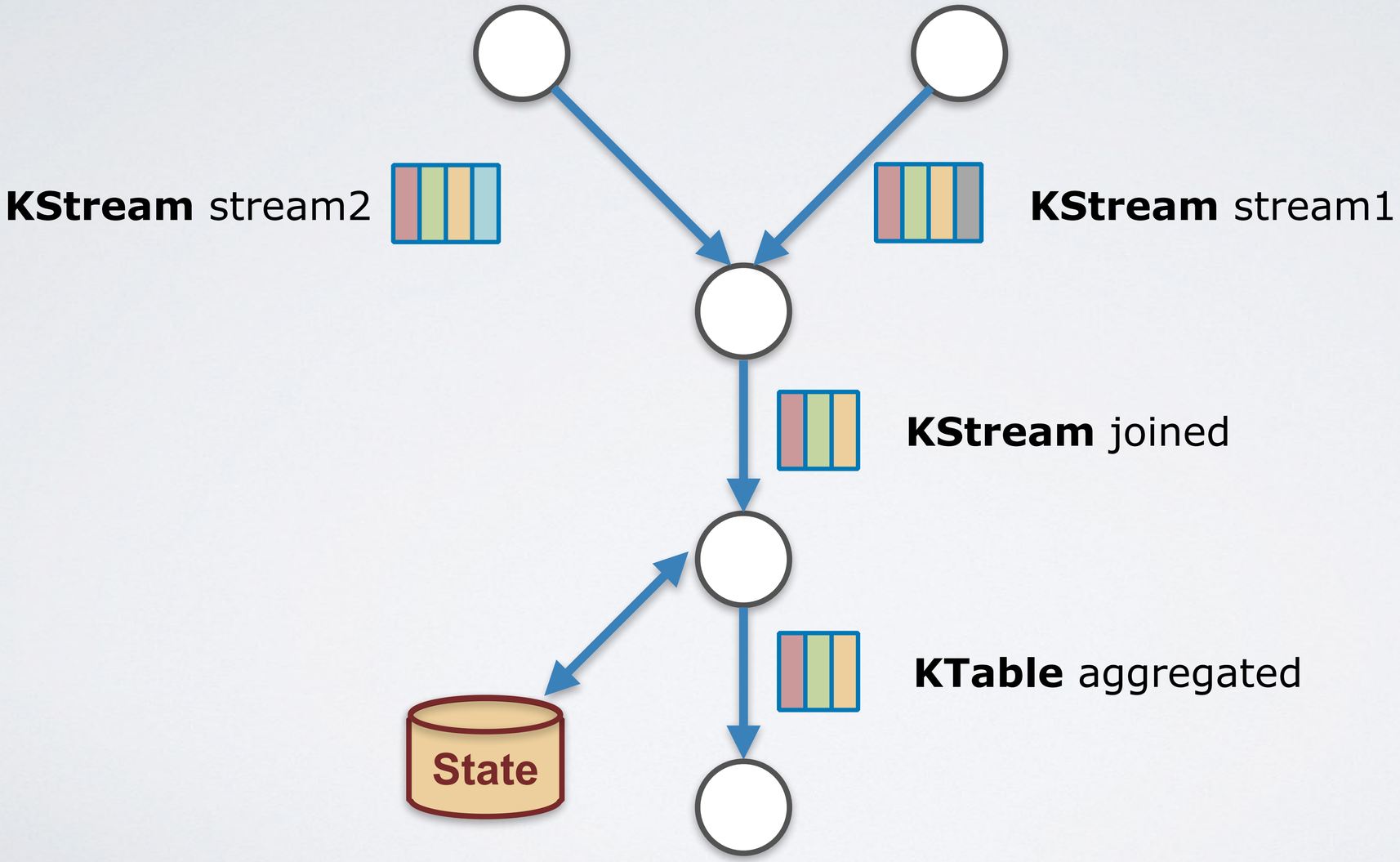
(key: Alice, value: **2+3**)

KTable.aggregate()

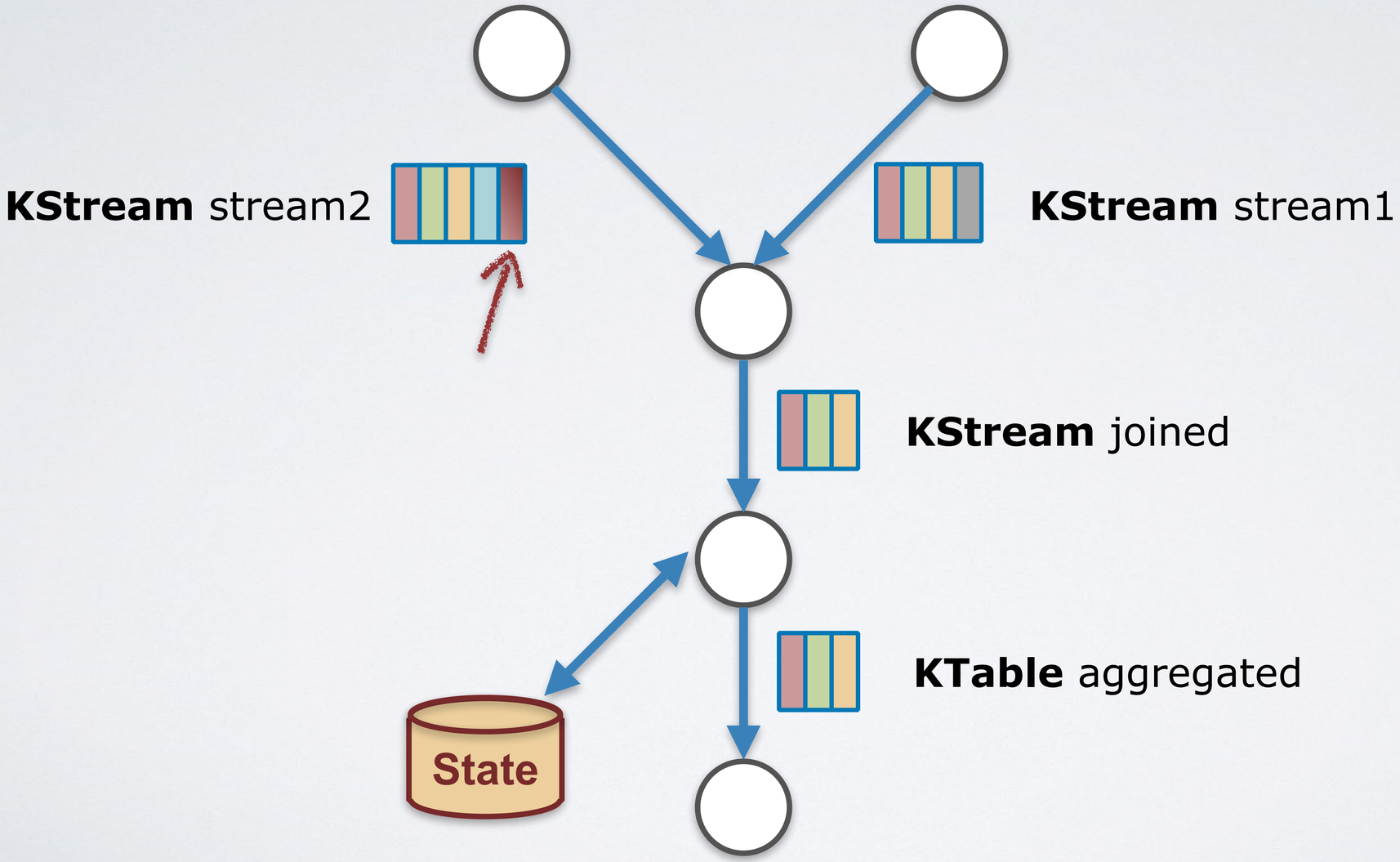
(key: Alice, value: ~~2~~ **3**)


reduce()
aggregate()
...
map()
filter()
join()
...
KStream
KTable
map()
filter()
join()
...
toStream()

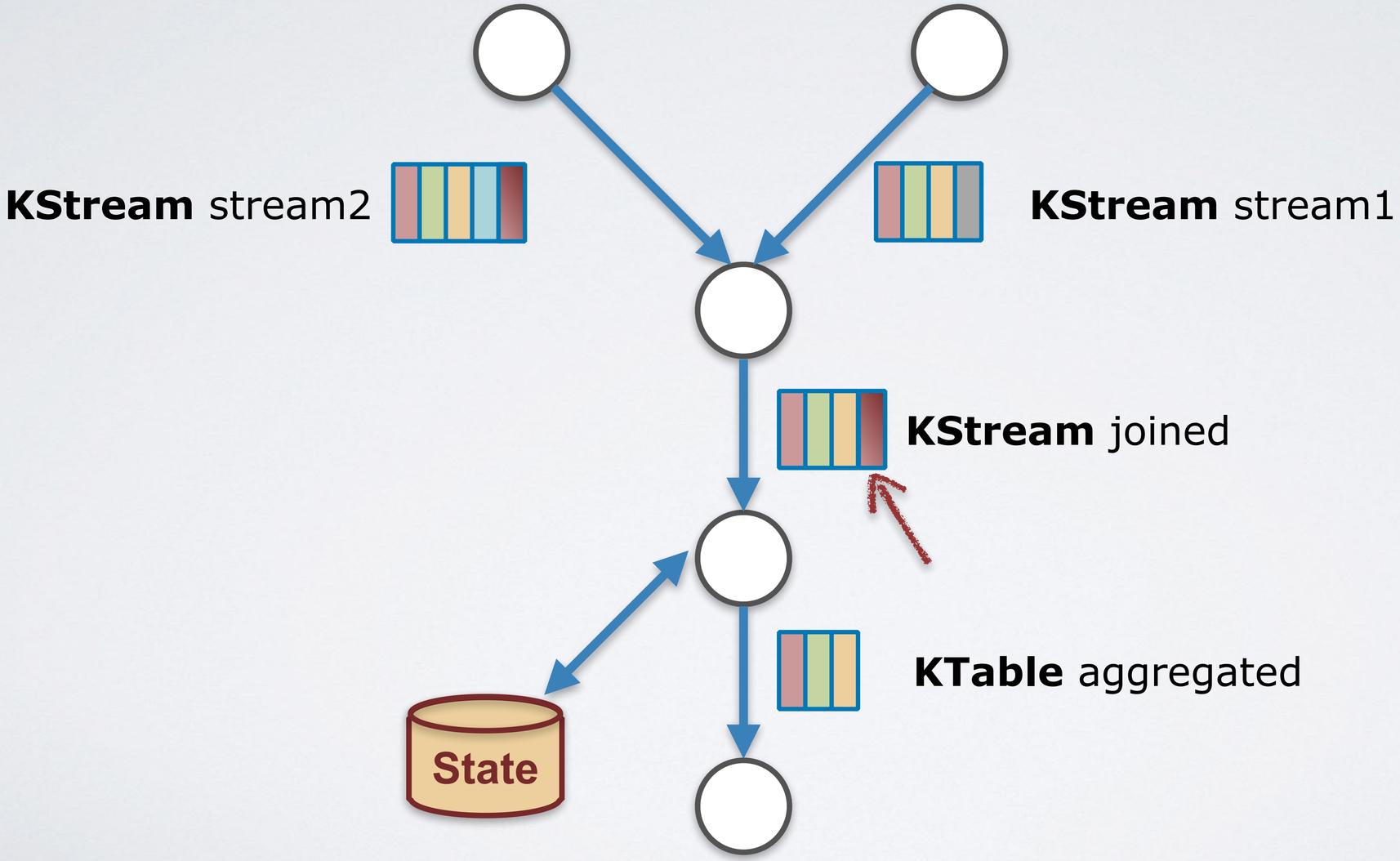
Updates Propagation in KTable



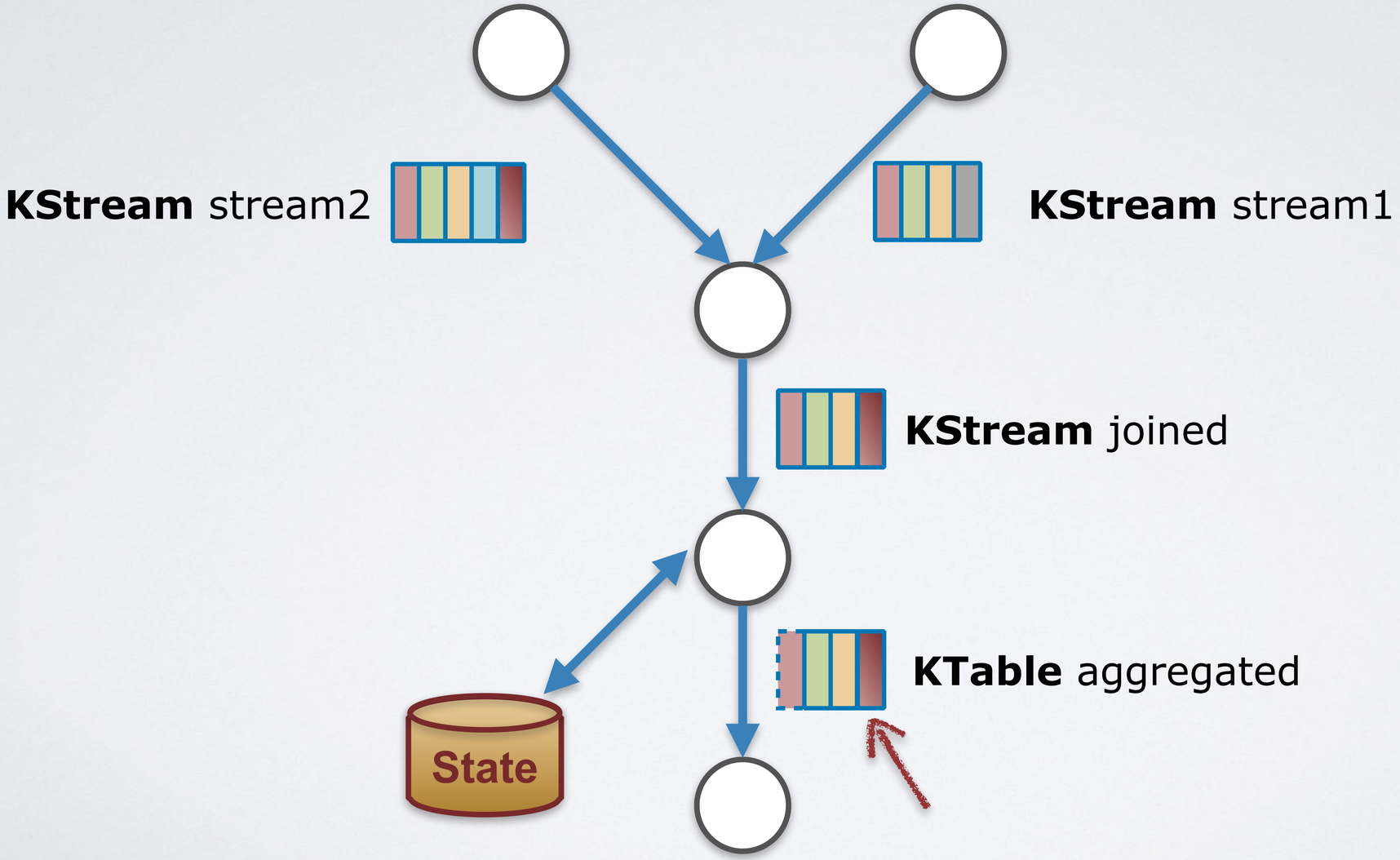
Updates Propagation in KTable



Updates Propagation in KTable

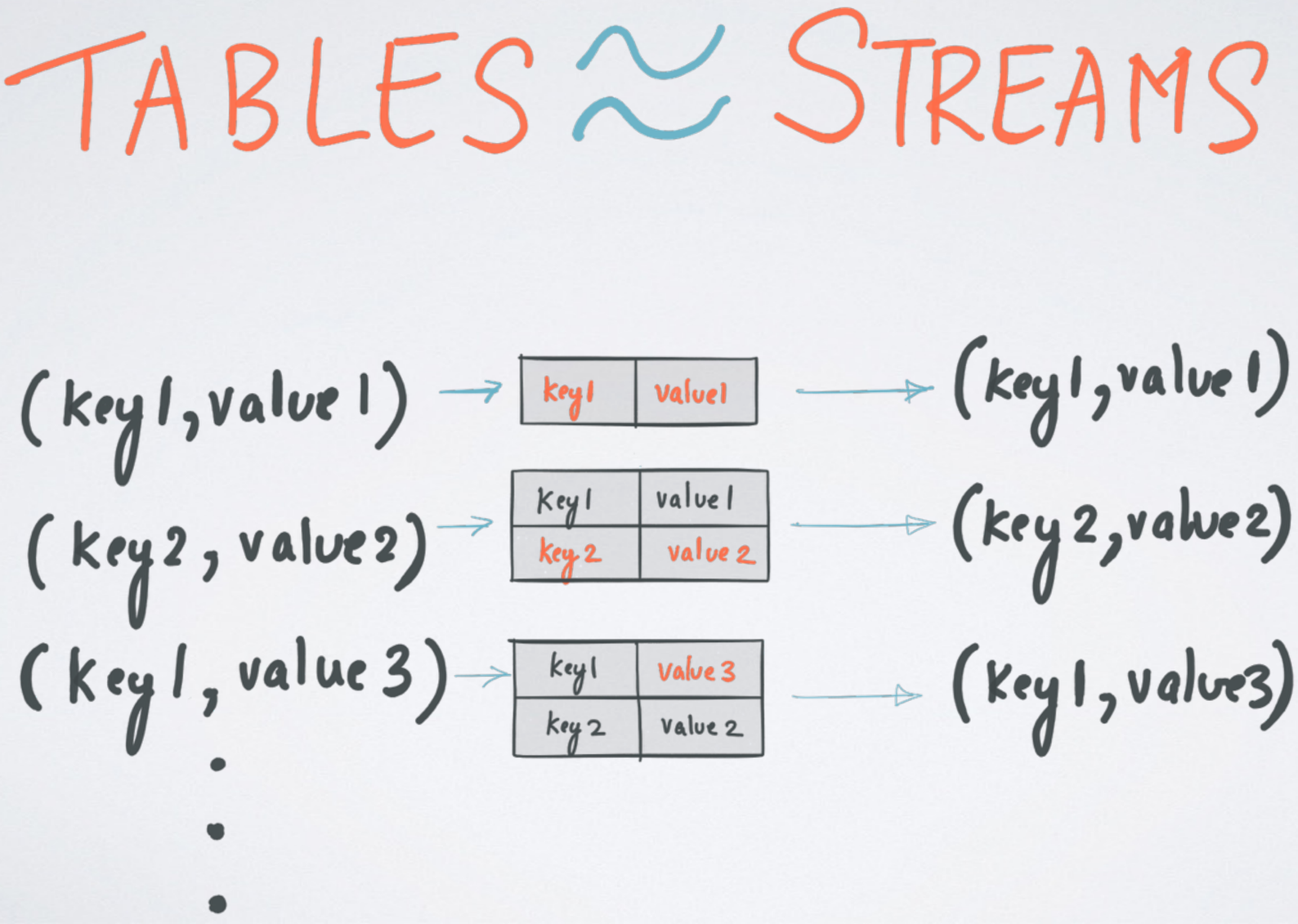


Updates Propagation in KTable

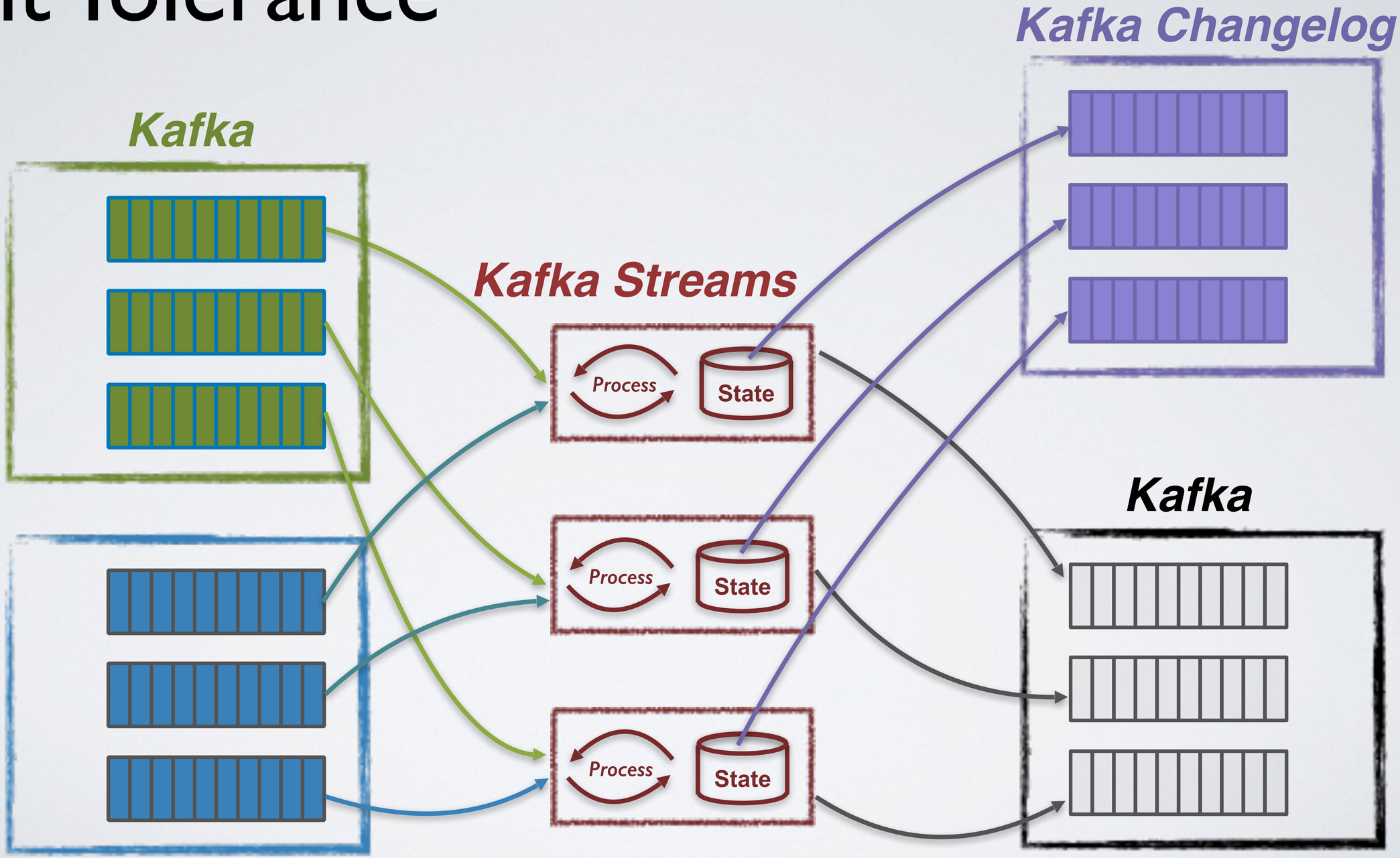


What about ***Fault Tolerance***?

