# Streams and Tables in Apache Kafka: Processing Fundamentals with Kafka Streams and ksqlDB

Technology > Use Cases JAN 15, 2020 READ TIME: 15 MIN

Part 2 of this series discussed in detail the storage layer of Apache Kafka: topics, partitions, and brokers, along with storage formats and event partitioning. Now that we have this foundation, our focus will move beyond storing events to *processing* events by looking at Kafka's processing fundamentals. This article will explore streams and tables along with data contracts and consumer groups, and how all this enables you to implement distributed applications that process data in parallel at scale.

Let's start with how your events stored in Kafka topics are made accessible for processing by turning them into streams and tables.

*This article is the third in our series on Kafka fundamentals:*

1. *Streams and Tables in Apache Kafka: A Primer*
2. *Streams and Tables in Apache Kafka: Topics, Partitions, and Storage Fundamentals*
3. ***Streams and Tables in Apache Kafka: Processing Fundamentals with Kafka Streams and ksqlDB (this article)***
4. *Streams and Tables in Apache Kafka: Elasticity, Fault Tolerance, and Other Advanced Concepts*

WRITTEN BY

## From storage to processing

Michael Noll

Product Manager

Topics live in Kafka's storage layer—they are part of the Kafka "filesystem" powered by the brokers. In contrast, streams and tables are concepts of Kafka's *processing layer*, used in tools like ksqlDB and Kafka Streams. These tools process your events stored in "raw" topics by turning them into streams and tables—a process that is conceptually very similar to how a relational database turns the bytes in files on disk into an RDBMS table for you to work with.

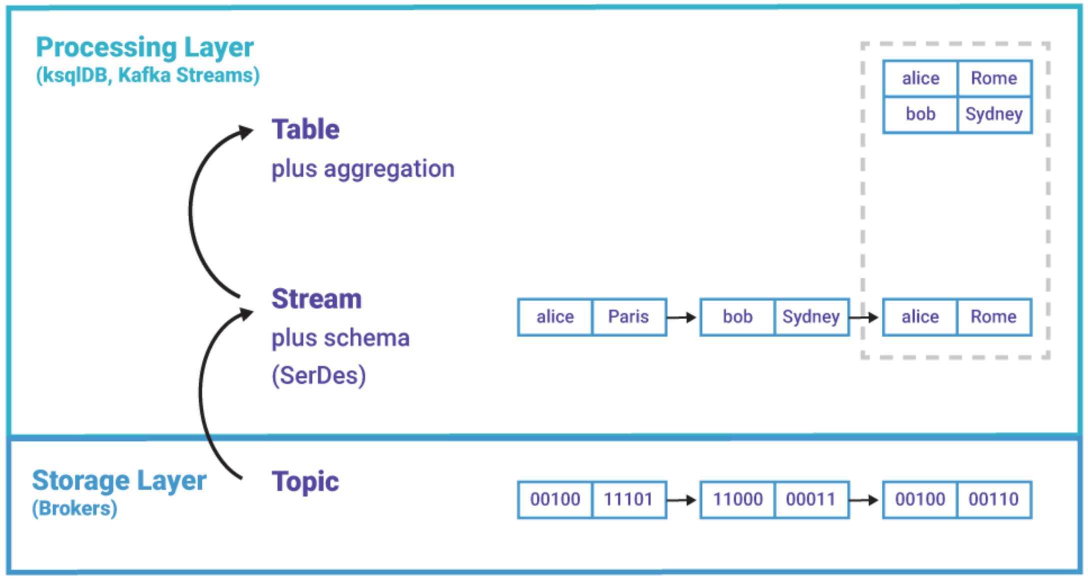


*Figure 1. Topics live in the storage layer. Streams and tables live in the processing layer.*

An **event stream** in Kafka is a topic with a schema. Keys and values of events are no longer opaque byte arrays but have specific types, so we know what's in the data. Like a topic, a stream is unbounded.

*The following examples use the Java notation of <eventKey, eventValue> for the data types of event key and value, respectively. For instance, <byte[], String> means that the event key is an array of raw bytes, and the event value is a String, i.e., textual data like a VARCHAR. More complex data types can be used, such as an Apache Avro™ schema that defines a GeoLocation type.*

**Example #1:** A <byte[], byte[]> topic is read and deserialized by a consumer client as a <String, String> stream of geolocation events. Or, we could use a better type setup and read it as a <User, GeoLocation> stream, which is what I'd prefer.

