Kafka Streams and ksqlDB Compared – How to Choose

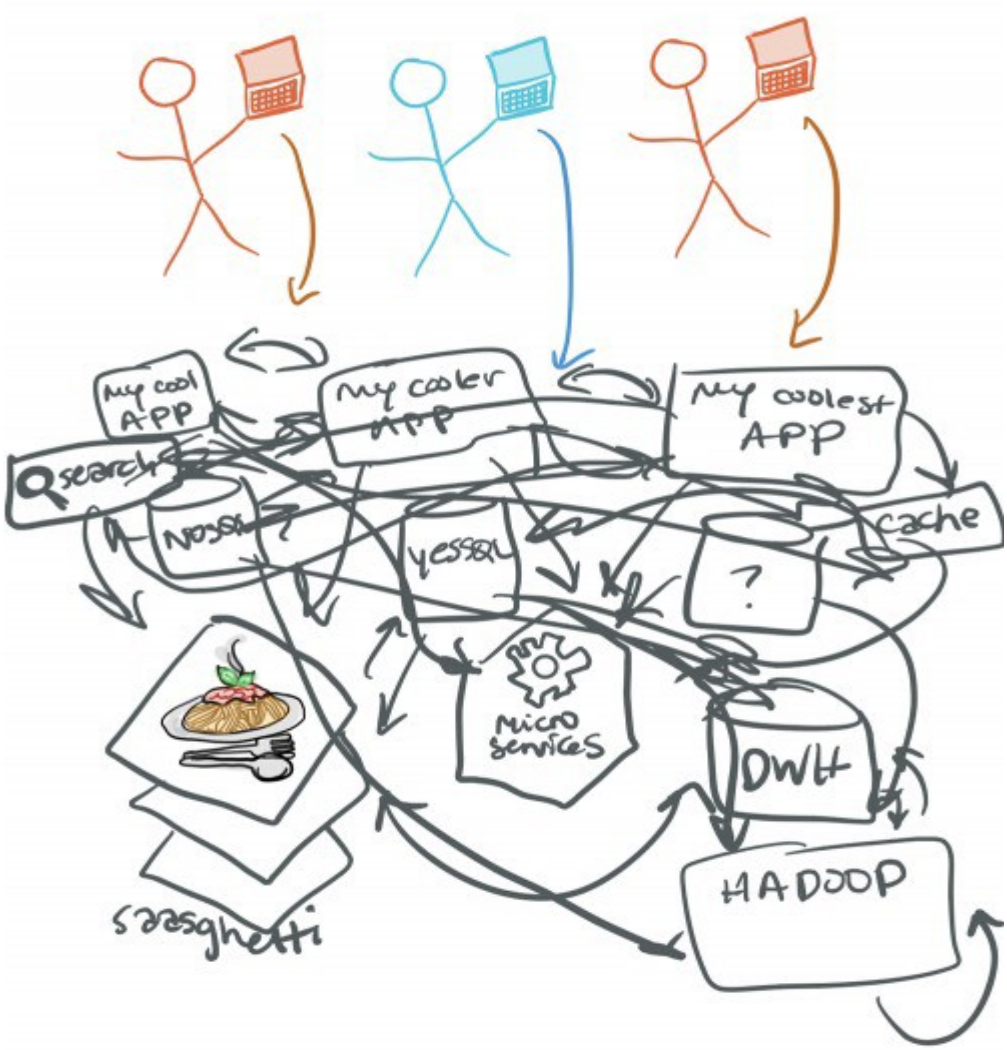
Technology

Use Cases

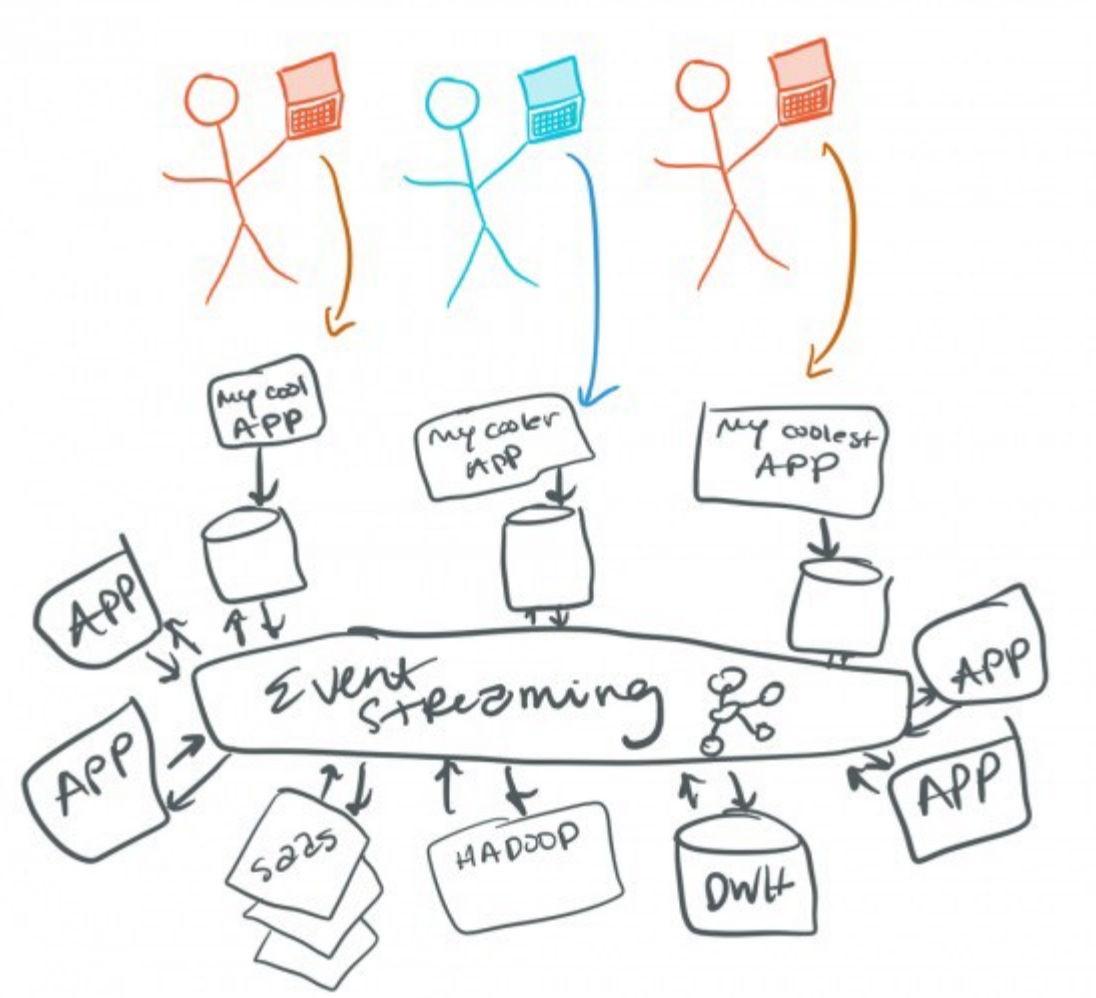
Nov 21, 2019Read Time: 10 min

ksqlDB is a new kind of database purpose-built for stream processing apps, allowing users to build stream processing applications against data in Apache Kafka® and enhancing developer productivity. ksqlDB simplifies maintenance and provides a smaller but powerful codebase that can add some serious rocketfuel to our event-driven architectures.

