confluent

Apache Kafka™ Fundamentals:

# What Every Software Engineer Should Know about Streams and Tables in Kafka

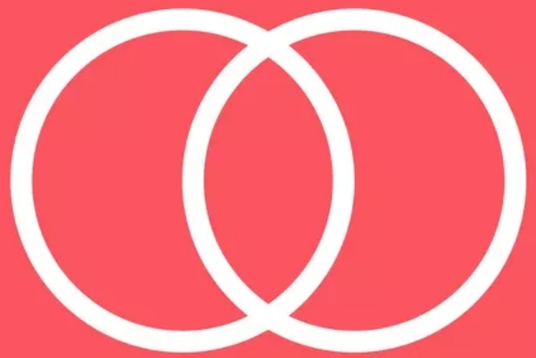
**Michael G. Noll**
Senior Technologist, Office of the CTO
@miguno

Kafka Meetup Zurich, Switzerland, Feb 2020

# Event Streaming

Streams  Tables

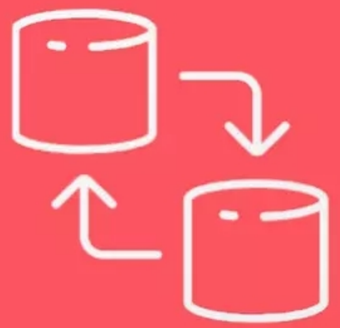
# Event Streaming

Streams  Tables

# Events, Streams, Tables

## A Primer

# An Event Streaming Platform gives you three key functionalities



Publish & Subscribe to Events

Store Events

Process & Analyze Events

# An Event
# records the fact that something happened

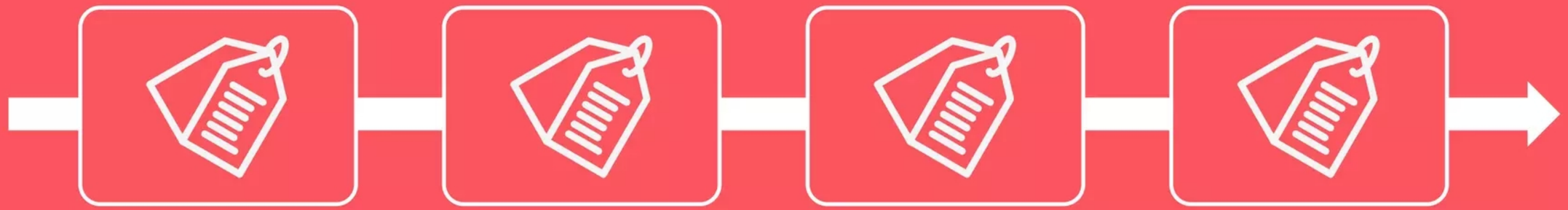
A good
was sold

An invoice
was issued to
Michael

Alice paid
$200 to Bob
on Feb 01, 2020
at 8:31am

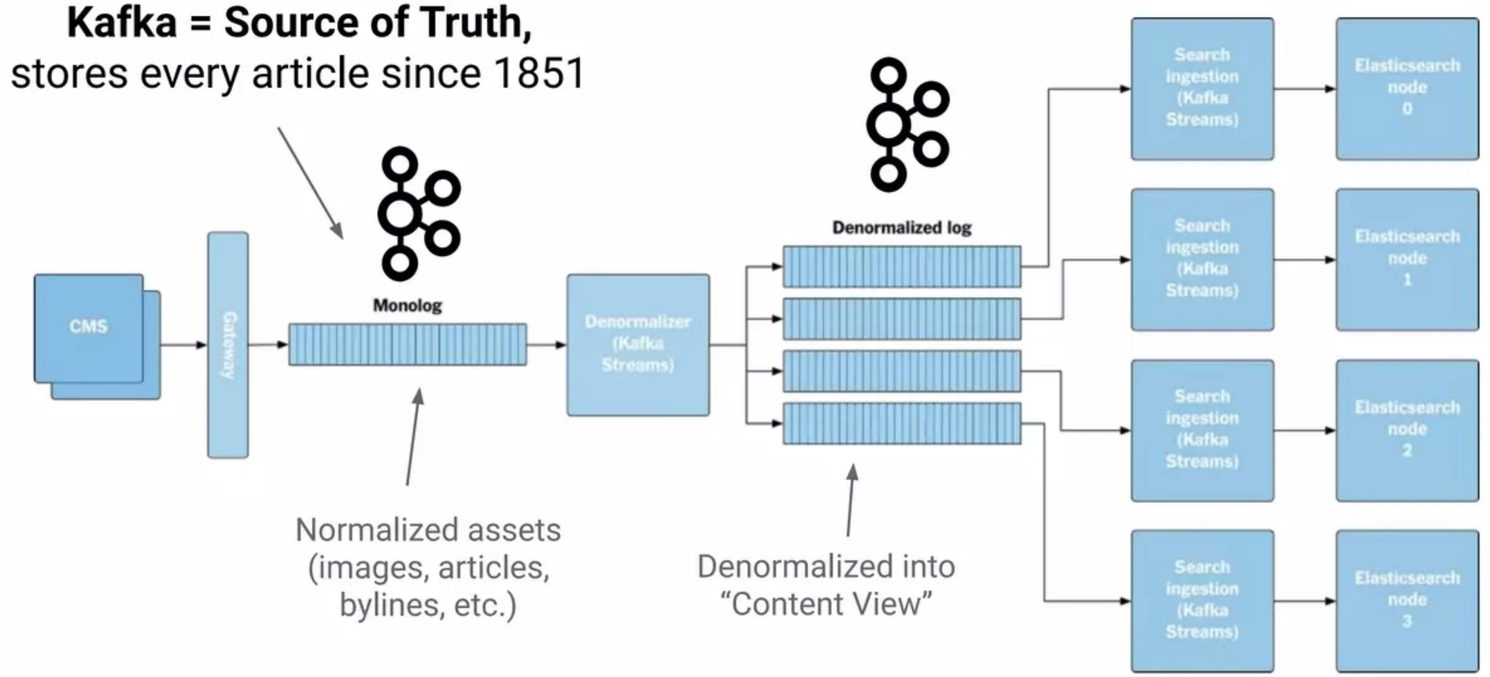
A new customer
registered

# A Stream
# records history as a sequence of Events



# Streams record history, even hundreds of years

The New York Times




https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/

# Streams record history

"The ledger of sales."

# Tables represent state

"The sales totals."

# Streams record history

1. e4 e5
2. Nf3 Nc6
3. Bc4 Bc5
4. d3 Nf6
5. Nbd2

“The sequence of moves.”

# Tables represent state

“The state of the board.”

**Streams = INSERT only**
**Immutable, append-only**

**Tables = INSERT, UPDATE, DELETE**
**Mutable, row key (event.key) identifies which row**

# The key to mutability is … the event.key!

| | Stream | Table |
|---|---|---|
| Has unique key constraint? | No | Yes |
| First event with key **`bob`** arrives | INSERT | INSERT |
| Another event with key **`bob`** arrives | INSERT | UPDATE |
| Event with key **`bob`** and value **`null`** arrives | INSERT | DELETE |
| Event with key **`null`** arrives | INSERT | *<ignored>* |

**RDBMS analogy: A Stream is ~ a Table that has no unique key and is append-only.**

# Creating a table from a stream or topic



```
KTable<String, String> table
  = builder.table("input-topic",
      Consumed.with(Serdes.String(),
                    Serdes.String()));
```


```
CREATE TABLE myTable
  (username VARCHAR, location VARCHAR)
  WITH (KAFKA_TOPIC='input-topic',
        VALUE_FORMAT='...', ...)
```

