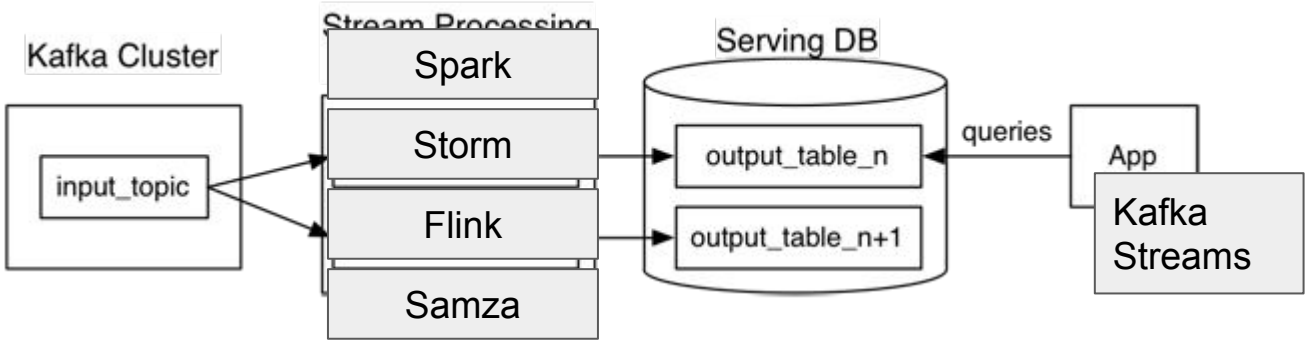
Kafka Streams - concepts

Kappa architecture: a Single Processing Framework



Source: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

Components of the Kappa architecture:
Apache Kafka (or another similar system) : retains the full log of the data that you need to re-process.
Streaming Processing Framework: When you want to re-process: start a second instance of your streaming job and start processing from beginning the retained data, but direct it to a new output table.

Streams

Messages in a stream are constantly arriving and being added to a Kafka topic

Messages in a stream are independent of each other, but arrive in a time ordered fashion

You can process the messages independent of each other

Examples: orders arriving in an website, twitter

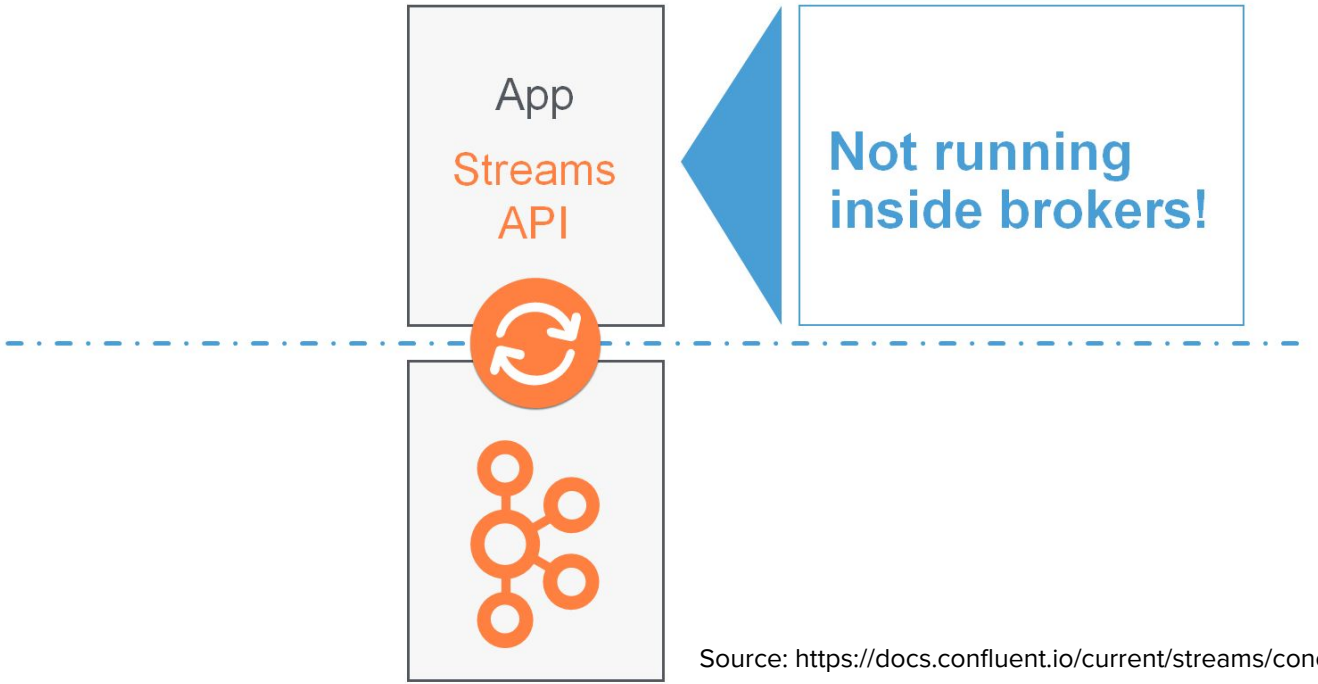
Streams Processor

A stream processor is anything that takes continuous streams of data from input topics, performs some processing on this input and produces a stream of data to output topics (or external services, databases, the trash bin, wherever really...).

2 main processing modes in Kafka:

- **simple processing directly** with the producer/consumer APIs (SMT's for Kafka Connect)
- **more complex transformations** like joining streams together, Kafka provides a integrated **Streams API** library (from Kafka 0.10.0 ->). Kafka Streams is a set of application API (currently in Java & Scala) that seamlessly integrates **stateless and stateful processing.**
  - **Processor API**
  - **Streams DSL (built on top of Streams Processor API) - Java based, Scala supported**
  - **KSQL server: write streaming queries in SQL against Apache Kafka**

Kafka Streams API



This API is intended to be used **within your own codebase**, it is **not running on a broker**.
Stream processing application doesn't run inside a broker.
Instead, it runs in a separate JVM instance, or in a separate cluster entirely.