Here are code examples for how to read a topic as a stream:

- ksqlDB:
- Kafka Streams:

A **table** is a, well, table in the ordinary technical sense of the word, and we have already talked a bit about tables before (a table in Kafka is today more like an RDBMS materialized view than an RDBMS table, because it relies on a change being made elsewhere rather than being directly updatable itself). Seen through the lens of event streaming however, a table is also an *aggregated stream*. This is a reference to the stream-table duality we discussed in part 1.

> If you are interested in further details, I recommend the academic paper Streams and Tables: Two Sides of the Same Coin, co-authored by my colleagues Matthias J. Sax and Guozhang Wang.

Of course, you can also create a table straight from a Kafka topic as a convenient usability shortcut. In most practical use cases, a table is effectively bounded—that is, it has a finite number of rows—because a company has a limited number of customers, a limited number of products to sell, and so on. But generally speaking, a table can be unbounded just like a stream. An example scenario is a table to which we keep adding rows (events) and where the key of each row is a UUID.

**Example #2:** A <String, String> stream of geolocation events is continuously aggregated into a <String, String> table that tracks the latest location of each user. This example is illustrated in Figure 1.

Here are the code examples for how to read a topic as a table:

- ksqlDB:
- Kafka Streams:

**Example #3:** A <String, String> stream is continuously aggregated into a <String, Long> table that tracks the number of visited locations (event value) per user (event key). The aggregation operation continuously counts and updates the table, per key, with the number of observed locations per key in the table. We have seen this example as an animation in the first section of this article, including the respective code using COUNT().

## Data contracts, schema on read, and schema on write

As already mentioned, it is the responsibility of the consuming client (whether it's ksqlDB, Kafka Connect, a custom Kafka consumer, etc.) to deserialize the raw bytes of a Kafka message into the original event by applying some kind of schema, be it a formalized schema in Avro or Protobuf, or an informal JSON format scribbled on the back of a napkin in the company canteen. This means it is, generally speaking, a *schema-on-read* setup.

But how does a consuming client know how to deserialize stored events, given that most likely a different client produced them? The answer is that producers and consumers must agree on a data contract in some way. Gwen Shapira covered the important subject of data contracts and schema management in an earlier blog post, so I'll skip over the details here. But in summary, the easiest option is to use Avro and Confluent Schema Registry. With a schema registry and a formalized schema (including but not limited to Avro), we are moving from schema on read into schema-on-write territory, which is a boon for pretty much everyone who is working with data, not just the few poor souls of us tasked to "go and do data governance."

And with Confluent Platform 5.4 or newer, you have the additional option to centrally enforce broker-side Schema Validation so that no misbehaving client can violate the data contract: incoming events are validated server side before they are stored in Kafka topics. This feature is a huge benefit for any Kafka user and especially for larger, regulated organizations.

## Processing is partitioned, too

In Kafka, the processing layer is partitioned just like the storage layer. Data in a topic is processed *per partition*, which in turn applies to the processing of streams and tables, too. To understand this, we must first talk about the concept of consumer groups in Kafka.

### Applications process data in groups for strength in numbers

One compelling reason to use Kafka is that it allows for highly scalable processing of your data *in parallel*. This is achieved by letting you implement applications that run in a distributed fashion across a number of *application instances*. You can picture this as a single application binary launched in multiple processes that run on different containers, virtual machines, or physical machines.

To process the data collaboratively, these multiple instances of your application dynamically form a so-called Kafka consumer group at runtime that reads from the same input topic(s). This group membership is configured with the application.id setting for Kafka Streams apps, the ksql.service.id setting for ksqlDB servers in a ksqlDB cluster, and the group.id setting for applications that use one of the lower-level Kafka consumer clients in your favorite programming language, as well as for worker nodes in a Kafka Connect cluster.

For example, we could write a distributed, multi-instance fraud detection application with Kafka Streams to process the events in a high-volume payments topic in parallel, and its app instances would all be members of the same group fraud-detection-app that represents the logical application. This application would then be deployed next to a Kafka cluster, from which it would read and write events via the network.