# Stream
## (facts)


| | |
|---|---|
| alice | Paris |
| bob | Sydney |
| alice | Rome |
| bob | Lima |
| alice | Berlin |

| | |
|---|---|
| bob | Lima |

| | |
|---|---|
| alice | Berlin |

# Table
## (dims)

| | |
|---|---|
| alice | Berlin |
| bob | Lima |

# Streams record history

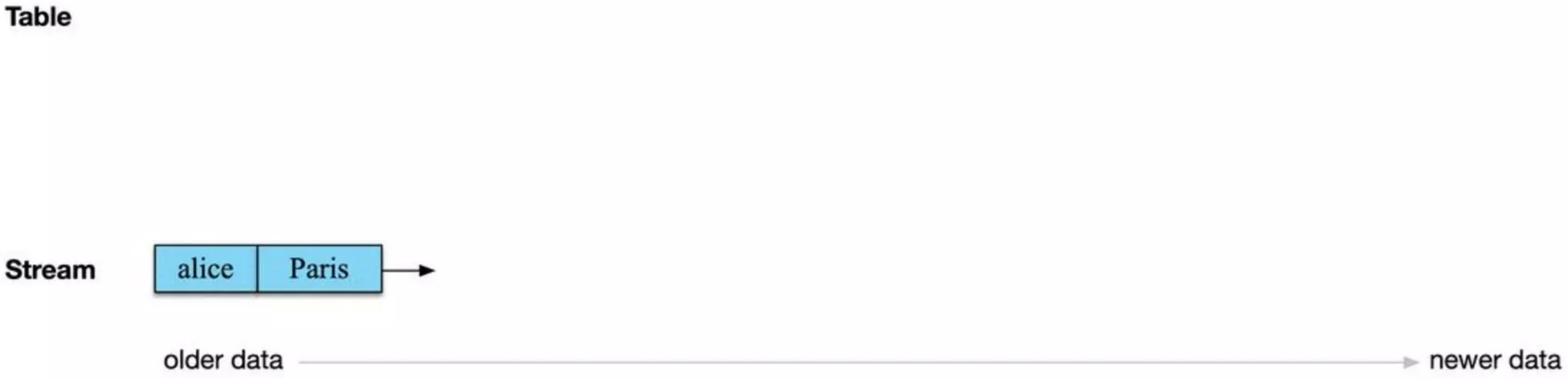
# Tables represent state

aggregation
like SUM( ), COUNT( )


table changes

*See the academic paper Streams and Tables: Two Sides of the Same Coin, M. Sax et al., BIRTE '18

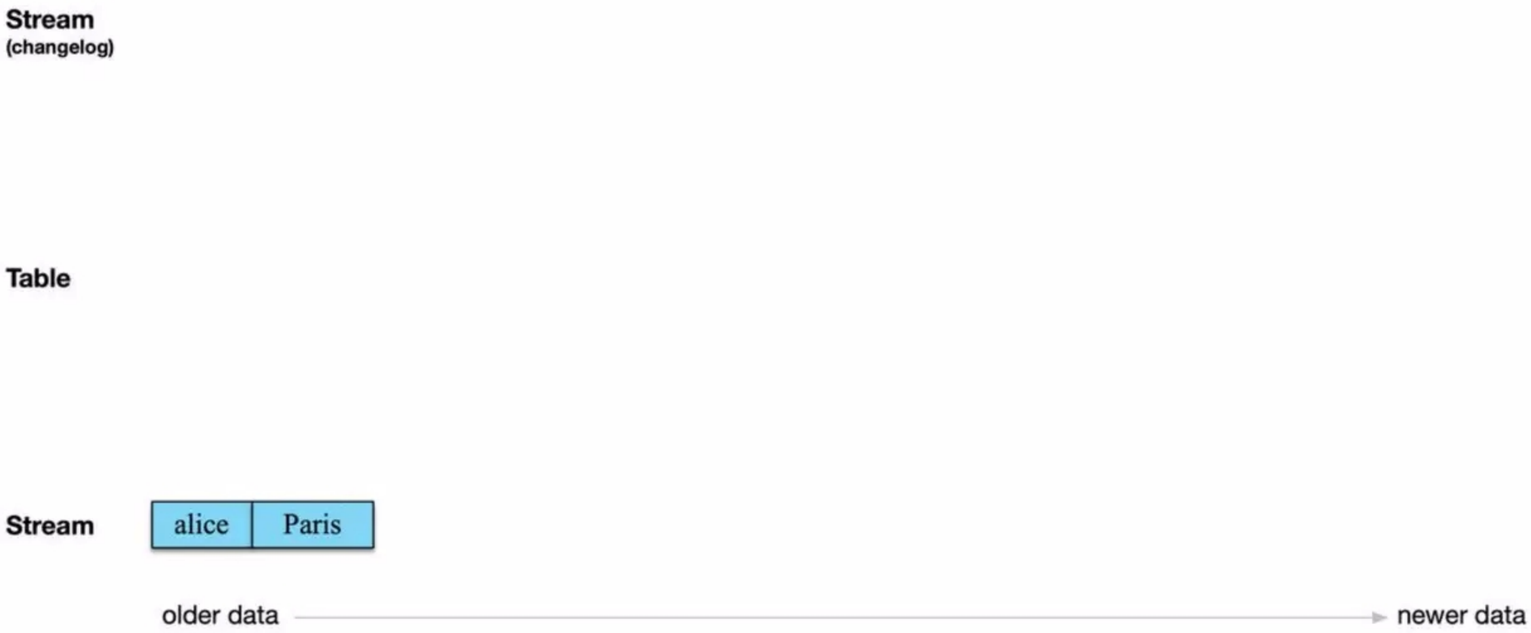
# Aggregating a stream (COUNT example)




```
KTable<String, Long> locationsPerUser
  = locationUpdatesStream
      .groupBy((k, v) -> v.username)
      .count();
```

```
CREATE TABLE locationsPerUser AS
    SELECT username, COUNT(*)
    FROM locationUpdatesStream
    GROUP BY username
    EMIT CHANGES;
```

# Aggregating a stream (COUNT example)




```
KTable<String, Long> locationsPerUser
  = locationUpdatesStream
      .groupBy((k, v) -> v.username)
      .count();
```

```
CREATE TABLE locationsPerUser AS
    SELECT username, COUNT(*)
    FROM locationUpdatesStream
    GROUP BY username
    EMIT CHANGES;
```

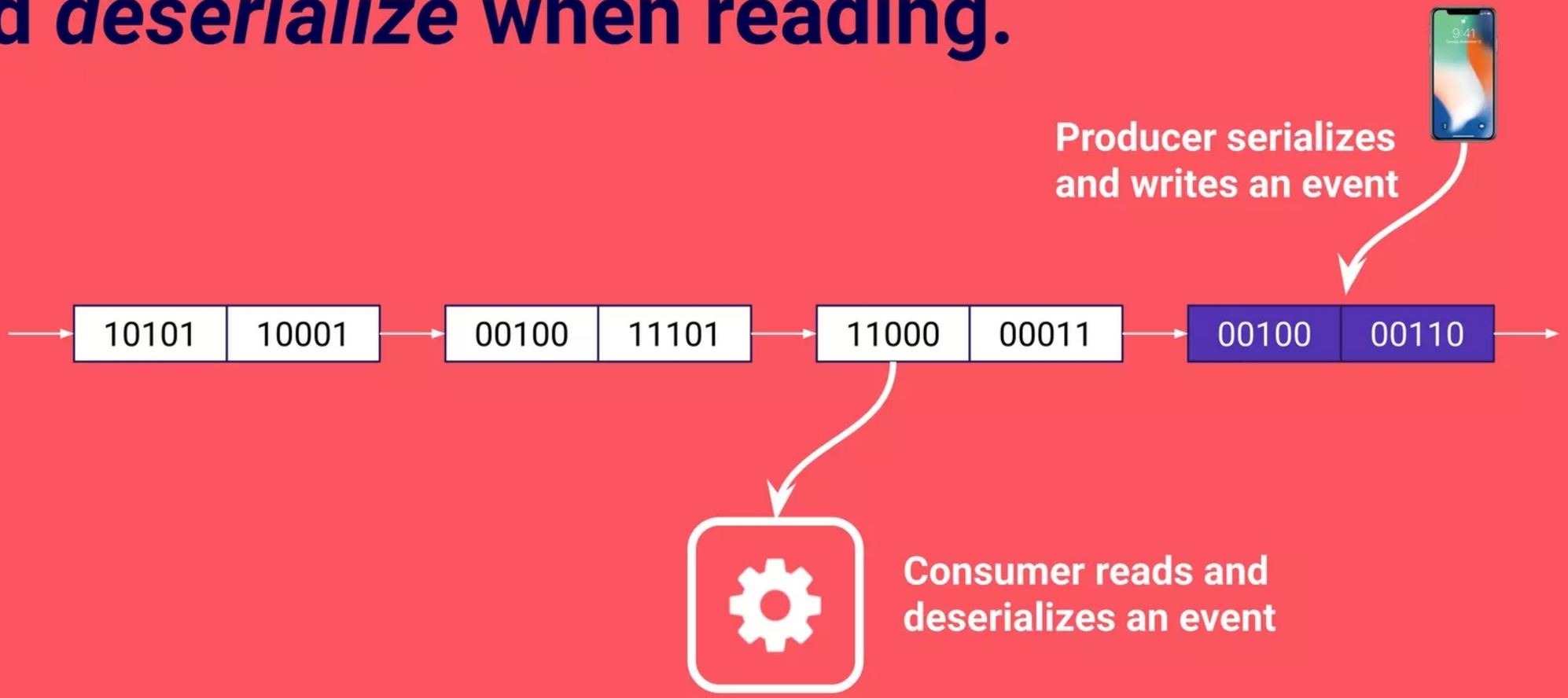
# Topics, Partitions, and Other Storage Fundamentals

# A Topic
# is where events are durably stored, like a file in a distributed file system


An unbounded sequence of serialized events, where each event is represented as an encoded key-value pair* or “Kafka message”.