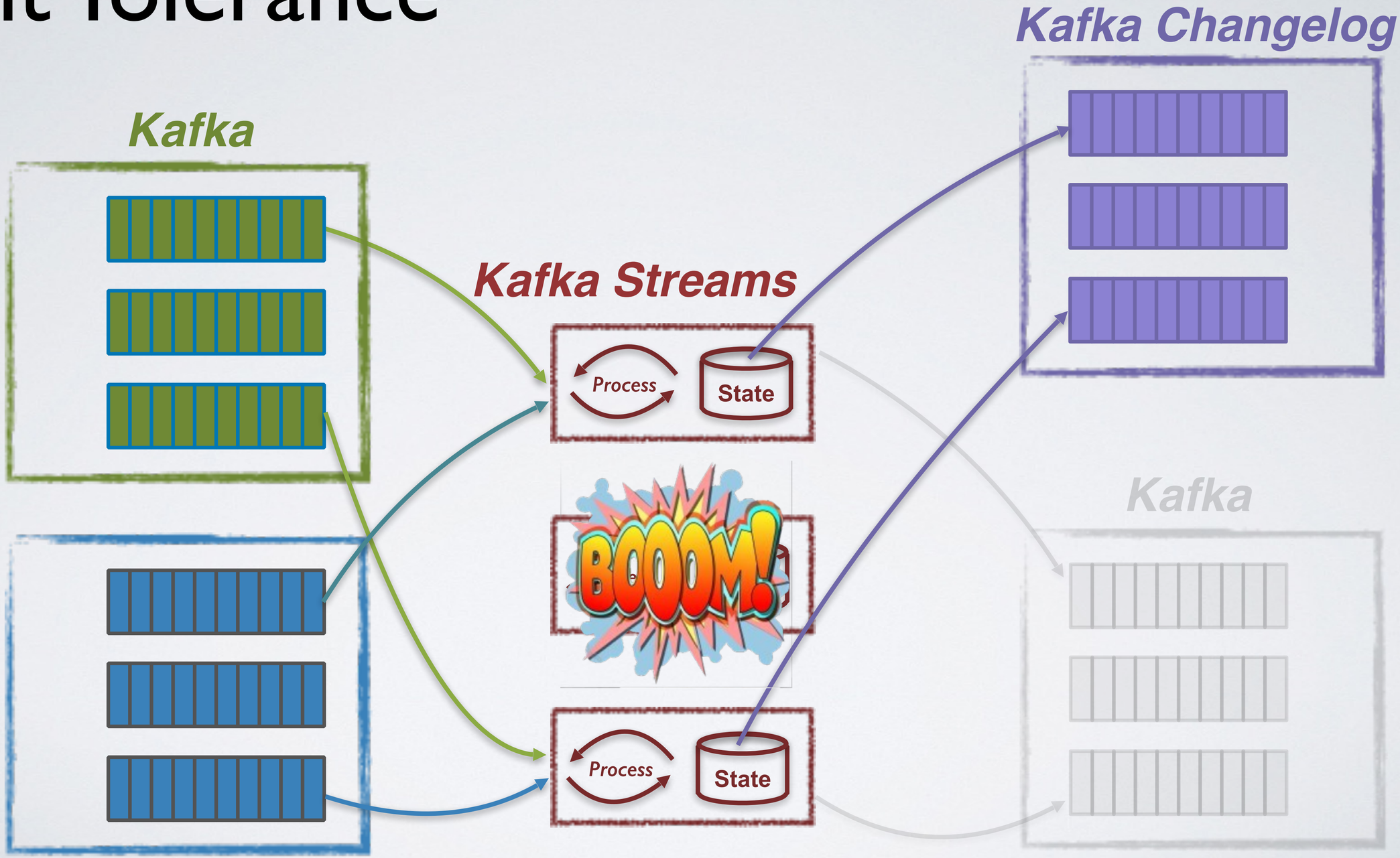
Remember?


TABLES ≈ STREAMS
(key1, value1)
(key2, value2)
(key1, value3)
key1
value1
key1
value1
key2
value2
key1
value3
key2
value2
(key1, value1)
(key2, value2)
(key1, value3)

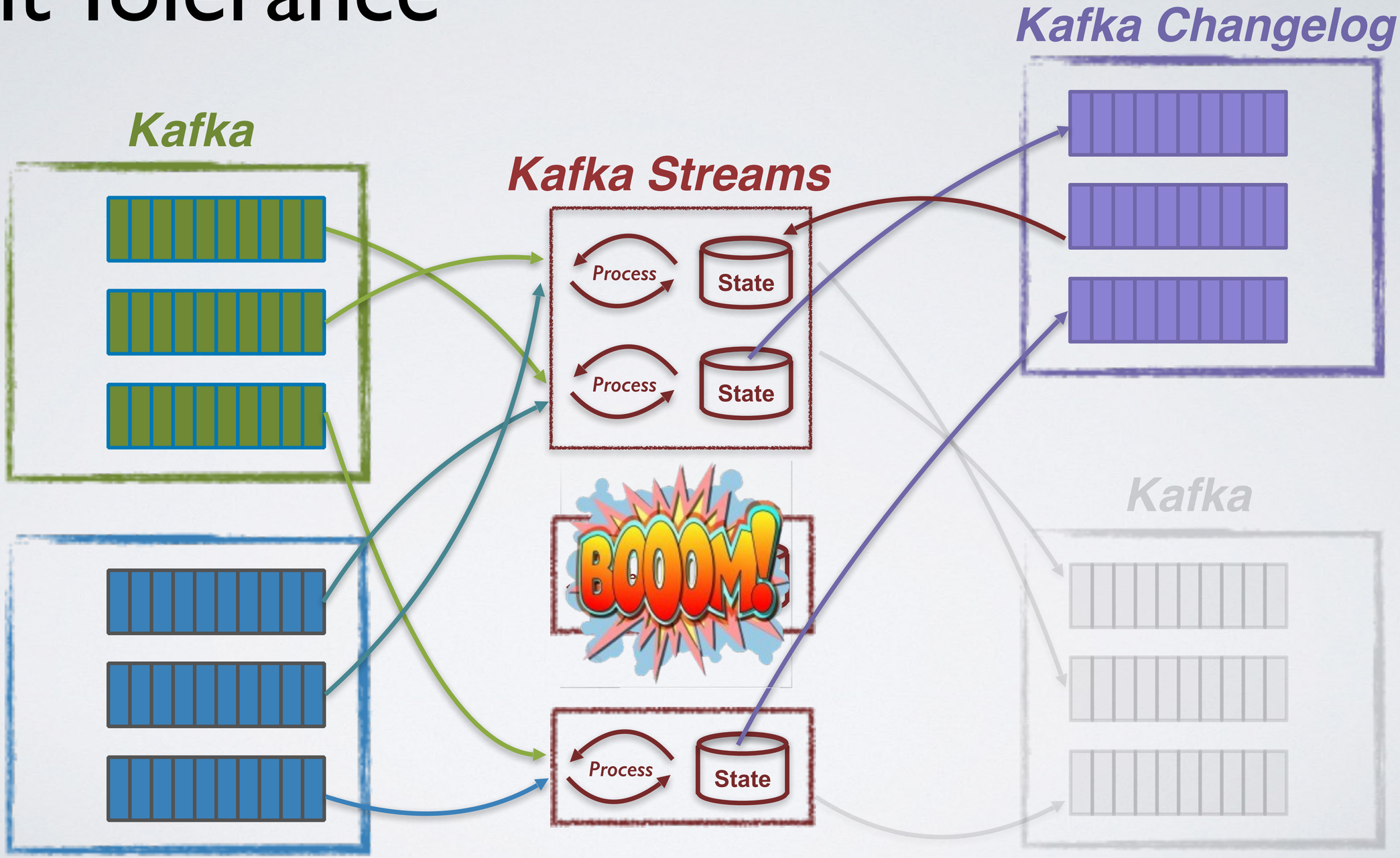
Fault Tolerance

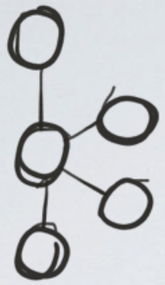


Fault Tolerance

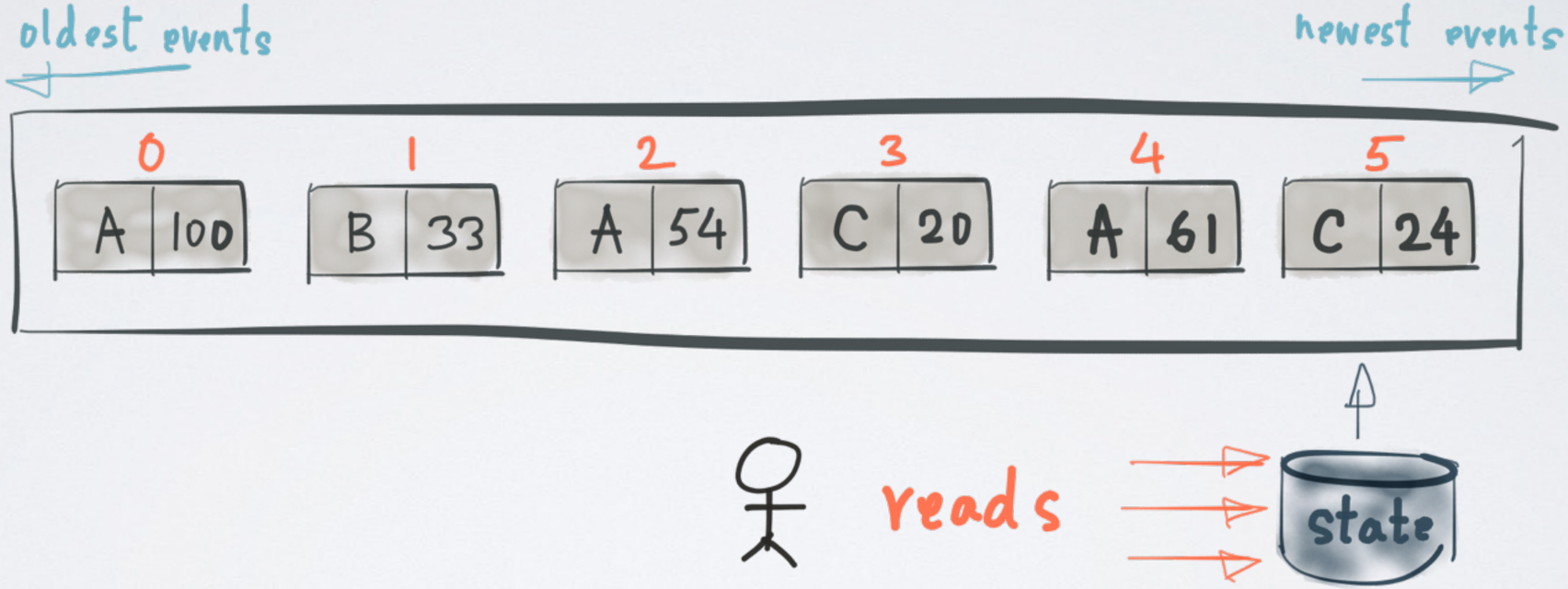


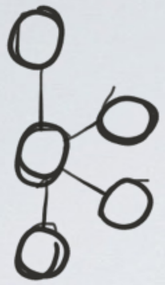
Fault Tolerance



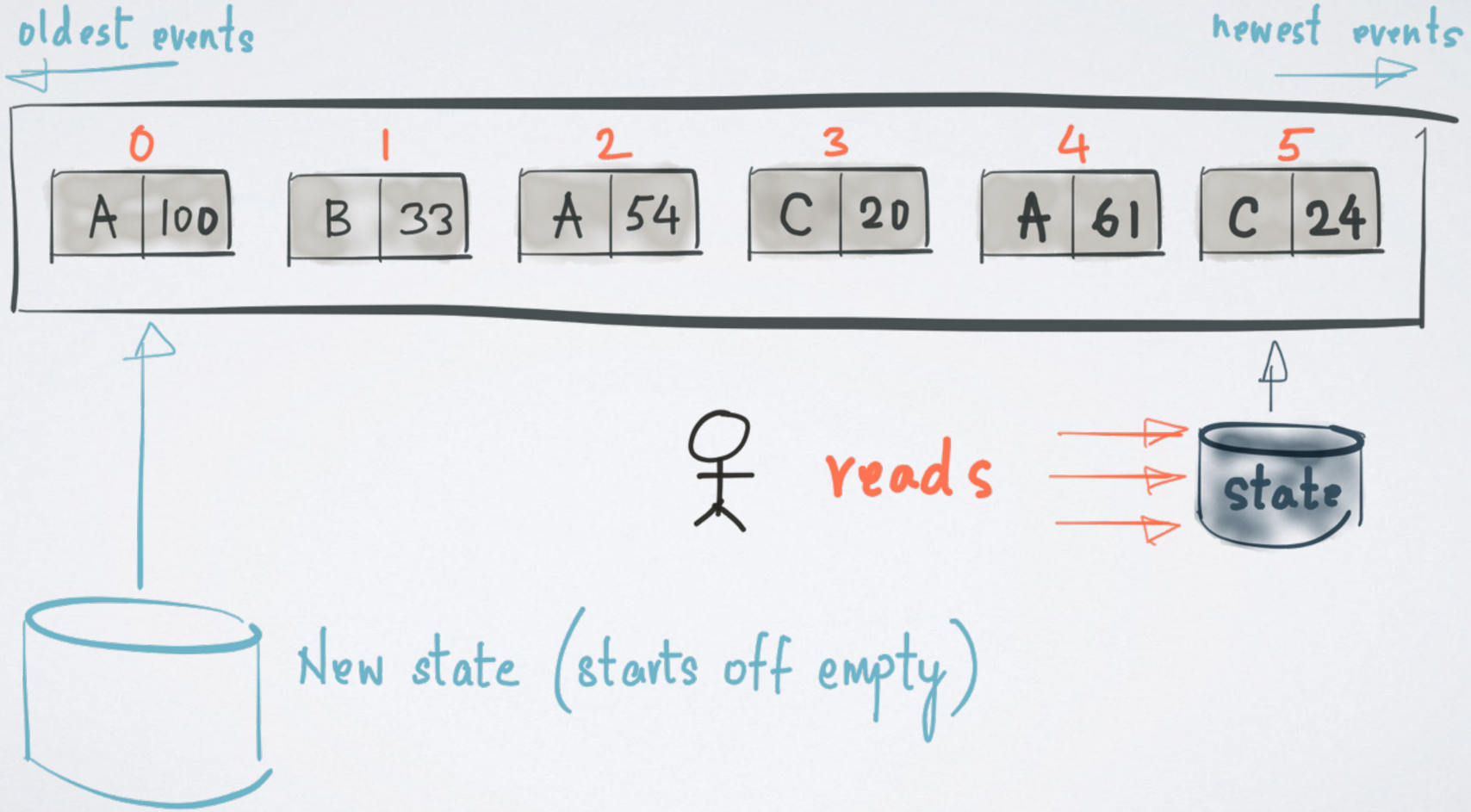


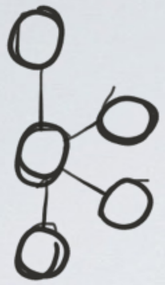
REPROCESSING



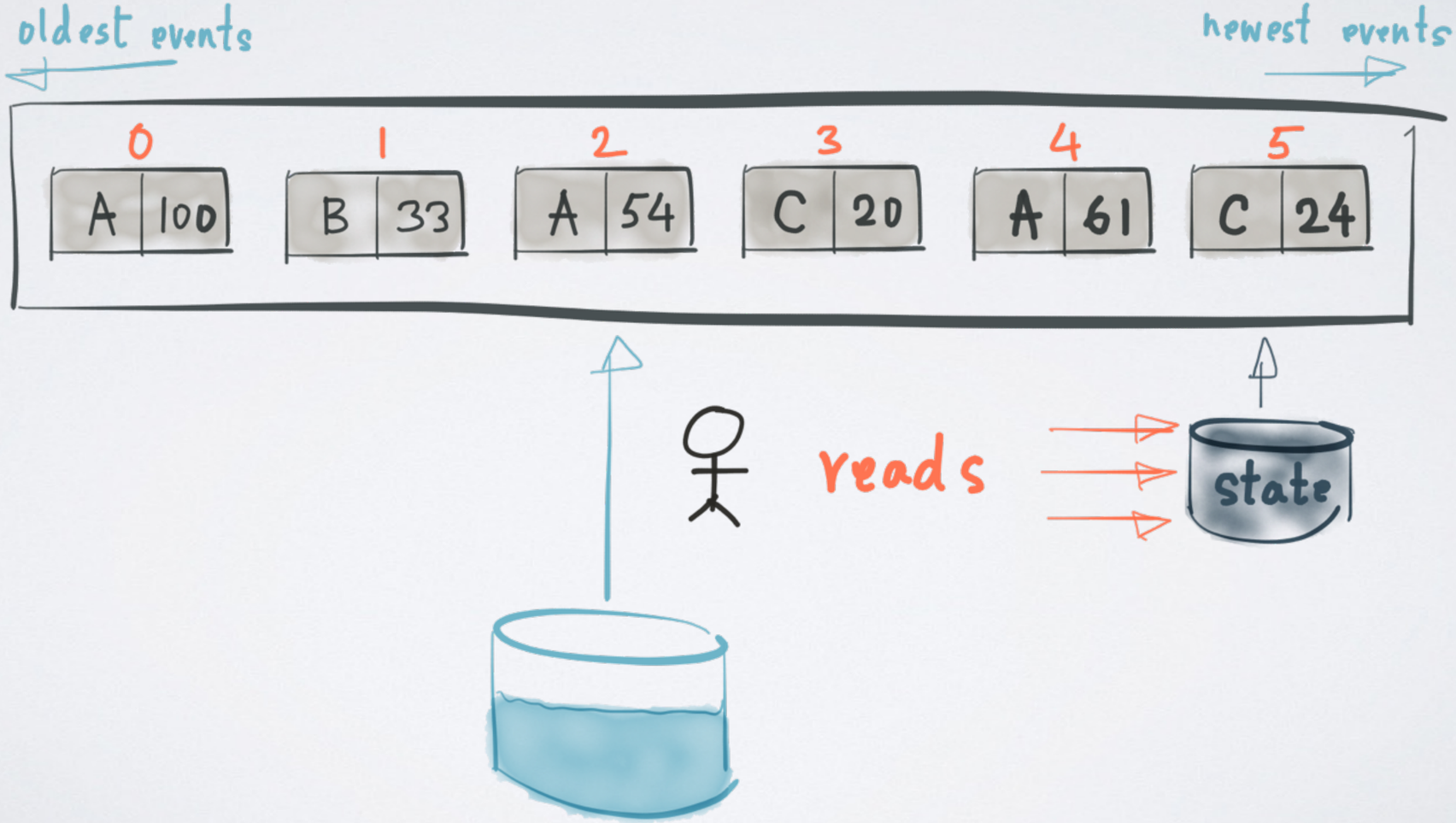


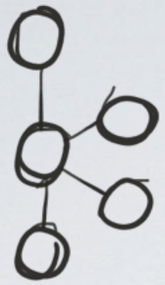
REPROCESSING



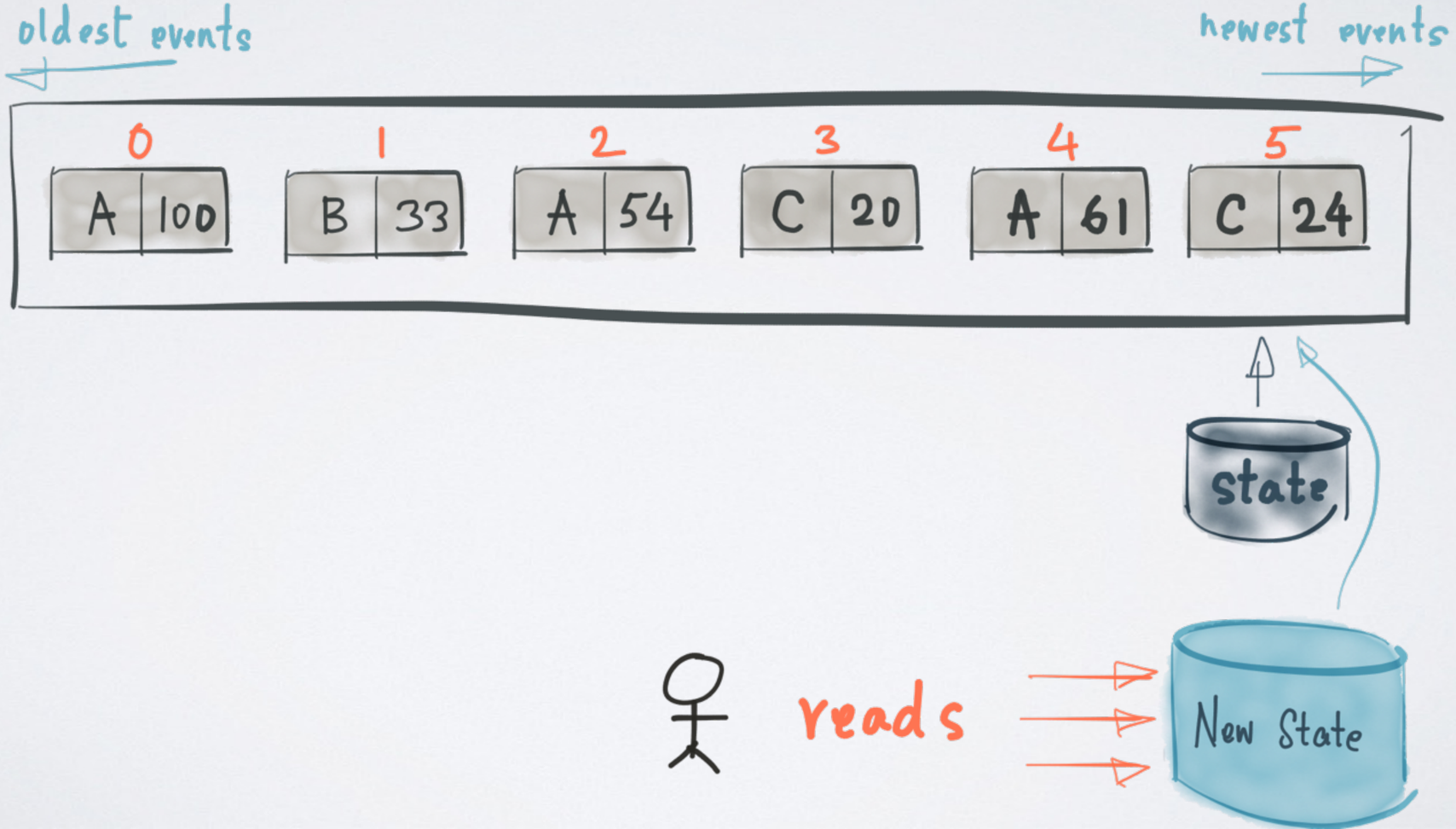


REPROCESSING





REPROCESSING



It's all about *Time*

- *Event-time* *(when an event is created)*
- *Processing-time* *(when an event is processed)*

Out-of-Order

Event-time	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Processing-time	*1999*	*2002*	*2005*	*1977*	*1980*	*1983*	*2015*

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

```
public long extract(ConsumerRecord<Object, Object> record) {
    return record.timestamp();
}
```

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

processing-time

```
public long extract(ConsumerRecord<Object, Object> record) {
    return record.timestamp();
}
```