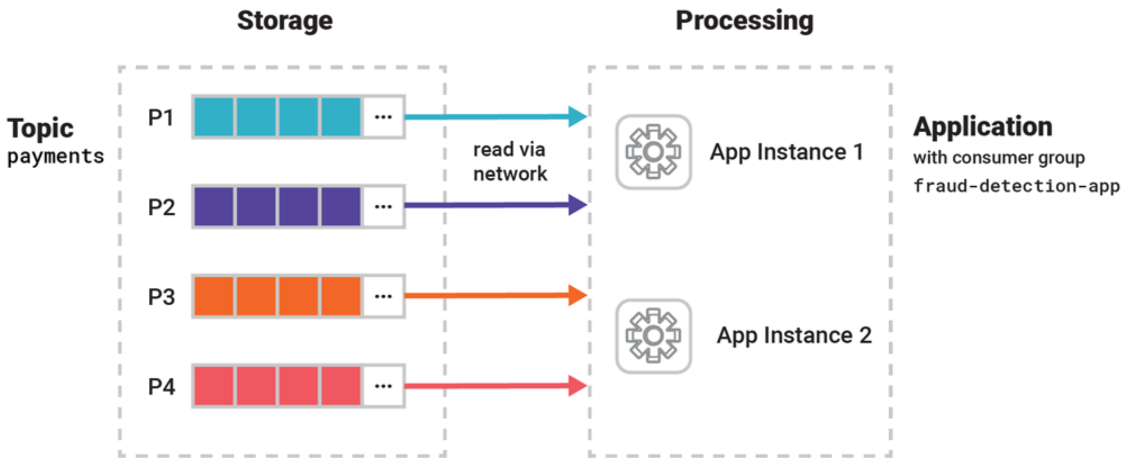


*Figure 2. This distributed application (right side) has two instances running, which form a Kafka consumer group for this application. They collaboratively process the input data in parallel, which they read from Kafka brokers over the network (left side).*

Within a group, every member instance gets an exclusive slice of the data (one or more topic partitions), which it processes in isolation from the other instances. And with the help of Kafka's consumer group protocol, the group automatically detects whenever new instances join or existing instances leave the group (e.g., because Kubernetes killed a container), and then it automatically redistributes the workload and partition assignments so that data processing continues automatically.

This coordination process is called rebalancing in Kafka, and it is a part of the consumer group protocol. For example, rebalancing is triggered when a new ksqlDB server node joins a ksqlDB cluster, or when an instance of a Kafka Streams application is terminated.

All this happens during live operations of your applications, without the risk of data loss or incorrect data processing. If you want to learn about rebalancing in more detail, I recommend the article Apache Kafka Rebalance Protocol, or the Magic Behind Your Streams Applications by Florian Hussonnois, who is a Confluent Community Catalyst.

## Stream tasks are the unit of processing parallelism

But, there's more. In ksqlDB and Kafka Streams, Figure 2 is actually a simplification. We need to zoom into the gear icons for application instances. That's because the unit of parallelism for processing is actually not an application instance, but a so-called stream task (which processes both streams and tables, despite its name). An application instance can be running zero, one, or multiple such tasks during its lifecycle. Input partitions from topics/streams/tables are assigned 1:1 to these tasks for processing. Then, with the help of Kafka's consumer group protocol, it is actually these stream tasks that are being distributed across an application's instances whenever one instance joins or leaves the group, i.e., whenever a *rebalancing* process is being carried out.

For example, if our previous application processes an input topic with four partitions P1–P4, then this results in four stream tasks 1–4 for their respective processing. And these four tasks will then be evenly distributed across an application's running instances. So if there are two app instances, then each will run two tasks for a total of four.

*Figure 3. Partitions are assigned 1:1 to stream tasks for processing. Tasks are automatically distributed evenly across running instances of your application.*

This partition-to-task assignment via stream tasks also means that data in a topic is being processed per partition. Do you remember "same event key to same partition" from the first article in this series? Here we can see why it is so important to always send "related events" to one and the same partition, because otherwise they can't be processed together and, even more important in many use cases, processed in the correct order.

To recap: processing is carried out by stream tasks. Every task is responsible for processing one particular input partition, which means there are exactly as many stream tasks as there are input partitions—neither more nor less. These stream tasks are evenly assigned by Kafka to the running instances of your application to allow for scalable, parallel processing of your data.

Given this primer on Kafka consumer groups and (for ksqlDB and Kafka Streams only) stream tasks, what does this mean for streams and tables?