# Storage formats

## Clients *serialize* events to bytes when writing, and *deserialize* when reading.

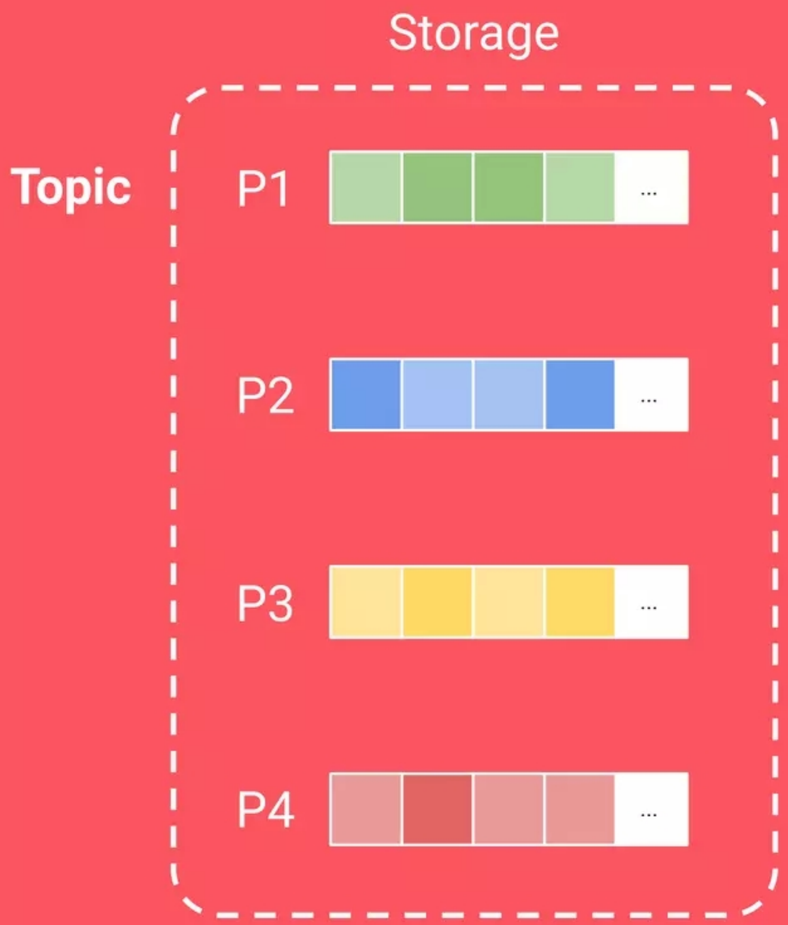


# Storage formats

## Brokers take no part in this. All they see is bytes. Being this “dumb” is actually really smart!

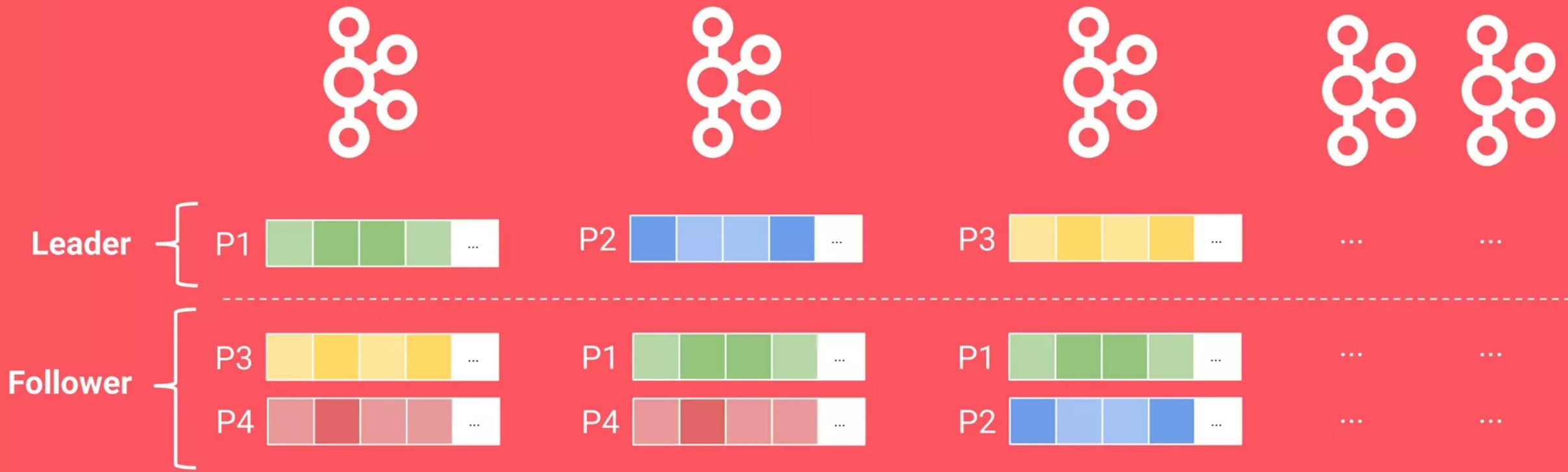
# Storage is partitioned

## And ‘partitions’ are the most important concept in Kafka



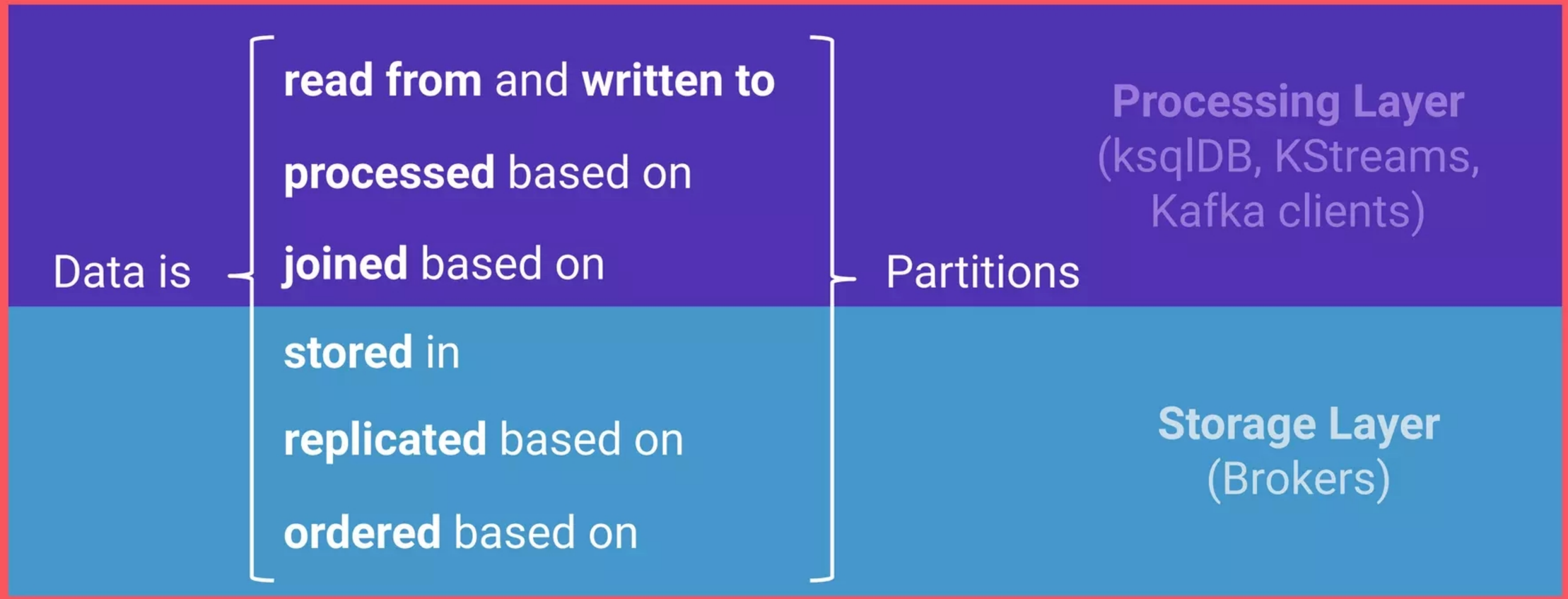
# Partitions are spread across Kafka brokers