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

processing-time

```
public long extract(ConsumerRecord<Object, Object> record) {
    return record.timestamp();
}
```

event-time

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

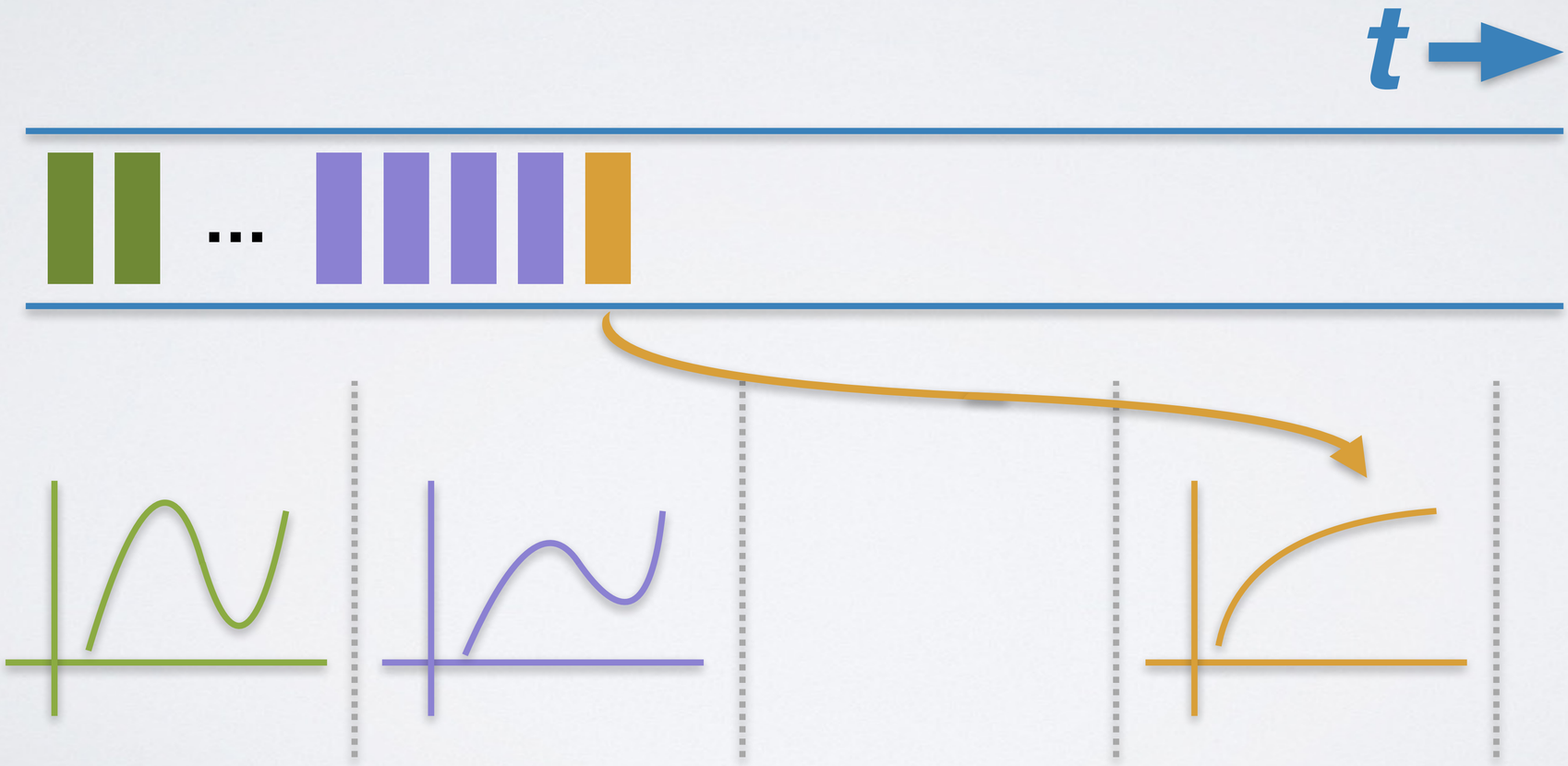
processing-time

```
public long extract(ConsumerRecord<Object, Object> record) {
    return ((JsonNode) record.value()).get("timestamp").longValue();
}
```

event-time

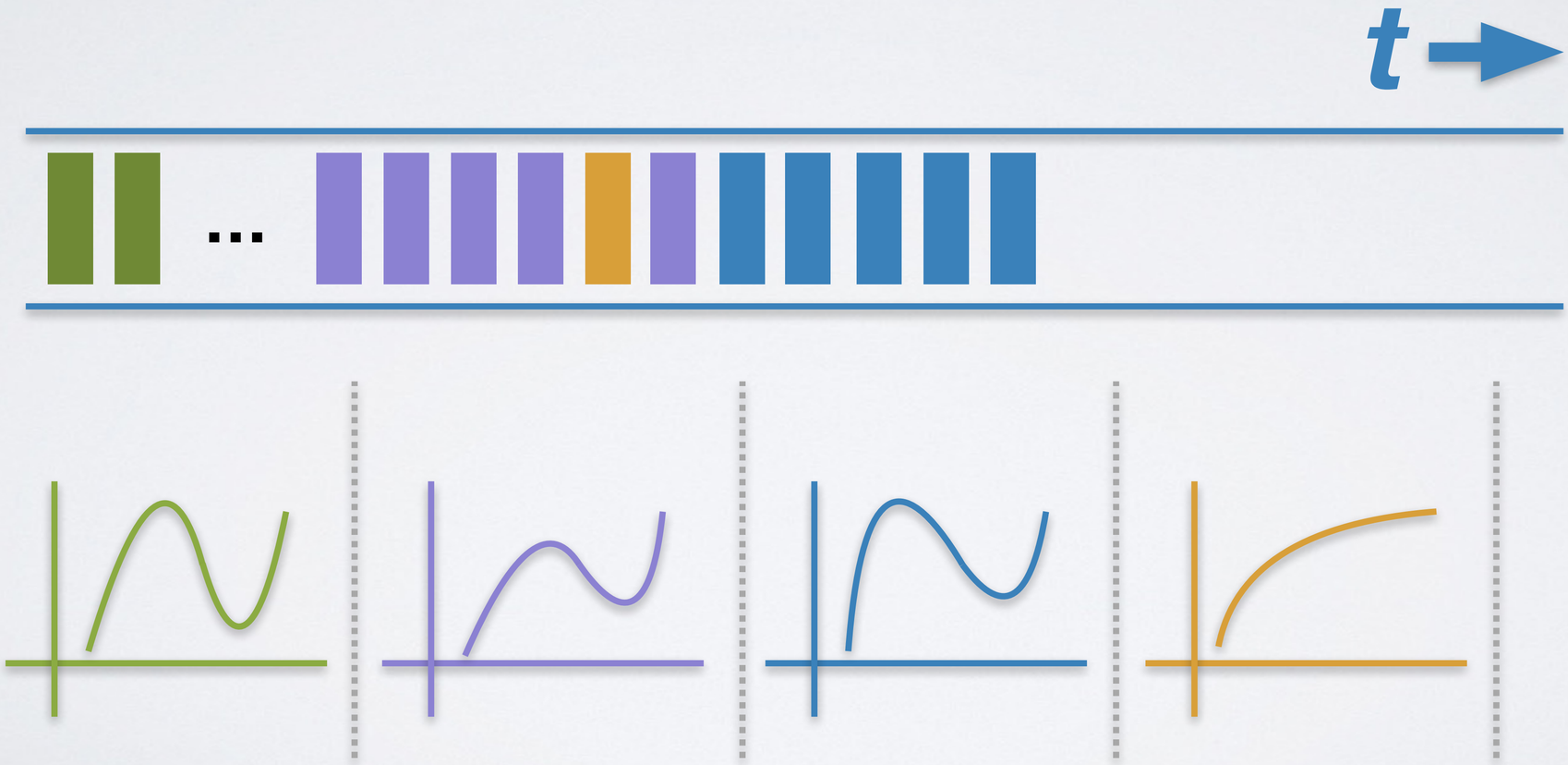
Windowing

Windowing

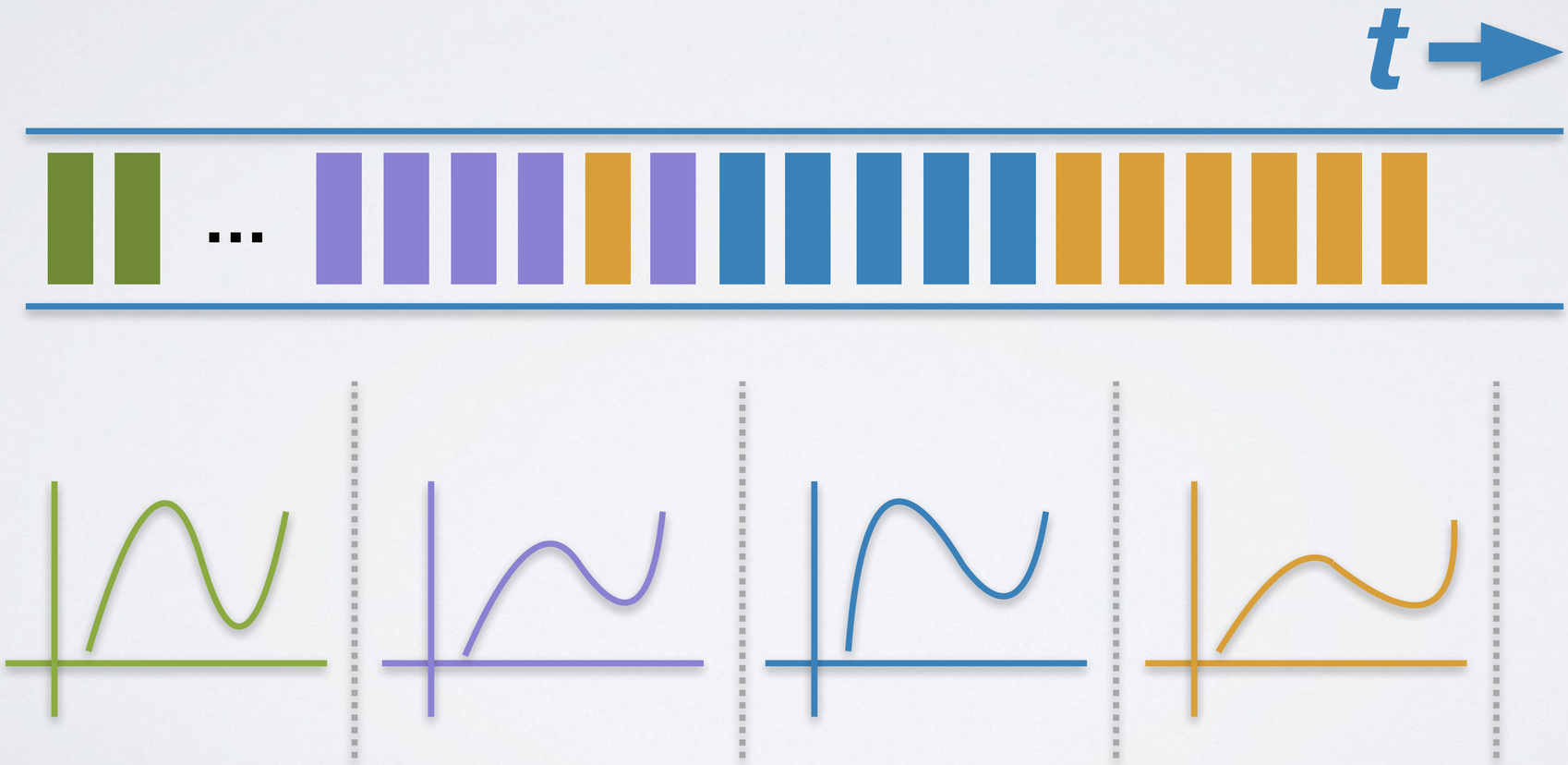


Windowing

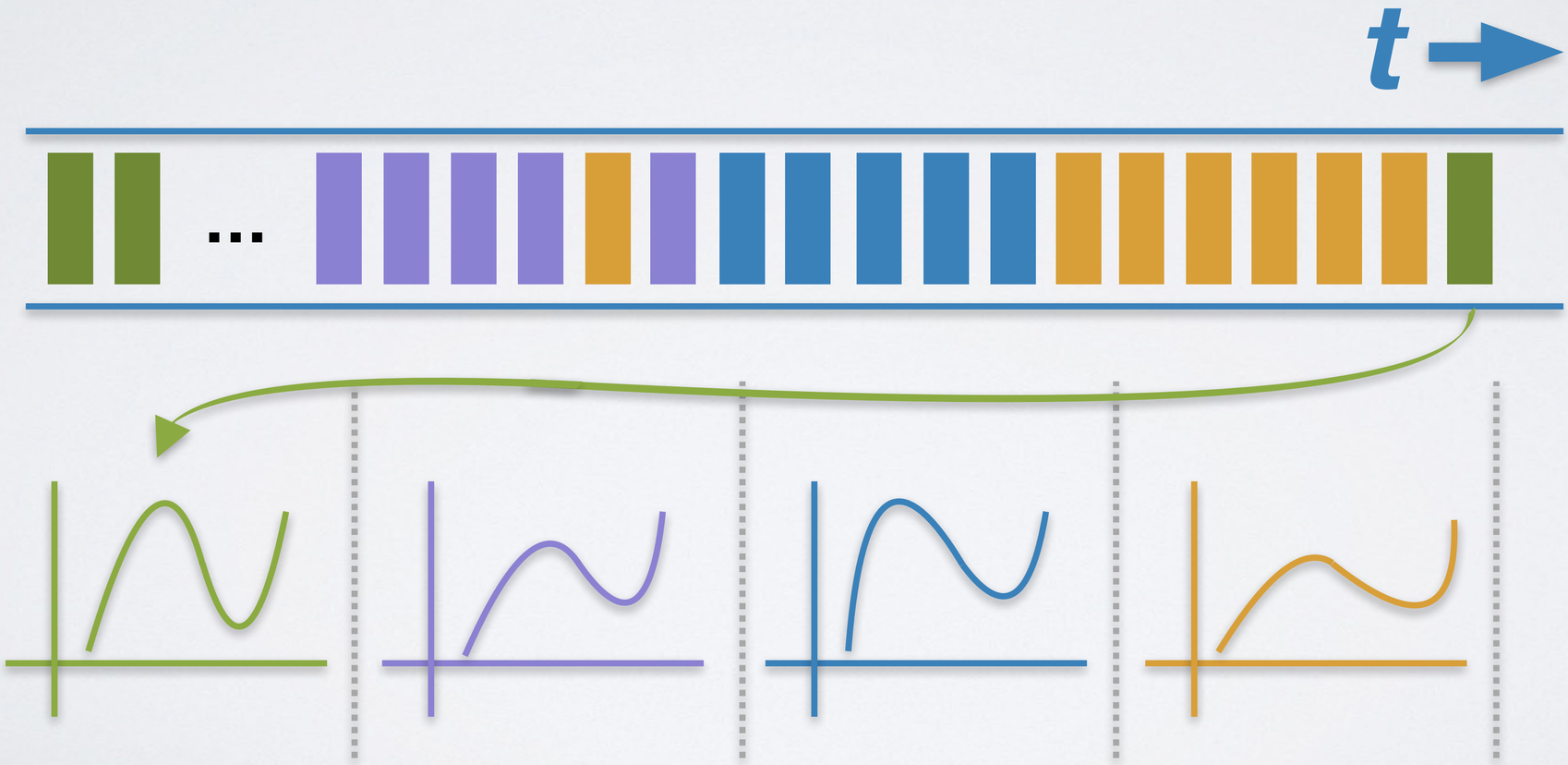
Windowing



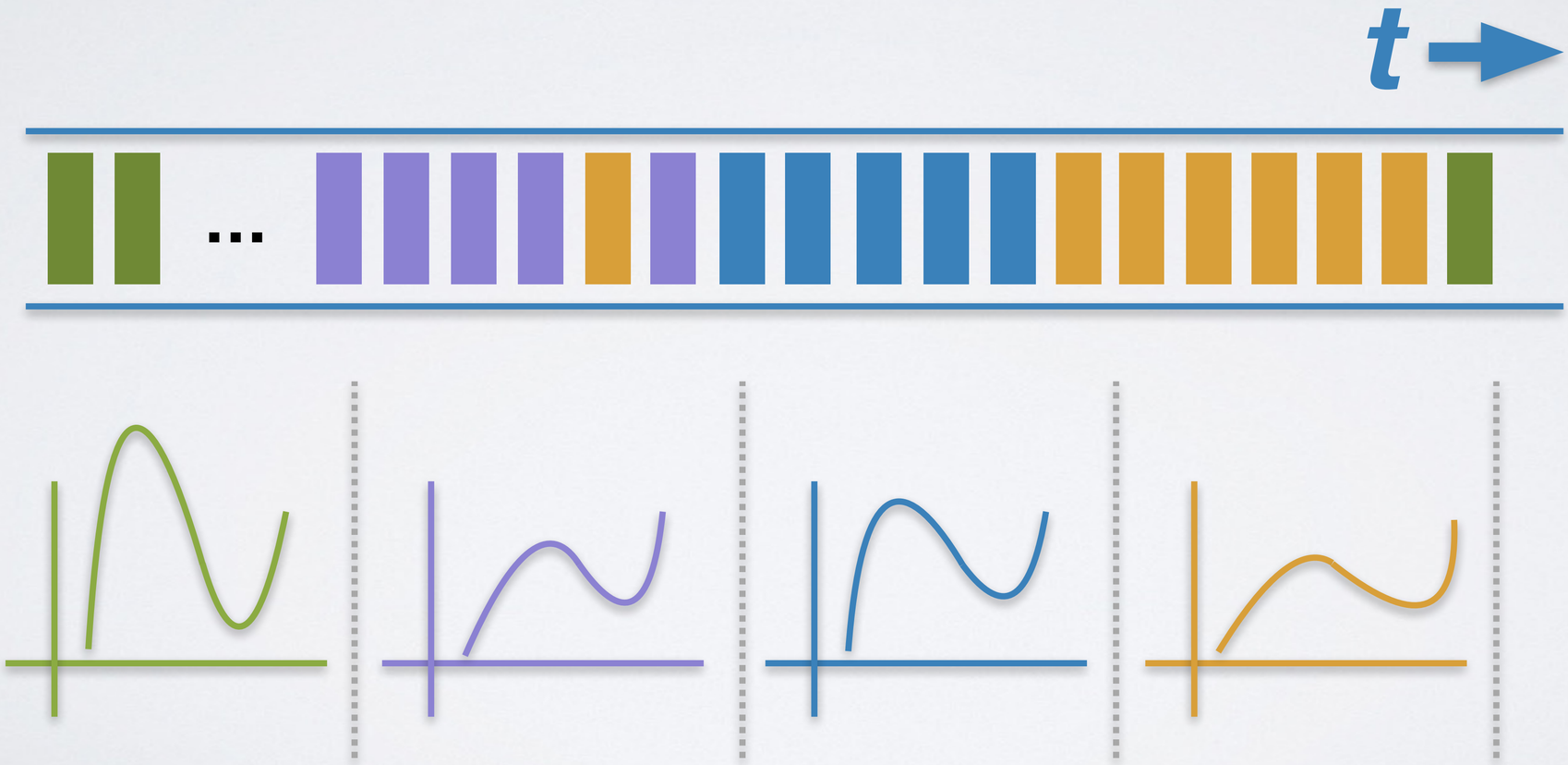
Windowing



Windowing



Windowing



Stream Processing *Hard Parts*

- *Ordering* ✓
- *State Management* ✓
- *Partitioning & Scalability* ✓
- *Time, Window & Out-of-order Data* ✓
- *Fault tolerance* ✓
- *Re-processing* ✓

For more details: http://docs.confluent.io/current

Stream Processing *Hard Parts*

- *Ordering* ✓
- *State Management* ✓

Simple is ***Beautiful***

- *Fault tolerance* ✓
- *Re-processing* ✓

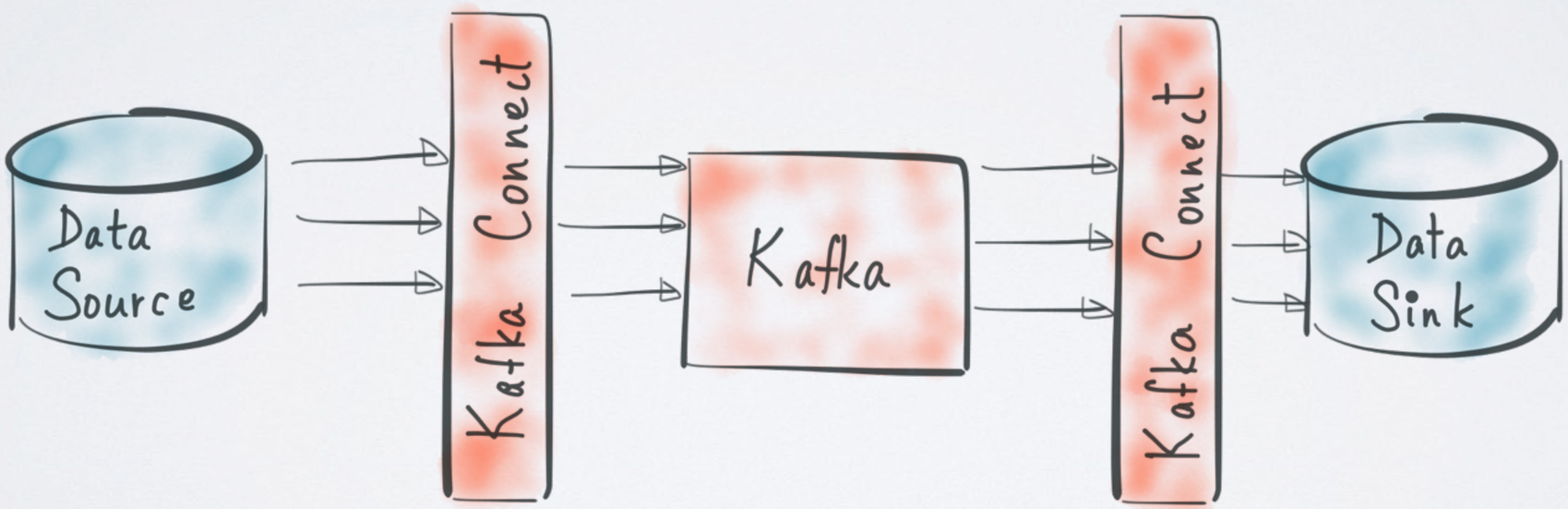
For more details: http://docs.confluent.io/current

Ongoing Work *(0.10+)*

- *Beyond Java APIs*
  - *Streaming SQL, Python client, etc*
- *Exactly-Once (end-to-end Semantics)*
- *… and more*

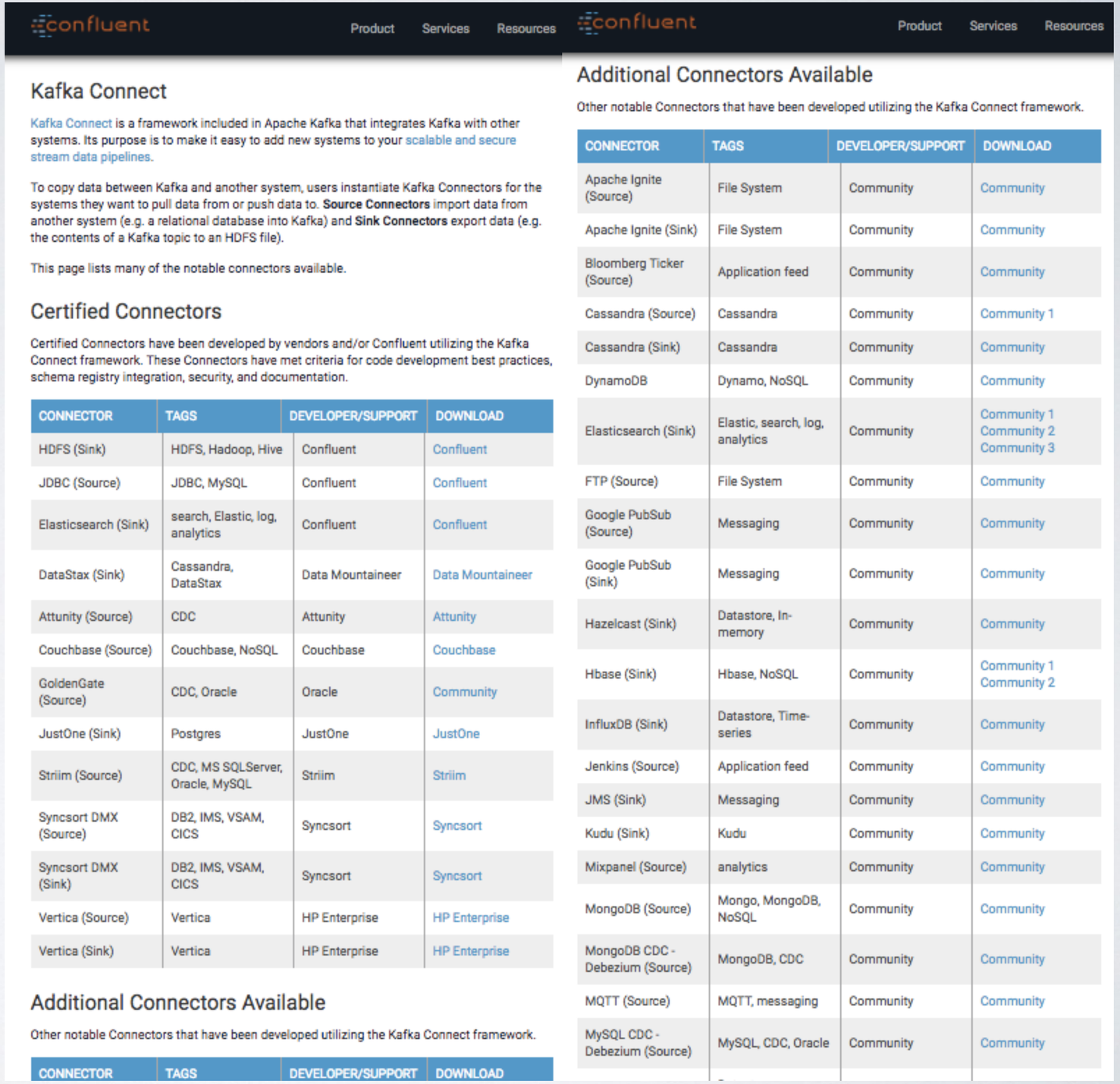
*But how to get data in / out **Kafka**?*

Kafka Connect



Connectors

- *40+ since first release this Feb (0.9+)*
- *13 from confluent & partners*



confluent Product Services Resources

Kafka Connect

Kafka Connect is a framework included in Apache Kafka that integrates Kafka with other systems. Its purpose is to make it easy to add new systems to your scalable and secure stream data pipelines.