As beginner Kafka users, we generally start out with a few compelling reasons to leverage Kafka in our infrastructure. An initial use case may be implementing Kafka to perform database integration. This is especially helpful when there are tightly coupled yet siloed databases—often the RDBMS and NoSQL variety—which can become single points of failure in mission-critical applications and lead to an unfortunate spaghetti architecture.

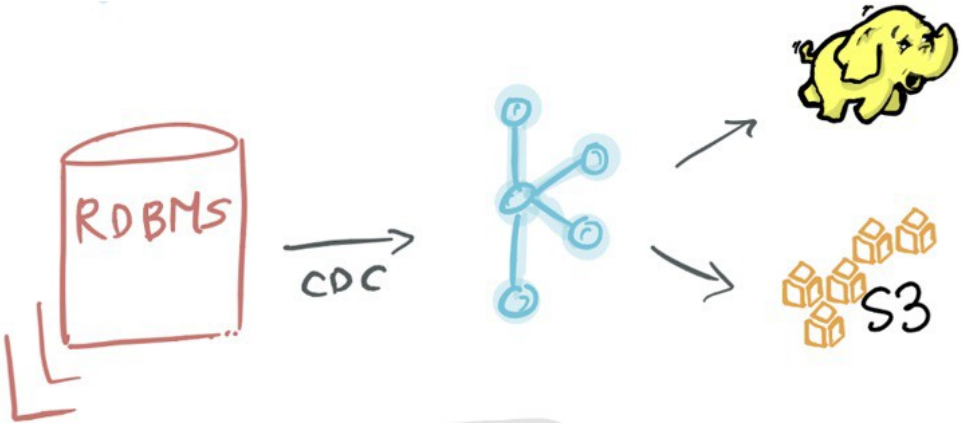


Enter: Kafka! Kafka provides buffering capabilities, persistence, and backpressure, and it decouples these systems because it is a distributed commit log at its architectural core.