## A broker can be leader and/or follower of 1+ partitions

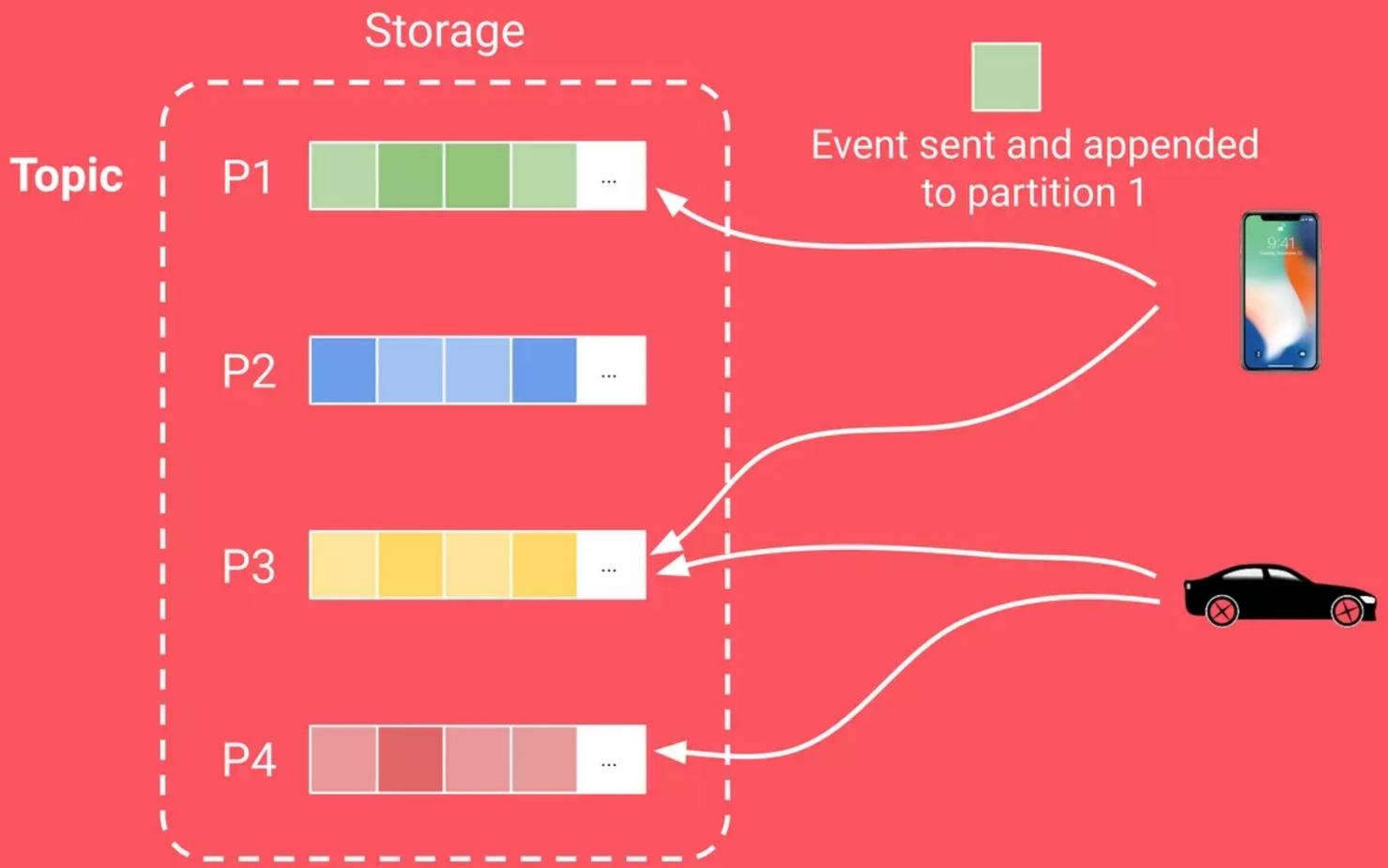


# Partitions play a central role in Kafka

Topics are partitioned. Partitions enable **scalability**, **elasticity**, **fault tolerance**.



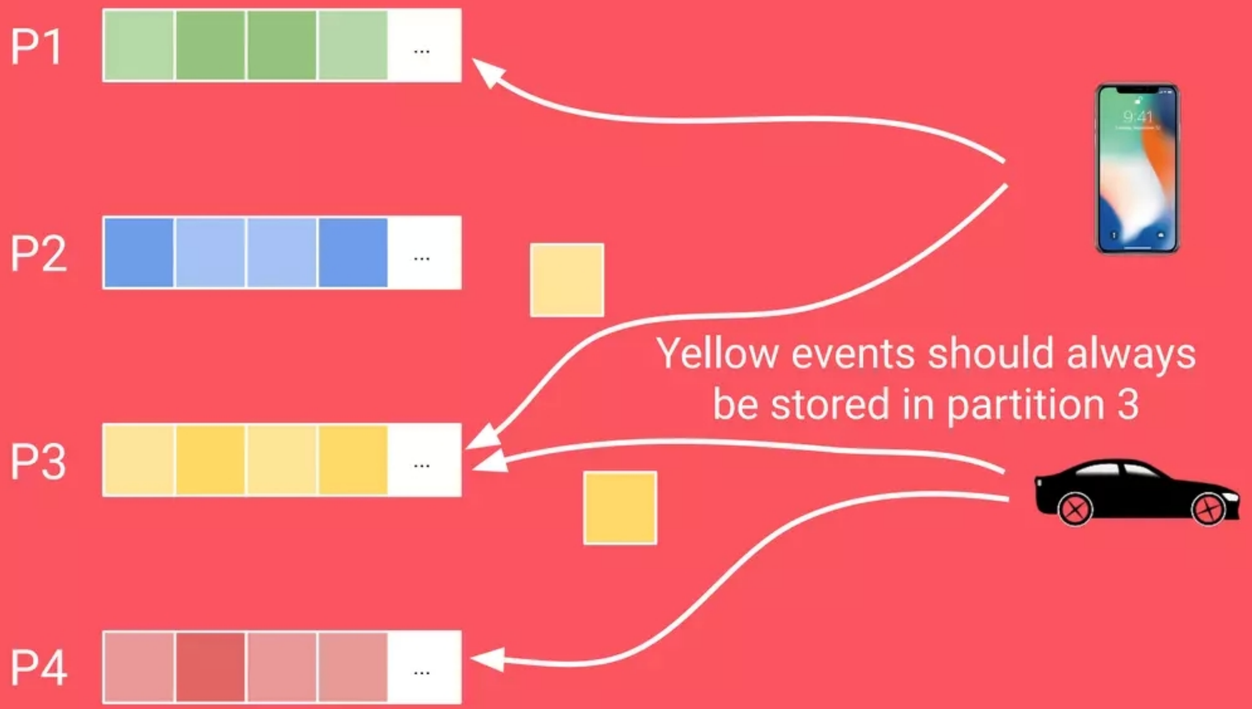
# Event producers determine event partitioning through a partitioning function *f*(event.key, event.value)*