To copy data between Kafka and another system, users instantiate Kafka Connectors for the systems they want to pull data from or push data to. **Source Connectors** import data from another system (e.g. a relational database into Kafka) and **Sink Connectors** export data (e.g. the contents of a Kafka topic to an HDFS file).

This page lists many of the notable connectors available.

Certified Connectors

Certified Connectors have been developed by vendors and/or Confluent utilizing the Kafka Connect framework. These Connectors have met criteria for code development best practices, schema registry integration, security, and documentation.

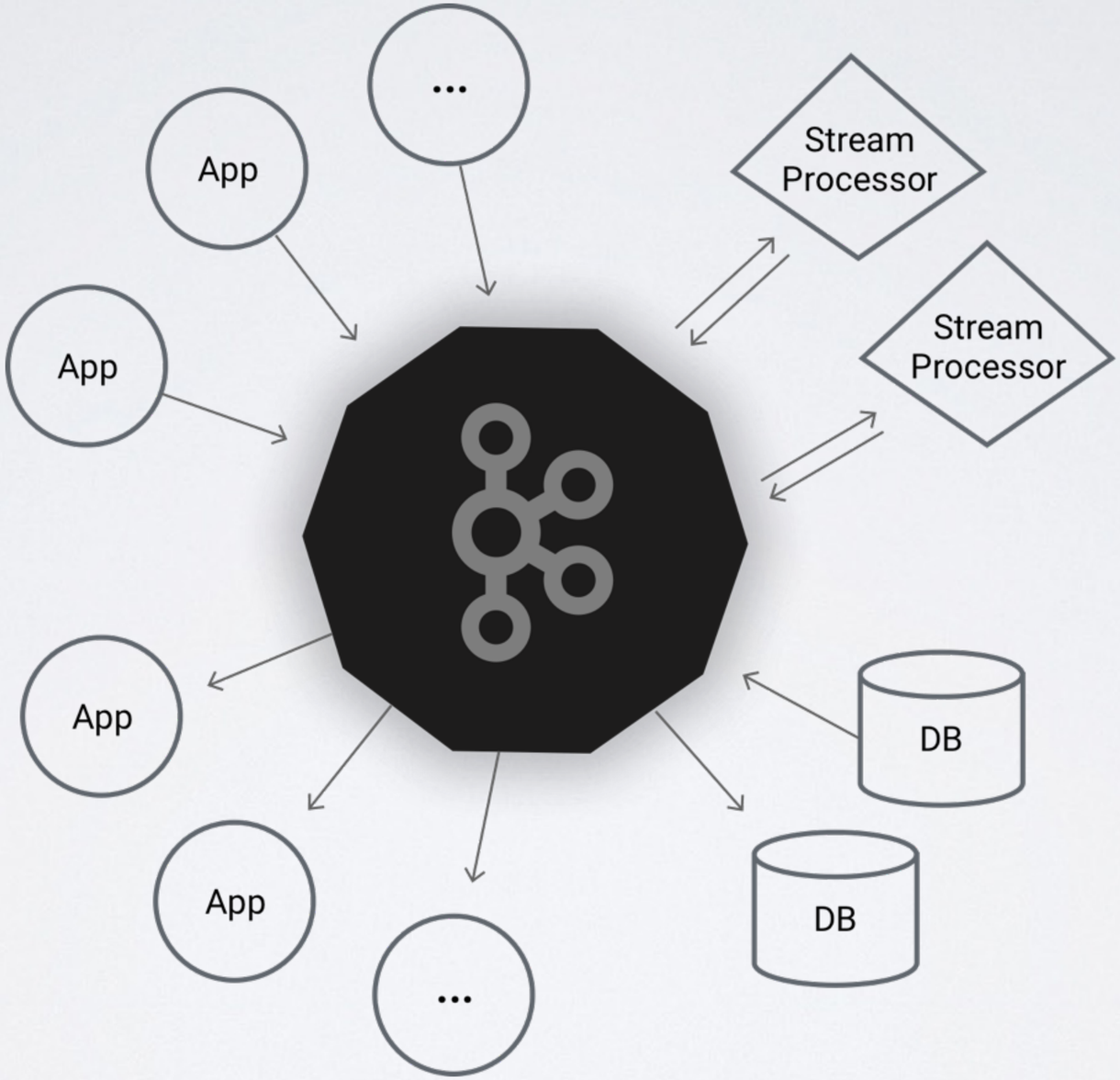
CONNECTOR	TAGS	DEVELOPER/SUPPORT	DOWNLOAD
HDFS (Sink)	HDFS, Hadoop, Hive	Confluent	Confluent
JDBC (Source)	JDBC, MySQL	Confluent	Confluent
Elasticsearch (Sink)	search, Elastic, log, analytics	Confluent	Confluent
DataStax (Sink)	Cassandra, DataStax	Data Mountaineer	Data Mountaineer
Attunity (Source)	CDC	Attunity	Attunity
Couchbase (Source)	Couchbase, NoSQL	Couchbase	Couchbase
GoldenGate (Source)	CDC, Oracle	Oracle	Community
JustOne (Sink)	Postgres	JustOne	JustOne
Striim (Source)	CDC, MS SQLServer, Oracle, MySQL	Striim	Striim
Syncsort DMX (Source)	DB2, IMS, VSAM, CICS	Syncsort	Syncsort
Syncsort DMX (Sink)	DB2, IMS, VSAM, CICS	Syncsort	Syncsort
Vertica (Source)	Vertica	HP Enterprise	HP Enterprise
Vertica (Sink)	Vertica	HP Enterprise	HP Enterprise

Additional Connectors Available

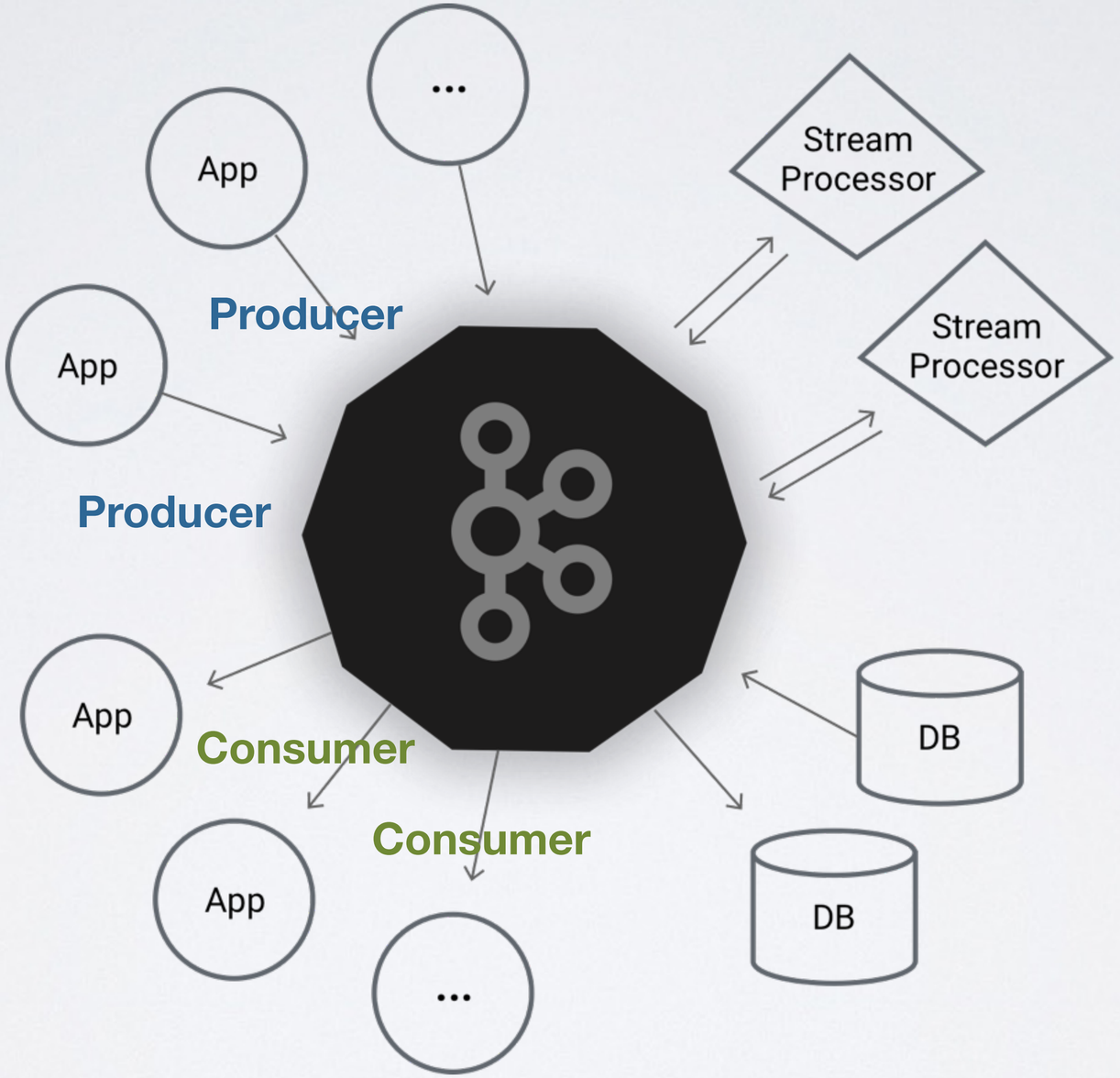
Other notable Connectors that have been developed utilizing the Kafka Connect framework.

CONNECTOR	TAGS	DEVELOPER/SUPPORT	DOWNLOAD
Apache Ignite (Source)	File System	Community	Community
Apache Ignite (Sink)	File System	Community	Community
Bloomberg Ticker (Source)	Application feed	Community	Community
Cassandra (Source)	Cassandra	Community	Community 1
Cassandra (Sink)	Cassandra	Community	Community
DynamoDB	Dynamo, NoSQL	Community	Community
Elasticsearch (Sink)	Elastic, search, log, analytics	Community	Community 1 Community 2 Community 3
FTP (Source)	File System	Community	Community
Google PubSub (Source)	Messaging	Community	Community
Google PubSub (Sink)	Messaging	Community	Community
Hazelcast (Sink)	Datastore, In-memory	Community	Community
Hbase (Sink)	Hbase, NoSQL	Community	Community 1 Community 2
InfluxDB (Sink)	Datastore, Time-series	Community	Community
Jenkins (Source)	Application feed	Community	Community
JMS (Sink)	Messaging	Community	Community
Kudu (Sink)	Kudu	Community	Community
Mixpanel (Source)	analytics	Community	Community
MongoDB (Source)	Mongo, MongoDB, NoSQL	Community	Community
MongoDB CDC - Debezium (Source)	MongoDB, CDC	Community	Community
MQTT (Source)	MQTT, messaging	Community	Community
MySQL CDC - Debezium (Source)	MySQL, CDC, Oracle	Community	Community

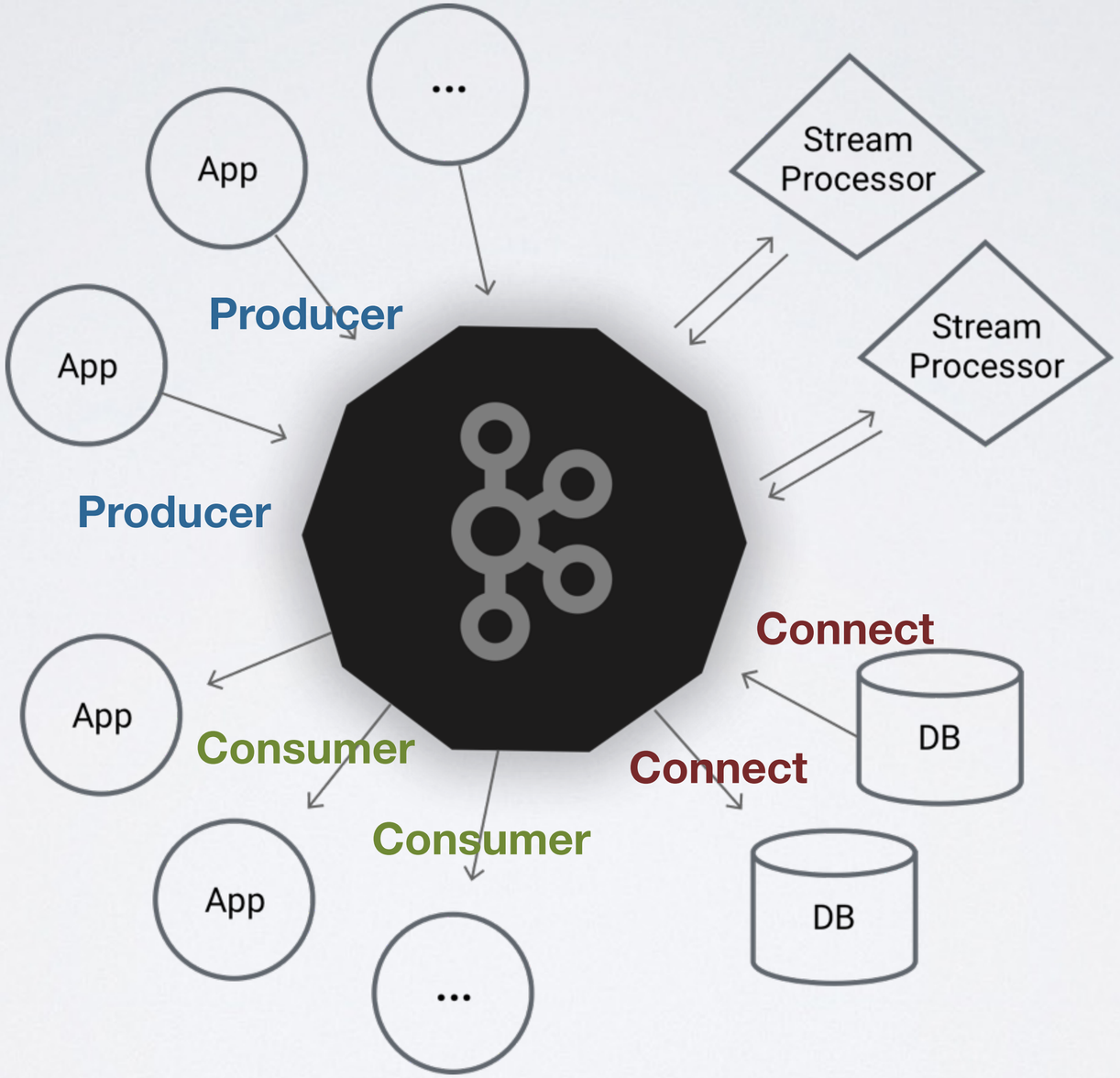
Data Streaming Platform



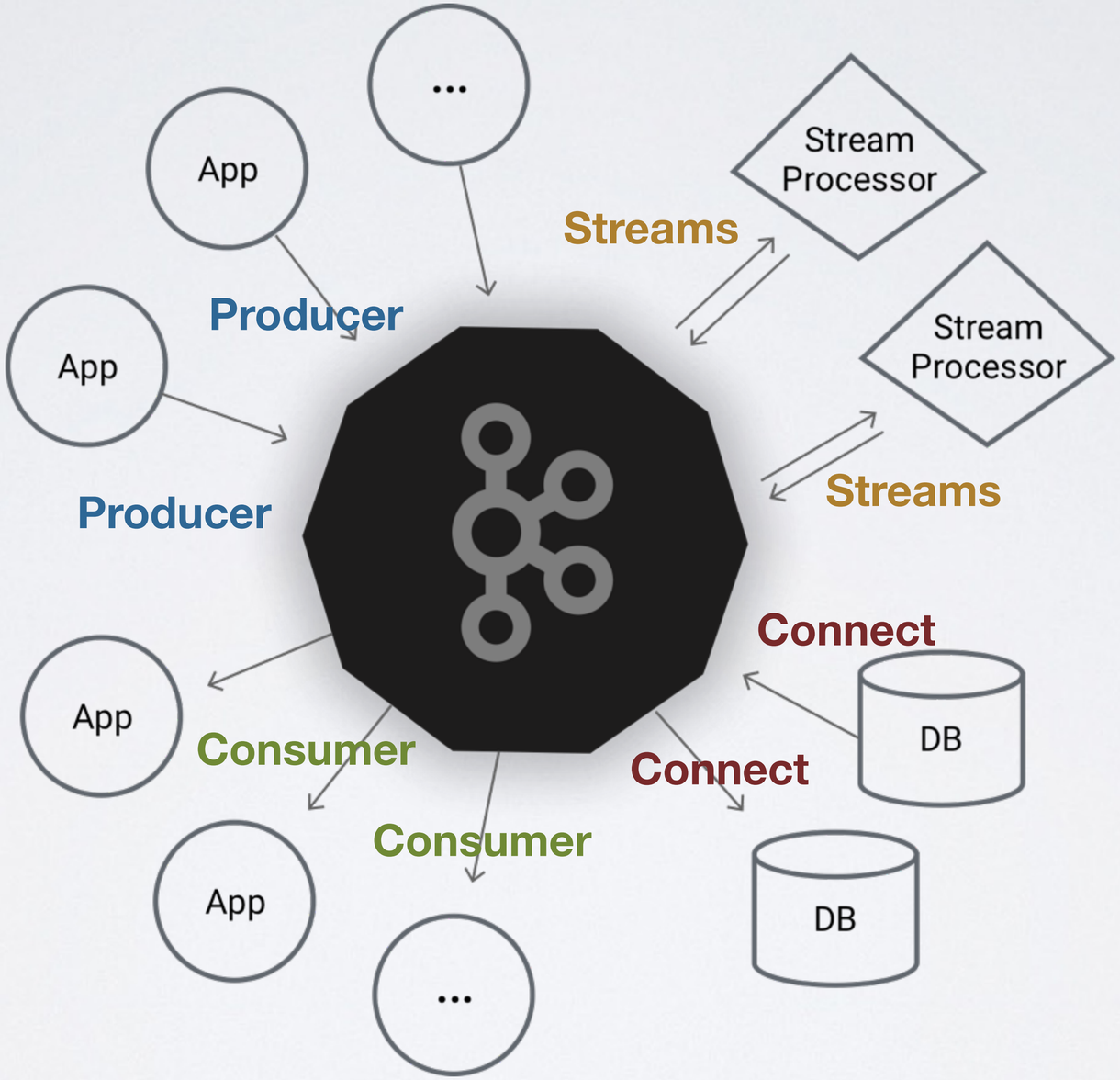
Data Streaming Platform



Data Streaming Platform



Data Streaming Platform



Take-aways

- **Stream Processing:** *a new programming paradigm*

Take-aways

- **Stream Processing:** *a new programming paradigm*
-  **Kafka Streams:** *stream processing made easy*

Take-aways

THANKS!