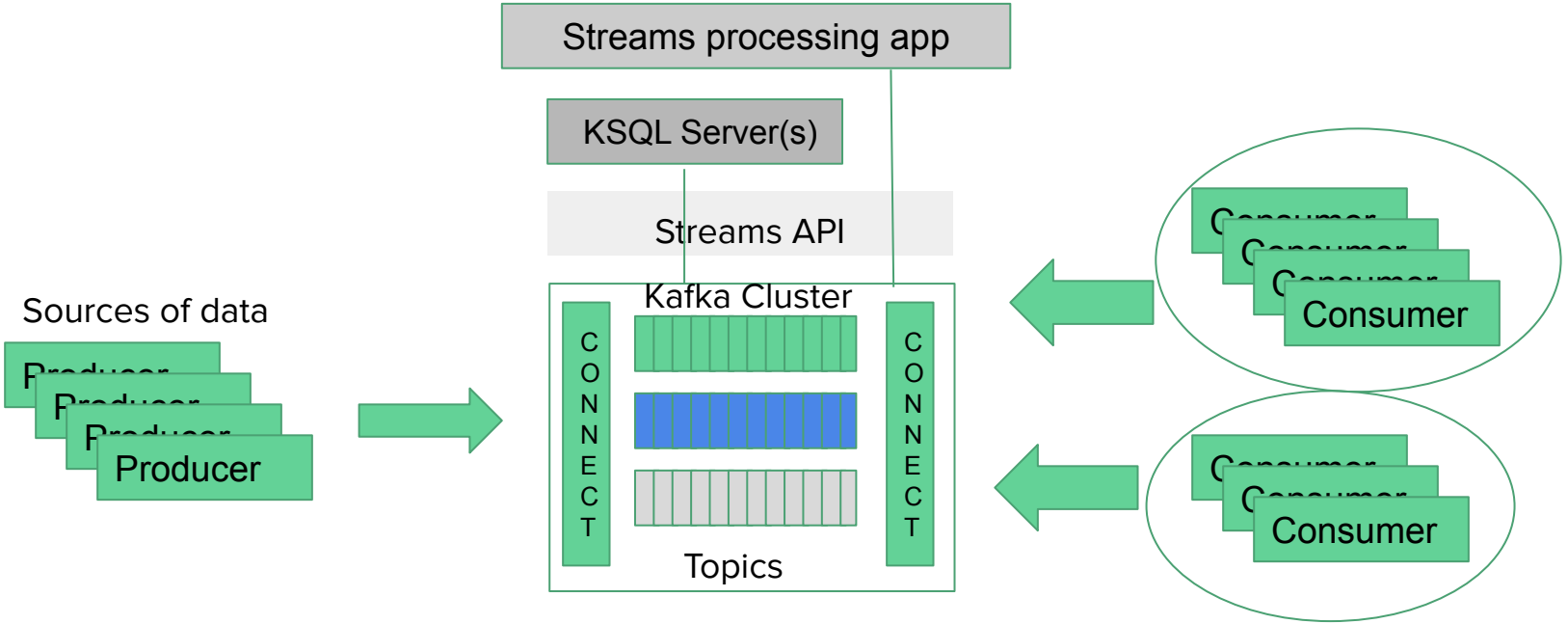
Source: https://docs.confluent.io/current/streams/concepts.html

Kafka Streams API

Despite being just a library, Kafka Streams directly addresses a lot of the hard problems in stream processing:

- Event-at-a-time processing (not microbatch) with millisecond latency;
- **Stateful processing** including distributed joins and aggregations;
- Windowing with out-of-order data;
- **Distributed processing and fault-tolerance with fast failover;**
- Reprocessing capabilities so you can recalculate output when your code changes;
- No-downtime rolling deployments;
- **3 options for developing apps: KSQL for the SQL aficionados, Streams DSL for the Java/Scala programmers, Processors API for advanced processing**

Processor API, Streams DSL and KSQL



Stream = continuous, unbounded, real time flow of events/records = Kafka Topic

Streams and Tables: KStreams, KTables, global KTables

When implementing stream processing use cases in practice, you typically need both streams and also tables. A stream can be interpreted as a series of updates for data, in which the aggregate is the final result of the table. An example use case that is very common in practice is an **e-commerce application that enriches an incoming stream of customer transactions with the latest customer profile information from a database table.**

Kafka Streams DSL:

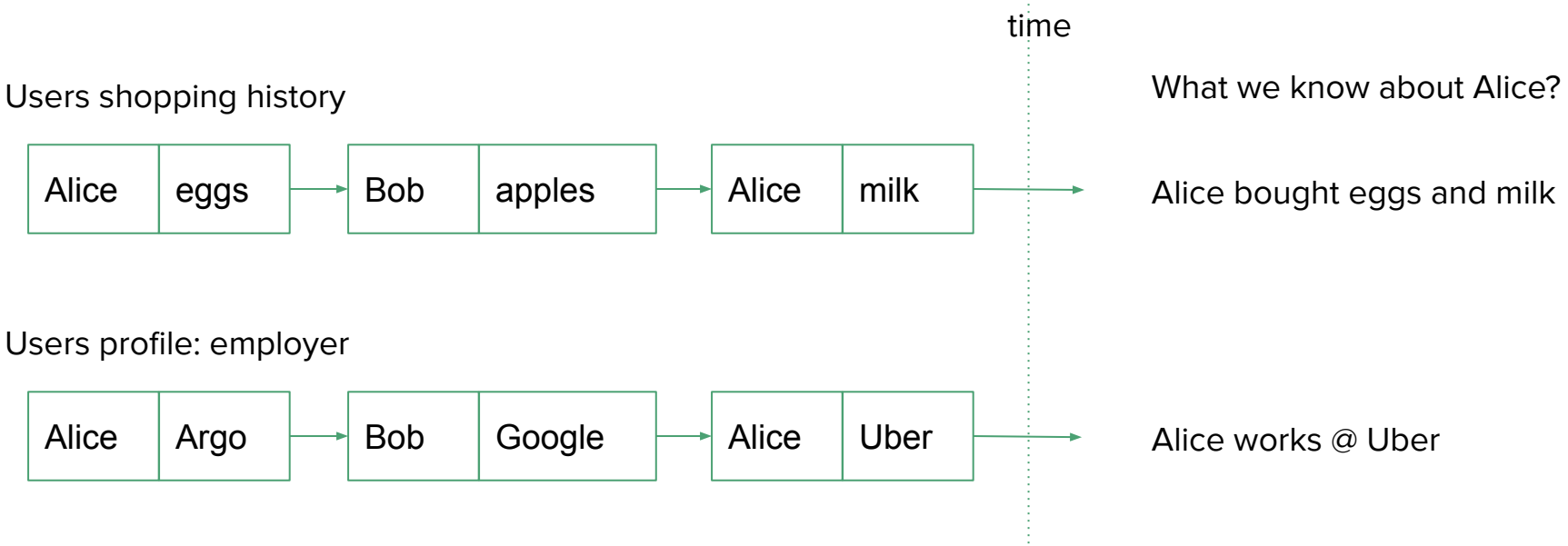
KStream is an abstraction of a record stream of Key-Value pairs, i.e., each record is an independent entity/event in the real world. Every time new data is added is an INSERT.

KTable / Global KTable is an abstraction of a stream from a primary-keyed table. Each record in this stream is an update on the primary-keyed table with the record key as the primary. Every time data is added for a key - we perform an UPDATE. **Tables are just a particular view of a stream.**

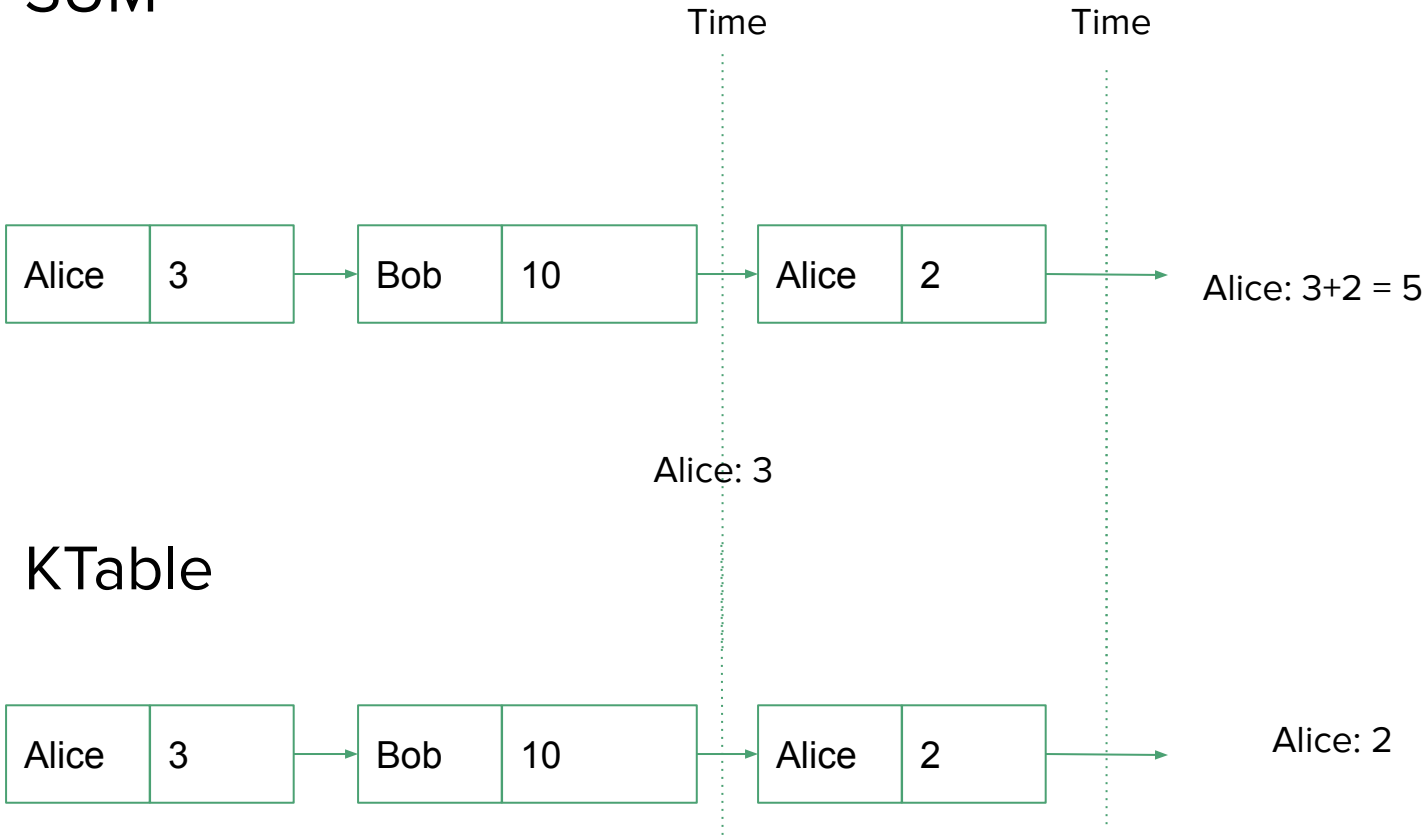
Stream - Table duality (KStreams, KTables)