# Events with same key should be in same partition

## to ensure proper ordering of related events

## to ensure processing by Consumers returns expected results



# Top causes for same key in *different* partitions

## 1. Topic configuration

You increased the number of partitions in a topic

## 2. Producer configuration

A producer uses a custom partitioner

## → Be careful in this situation!

# Processing Fundamentals with ksqlDB and Kafka Streams

## From Topics to Streams and Tables

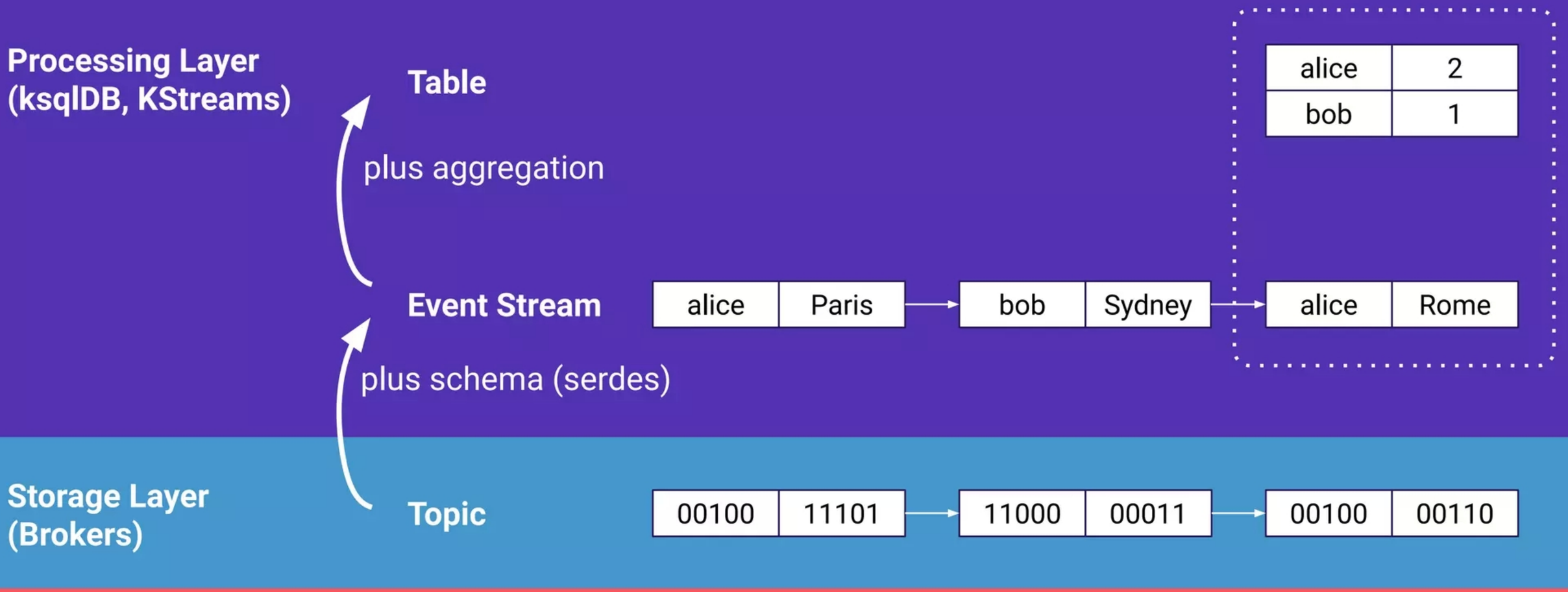
# Topics

**live in the Kafka *storage* layer, ‘filesystem’ (Brokers)**

# Streams and Tables

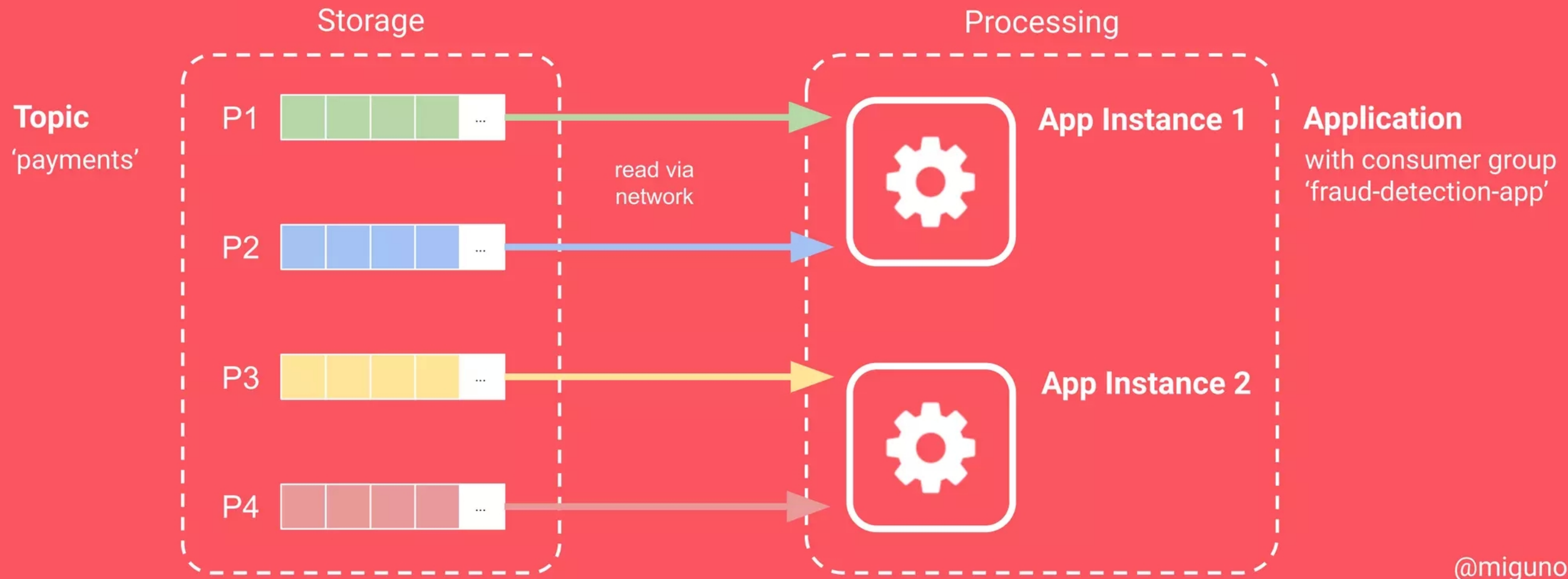
**live in the Kafka *processing* layer (ksqlDB, KStreams)**