- **Stream Processing:** *a new programming paradigm*
-  **Kafka Streams:** *stream processing made easy*

Join Kafka Summit 2017 @ NYC & SF

Confluent Webinar: http://www.confluent.io/resources

Guozhang Wang | guozhang@confluent.io | @guozhangwang

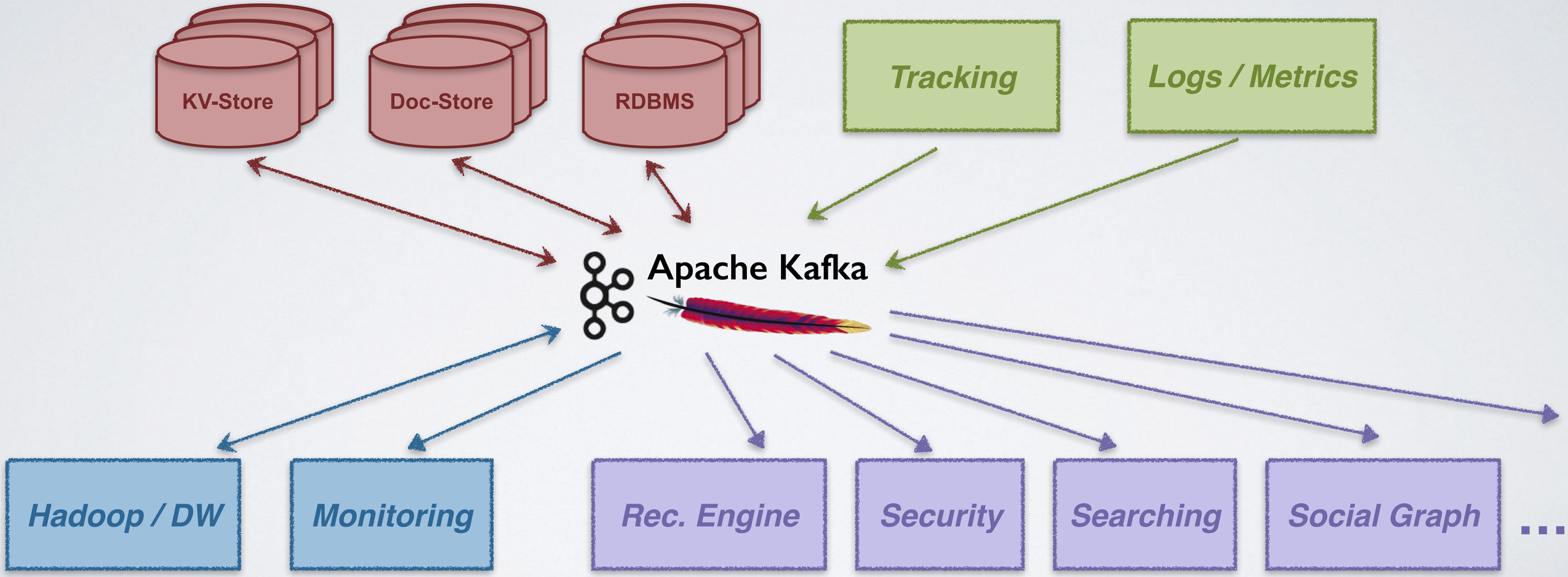
BACKUP SLIDES

confluent

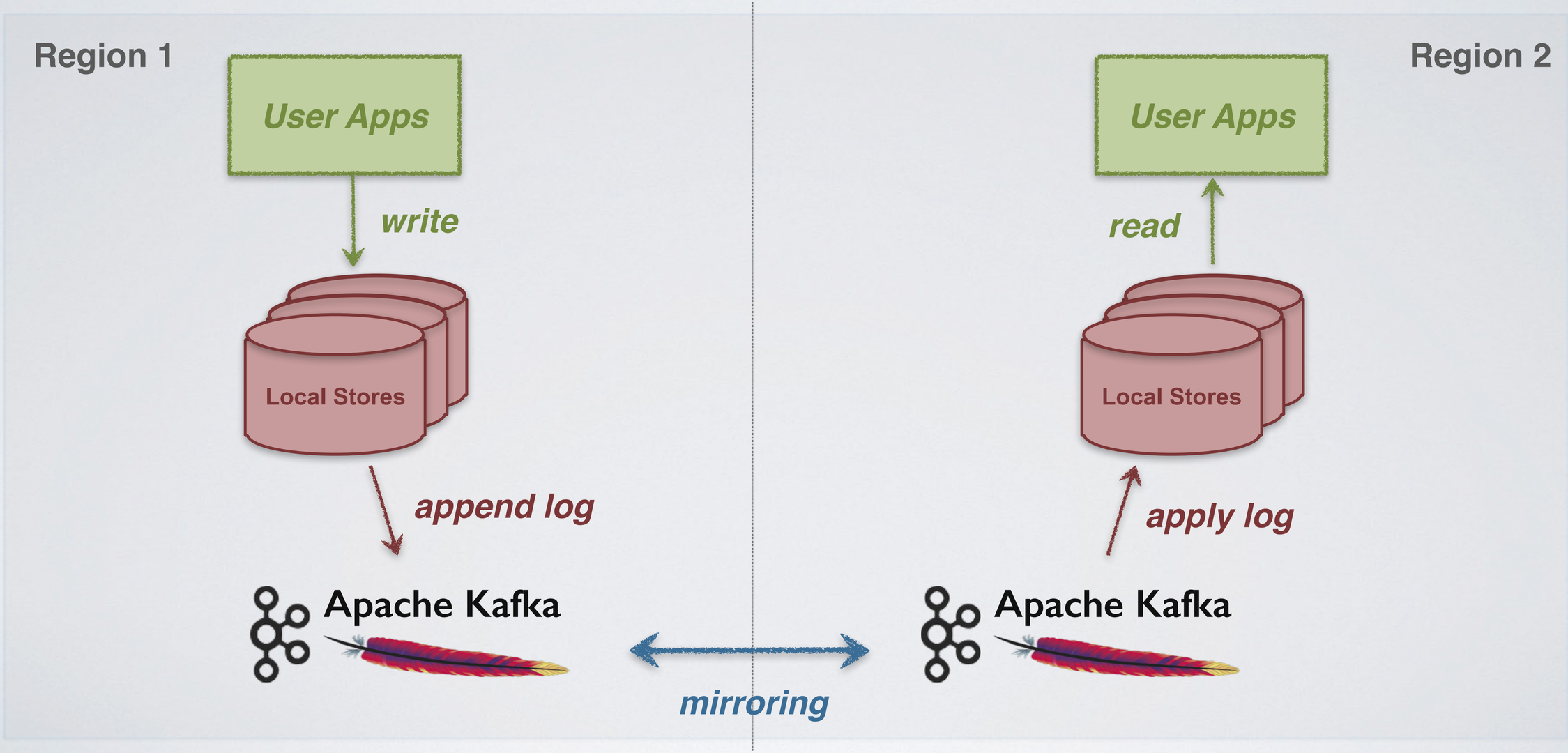
Real-time streams powered by Apache Kafka.

We are Hiring!

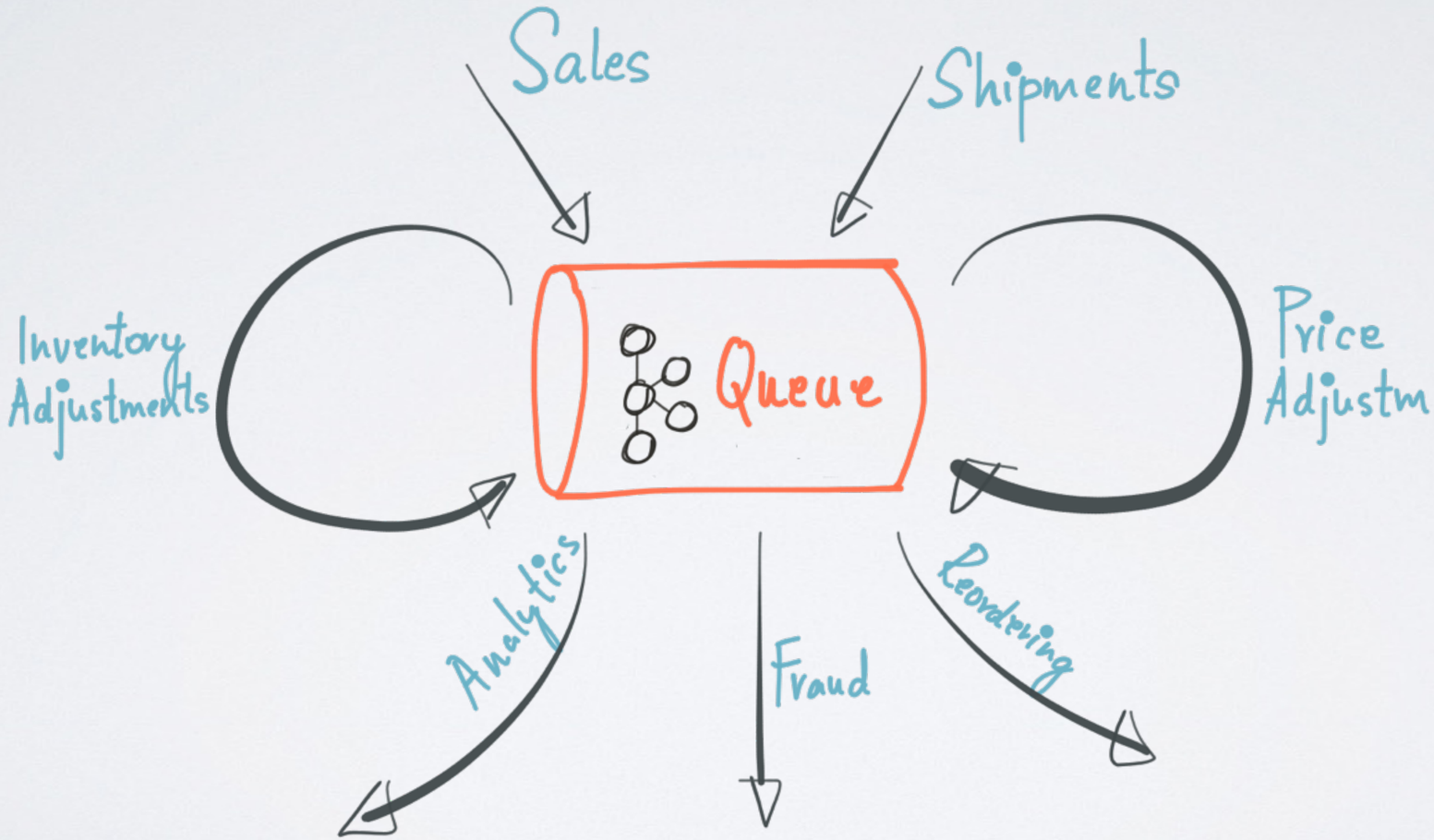
Example: Centralized Data Pipeline



Example: Data Store Geo-Replication



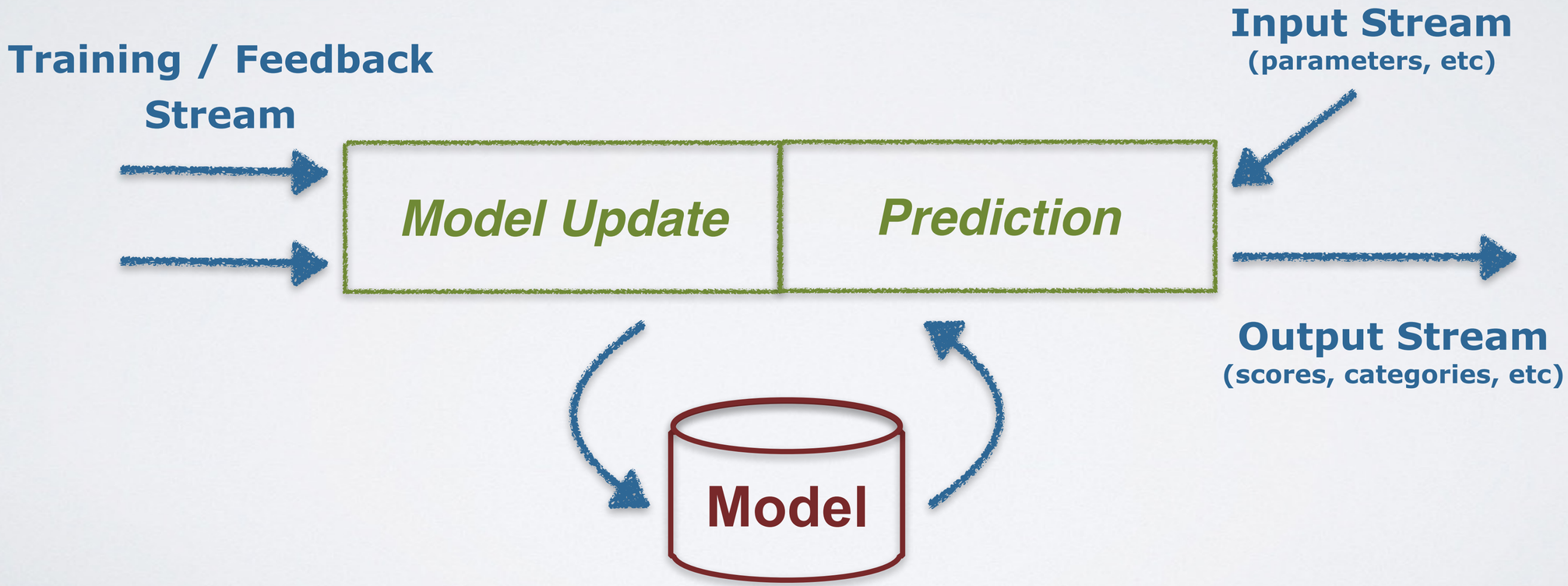
Example: Async. Micro-Services



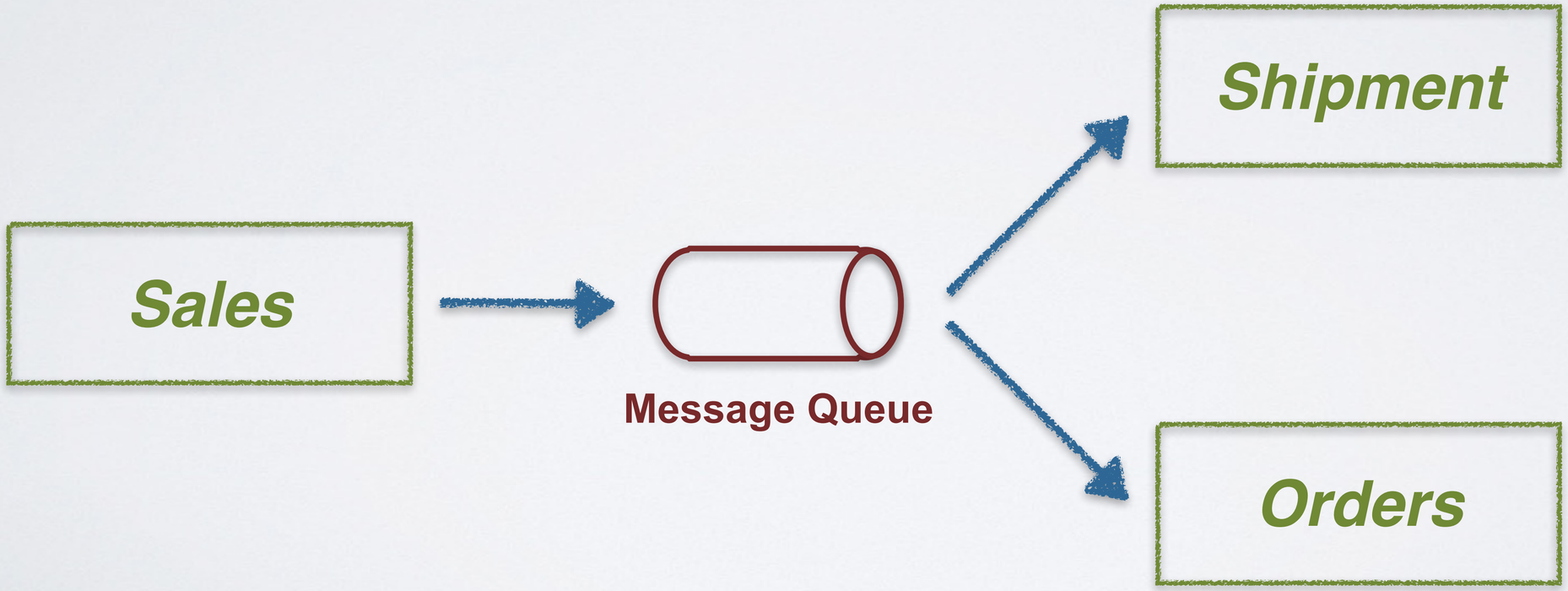
Ongoing Work *(0.10+)*

- *Beyond Java APIs*
  - *SQL support, Python client, etc*
- *End-to-End Semantics (exactly-once)*
- *… and more*

Online Machine Learning

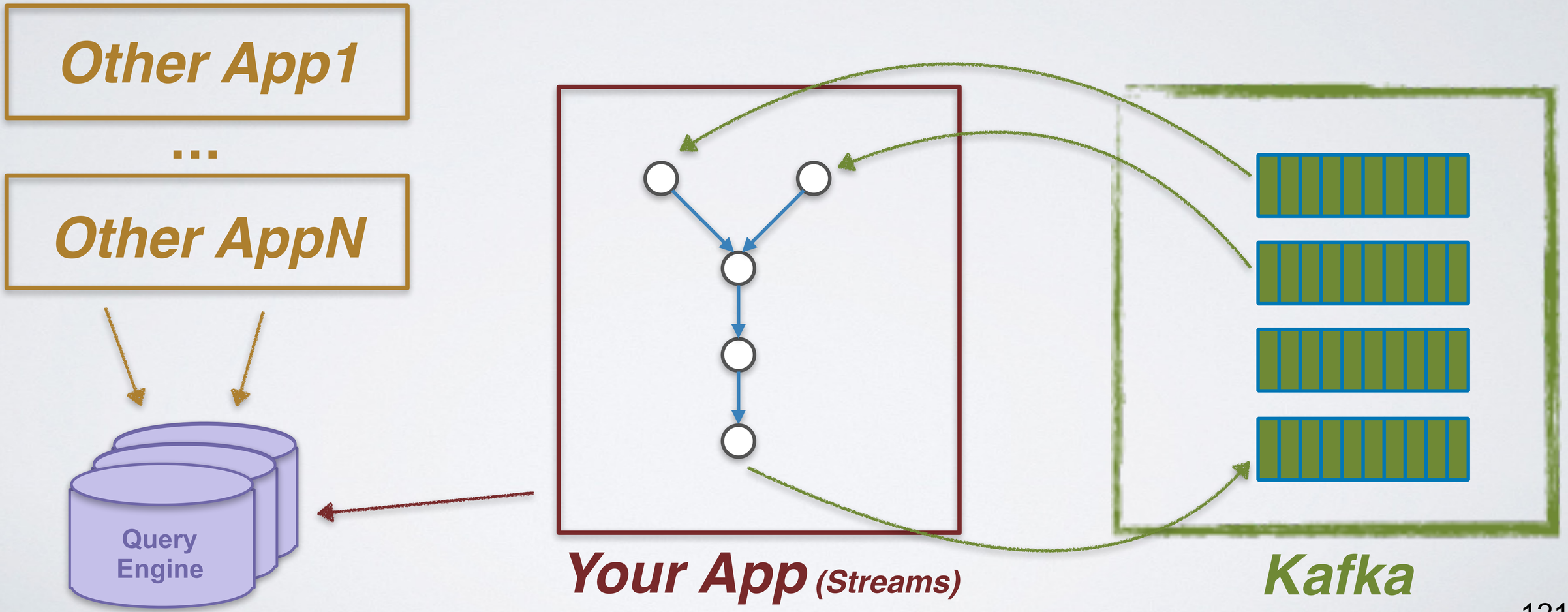


Async. Micro-services



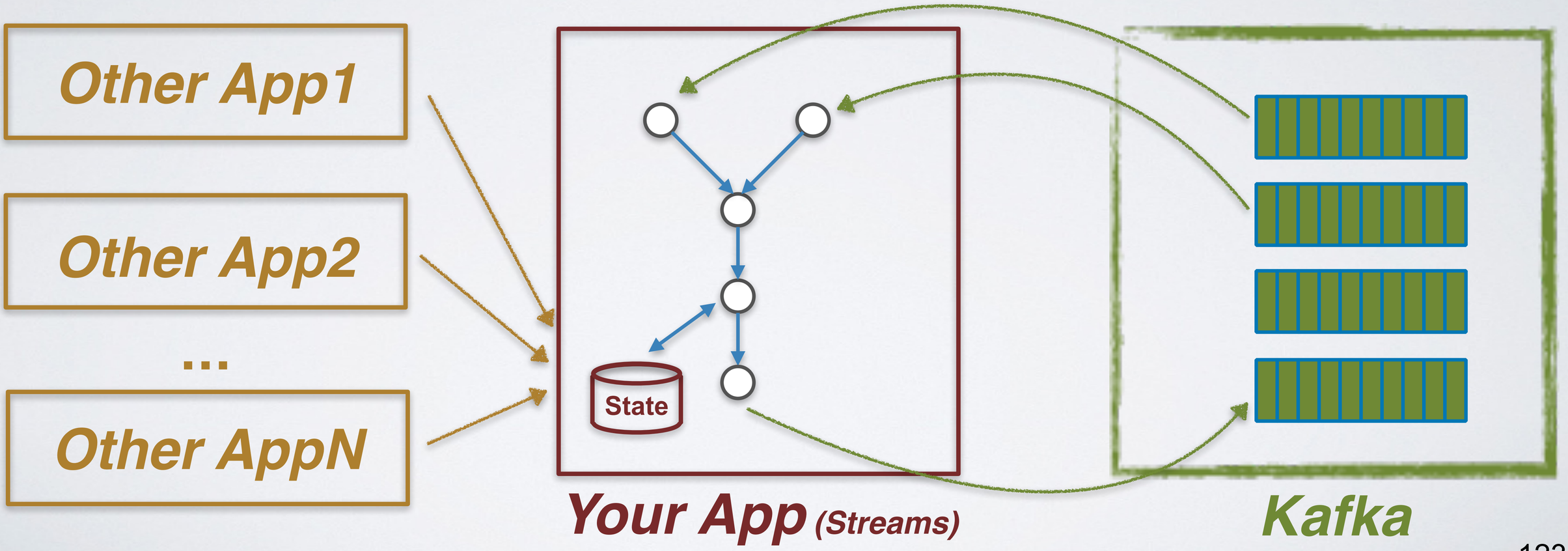
Real-time Analytics

Interactive Queries on States



Interactive Queries on States

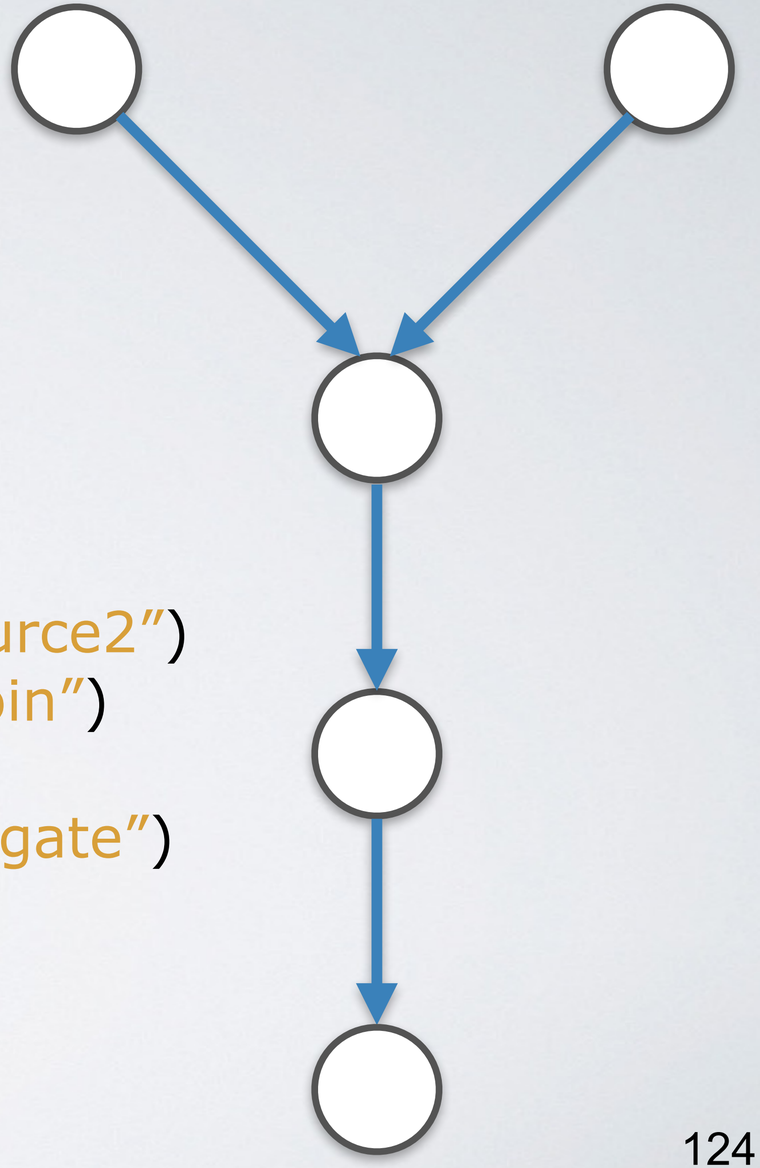
Interactive Queries on States (0.10.1+)