User purchase history **KStream**

User Profile **KTable/GlobalKTable**

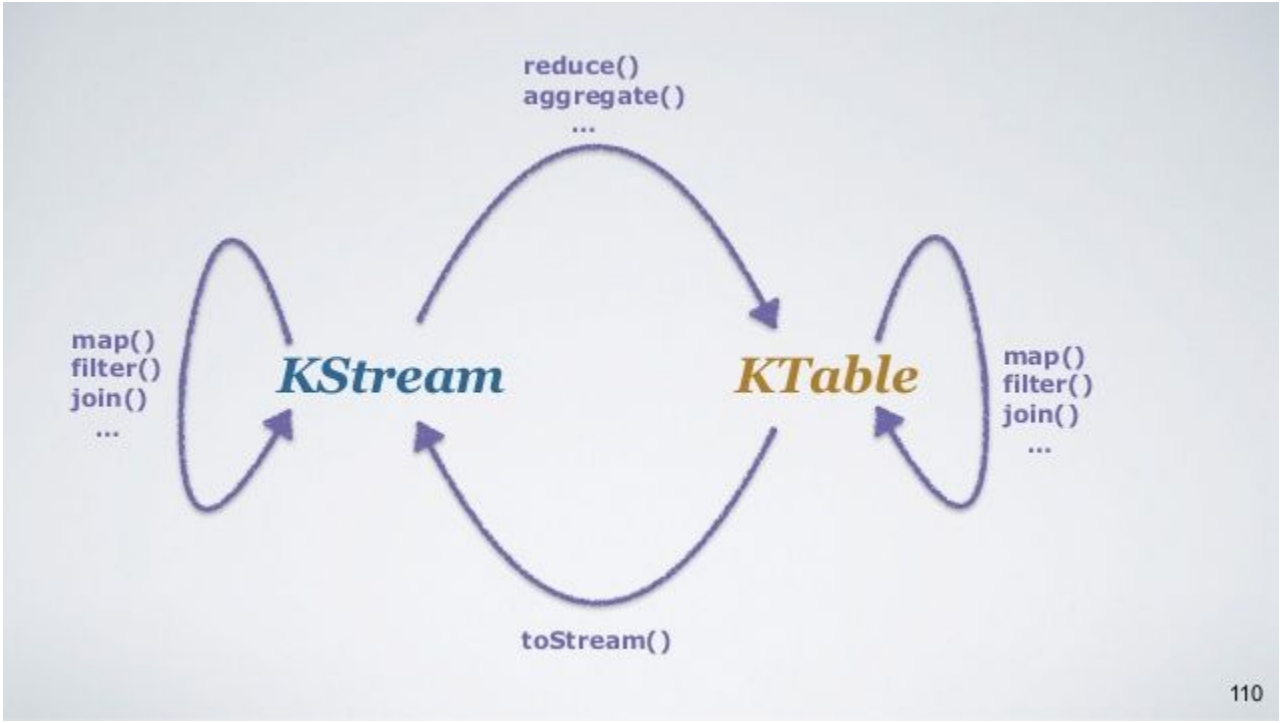


KStream.aggregate()
SUM



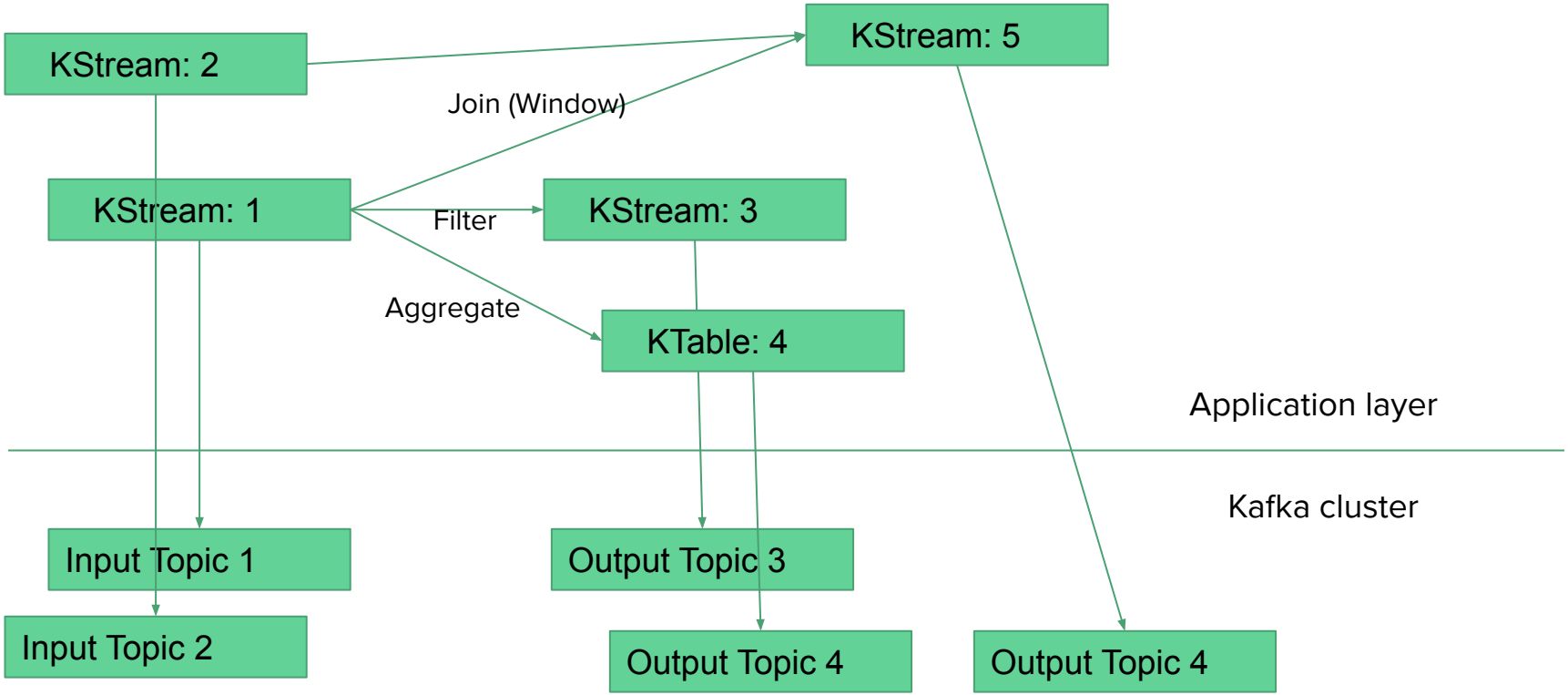
KTable

Transformations on KStreams and KTables

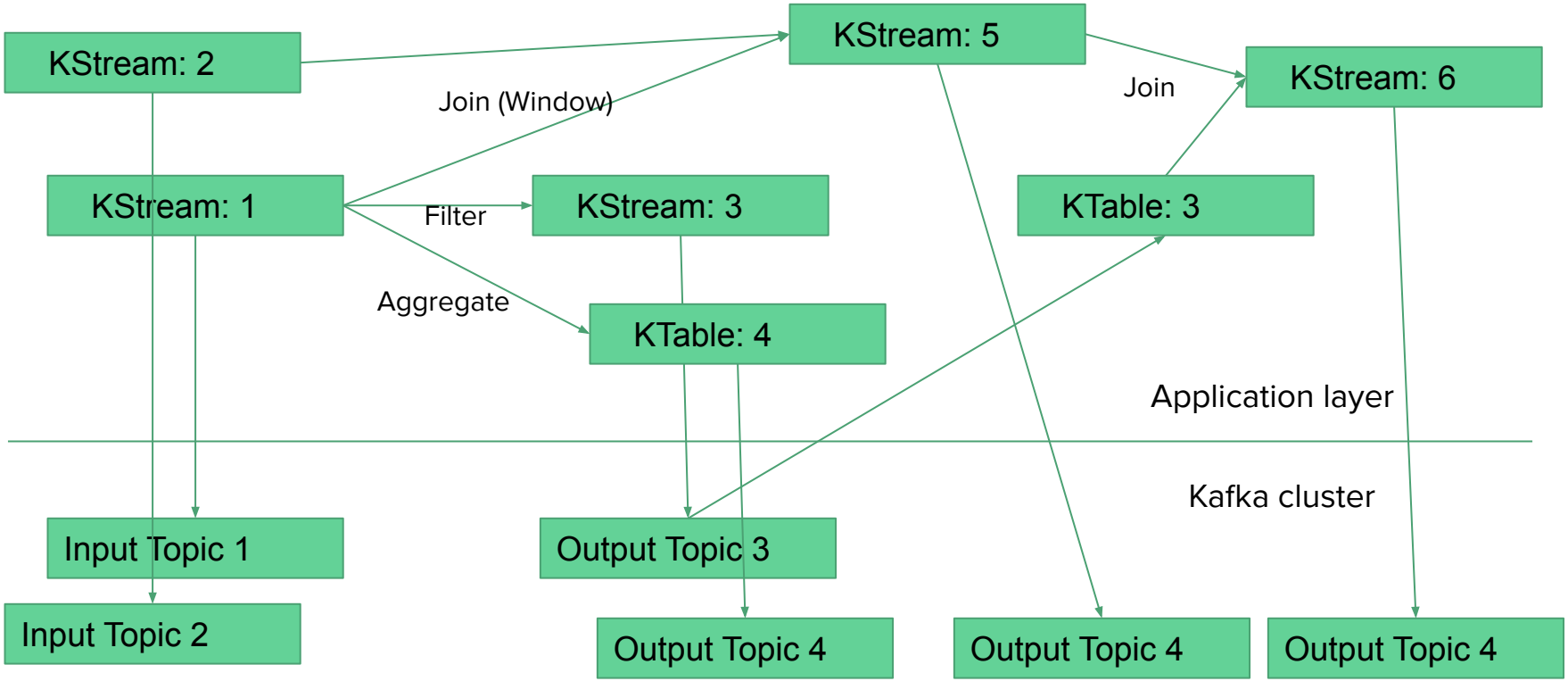


Source: https://www.slideshare.net/GuozhangWang/introduction-to-kafka-streams

Simplified :-) view of transformations



Simplified :-) view of transformations



Persistent and Non-Persistent queries

Select * => non - persistent Query

Create Stream/Table as Select => Persistent Query => Terminate Query for stopping the query to run

	Persistent	**Non - persistent**
Where does it run	Background	Foreground (goes away when you stop it)
Result of Query	Written back to Kafka	Show only in your CLI

Application fails/stops => the query is terminated.

Stateless and Stateful transformation operations

Stateless operations: imagine you wish to filter a stream for all keys starting with a particular string. In this case, Kafka Streams doesn’t require knowing the previous events in the stream.

You want to calculate a forecast of how many users you'll have in the next second on your website. You want to **average the last 10 minutes, so you need to keep a queue with the last 10 * 60 seconds** - that's the state you need to keep for your processing, and you need to update it every second, to keep the most recent 10 minutes of state. That's of course a **stateful operation**. A simpler stateful operation is just **counting the total number of page view since the beginning of the site.**

One critical difference between the two operations is that if the stream stops and you reset the system, you gotta take care of saving the state. A stateless operation does not have any state to save so its processing can continue immediately after being restarted.