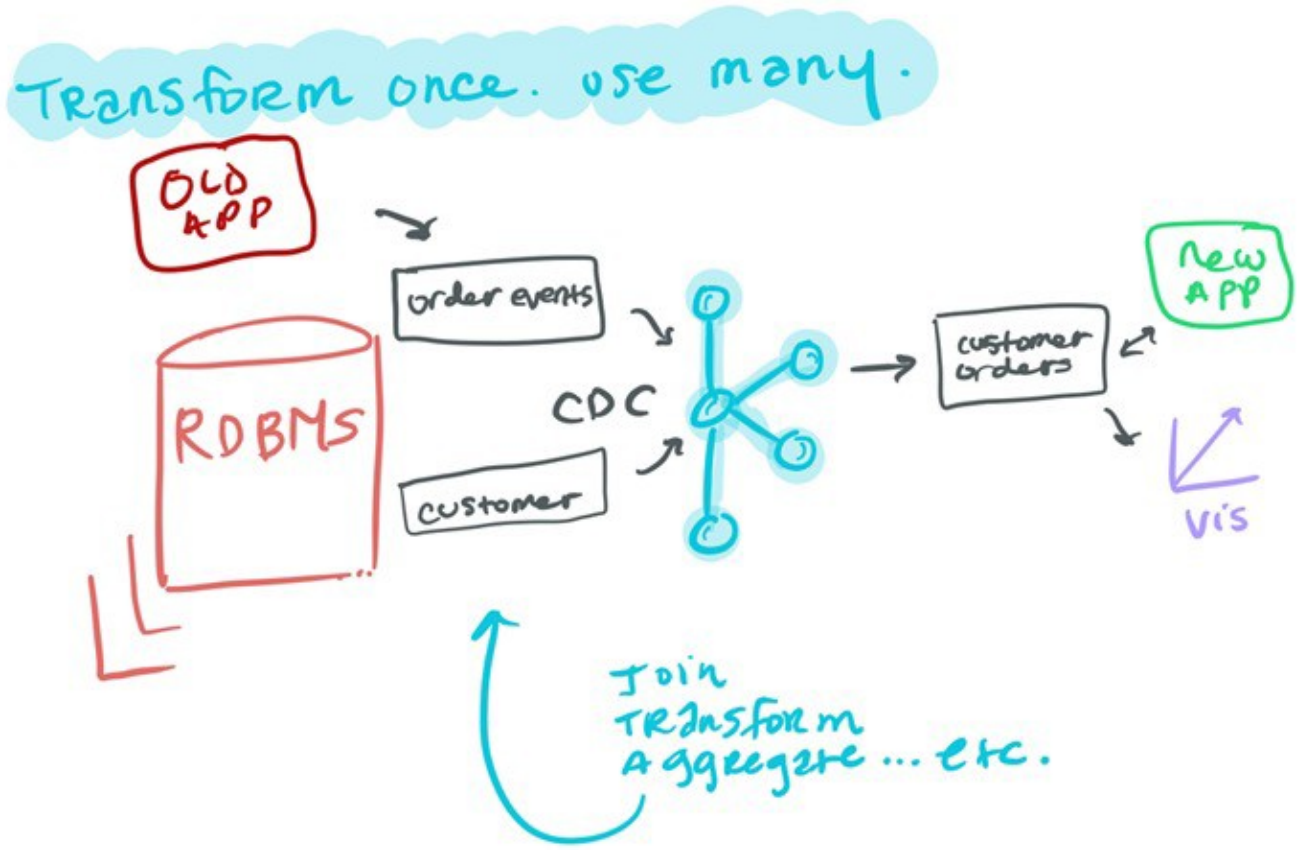


But wait, there are more benefits as to why we might consider Apache Kafka. Perhaps we want to leverage it as a “message bus” or for “pub/sub” (read more about how it compares to those approaches in this blog post). Apache Kafka is distributed unlike

other enterprise service bus (ESB) or pub/sub solutions, with a leader-follower design. It is highly available, fault tolerant, low latency, and foundational for an event-driven architecture for the enterprise. Our initial Kafka use case might even look a little something like change data capture (CDC), where we are capturing the changes derived from a customer table, as well as changes to an order table in our relational store.



