CONFLUENT

Whitepaper

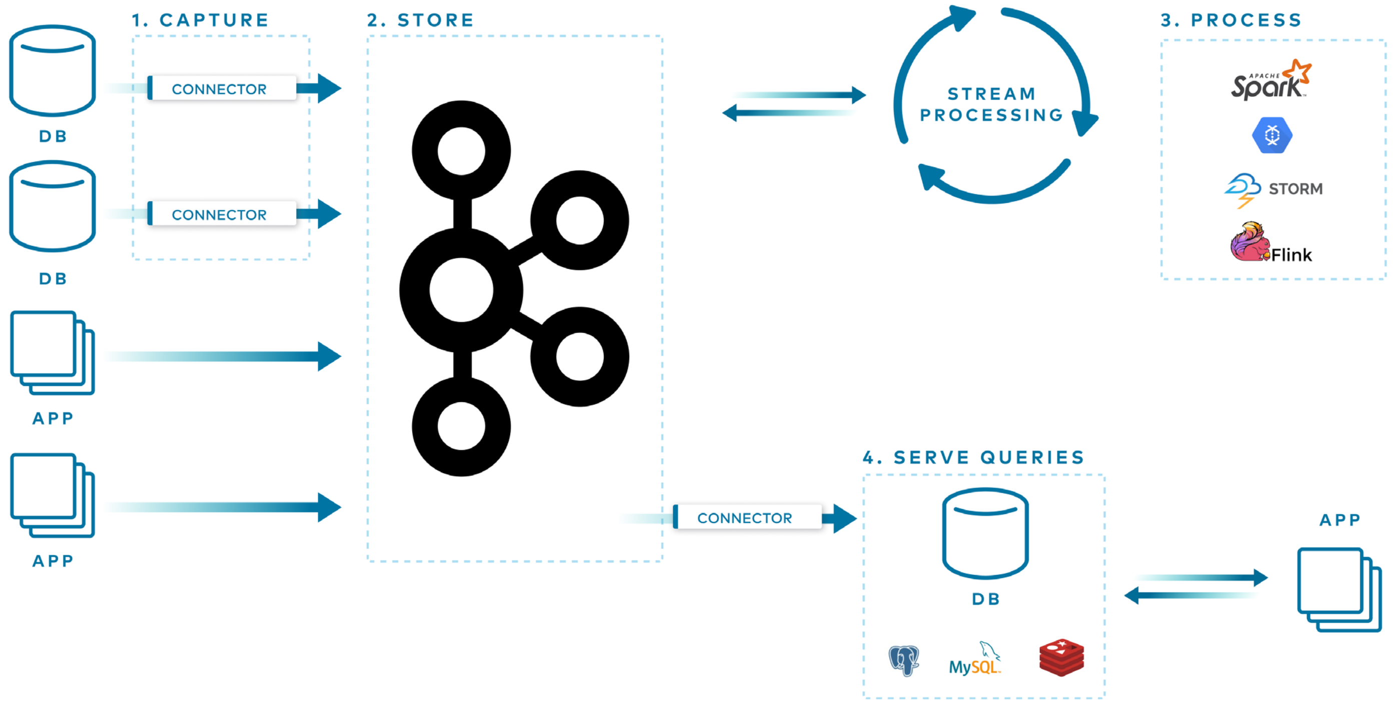
Building Stream Processing Applications with Confluent