## Streams are partitioned, and so is their processing

Streams are simply topics with a schema, so there's little that's new to explain regarding how their processing works compared to what we just learned in the previous section. Essentially, the only additional step a processing application performs is applying a schema to a topic to turn it into a "typed" stream when reading from storage, and vice versa for writing. Everything else—every stream task getting its own stream partition to process, automatic (re)distribution of tasks as application instances come and go, etc.—is identical to what we discussed in the previous section.

## Tables are partitioned, and so is their processing

The situation for tables is much more interesting, at least if you enjoy learning about the internals of distributed systems! Whenever an application—not just ones using Kafka—must remember something about previous events when processing the next one, we call this application to be stateful, and what it remembers is called the state.

Tables are one form of such application state. Imagine you need to compute the aggregate table of total sales per georegion. Here, we must be able to remember, say, the current total of Germany (state) so that we can add the next German sale (new event) to the total, thus updating it. Dealing with state, and doing this efficiently at scale and in the face of infrastructure failures, turns out to be a key challenge in distributed systems. It's also what led to the rise of NoSQL databases like Apache Cassandra™ when traditional RDBMS could no longer keep up with the scalability requirements and data volumes we have today. So let's take a closer look at how state management works for tables in Kafka.

Like streams, tables are partitioned. This means what we previously discussed about the nature of processing streams generally applies to tables, too. But unlike streams, tables need to additionally maintain their respective state in between events so that operations like aggregations (e.g., COUNT()) can work properly. This is achieved by implementing tables on top of so-called state stores, which are essentially lightweight key-value databases.

Every table has its own state store. Any operation on the table such as querying, inserting, or updating a row is carried out behind the scenes by a corresponding operation on the table's state store. These state stores are being *materialized on local* *disk* inside your application instances or ksqlDB servers for quick access. Storing on local disk has the great advantage that tables and other state don't need to fit into the RAM of the container/VM/machine on which the respective stream task is running. This allows you to work with use cases that require large amounts of state (dozens of GBs or more) as well as to run your applications on cheaper cloud instances.

But more importantly, the state stores are stored remotely in Kafka to enable fault tolerance and elasticity, which we cover later on. That is, Kafka is the source of truth for table data, just like for streams. The aforementioned local materialization of tables, which uses embedded RocksDB as the default engine, is immaterial for the "safety" of your data. I emphasize this point because I often get RocksDB questions in this context, as if the color of your parachute was significant for its safety (it's not).

Now, a table's state store is split into multiple state store partitions (sometimes called *state store instances* in the Kafka documentation) just like the table's partitions. If a table has 10 partitions, then its state store will have 10 partitions, too. During the actual processing, each state store partition will be maintained by one and only one stream task, which has exclusive access to the data. We can thus say that such tasks are stateful stream tasks.

For example, in the diagram below, task 1 of our application will read events from the blue partition P1 to maintain whatever is in the blue state store partition and thus the blue table partition. Yes, the presence and usage of partitions in Kafka is really ubiquitous! This distributed, shared-nothing design based on partitions for stateful processing (tables) as well as for stateless processing (streams) is a major reason why Kafka's processing layer scales so well.

*Figure 4. When your application works with tables, then this means the respective stream tasks will additionally maintain and interact with state stores behind the scenes. Note the additional "state" icons compared to the previous diagram.*

You may wonder at this point what the difference is between a table and a state store. Firstly, every table generally has a state store but not vice versa. Developers writing Kafka Streams applications can use the Processor API to create and interact with the lower-level state stores directly without having to use tables at all. If you are using ksqlDB, however, you will never see state stores as they are an implementation detail. Secondly, as discussed previously, one cannot mutate a table directly today, which makes it similar to an RDBMS materialized view. State stores differ in that they give developers direct read and write access through typical operations, such as put(key, value) or get(key) for querying, inserting, updating, and deleting data.

Let's zoom out a little bit from the previous diagram showing the nuts and bolts of state stores to see some practical effects of this setup for tables. What about the local footprint of a table inside your application or ksqlDB server, for example? Imagine that the example topic with four partitions P1–P4 is being read into a table, whose data has a total size of 12 GB for your application. These 12 GB will be split across the table's four partitions along the partition boundaries of the table's input topic (or input stream/table). The sizes of individual table partitions will vary depending on the characteristics of the data: the second partition may be 3 GB, the third partition may be 5 GB, and so on. An effective partitioning function ƒ(event.key, event.value) for your use case is therefore a crucial ingredient for an even distribution of your data.