Maybe we find that there's opportunity to optimize Kafka for benefits beyond the above-mentioned purposes. We could be doing more—processing and analyzing data as it occurs, and deriving real-time insights by joining streams and enabling actionable logic instead of waiting to process it at a later point in time in a nightly batch. What can we do to enhance this data pipeline?

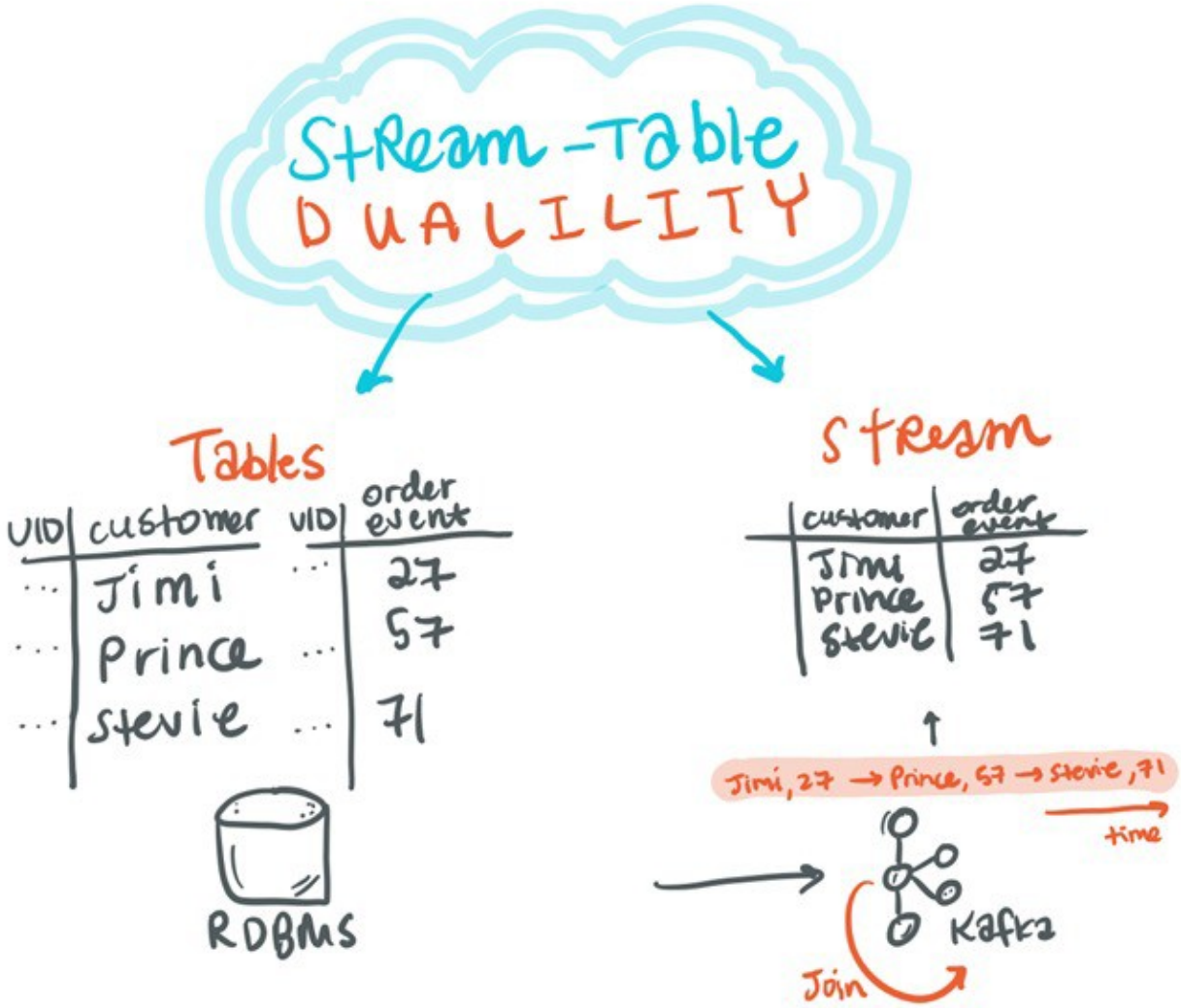


If we expand upon the initial CDC use case presented, we see that we can transform our data once but use it for many applications. By joining the "customer" and "order events" streams together to give us "customer orders," we enable developers to write new apps

using this enriched data available as a stream, as well as land it to additional datastores as required. Plus, since this new stream is consumed from Kafka, it still has all the benefits that we listed before. So how do we get from our RDBMS tables to become real-time streams that we can process and enrich? To answer this, we must first understand the stream-table duality concept.