How to process, manipulate, and enrich streams of events using ksqlDB

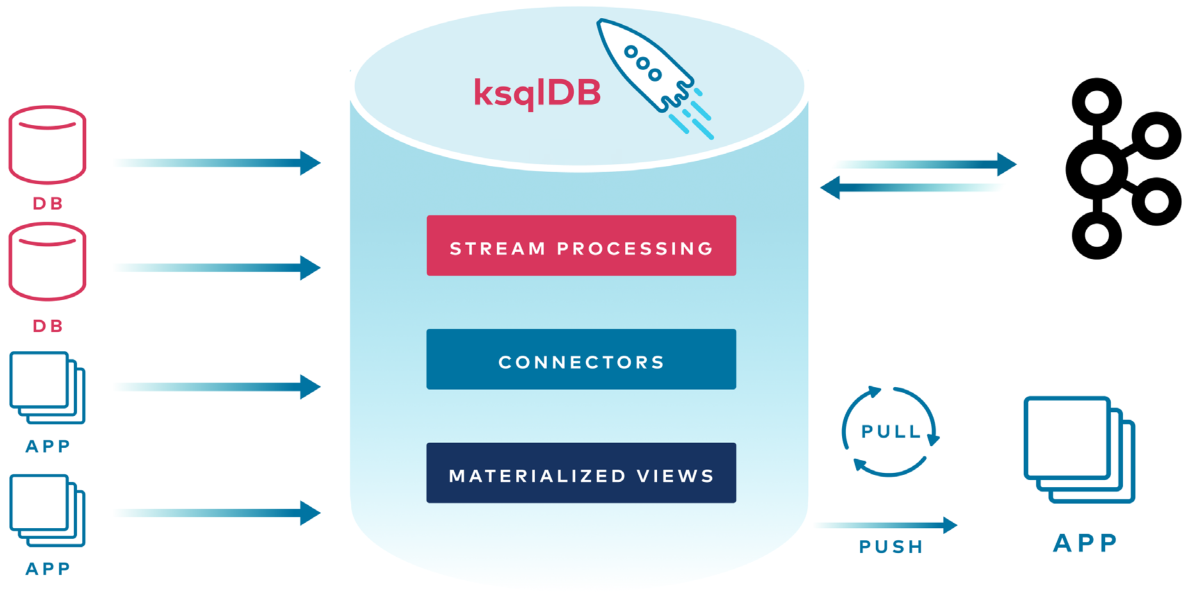
Introduction

Customers demand immediacy in every area of their lives. Expectations for how they shop, bank, hail rides—and do just about anything else—have been altered dramatically over the last decade by tech upstarts that have built on modern platforms and are racing to get scale. To fend off these disruptors and compete in a digital-first world, businesses must create rich, real-time customer experiences and transition to more sophisticated data-driven operations. These types of experiences are powered by stream processing applications, which continuously process and immediately react to data in motion rather than periodically querying data at rest.

However, developing the underlying architecture to support stream processing applications can be an immensely challenging exercise. In practice, several distributed systems need to be integrated together to capture streams of events from external sources, store them, process them, and serve queries against them to applications. Almost all stream processing architectures today are piecemeal solutions cobbled together from different projects. This equates to platform teams needing to understand three to five separate ways of scaling, monitoring, securing, and debugging their data infrastructure.



When looking at all of these pieces, it's easy to question if this is all worth it. Confluent believes that building stream processing applications should be as easy as building CRUD applications on top of a regular database.



That's why Confluent developed ksqlDB, the database purpose-built for stream processing applications. With ksqlDB, the complex architecture once required to build stream processing applications is reduced to two components that are both fully managed as part of Confluent Cloud: ksqlDB and Apache Kafka®. Rather than needing to work across several complex systems, developers can leverage ksqlDB to unlock real-time business insights and customer experiences with just a few SQL statements.

Unlocking the value of real-time data with ksqlDB

While stream processing architectures previously required separate systems for event capture, stream processing, and query serving, ksqlDB provides one mental model for working across the entire stack:

- **Stream processing**: Execute continuous computations over unbounded streams of events, ad infinitum
- **Embedded connectors**: Leverage pre-built connectors to easily move data from existing data systems in and out of ksqlDB
- **Materialized views**: Store the results of materialized views to serve queries to applications

ksqlDB's one and only dependency is Kafka, which is used to store streams of events. With fewer moving parts in the underlying architecture, developers can more easily build stream processing applications, and platform operators can scale, monitor, and secure their architecture with a more uniform approach. Ultimately, this provides enterprises a number of advantages by unlocking a host of new design techniques to build real-time applications that were previously out of reach.