# Topics vs. Streams and Tables



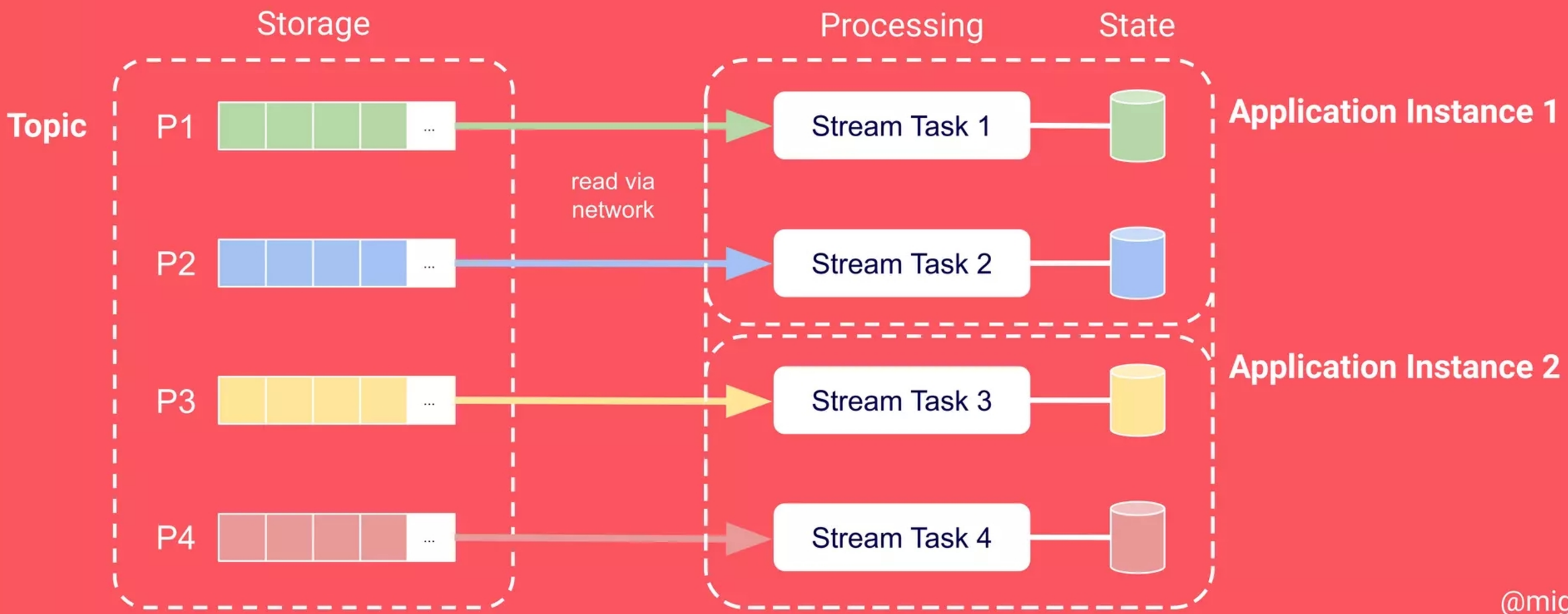
# Kafka Processing

## Data is processed per-partition

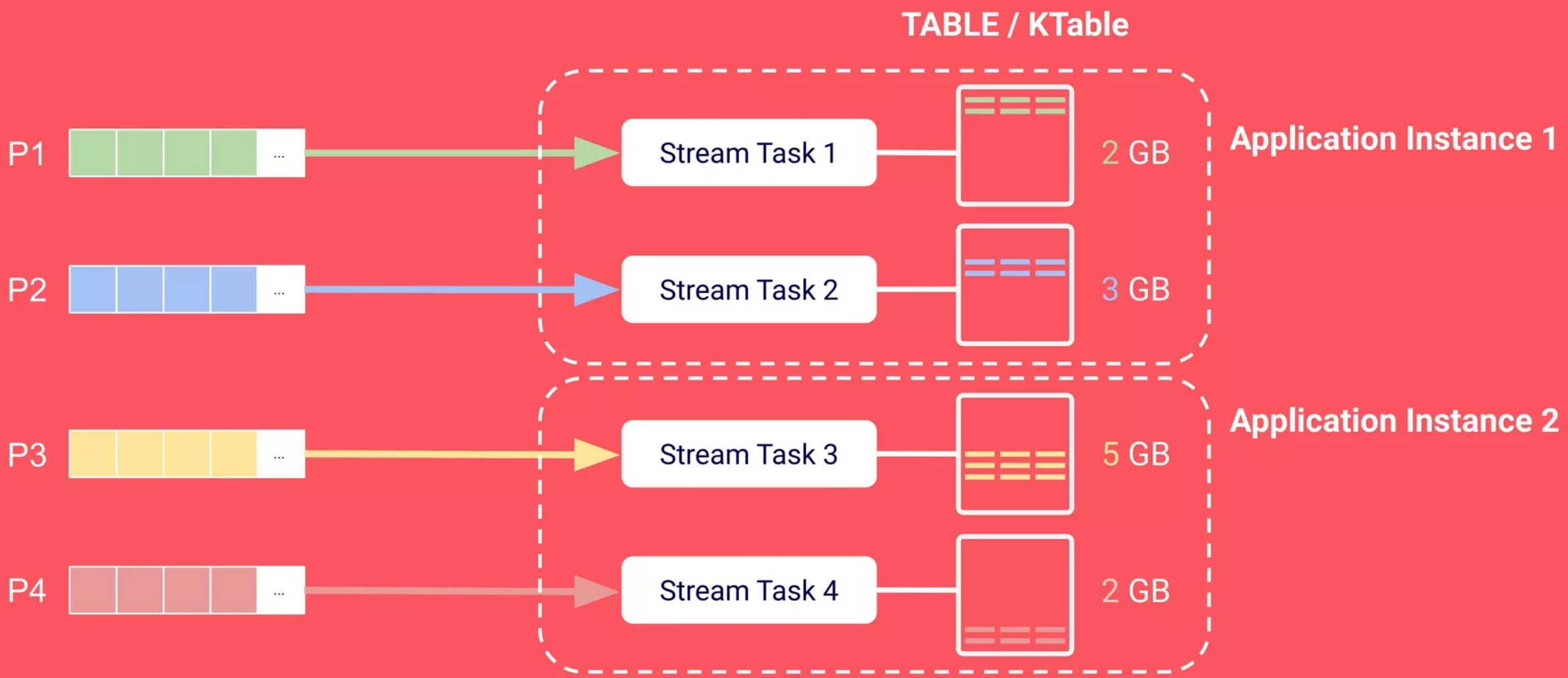


# Kafka Processing

## Stream tasks are the unit of parallelism

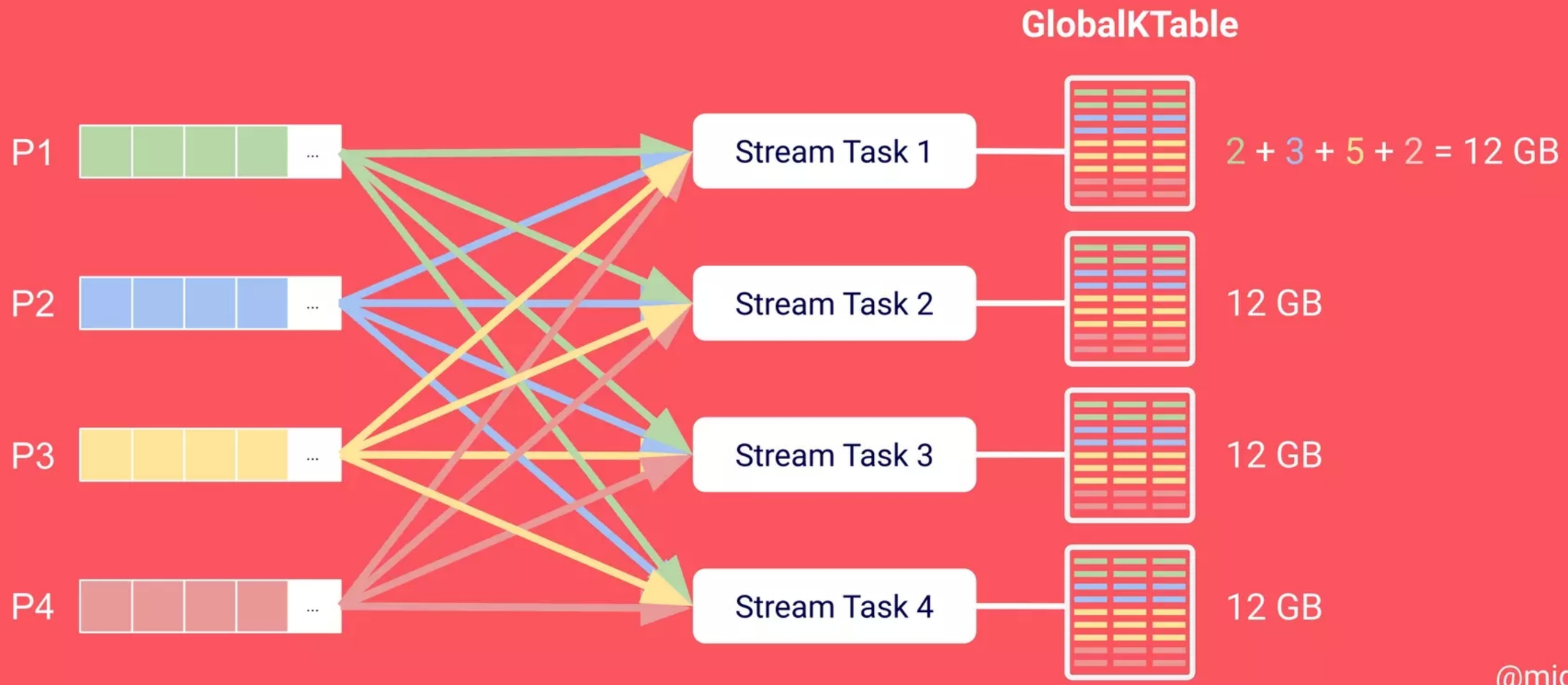


# Streams and Tables are partitioned, too



# Global Tables give *complete* data to every task

## Great for joins without re-keying the input or to broadcast info



# Topics vs. Streams and Tables

| Concept | Schema | Partitioned | Unbounded | Ordering Guarantee*** | Mutable | Single Value Per Key | Fault Tolerant |
|---|---|---|---|---|---|---|---|
| **Storage Layer** | | | | | | | |
| Topic | No* (raw bytes) | Yes | Yes | Yes | No | No | Yes |
| **Processing Layer** | | | | | | | |
| Stream | Yes | Yes | Yes | Yes | No | No | Yes |
| Table | Yes | Yes | No** | No | Yes | Yes | Yes |
| Global Table | Yes | No | No** | No | Yes | Yes | Yes |

*When using Schema Registry with Avro, then a schema is technically available for the data in a topic.
**Generally speaking the answer is Yes but, in practice, tables are almost always bounded due to finite key space.
***Ordering per partition, not across the multiple partitions in a topic.