The stream-table duality

Moving from the RDBMS world to the event-driven world—everything begins with events, but we still have to deal with the reality that we have data in tables. With our examples above, we have two separate tables for the customer and order event. These tables are a static view of our data at a point in time. When we translate our key/value data into Kafka, we do so via a Kafka topic. The concept of streams allows us to read from the Kafka topic in real time and process the data. Understanding how data is converted from a static table into events is a core concept of understanding Kafka Streams and ksqlDB.



Due to the stream-table duality, we can convert from table to stream and stream to table with fidelity. When we get our relational data into a Kafka-friendly format, we can start to do more and develop new applications in real time.

There are numerous ways to do stream processing out there, but the two that I am going to focus on here are those which integrate the best with Apache Kafka in terms of security and deployment: Kafka Streams, which is a native component of Apache Kafka, and ksqlDB, which is an event streaming database built and maintained by the original co-creators of Apache Kafka.

As a Java library, Kafka Streams allows you to do stream processing in your Java apps. By contrast, ksqlDB is an event streaming database that runs on a set of servers. It enables developers to build stream processing applications with the same ease and familiarity that comes with building traditional apps on a relational database.

Stream processing

The generic stream processing operations are filter, transform, enrich, and aggregate. ksqlDB allows you to seamlessly integrate stream processing functionality onto an existing Kafka cluster with an interface as familiar as a relational database. It is also valuable in its ease of use for diverse development teams Python, Go, and .NET, given that it speaks language-neutral SQL.

All of these elements are great, but recall the stream-table duality. We can not only do normal things like extract, transform, and load ETL our data but cleaning our data and making sure we get the right data in the right places is also a really common pattern that a lot of companies are using in production today. Simple use cases such as data filtering, filtering out some bit of data, and utilizing that stream in a specific application or to satisfy compliance are other patterns of utility.

When working within the context of a stream processing application, time becomes crucial. Stream joins and aggregations utilize windowing operations, which are defined based upon the types of time model applied to the stream. Examples include the time an event was processed (event time), when the data was captured by the app (processing time), and when Kafka captured the data (ingestion time). Configuring Kafka and developing our specific streams' apps depend on time semantics which vary given the business use cases at hand.