Benefit #1: Enable the business to innovate faster and operate in real-time by processing data in motion rather than data at rest

Databases are designed to query and store data at rest, allowing processing only at a point in time, and computing an answer which is immediately out of date as the business continues to evolve around it. This means each application acts as an isolated island of stored data, disconnected from the rest of the business. This is simply too slow to serve the real-time nature of modern customer experiences and operational needs.

Processing data in motion allows enterprises to tap into streams of data being generated anywhere in the company and continually process it, enabling enterprises to get value from their data faster. ksqlDB does so by modeling data in motion through streams, which are immutable, append-only collections that are useful for representing a series of historical facts. ksqlDB can also handle out-of-order data based on timestamps put into the data, rather than processing everything in the order received.

All of ksqlDB's programming model is built around manipulating these streams, allowing application development teams to seamlessly create real-time innovative customer experiences and fulfill data-driven operational needs.

Benefit #2: Simplify your stream processing architecture

Previously, development teams were forced to stitch together multiple distributed systems to support stream processing applications that harness data in motion. ksqlDB doesn't require separate clusters for event capture, processing, or query serving—meaning it has very few moving parts. There is one model for working across the entire stack for scaling, monitoring, and security. So what does this mean for the business? Less infrastructure to maintain, fewer vendor contracts, and fewer relationships to manage.

Benefit #3: Empower developers to start building stream processing applications with ksqlDB's simple SQL syntax

Often, the logic used in stream processing is complex or cumbersome to express in Java, deterring many development teams. SQL, on the other hand, is a high-level language that's easier to use because it's declarative. ksqlDB provides much of the functionality of other stream processing frameworks through a familiar, lightweight SQL syntax. Developers can build stream processing applications with the same ease and familiarity as they build traditional apps on a standard database.

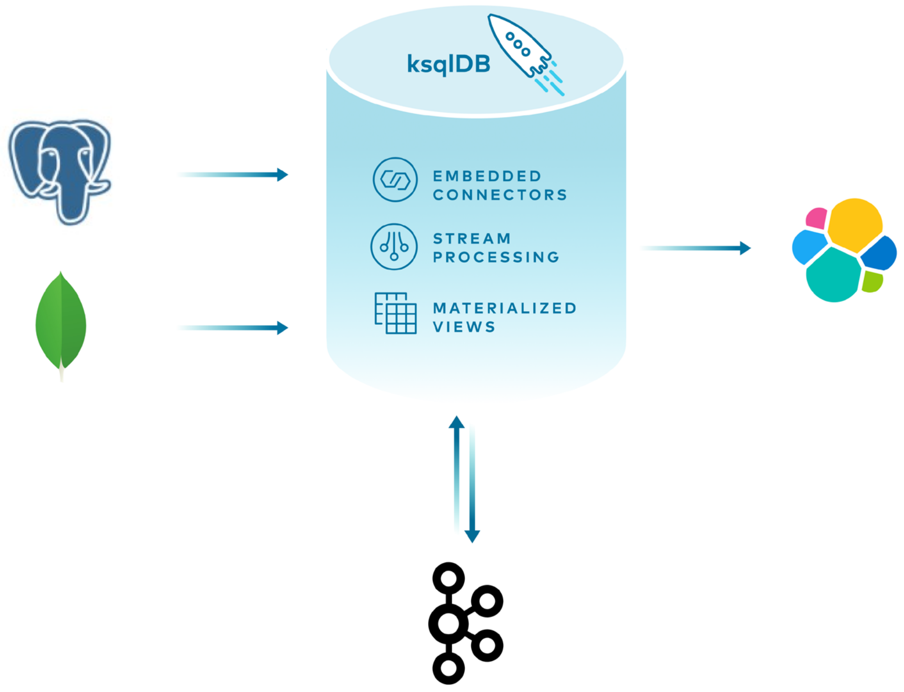
Data in motion supports new technical use cases across the enterprise

ksqlDB supports a number of real-time applications and use cases that can change how a business operates. Below are three of the most popular use cases for this new database.