# Elasticity, Fault Tolerance, and Other Advanced Concepts

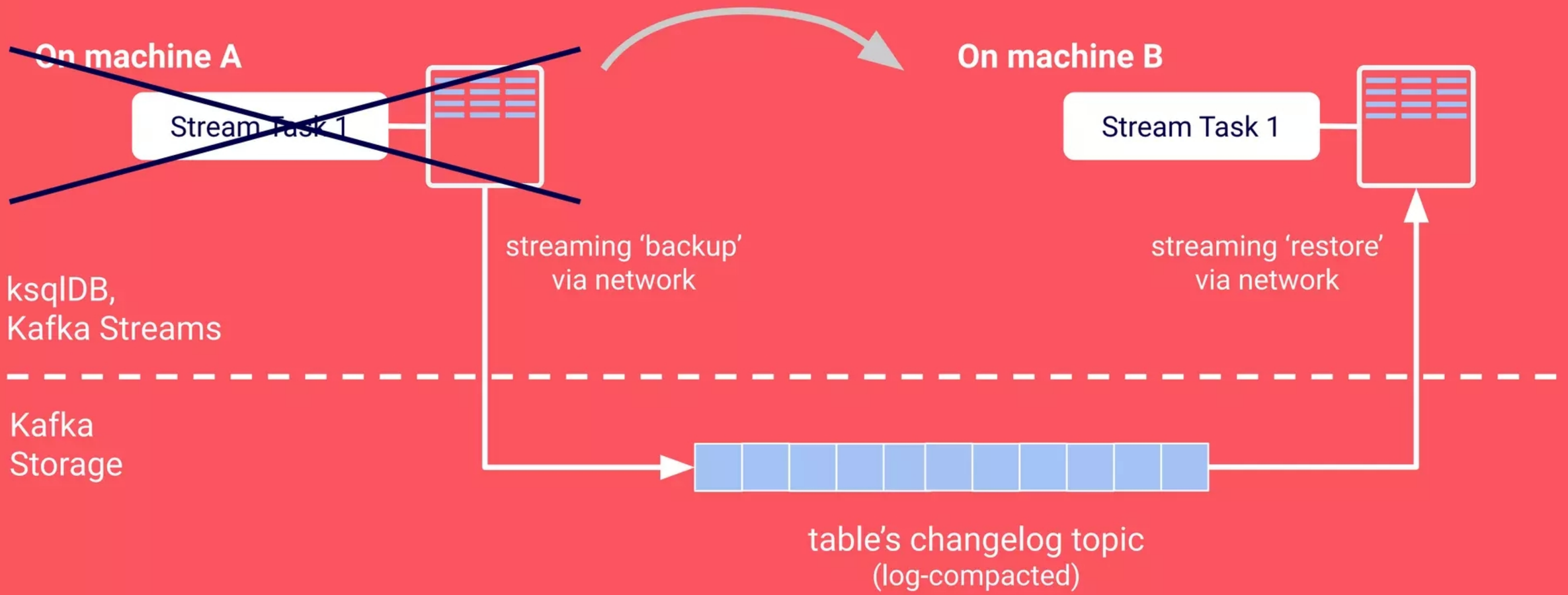
# Tables are materialized on local disk

## so that processing is fast, and tables can be larger than RAM. Enables large-scale state. Saves $$$ on cloud instances.


Materialized on local disk under /var
(via RocksDB by default)

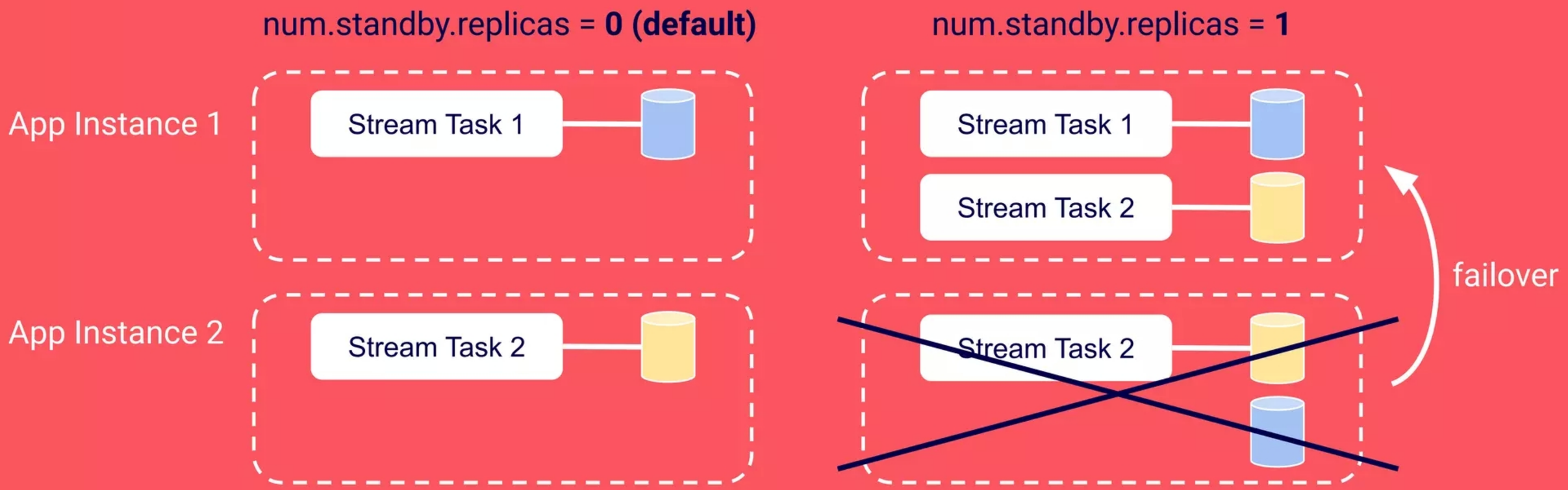
## But machines and containers can be lost!
## How can we make this fault-tolerant?

# Tables and other state are made fault-tolerant by storing their data durably, remotely in Kafka topics



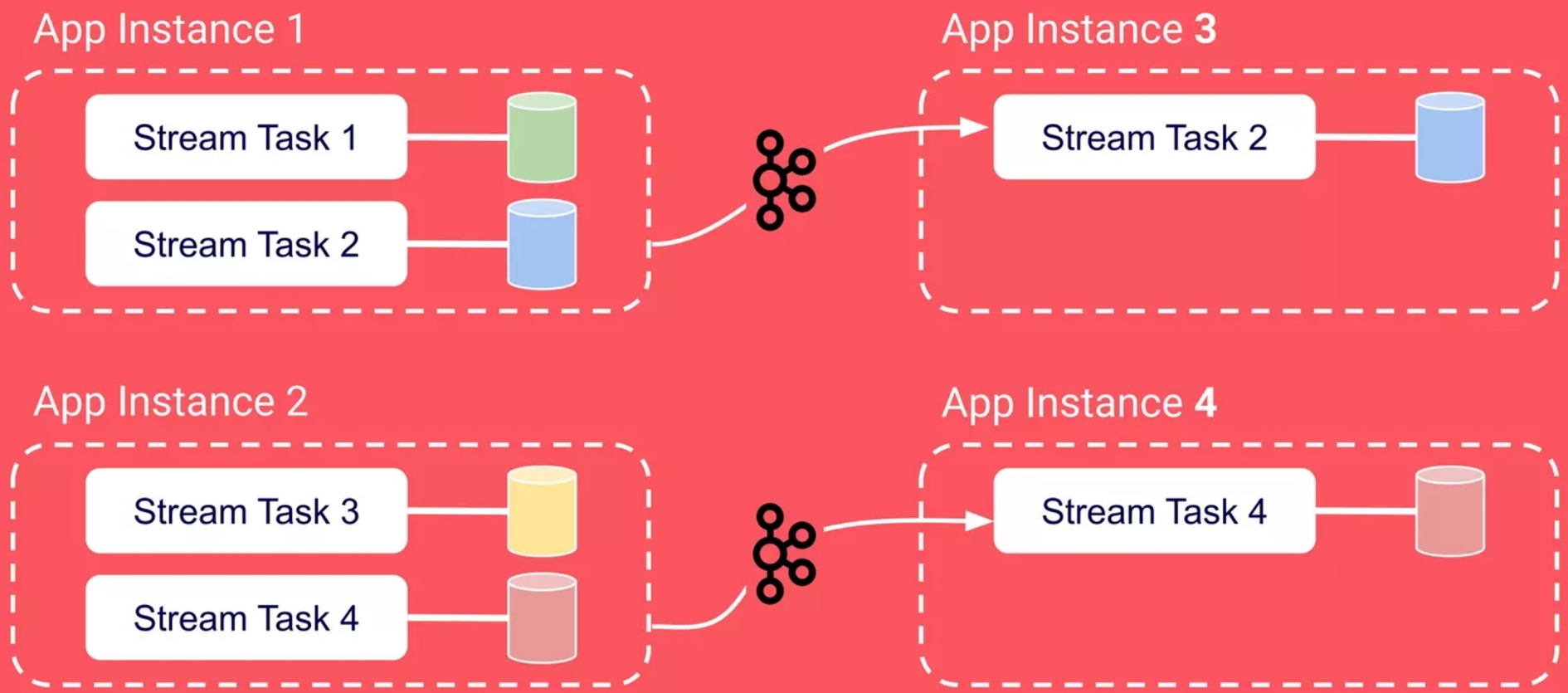
# Standby Replicas speed up application recovery