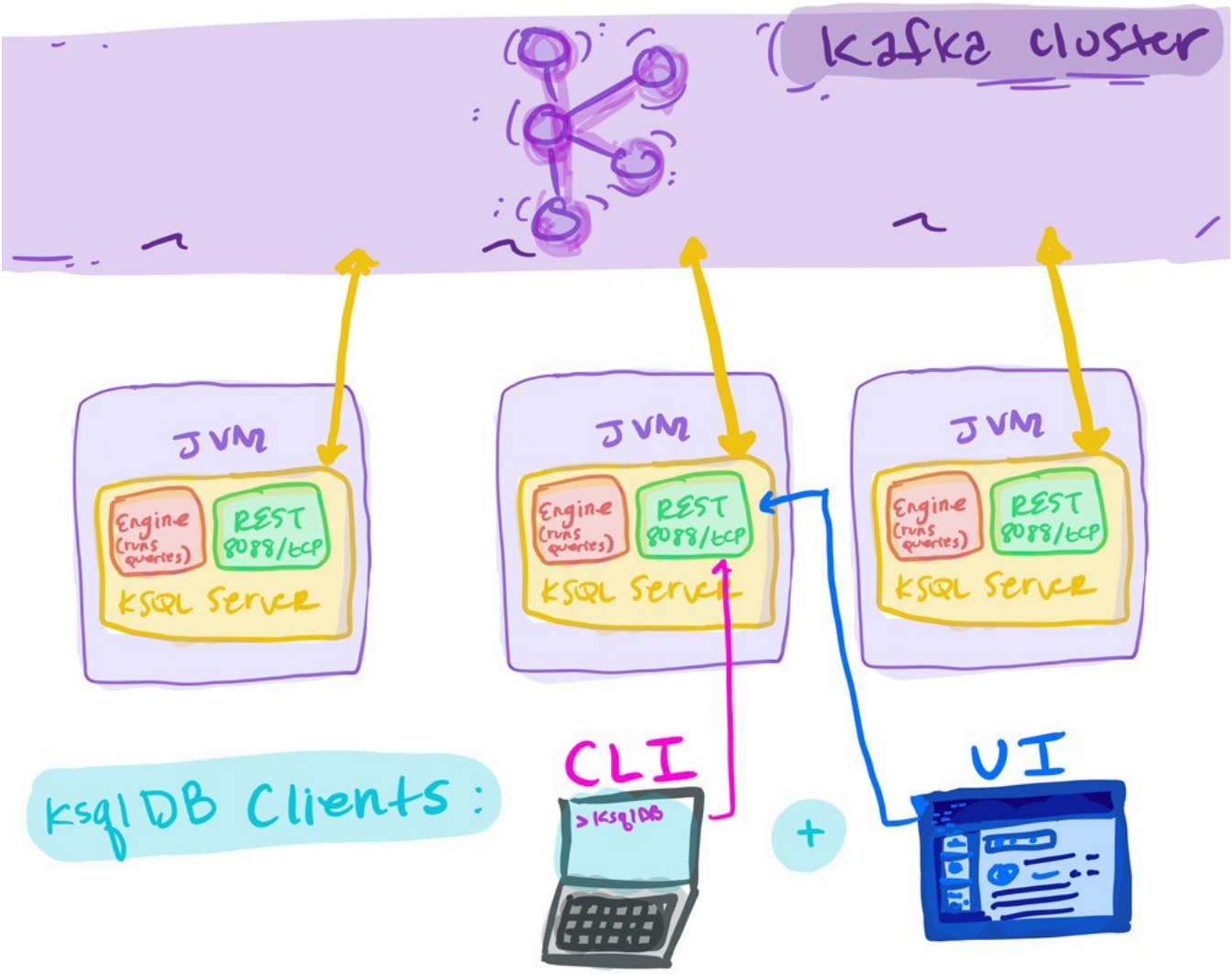
Common stream processing use cases include:

- **Anomaly detection:** detecting anti-patterns in the data; particularly relevant in banking or any transaction-based activities where rapid detection can be acted upon and fraud can be handled intelligently for the business
- **Manufacturing and the Internet of Things (IoT):** consider the notion of detecting manufacturing device failures and preventing loss of revenue in the production line

Stream processing with ksqlDB

With ksqlDB, we can create continuously updating, materialized views of data in Kafka,

and query those materializations in a variety of ways with SQL-based semantics. The ksqlDB clients are its command line interface CLI, Confluent Control Center UI, and the REST API. ksqlDB's server instances talk to Kafka directly, and you can add more servers without restarting your applications.