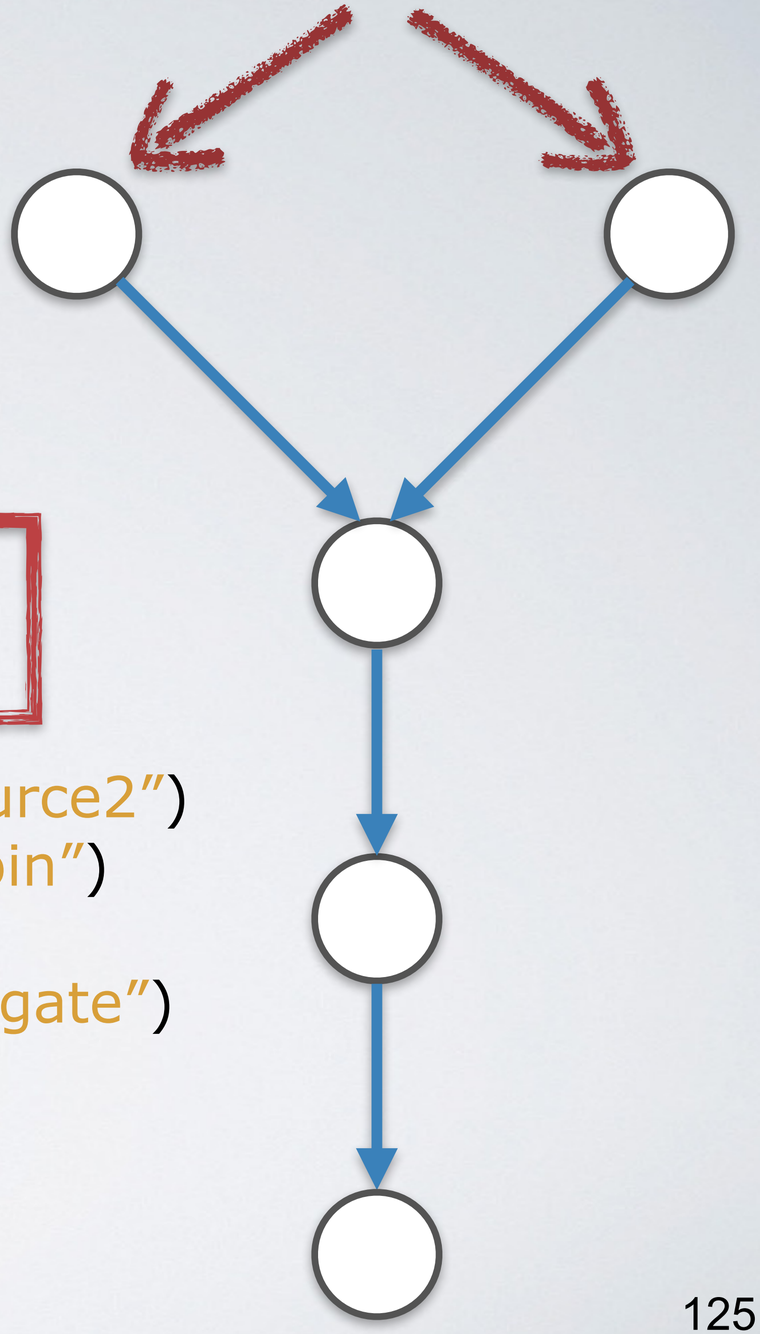
Processor Topology

```
KStream<..> stream1 = builder.stream(”topic1”);
KStream<..> stream2 = builder.table(”topic2”);

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(..);

aggregated.to(”topic3”);
```

```
builder.addSource(”Source1”, ”topic1”)
        .addSource(”Source2”, ”topic2”)
        .addProcessor(”Join”, MyJoin:new, ”Source1”, ”Source2”)
        .addProcessor(”Aggregate”, MyAggregate:new, ”Join”)
        .addStateStore(Stores.persistent().build(), ”Aggregate”)
        .addSink(”Sink”, ”topic3”, ”Aggregate”)
```



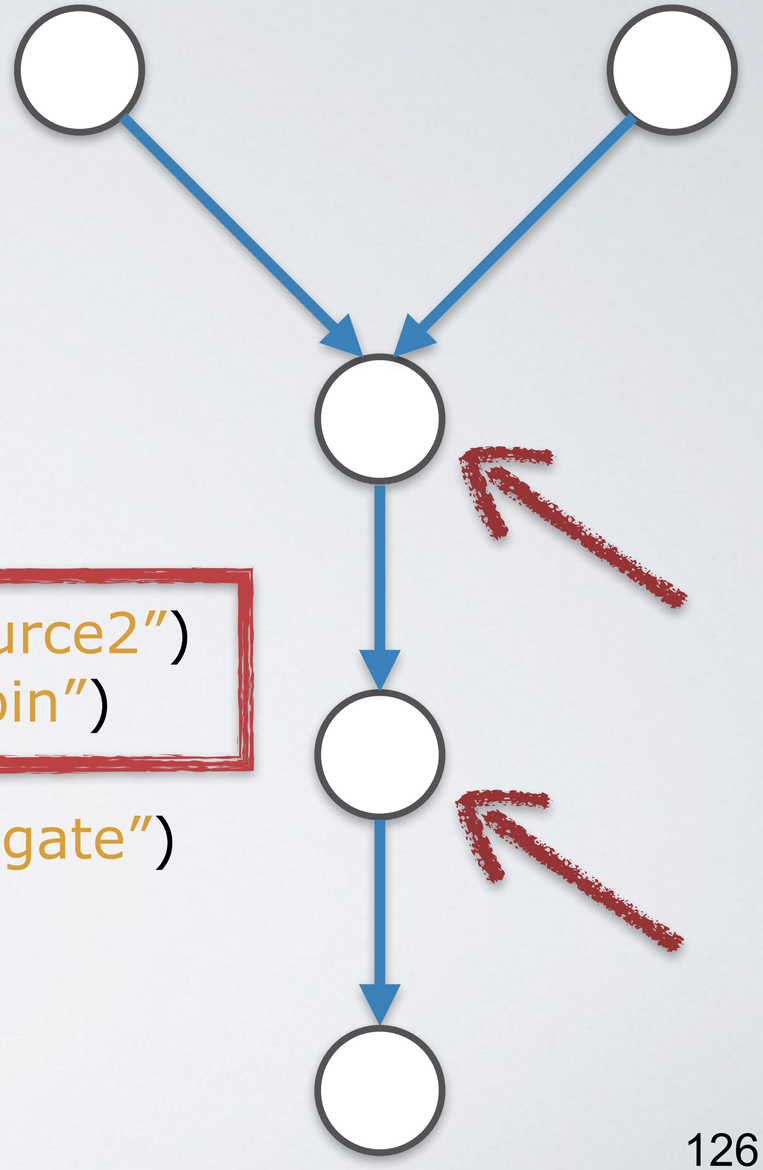
Processor Topology



```
builder.addSource(”Source1”, ”topic1”)
       .addSource(”Source2”, ”topic2”)

       .addProcessor(”Join”, MyJoin:new, ”Source1”, ”Source2”)
       .addProcessor(”Aggregate”, MyAggregate:new, ”Join”)

       .addStateStore(Stores.persistent().build(), ”Aggregate”)

       .addSink(”Sink”, ”topic3”, ”Aggregate”)
```

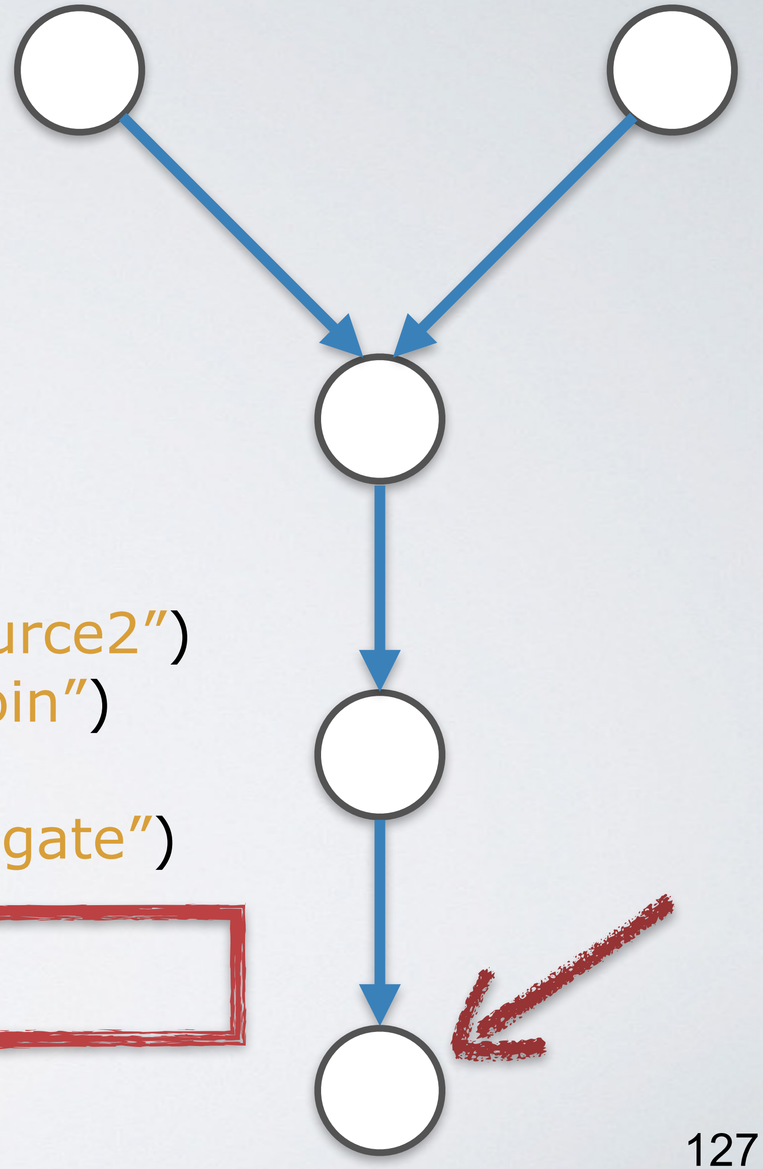
Processor Topology

```
builder.addSource(”Source1”, ”topic1”)
       .addSource(”Source2”, ”topic2”)

       .addProcessor(”Join”, MyJoin:new, ”Source1”, ”Source2”)
       .addProcessor(”Aggregate”, MyAggregate:new, ”Join”)

       .addStateStore(Stores.persistent().build(), ”Aggregate”)

       .addSink(”Sink”, ”topic3”, ”Aggregate”)
```



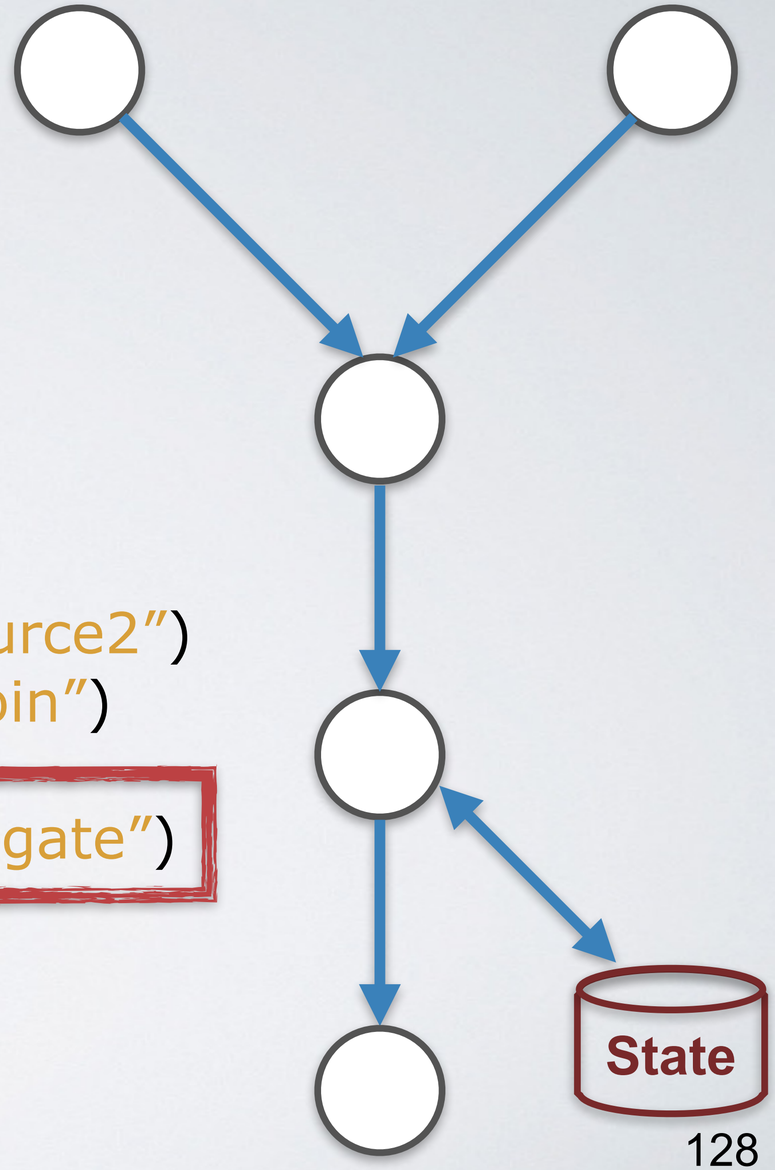
Processor Topology

```
builder.addSource(”Source1”, ”topic1”)
       .addSource(”Source2”, ”topic2”)

       .addProcessor(”Join”, MyJoin:new, ”Source1”, ”Source2”)
       .addProcessor(”Aggregate”, MyAggregate:new, ”Join”)

       .addStateStore(Stores.persistent().build(), ”Aggregate”)

       .addSink(”Sink”, ”topic3”, ”Aggregate”)
```



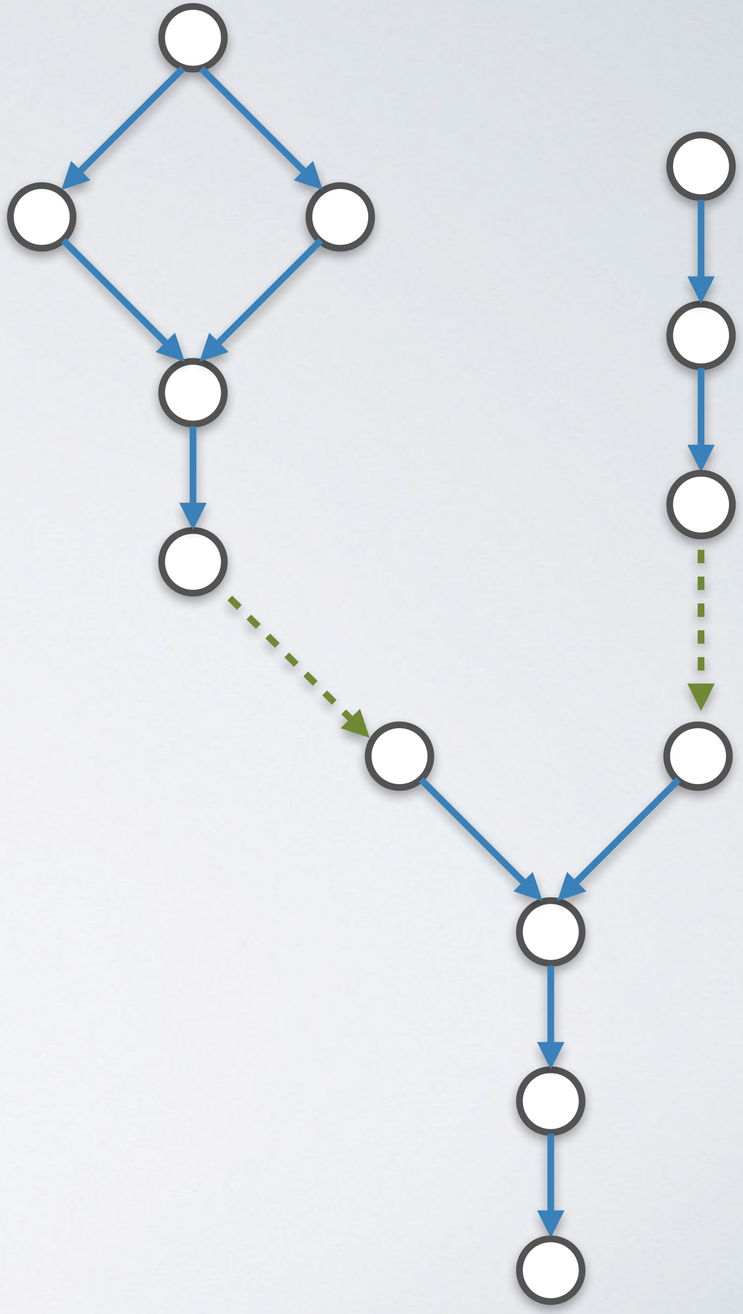
States in Stream Processing

```
builder.addSource(”Source1”, ”topic1”)
       .addSource(”Source2”, ”topic2”)

       .addProcessor(”Join”, MyJoin:new, ”Source1”, ”Source2”)
       .addProcessor(”Aggregate”, MyAggregate:new, ”Join”)

       .addStateStore(Stores.persistent().build(), ”Aggregate”)

       .addSink(”Sink”, ”topic3”, ”Aggregate”)
```



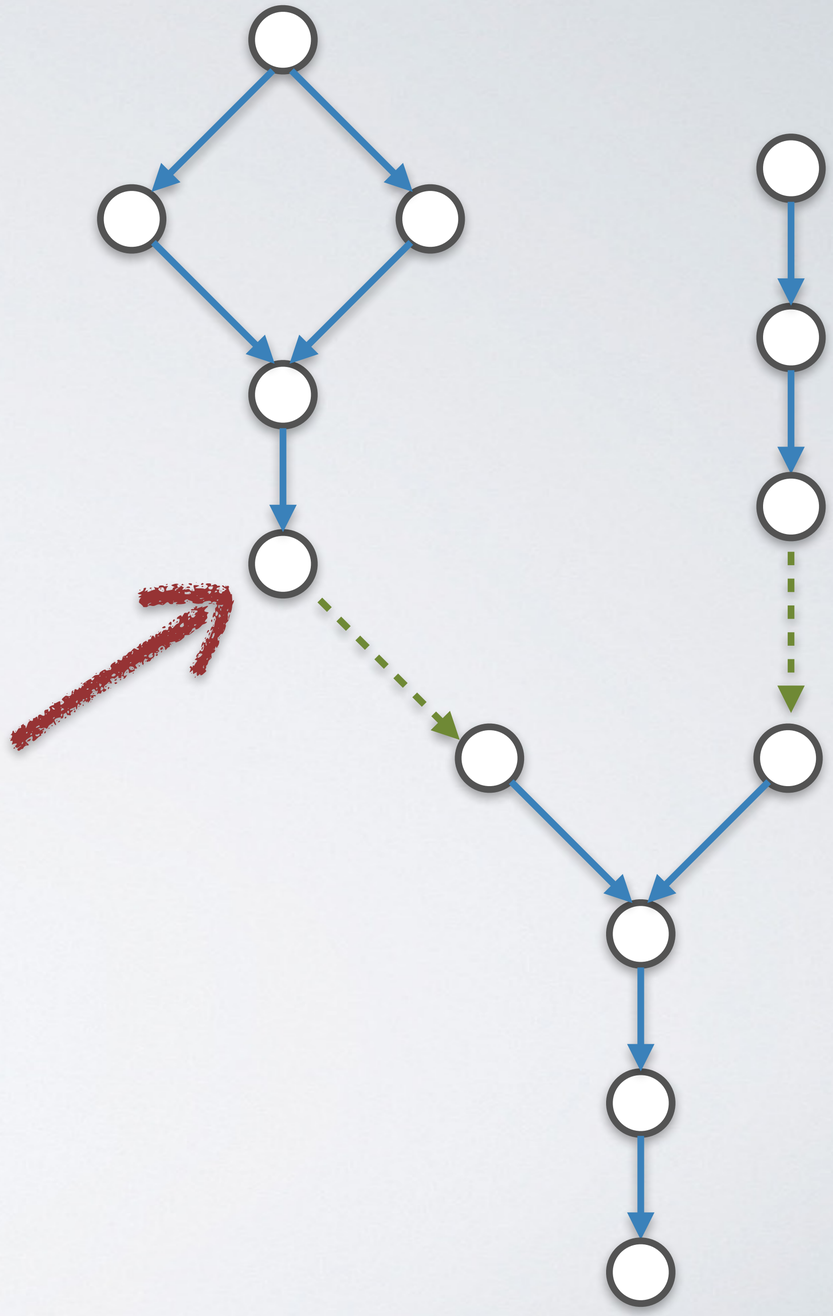
Processor Topology

```
…

sink1.to(”topic1”);

source1 = builder.table(”topic1”);

source2 = sink1.through(”topic2”);

…
```