Stateless vs Stateful queries

Stateless:

- Create new streams/tables from existing ones using: branch/split, filter, FlatMap, Map, GroupbyKey, SelectKey operations;
- change data format: delimited, AVRO (Schema Registry needed), JSON;

Stateful

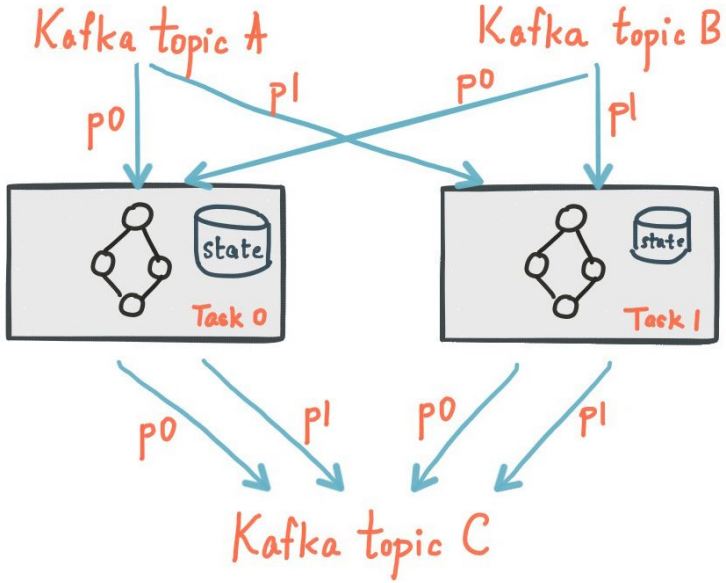
- Aggregating (e.g count)
- Joining
- Windowing (as part of aggregations and joins)

State Stores

For stateful transformations, in order to maintain the current state of processing the input and outputs, Kafka Streams introduces a construct called a **State Store**.

Local state is just data that is kept in memory or on disk on the machines doing the processing. The job can query or modify this data in response to its input. The simplest way of maintaining state might be a simple local key-value store.

State stores, offering an abstraction of a fast local Key-Value Store that can be read and written to when processing messages with Kafka Streams. These Key-Value stores can be continuously filled with new messages from a Kafka topic by defining an appropriate stream processor, so that it is now possible to quickly retrieve messages from the underlying topic.



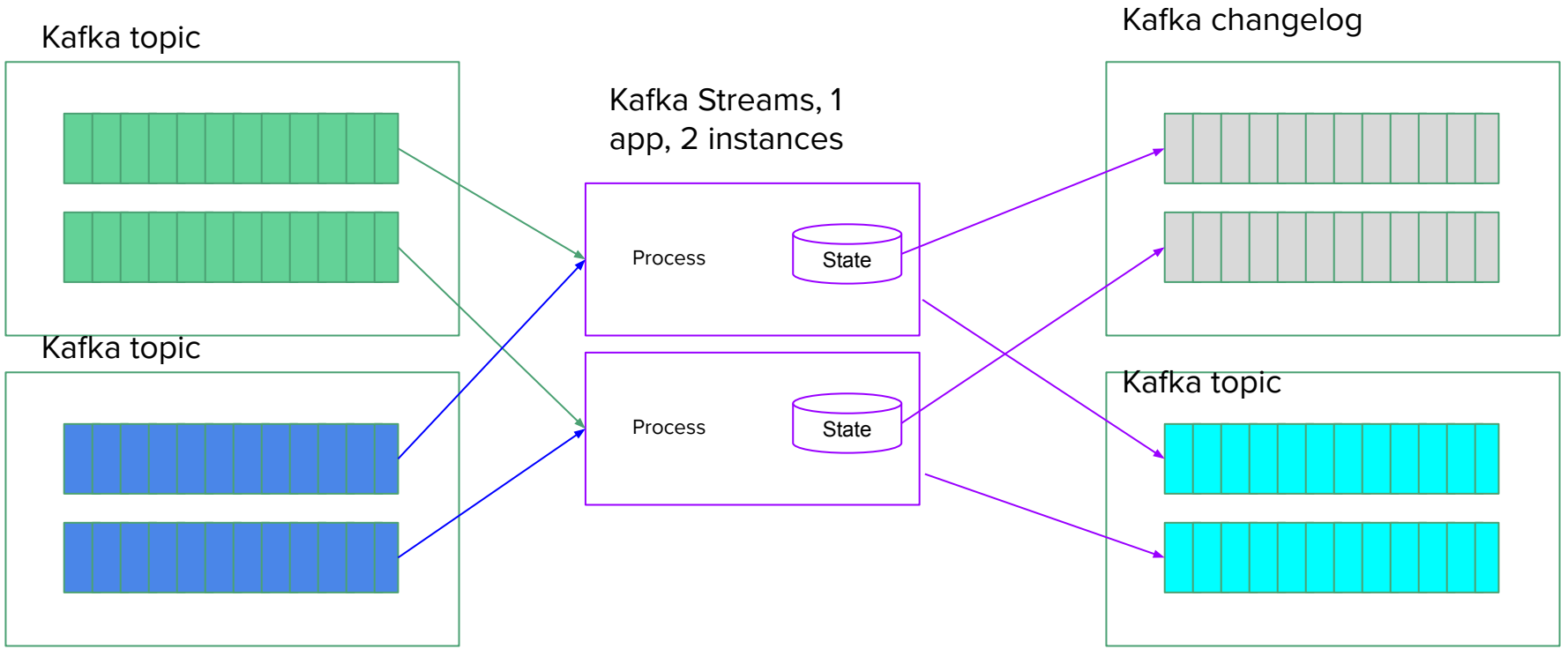
State stores

Kafka Streams creates a state store to perform the aggregation (here called metrics-agg-store), **and this state store is backed by a changelog** (effectively another internal topic) to make it fault-tolerant. The changelog topic basically keeps track of the updates made to the state store, and it is read from if the application has to recover from an interruption. In this topic, the key is a compound key made of the aggregation key (valid / invalid) and of the time window, and the value is the running count as well as an offset of the message in the input topic.