Streaming ETL: Manipulate in-flight data to connect arbitrary sources and sinks

A streaming ETL pipeline, sometimes called a "streaming data pipeline," is a set of software services that ingests events, transforms them, and loads them into destination storage systems. It's often that data needs to be moved from one place to another as soon as it is received, but changes need to be made to the data as it is transferred. This may include stripping personal identifiable information or something more complex like enriching events by joining them with data from another system. A streaming ETL pipeline enables streaming events between arbitrary sources and sinks, and helps make changes to the data while it's in flight. Typically this would include gluing a bunch of services together, which can pose as a challenge.

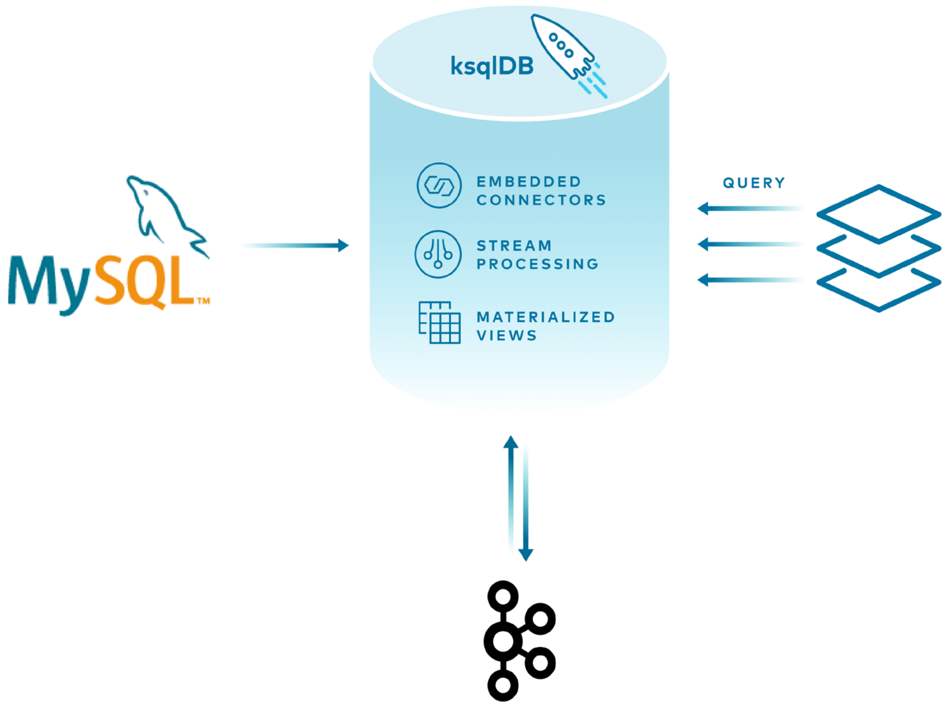
ksqlDB helps streamline how you write and deploy streaming data pipelines by boiling it down to just two things: storage (Kafka) and compute (ksqlDB).



Using ksqlDB, it is simple to run any connector by embedding it in ksqlDB's servers. Then easily transform, join, and aggregate all streams together by using a coherent, powerful SQL language. This gives the enterprise a slender architecture for managing the end-to-end flow of its data pipeline.

Materialized cache: Build and serve incrementally updated stateful views

A materialized view, sometimes called a "materialized cache," is an approach to precomputing the results of a query and storing them for fast read access. In contrast with a regular database query, which does all of its work at read time, a materialized view does nearly all of its work at write time. This is why materialized views can offer highly performant reads.