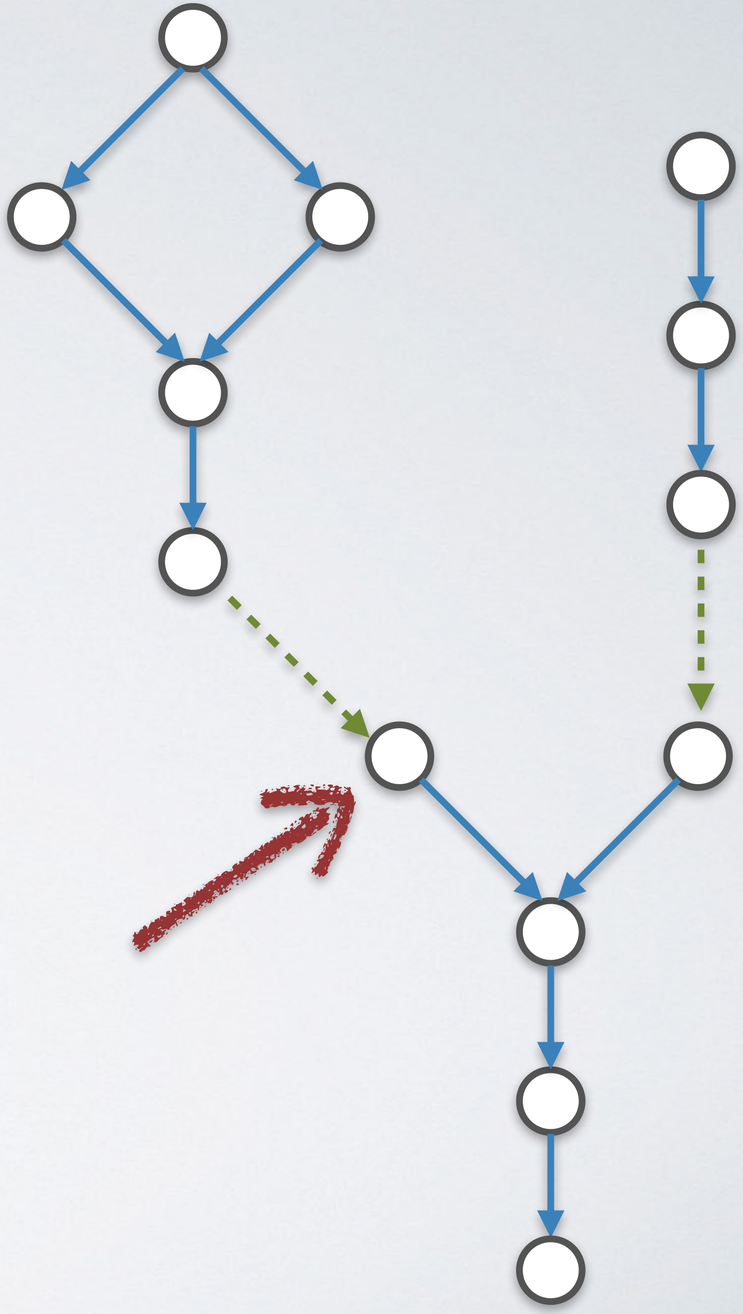
Processor Topology

```
...

sink1.to(”topic1”);

source1 = builder.table(”topic1”);

source2 = sink1.through(”topic2”);

...
```



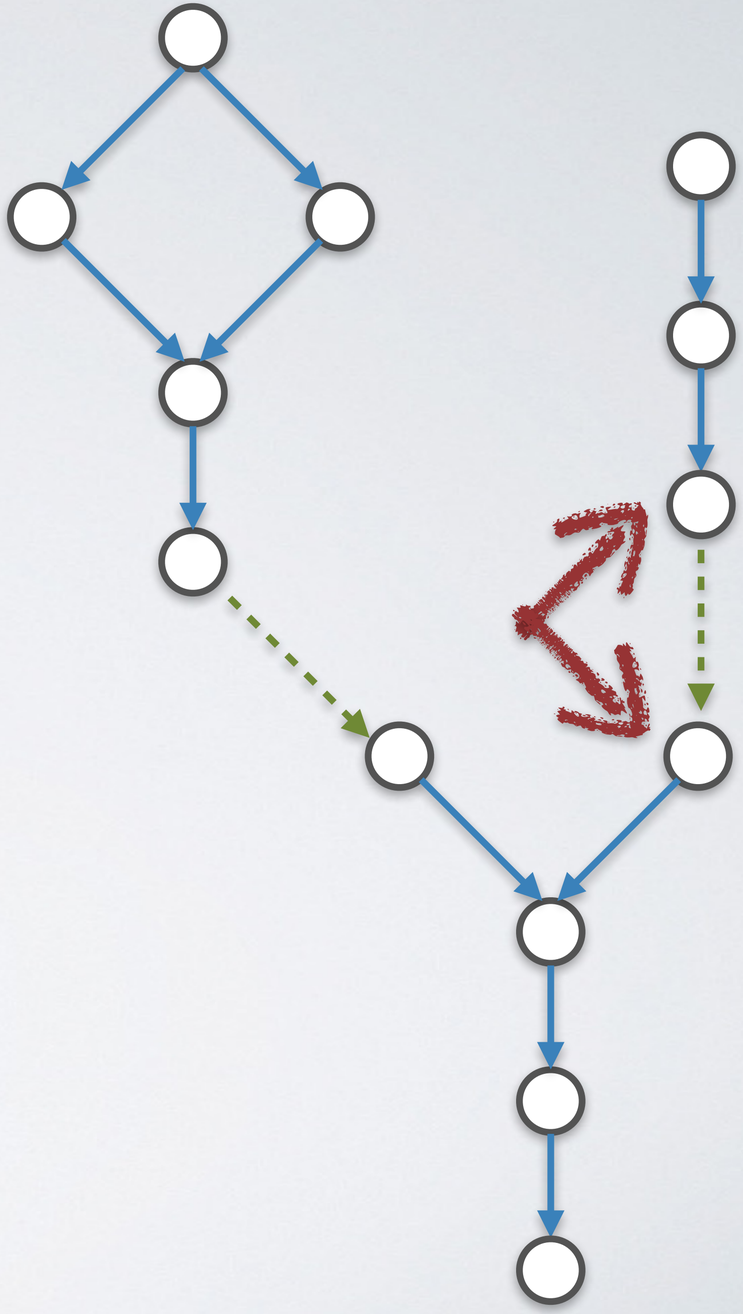
Processor Topology

```
…

sink1.to(”topic1”);

source1 = builder.table(”topic1”);

source2 = sink1.through(”topic2”);

…
```



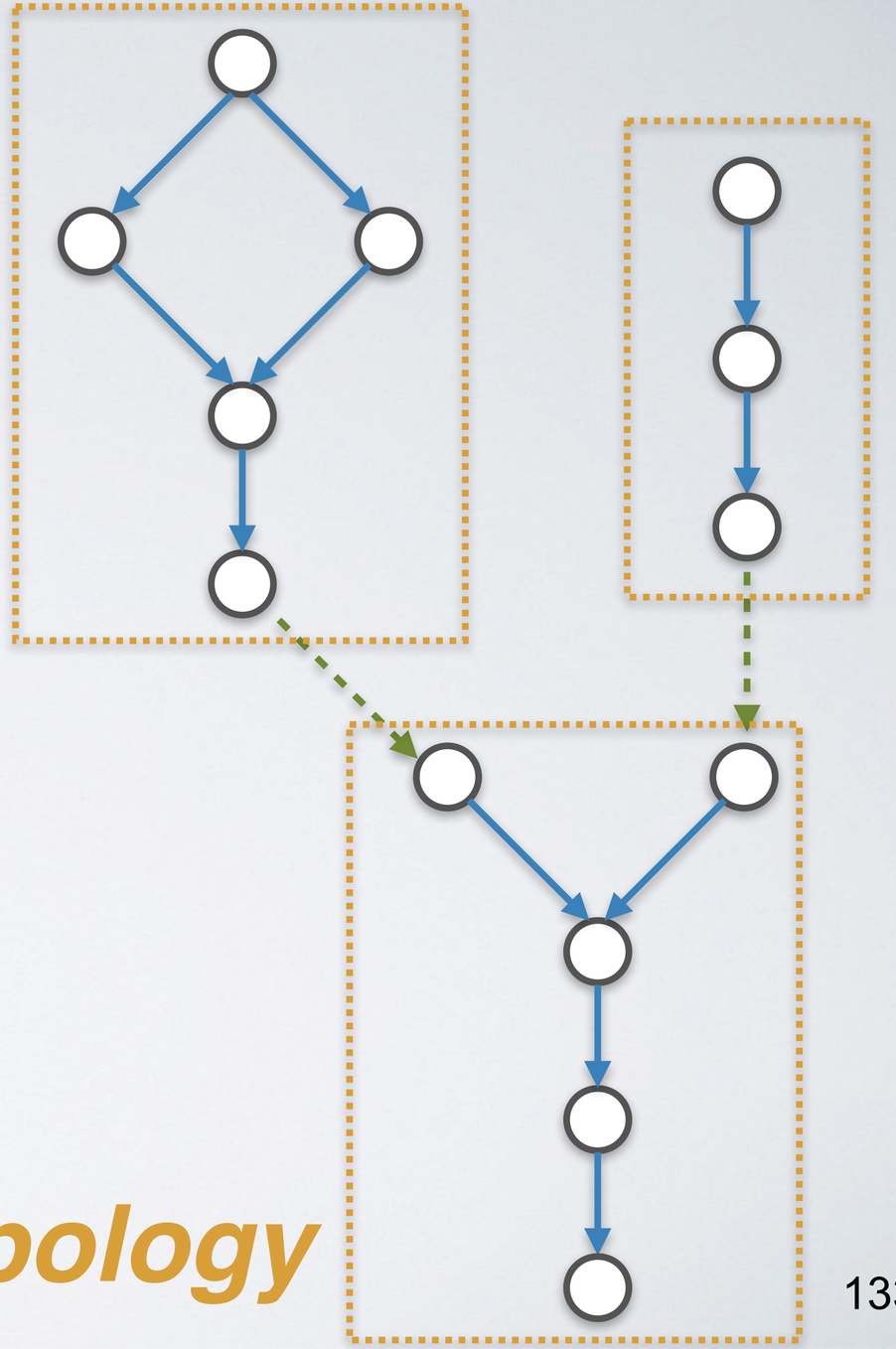
Processor Topology

```
...

sink1.to(”topic1”);

source1 = builder.table(”topic1”);

source2 = sink1.through(”topic2”);

...
```



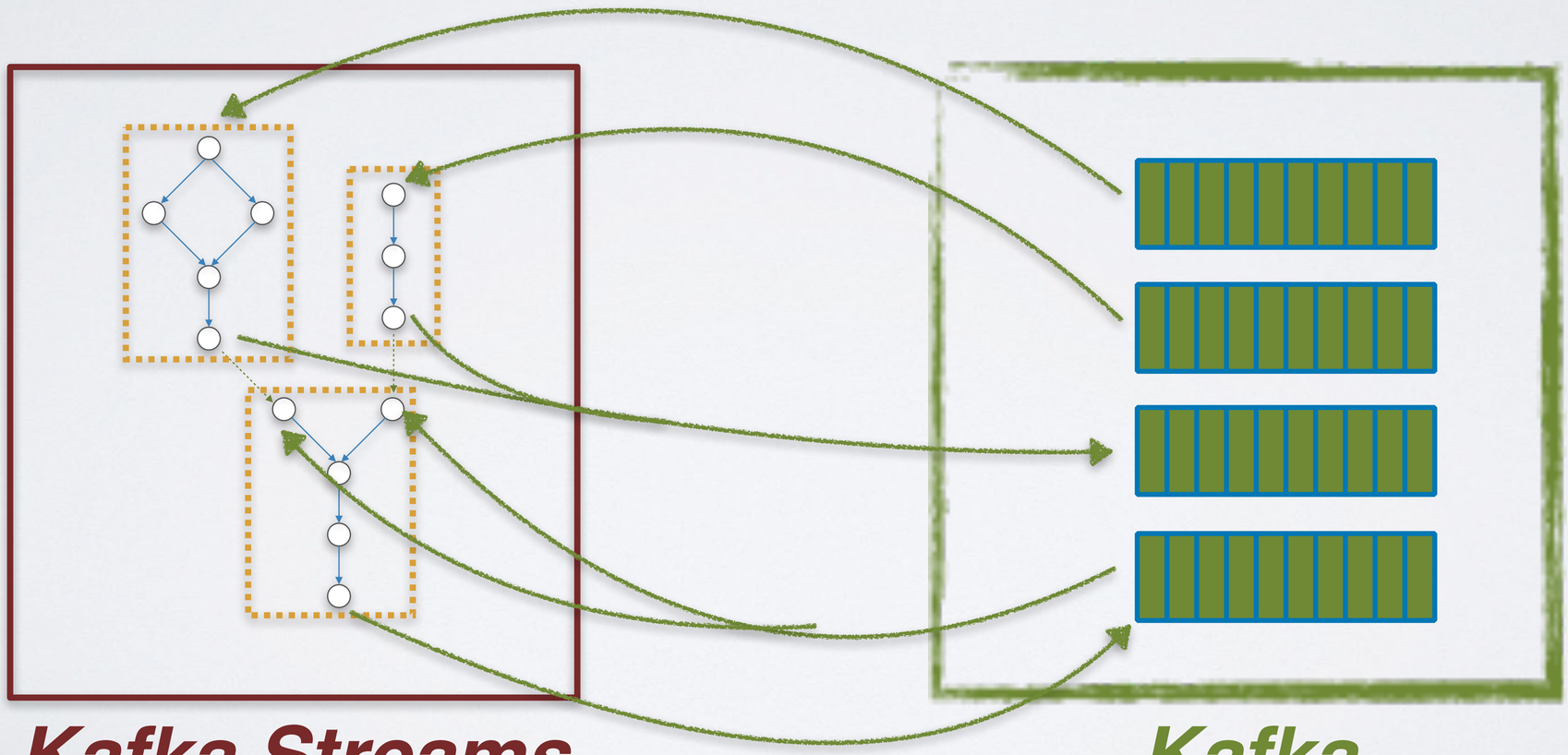
Processor Topology

```
…

sink1.to(”topic1”);

source1 = builder.table(”topic1”);

source2 = sink1.through(”topic2”);

…
```