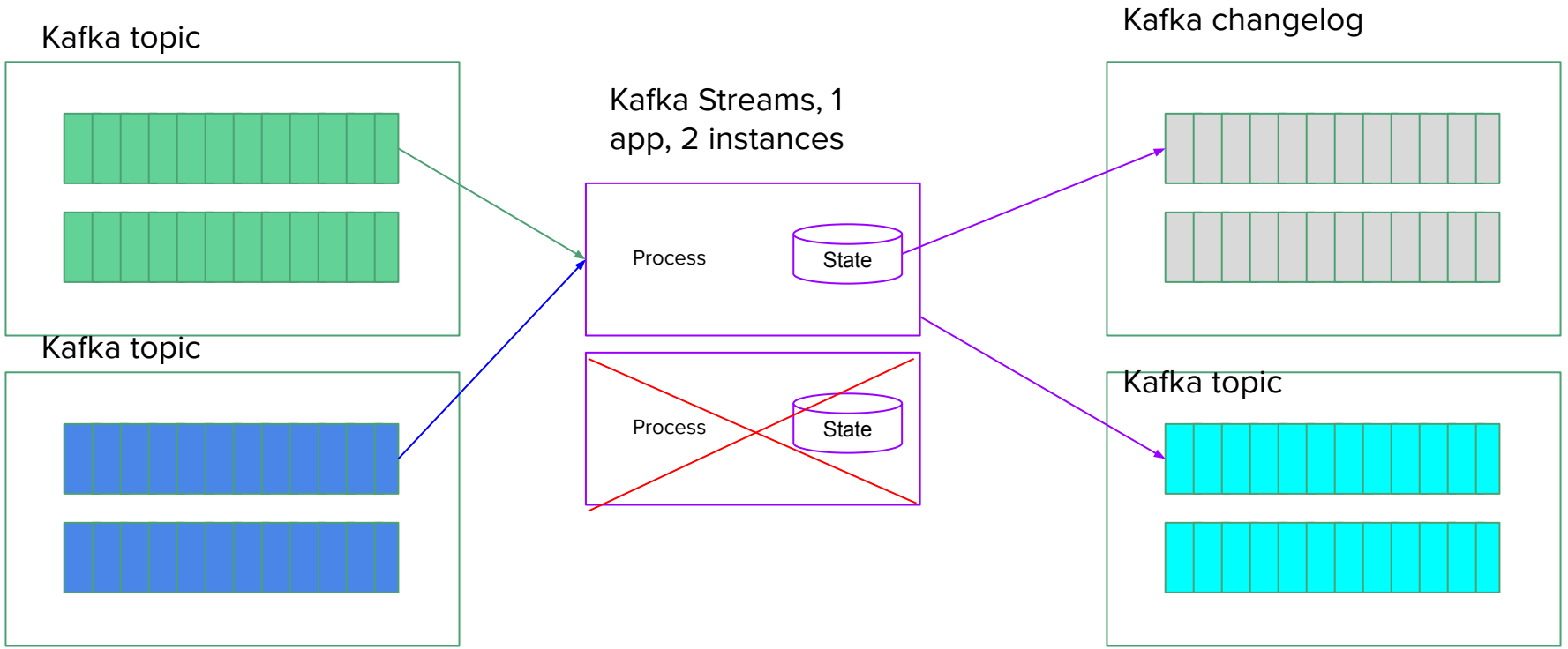
Kafka State Store is implemented with **RocksDB at application level (fast key-value lookup)** and a **Kafka internally created and compacted changelog topic (used only for fault-tolerance)**.

Fault Tolerance & Scalability of Streaming Apps

Streaming Apps instances behave just like consumers in the same group.