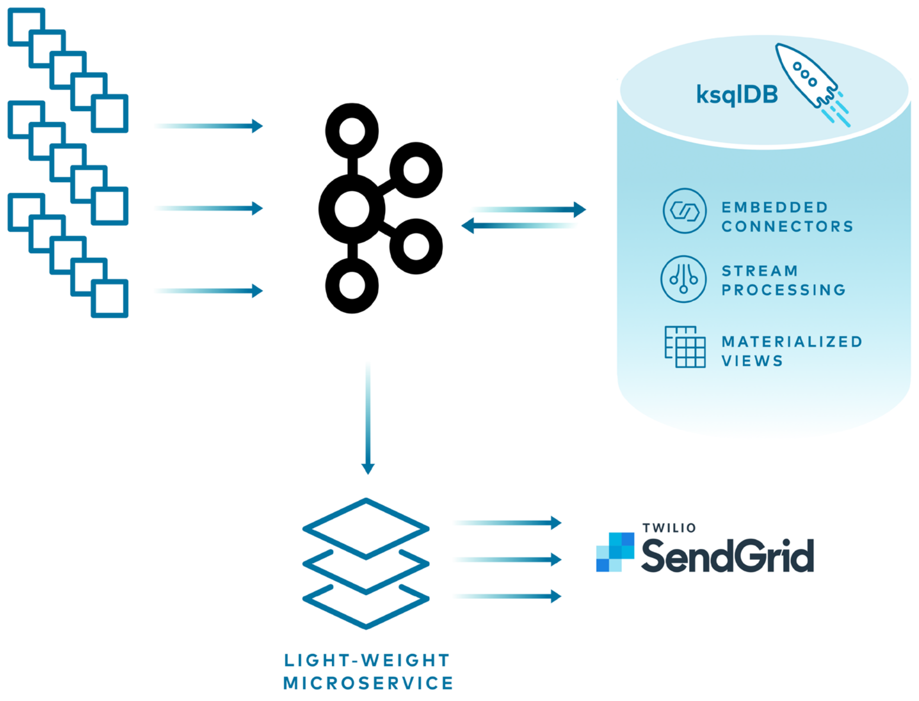


A standard way of building a materialized cache is to capture the changelog of a database and process it as a stream of events. This creates multiple distributed materializations, each best suited to each application's query patterns. This results in running multiple systems, which can be a lot to manage—the database, Kafka clusters, connectors, the stream processor, and another data store. It's challenging to monitor, secure, and scale all of these systems as one. Again, ksqlDB helps slim the architecture down to just storage (Kafka) and compute (ksqlDB).

Event-driven microservice: Trigger changes based on observed patterns of events in a stream

An event-driven microservice is a pattern in which a piece of code only communicates with the outside world through streams of events. This technique can dramatically simplify an architecture because each microservice only receives and emits information from clearly defined communication channels. Because state is localized within each microservice, complexity is tightly contained.

Scaling stateful services is challenging. Coupling a stateful service with the responsibility of triggering side effects makes it even harder. It's up to the development team to manage both as if they were one, even though they might have completely different needs. If there are changes in how side effects behave, development teams must redeploy the stateful stream processor. ksqlDB helps simplify this by splitting things up: stateful stream processing is managed on ksqlDB, while side effects run inside the stateless microservice.



Using ksqlDB, you can isolate complex stateful operations within ksqlDB's runtime. This way the application can stay simple because it is stateless. It merely reads events from a Kafka topic and executes side effects as needed.

Understanding the constructs of ksqlDB

Because it was specifically designed to support stream processing applications, ksqlDB has a number of features that are distinct from traditional databases. Let's take a deep dive into some of the key features that define the constructs of ksqlDB.

Streams and tables

It's foundational to understand ksqlDB's two constructs for representing collections of data and how they're connected to one another: streams and tables. Both operate under a simple key-value model. Streams are immutable, append-only collections. They're useful for representing a series of historical facts and seeing data in motion. Adding multiple events with the same key means they are simply appended to the end of the stream. In the example below, the locations of various vehicles are represented in a stream as they occur in real time. Multiple locations from the same vehicle simply get appended to the end.

Example stream

vehicleID	latitude	longitude
a1rc4r	43.683117	-79.611421
wh4rfx	51.509855	-0.123746
a1rc4r	43.642826	-79.387123
ffk1t3	45.716540	-121.526191
wh4rfx	51.503801	0.048346

Tables are mutable collections. They represent the latest version of each value per key. They're helpful for modeling change over time and are often used to represent aggregations. Continuing our previous example, the latest locations for each vehicle are shown on the table and updated in real-time as more recent data is collected.