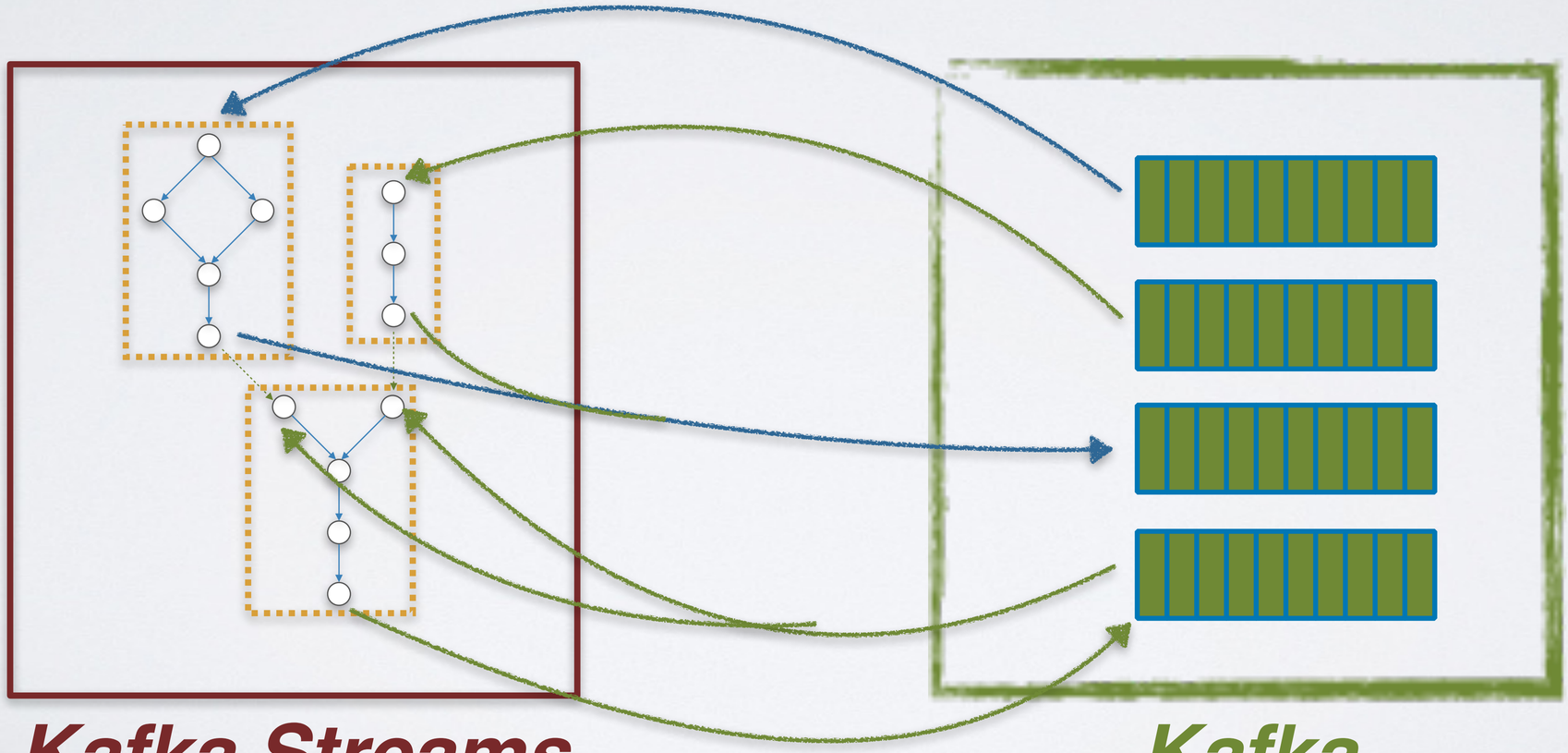


Sub-Topology

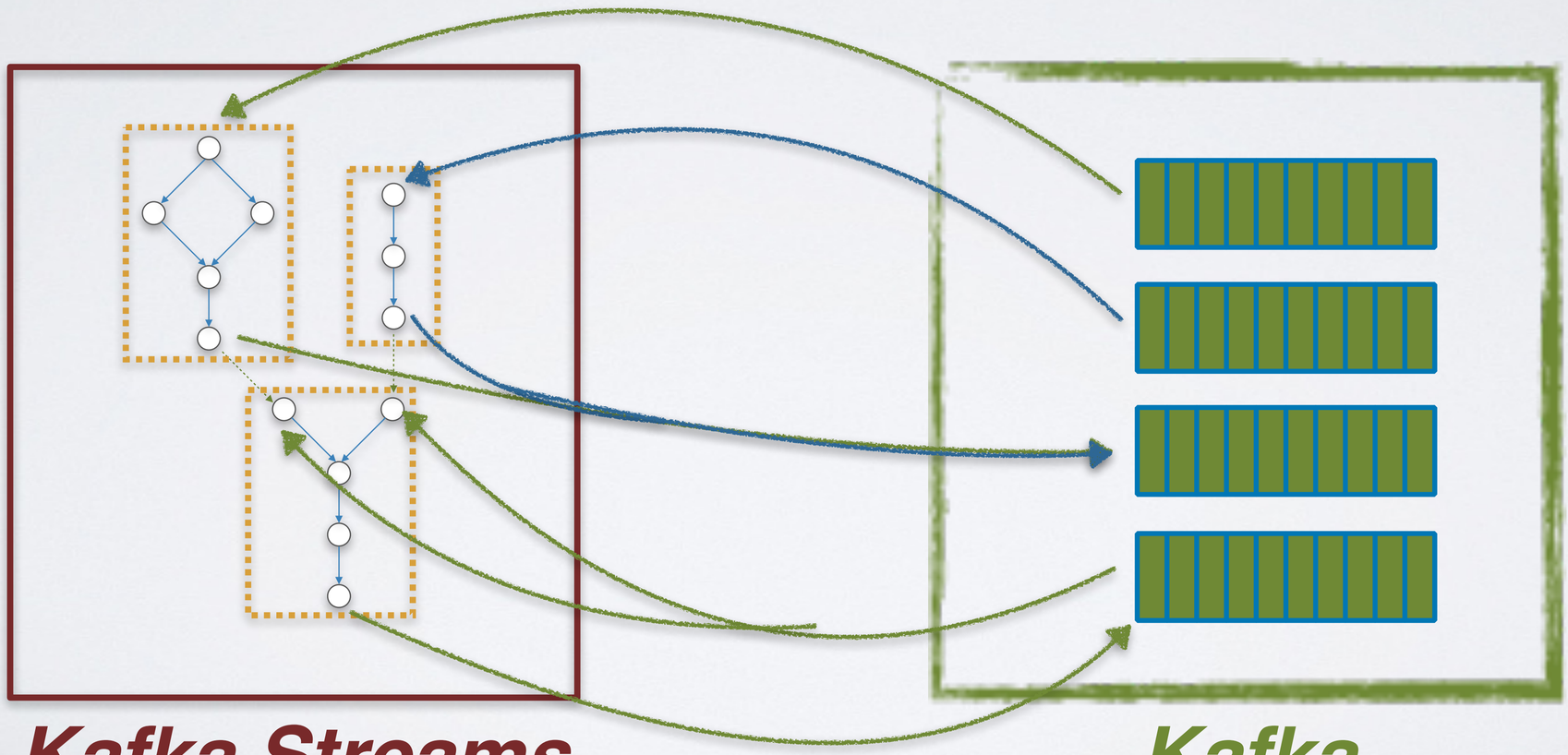
Processor Topology



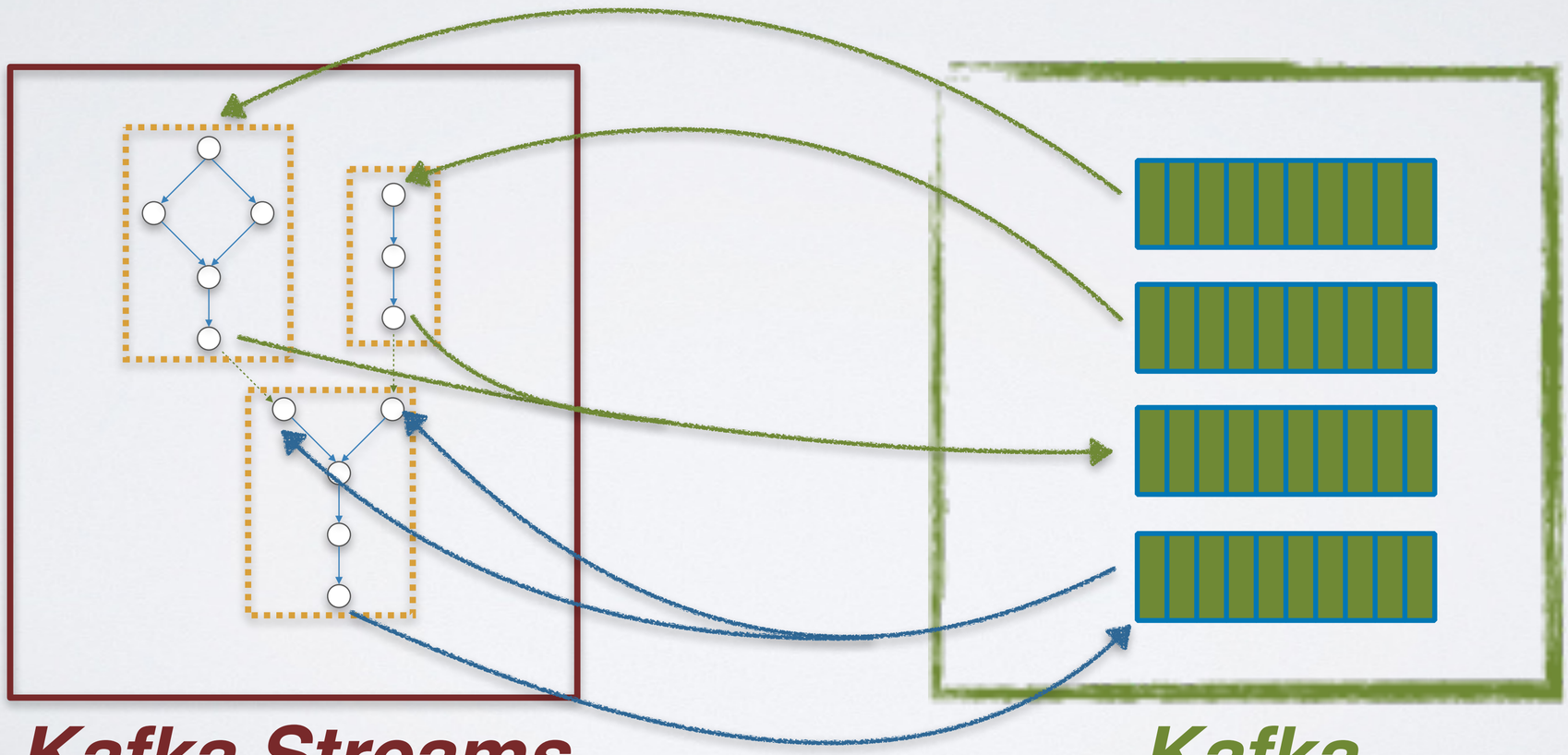
Processor Topology



Processor Topology

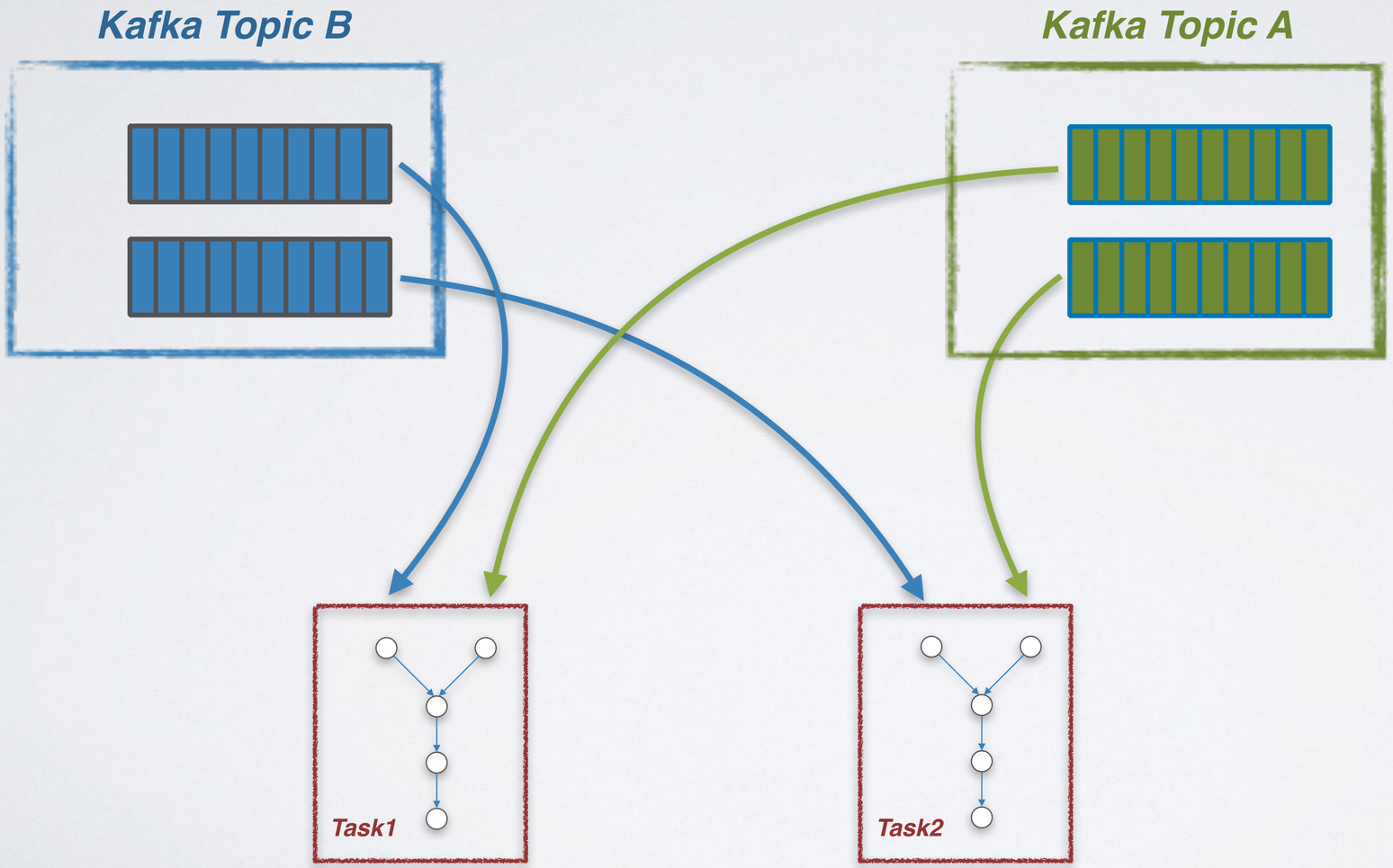


Processor Topology

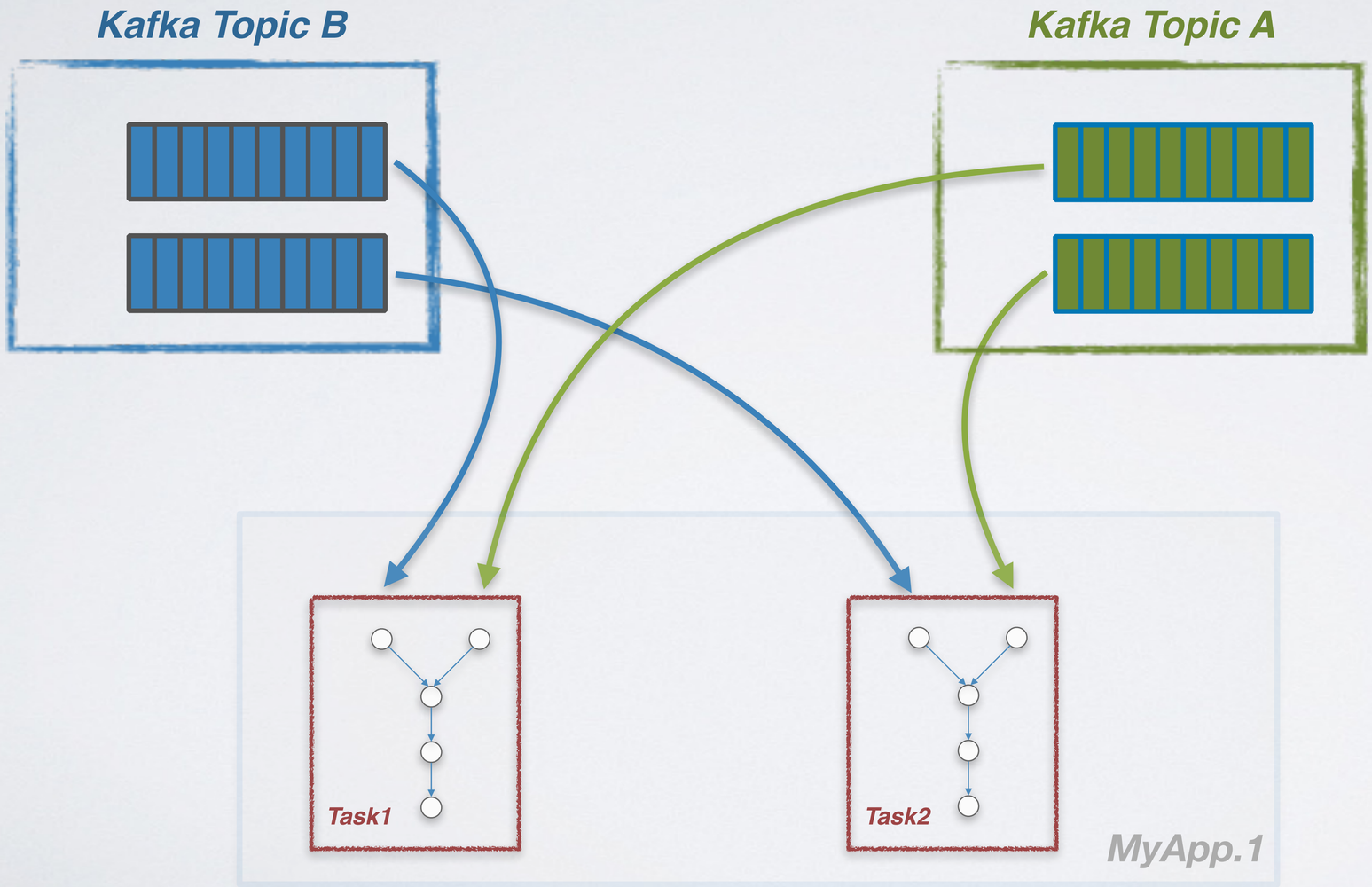


Stream Partitions and Tasks

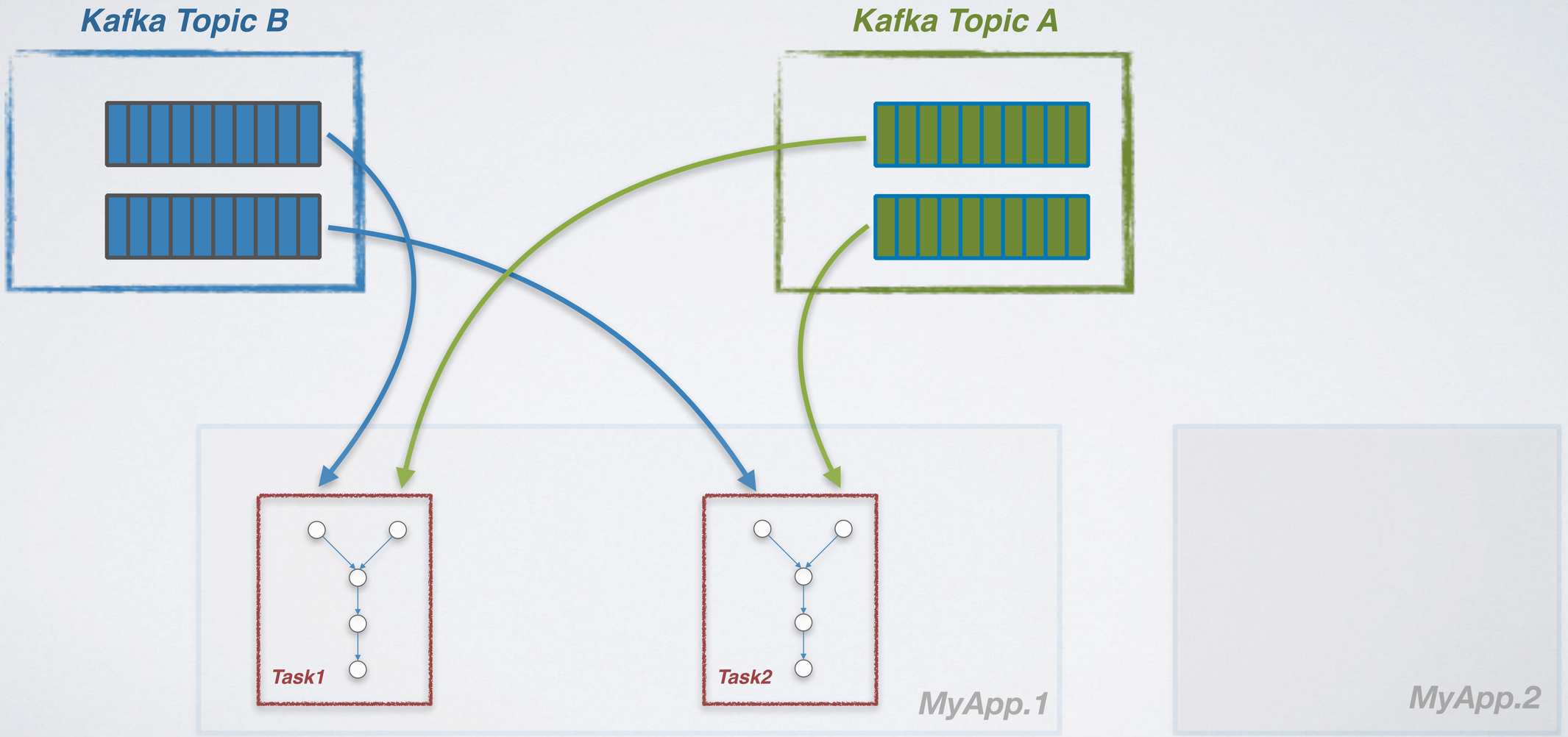
Stream Partitions and Tasks



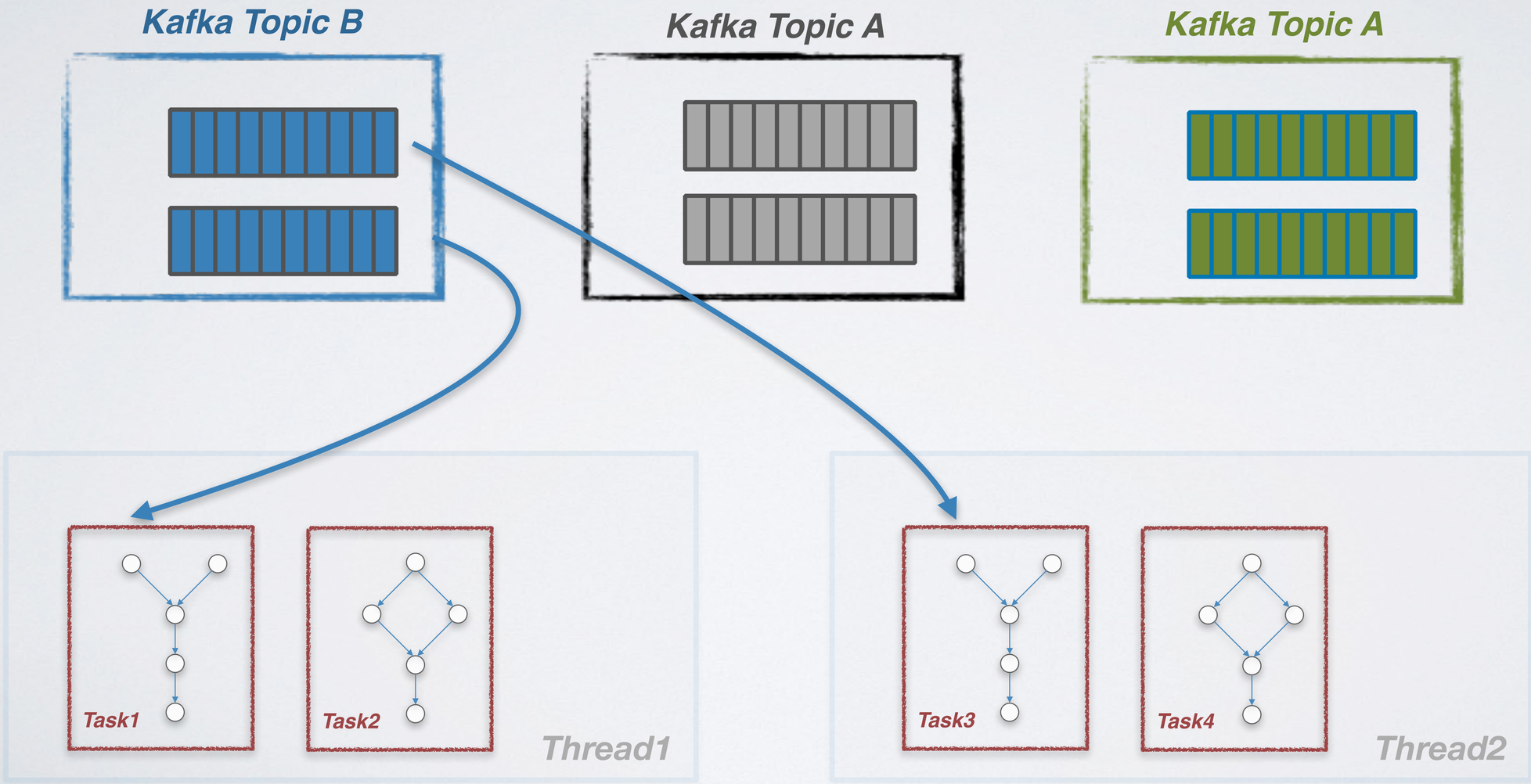
Stream Threads



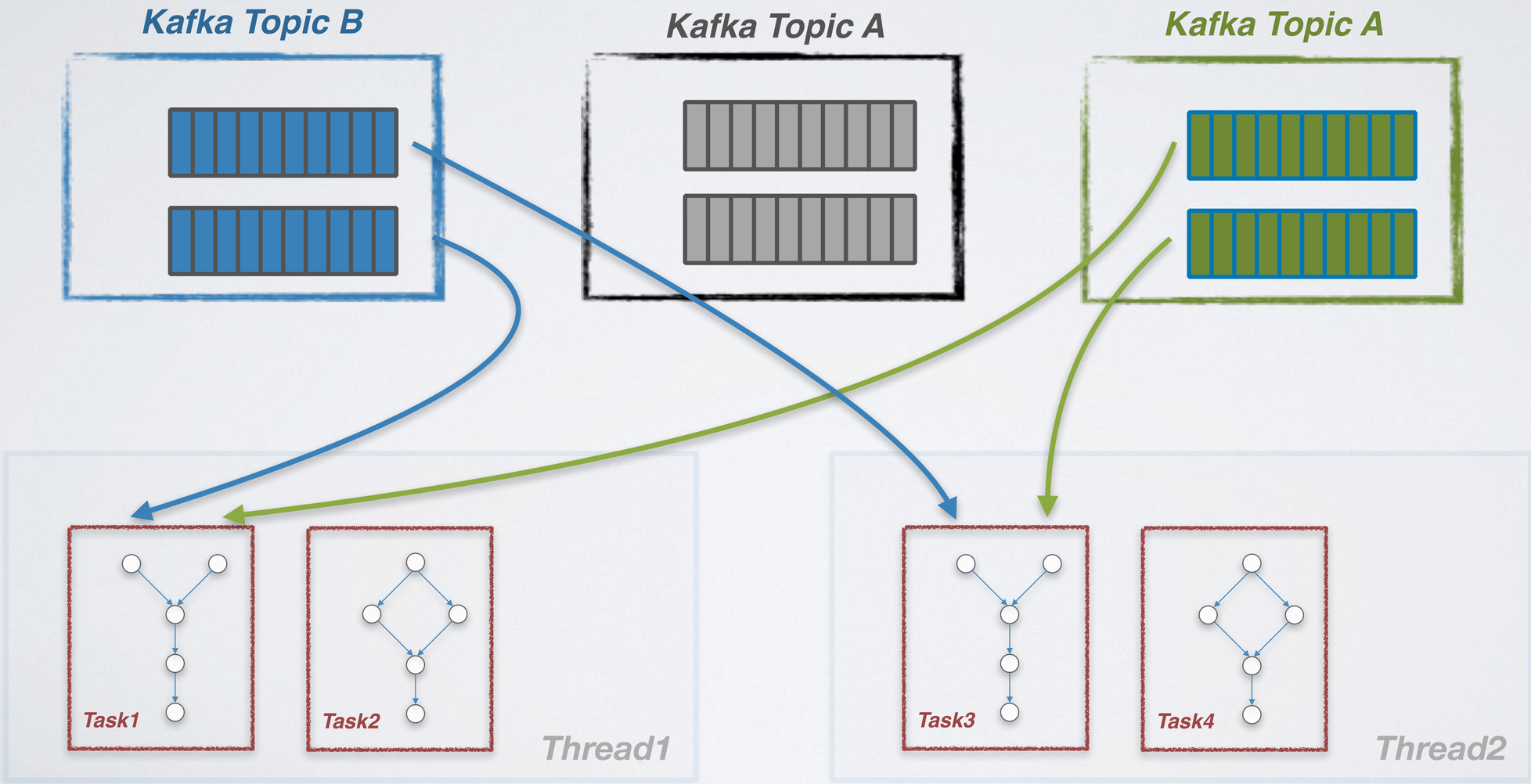
Stream Threads



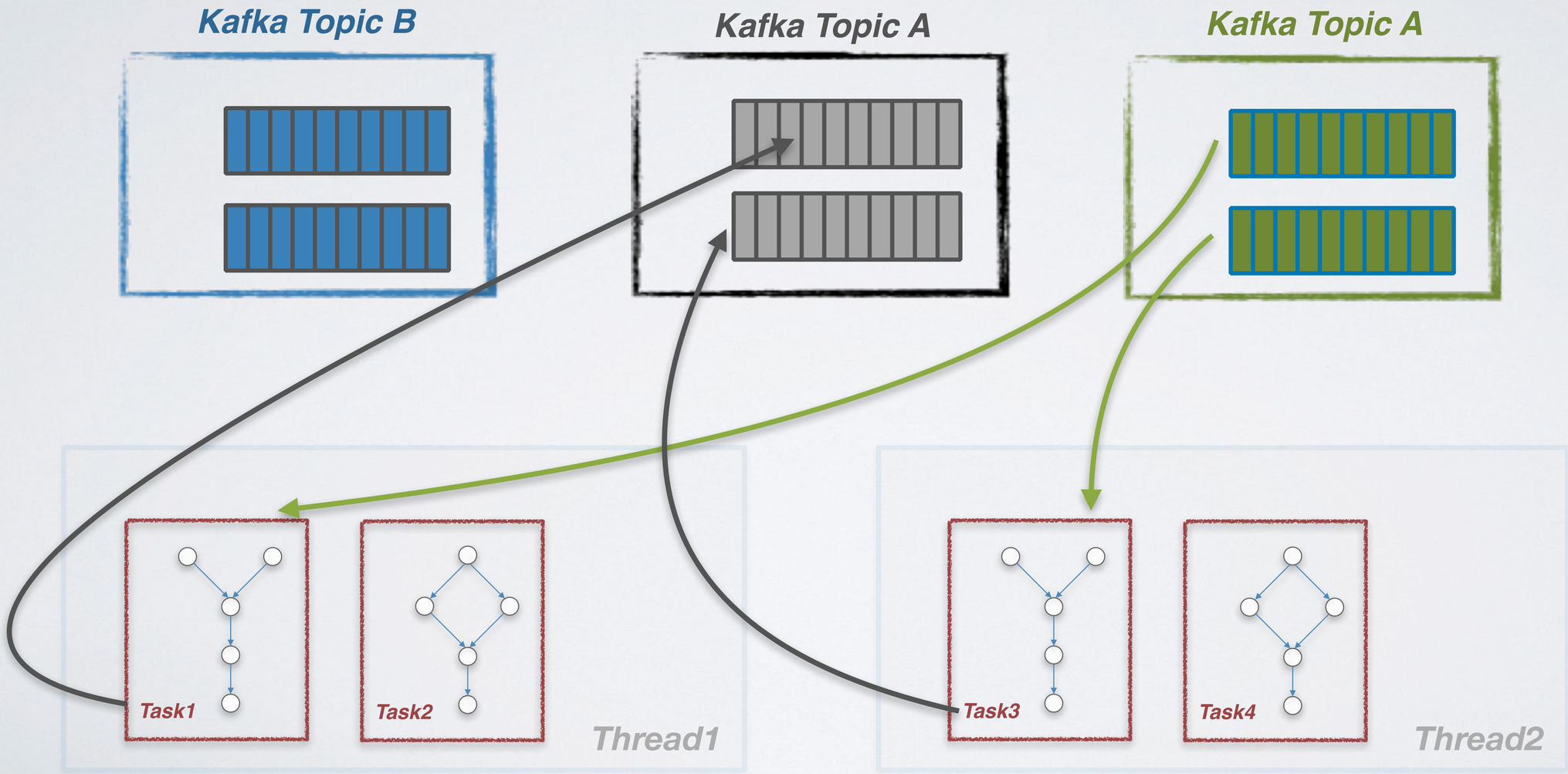
Stream Threads



Stream Threads



Stream Threads



Stream Threads