*Figure 5. An application computing a table from an input topic. This example table's complete data of 12 GB is split along partition boundaries to respective stream tasks, each of which manages exactly one table partition.*

## Global tables are not partitioned

The partitioned design of Kafka's processing layer is tremendously useful for scaling and performance. Sometimes, however, we have a use case that requires global knowledge of all events. That's where global tables come in handy. Unlike a normal table, a global table is not partitioned but instead gives complete data to every stream task. Going back to the previous example, every task would now have 12 GB of table data locally available. Global tables are very useful for, say, broadcasting information to all tasks or when you want to do joins without having to first re-partition your input data. Note that global tables are currently only supported by Kafka Streams and not by ksqlDB (yet).

*Figure 6. A global table gives complete data to every stream task, unlike a normal, partitioned table. In contrast to Figure 5, every task now has the full 12 GB of table data locally available.*

## Comparing streams, tables, and topics

Here's a quick recap of what we covered thus far:

*Table 1. Topics vs. Streams and Tables*

## Summary

This completes the third part of this series, where we learned about the processing layer of Apache Kafka by looking at streams and tables, as well as the architecture of distributed processing with the Kafka Streams API and also ksqlDB. Part 4, the final article in this series, will revisit the processing layer again and dive into how elastic scaling and fault tolerance are architected and implemented—including a return of the stream-table duality, which turns out to underpin many of these capabilities.

## Say Hello World to event streaming

If you're ready to get more hands on, there is a way for you to learn how to use Apache Kafka the way you want: by writing code. Apply functions to data, aggregate messages, and join streams and tables with Kafka Tutorials, where you'll find tested, executable examples of practical operations using Kafka, Kafka Streams, and ksqlDB.

## Other articles in this series

- Part 1 – Streams and Tables in Apache Kafka: A Primer
- Part 2 – Streams and Tables in Apache Kafka: Topics, Partitions, and Storage Fundamentals
- Part 4 – Streams and Tables in Apache Kafka: Elasticity, Fault Tolerance, and Other Advanced Concepts (next article)

---

Michael is a former principal technologist in the Office of the CTO at Confluent, the company founded by the original creators of Apache Kafka®. He focuses on longer-term product and technology strategy. Previously, Michael was the lead product manager for stream processing at Confluent, where his team created Kafka Streams and the streaming database ksqlDB. He is a well-known technology blogger in the big data community (WWW.MICHAEL-NOLL.COM) and a committer/contributor to open source projects such as Apache Storm and Apache Kafka.

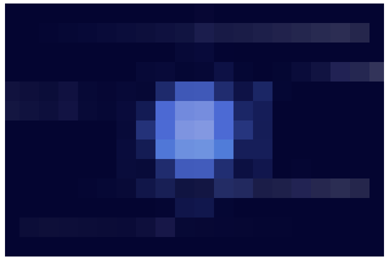
---

## Did you like this blog post? Share it now

Technology < Use Cases

## Subscribe to the Confluent blog

Subscribe

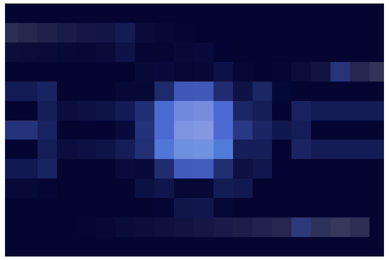


### Introducing Versioned State Store in Kafka Streams

AUG 24, 2023

Versioned key-value state stores, introduced to Kafka Streams in 3.5, enhance stateful processing capabilities by allowing users to...

VICTORIA XIA



### Delivery Guarantees and the Ethics of Teleportation

APR 25, 2023

This blog post discusses the two generals problems, how it impacts message delivery guarantees, and how those guarantees would...

WADE WALDRON

| Product | Cloud | Solutions | Developers | About |
|---|---|---|---|---|
| Confluent Cloud | Confluent Cloud | Financial Services | Confluent Developer | Investor Relations |
| Confluent Platform | Support | Insurance | What is Kafka? | Startups |
| Connectors | Sign Up | Retail and eCommerce | Resources | Company |
| Flink | Log In | Automotive | Events | Careers |
| Stream Governance | Cloud FAQ | Government | Webinars | Partners |
| Confluent Hub | | Gaming | Meetups | News |
| Subscription | | Communication Service Providers | Current: Data Streaming Event | Contact |
| Professional Services | | Technology | Tutorials | Trust and Security |
| Training | | Manufacturing | Docs | |