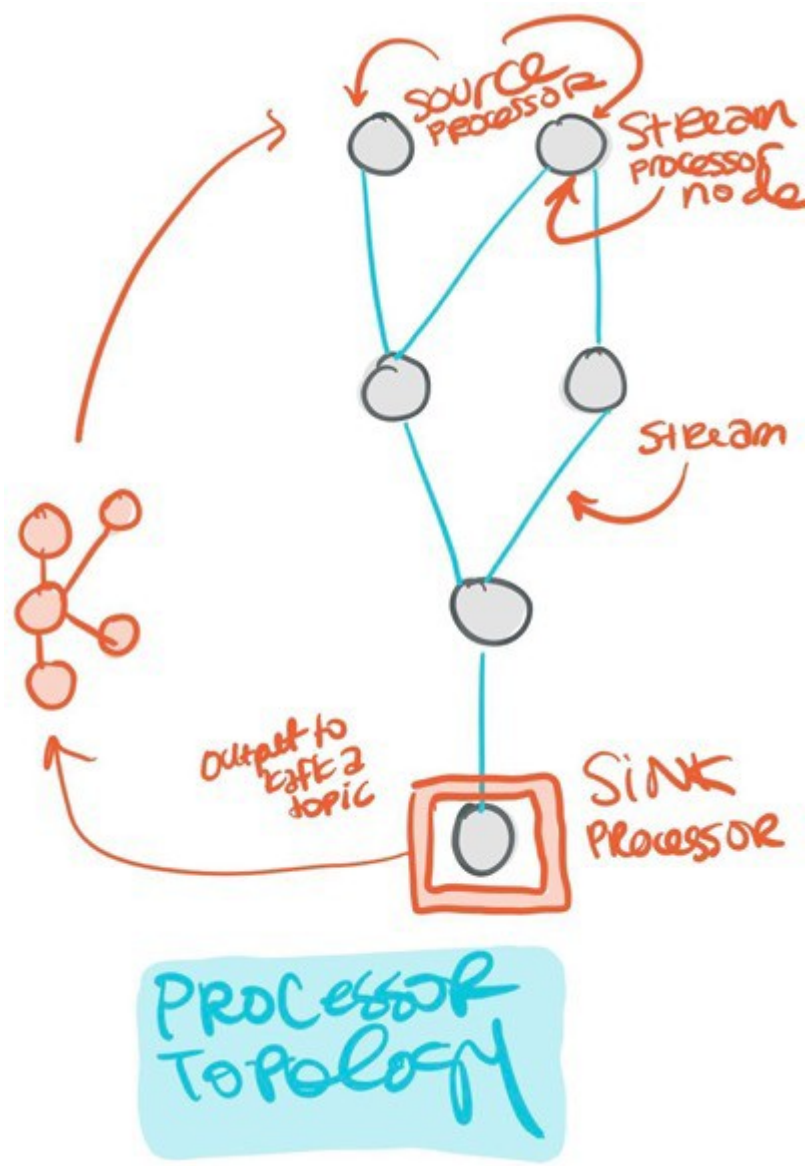
ksqlDB is deployed as a cluster of servers. To appropriately size our cluster, factors that impact server processing capabilities, such as query complexity and the number of concurrent queries running, should be considered. Plan for capacity around CPU utilization, good network throughput, and SSDs. As ksqlDB compiles to Kafka Streams (more on this soon), ksqlDB keeps the same fault tolerance. The ksqlDB cluster load balances and fails over between server nodes.

Another tidbit of advice is to not think of deploying ksqlDB as big clusters, but instead adhere to a per-use-case-per-team rule. This is very similar to the concept of database per use case. Ensuring proper resource isolation is important for the success of our deployment.

Kafka Streams

Kafka Streams, a part of the Apache Kafka project, is a client library built for Kafka to

allow us to process our event data in real time. Kafka Streams enables resilient stream processing operations like filters, joins, maps, and aggregations. It also gives us the option to perform stateful stream processing by defining the underlying topology.



For any given stream processing application, data generally arrives from Kafka in the form of one or more Kafka topics to an initial source processor that generates an input stream for the processing to begin. Next, the downstream stream processor nodes transform the streams of data as specified by the application. This may be a single step or multiple steps. The sink processor then supplies the completely transformed data back into a Kafka topic.

How Kafka Streams differs from ksqlDB

ksqlDB is actually a Kafka Streams application, meaning that ksqlDB is a completely different product with different capabilities, but uses Kafka Streams internally. Hence, there are both similarities and differences.

- Deployment: Unlike ksqlDB, the Kafka Streams API is a library in your app code!

Thus, the main difference is that ksqlDB is a platform service while Kafka Streams is a customer user service. You do not allocate servers to deploy Kafka Streams like you do with ksqlDB. You do need to allocate server (or container) resources to deploy your applications that use Kafka Streams under the hood.

- **User Experience:** Kafka Streams runs tasks in parallel for the relevant partitions that need to be processed, and the complexity and amount of partitions utilized will increase your CPU utilization. Since we are working with writing an application and deploying our code, it's a totally different user experience from that of ksqlDB. Rather, Kafka Streams is ultimately an API tool for Java application teams that have a CI/CD pipeline and are comfortable with distributed computing.

A side-by-side comparison of ksqlDB and Kafka Streams

To fully grasp the difference between ksqlDB and Kafka Streams—the two ways to stream process in Kafka—let's look at an example. While we wouldn't see the following fraud detection use case in production, it gives us an idea of the additional lines of code necessary in Kafka Streams to get the same output from ksqlDB.