## App instances can optionally maintain copies of another instance's table/state data to minimize failover times



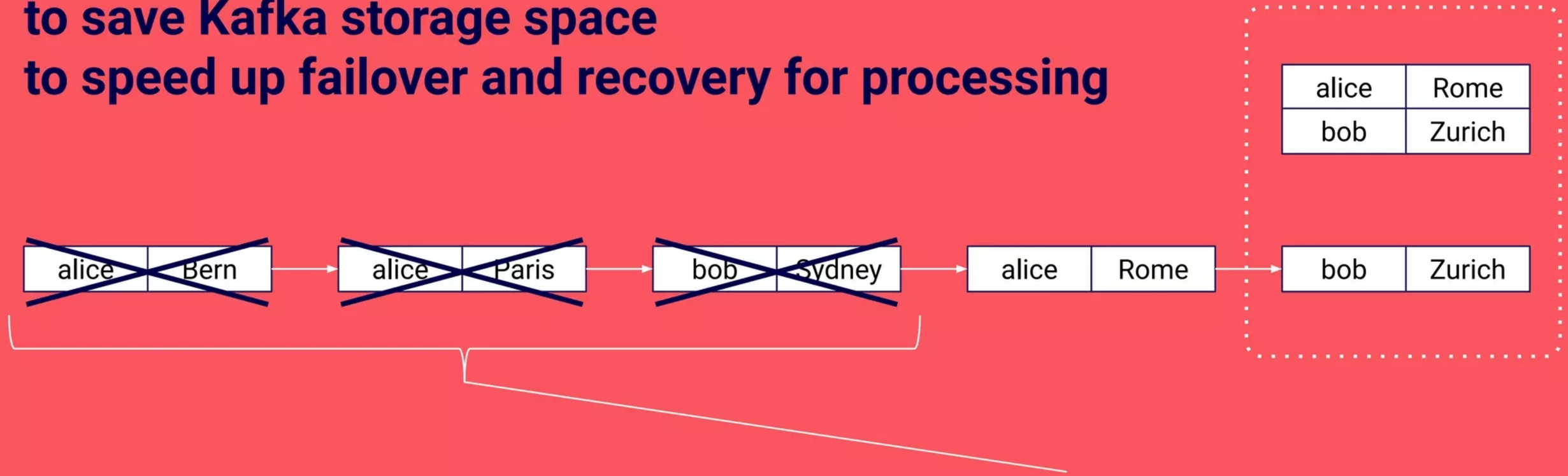
# Elastic scaling

## Stream tasks are migrated, including tables/state (via Kafka)



# Tables <-> topic log-compaction

**A table's underlying topic is *compacted* by default**
**to save Kafka storage space**
**to speed up failover and recovery for processing**



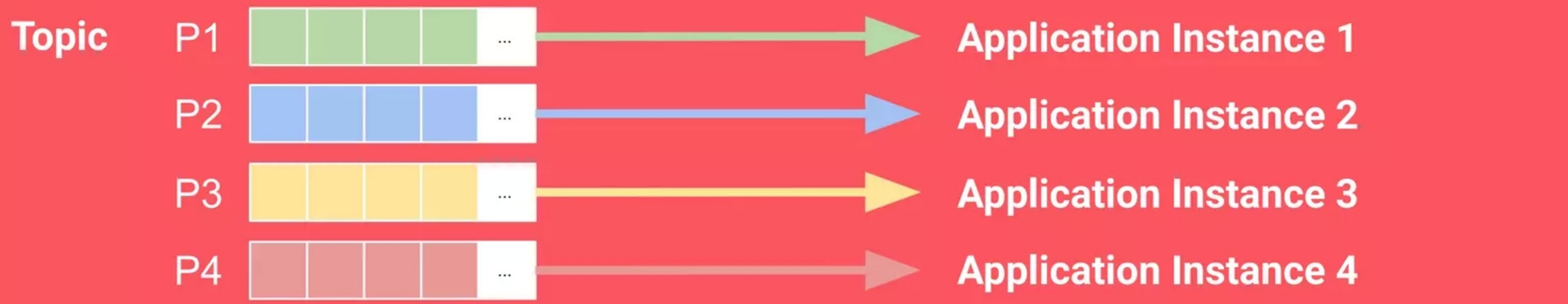

**Note: Compaction intentionally removes part of a table's history. If you need the full history and don't have the historic data elsewhere, consider disabling compaction.**

# TL;DR for Log Compaction

**Have a Stream?**
**→ Disable log compaction for its topic (= default)**

**Have a Table?**
**→ Enable log compaction for its topic (= default)**
**Disable only when needed, see previous slide**

# Max processing parallelism = #input partitions




→ Need higher parallelism? Increase the original topic's partition count.

→ Higher parallelism for just one use case? Derive a new topic from the original with higher partition count. Lower its retention to save storage.

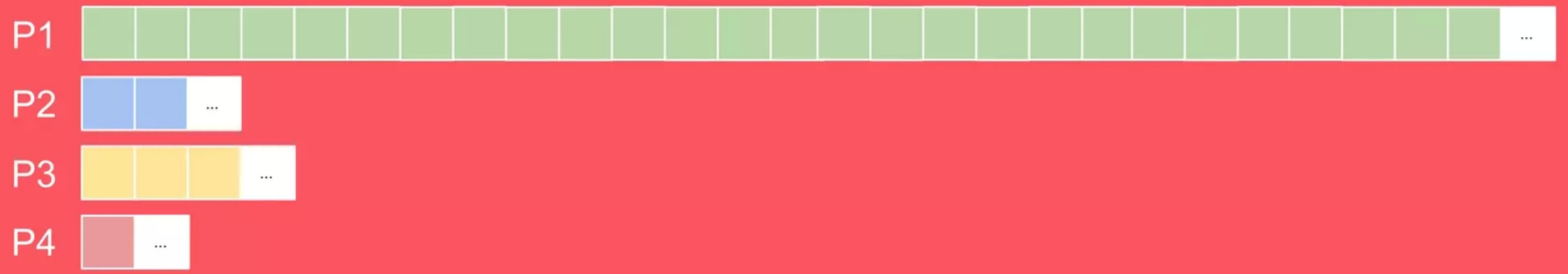
# How to increase number of partitions when needed

## ksqlDB example: statement below creates a new stream with the desired number of partitions.

```
CREATE STREAM products_repartitioned
  WITH (PARTITIONS=30) AS
  SELECT * FROM products
  EMIT CHANGES
```

# ‘Hot’ partitions can be problematic, often caused by

1. Events not evenly distributed across partitions
2. Events evenly distributed but certain events take longer to process




## Strategies to address hot partitions include

1a. Ingress: Find better partitioning function *f*(event.key) for producers
1b. Storage: Re-partition data into new topic if you can’t change the original
2. Scale processing vertically, e.g. more powerful CPU instances

# How to re-partition your data when needed

## ksqlDB example: statement below creates a new stream with changed number of partitions and a new field as event.key (so that its data is now correctly co-partitioned for joining)

```
CREATE STREAM products_repartitioned
  WITH (PARTITIONS=42) AS
  SELECT * FROM products
  PARTITION BY product_id
  EMIT CHANGES
```

# Capacity Planning and Sizing

## Sorry, not covered here!  I recommend:

ksqlDB Performance Tuning for Fun and Profit, by Nick Dearden
https://kafka-summit.org/sessions/ksql-performance-tuning-fun-profit/

ksqlDB documentation
https://docs.ksqldb.io/en/latest/operate-and-deploy/capacity-planning/

Kafka Streams documentation
https://docs.confluent.io/current/streams/sizing.html

# Where to go from here?

My blog series on Kafka Fundamentals to learn more
https://www.confluent.io/blog/kafka-streams-tables-part-1-event-streaming

Confluent Cloud, the fastest & easiest way to get started with Kafka
https://www.confluent.io/confluent-cloud/

Confluent Platform, if you want to operate Kafka yourself
https://www.confluent.io/product/confluent-platform/

confluent

# THANK YOU


## @miguno
## michael@confluent.io


---

 cnfl.io/meetups

 cnfl.io/blog

 cnfl.io/slack