Example table

vehicleID	latitude	longitude
a1rc4r	43.642826	-79.387123
wh4rfx	51.503801	0.048346
ffk1t3	45.716540	-121.526191

Persistent queries

Persistent queries are used to process data.

Persistent queries can execute continuous computations over unbounded streams of events, ad infinitum. Transform, filter, aggregate, and join collections together to derive new collections or materialized views that are incrementally updated in real time as new events arrive.

Push and pull queries

Push and pull queries are used to query data.

Push queries can subscribe to a query's result as it changes in real time. When new events arrive, push queries emit refinements, which quickly react to new information. They're a perfect fit for asynchronous application flows.

Pull queries fetch the current state of a materialized view. Because materialized views are incrementally updated as new events arrive, pull queries run with predictably low latency. They're a great match for request/response flows.

Case Study: Ticketmaster

"Each technology [only] has to be connected to Kafka, and through that, they're effectively connected to each other."

— Vice President of Engineering and Data Science, Ticketmaster

Ticketmaster, the leading ticket sales and distribution platform, made a huge leap from an architecture built on monolithic applications to a microservices architecture—a strategic DevOps transformation that made data management both more critical and more challenging.

The team needed visibility into each specific component, but rather than try to integrate all the components of 300+ applications hosted on more than 4,500 virtual machines, Ticketmaster set up a centralized data lake for application telemetry and fed it using a set of stream processing pipelines, in partnership with Confluent.

When tickets go on sale, there are heavy traffic spikes that place an intense load on certain systems. In order to drive better business decisions, different engineering and data science teams use ksqlDB for data enrichment. Now anyone can submit a query and get results instantaneously. Ticketmaster is also using ksqlDB for anomaly detection, so they can prioritize ordinary customers over bad actors who are abusing the system to resell tickets at higher prices. By building a machine learning model, Ticketmaster has combated this abuse and reacted quickly when malicious users change their strategy.

By partnering with Confluent, Ticketmaster takes the friction out of software development and enables the business to deliver better digital customer experiences.

Read the full case study

Leveraging a complete platform for data in motion with Confluent

In addition to ksqlDB, Confluent includes other crucial components that together provide a complete platform for harnessing data in motion:

- **Rich pre-built ecosystem**: When partnering with Confluent, you tap into our extensive connector ecosystem to fulfill the vision of creating a central nervous system throughout the enterprise. From popular data warehouses to cloud-based object storage, Confluent offers a portfolio of 120+ pre-built connectors to help enterprises integrate a diverse set of data sources and sinks. Seamlessly modernize your infrastructure stack by streaming valuable legacy data to cloud-native data systems to accelerate, simplify, and de-risk your migration efforts.

- **Data compatibility and governance**: A key component for supporting data in motion is to enable broad compatibility between applications connecting to Confluent. That is why Confluent developed Schema Registry and Schema Validation. Schema Registry is a central repository with a RESTful interface for developers to define standard schemas and register applications to make sure they adhere to those schemas. Data compatibility at scale also often requires a more centralized approach in case individual applications do not register with Schema Registry. Schema Validation delivers a programmatic way of validating and enforcing schemas directly on the broker so all data inbound to a topic is robust and compatible with downstream applications.

- **Industry-leading reliability and security**: Confluent provides highly available Kafka clusters backed by a 99.95% SLA that are always on the latest stable version of Kafka and get upgraded and patched in the background like any other cloud-native service. Confluent secures existing Kafka environments with encryption at rest and in-transit, role-based access controls, Kafka ACLs, and SAML/SSO for authentication. More granular control with private networking and BYOK for your Kafka environments is available as well.

With these capabilities in hand, enterprises can feel well-equipped to build rich, real-time customer experiences and data-driven operations that change their business.

To learn more, visit https://www.confluent.io/product/ksql/