In this example, we are reading from a payments topic, analyzing each message for fraud. If the probability of it being fraudulent is greater than 0.8, then the message is written to the fraudulent_payments topic.

ksqlDB example

We are creating a stream with the CREATE STREAM statement that outputs a Kafka topic for fraudlent_payments. We SELECT the fraudProbability(data) from the payments stream where our probability is over 80% and publish it to the fraudlent_payments stream. An important note about the fraudProbability function: it is actually a user-defined function (UDF)! Scalar and aggregate UDFs were released as a part of Confluent Platform 5.0, and you can read about some examples on how to implement them in this blog post. These UDFs provide a crossover between both the Java and SQL worlds, allowing us to further customize our ksqlDB operations.

```
CREATE STREAM fraudulent_payments AS
 SELECT fraudProbability(data) FROM payments
 WHERE fraudProbability(data) > 0.8;
```

Now let's consider what we have to do differently using Kafka Streams to achieve the same outcome. We have to understand the API, be comfortable enough with Kafka to create streams from the Java context, write the filter, point to our BOOTSTRAP_SERVER, and execute, among other tasks. This is a bit more heavy lifting for a basic filter.

Kafka Streams example

```
// Example fraud-detection logic using the Kafka Streams API.
object FraudFilteringApplication extends App {

  val builder: StreamsBuilder = new StreamsBuilder()
  val fraudulentPayments: KStream[String, Payment] = builder
    .stream[String, Payment]("payments-kafka-topic")
    .filter((_ ,payment) => payment.fraudProbability > 0.8)
  fraudulentPayments.to("fraudulent-payments-topic")

  val config = new java.util.Properties
  config.put(StreamsConfig.APPLICATION_ID_CONFIG, "fraud-filtering-app")
  config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092")

  val streams: KafkaStreams = new KafkaStreams(builder.build(), config)
  streams.start()
}
```

This might actually be what we want though. There is an engineering tradeoff here between ease of use and customization. For broadening stream processing usage with clusterized deployment, ksqlDB makes sense. If we want to design more complex applications, we can do so with the Kafka Streams API. The two flavors of Streams APIs: Processor API (imperative)— low level and customizable, and the Streams API (functional) with built-in abstractions and stateless and stateful transformations, give us the ability to build what we want how we want. When we opt in for a SQL-flavored abstraction layer, we naturally lose some customization power. It really just comes down to what works best for our use case, resources, and team aptitude.

When to choose ksqlDB and when to choose Kafka Streams

Think of ksqlDB as a specialized database for event streaming applications.

In addition, some teams are leveraging ksqlDB to validate their Kafka Streams logic. This can be productive if development teams want to invest into an application or work out conceptual kinks without having to build it out from brass tacks.

The biggest question when evaluating ksqlDB and Kafka Streams is which to use for our stream processing applications and why. The answer boils down to a composite of resources, team aptitude, and use case.

With regard to use case, ksqlDB is a great place to start evaluation. If our use case isn't supported by ksqlDB, we should try to write a UDF. If neither of these are feasible and we have a use case that rules out ksqlDB as a viable option, then consider Kafka Streams. If we need to create an end-to-end stream processing application with highly imperative logic, the Streams API makes the most sense as SQL is best used for solving declarative-style problems. If we need to join streams, employ filters, and perform aggregations and the like, ksqlDB works great.

The future of ksqlDB is bold. It is a fast-moving project that is bound to become a powerful part of the Confluent Platform. We believe that ksqlDB represents a powerful new category of stream processing infrastructure. More robust database features will be added to ksqlDB soon—ones that truly make sense for the de facto event streaming database of the modern enterprise. For a new data paradigm where everything is based upon events, we need a new kind of database for it. We are truly excited for the future of stream processing with the Confluent Platform, and we hope you are too!

Getting started

Ready to check ksqlDB out? Head over to ksqldb.io to get started. Follow the quick start, read the docs, and check out the project on Twitter!

Let us know what you think is missing or ways it can be improved—we invite your feedback within the community.

Shift Left: Bad Data in Event Streams, Part 2

OCT 11, 2024

Event design plays a big role in your ability to fix bad data in your streams. But if you've wrecked a stream with bad data (i.e., it's unavoidably contaminated), you'll need to employ a "rewind, rebuild, and retry" strategy.

ADAM BELLEMARE

Shift Left: Bad Data in Event Streams, Part 1

OCT 4, 2024

At a high level, bad data is data that doesn't conform to what is expected, and it can cause serious issues and outages for all downstream data users. This blog looks at how bad data may come to be, and how we can deal with it when it comes to event streams.

ADAM BELLEMARE