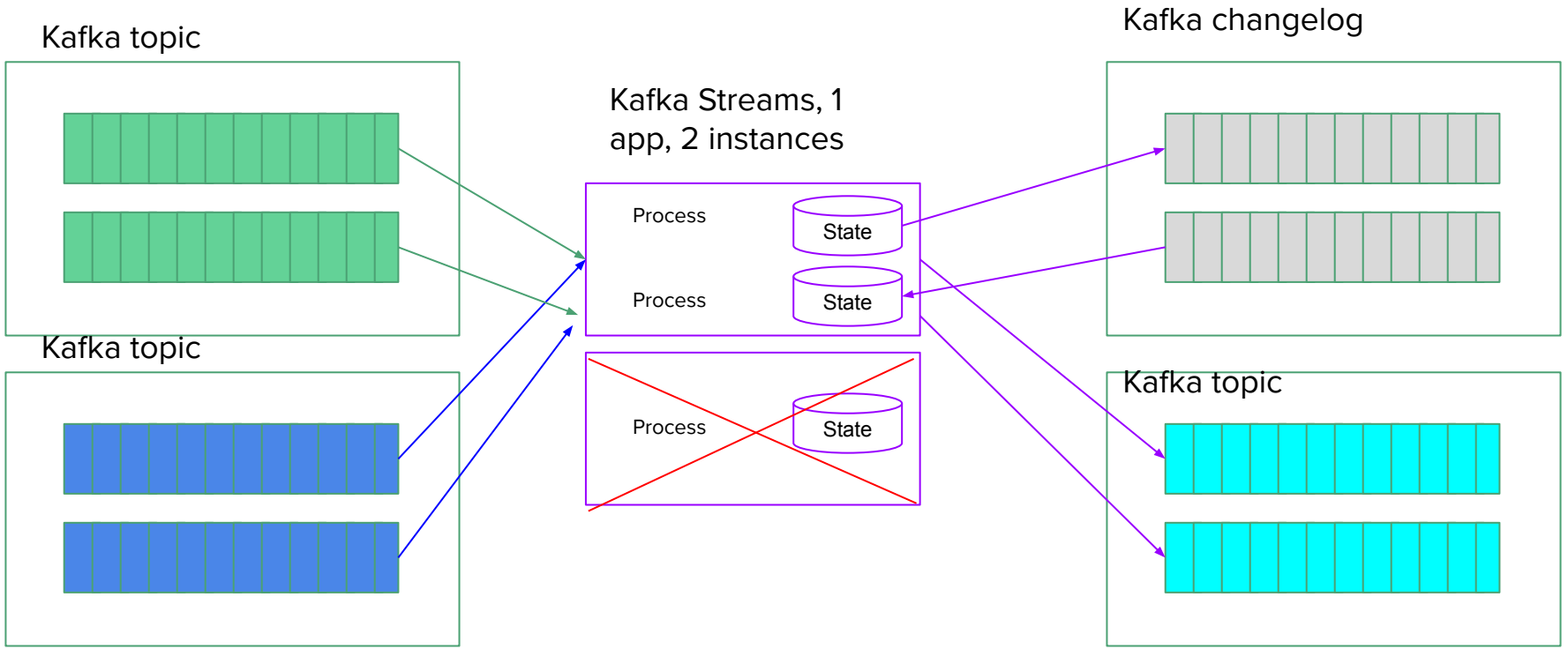


Fault Tolerance & Scalability of Streaming Apps



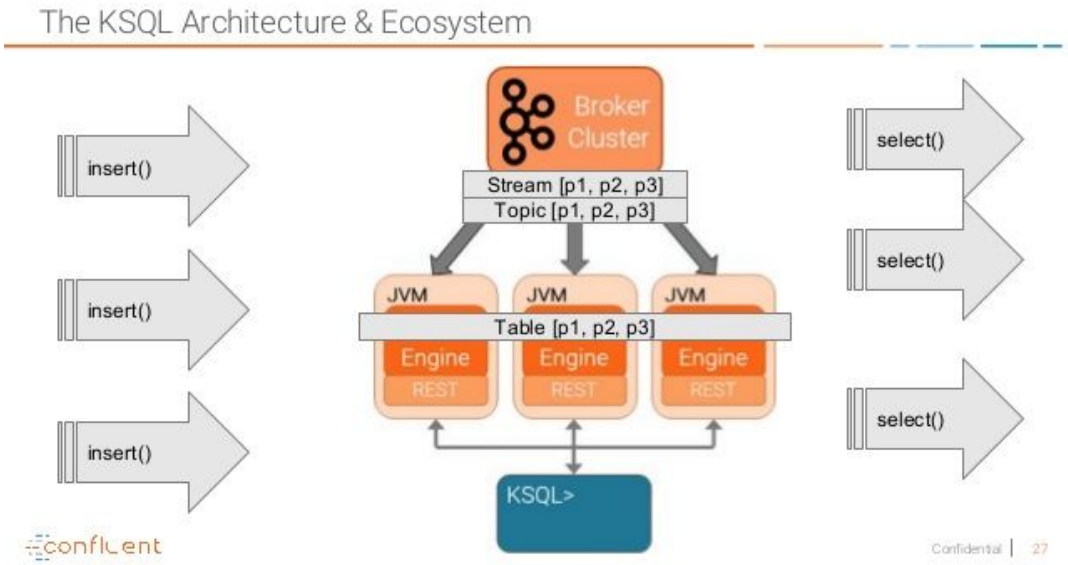
Fault Tolerance & Scalability of Streaming Apps

Streaming Apps instances behave just like consumers in the same group.

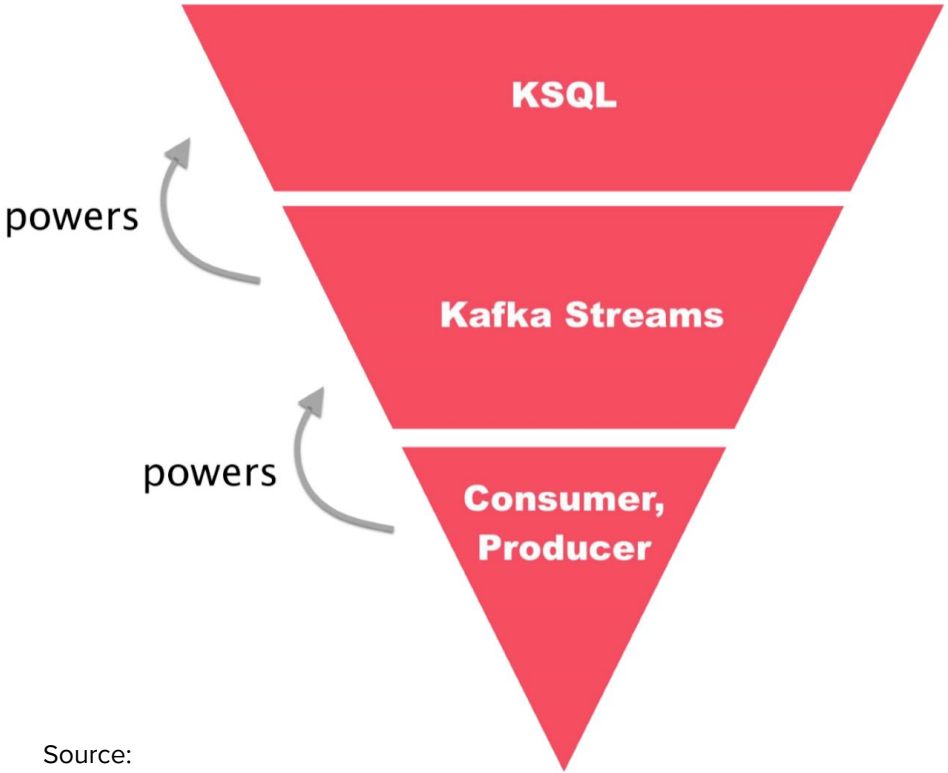


KSQL (https://github.com/confluentinc/ksql)

ksql is the **open-source SQL streaming engine for Apache Kafka**, and makes it possible to build stream processing applications at scale, written using a familiar SQL interface.



ksql = SQL Engine for Apache Kafka



ease of use

```
CREATE STREAM, CREATE TABLE,
SELECT, JOIN, GROUP BY, SUM, …
```

```
KStream, KTable,
filter(), map(), flatMap(),
join(), aggregate(), …
```

```
subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

flexibility

Source:
https://docs.confluent.io/current/ksql/docs/concepts/ksql-and-kafka-streams.html

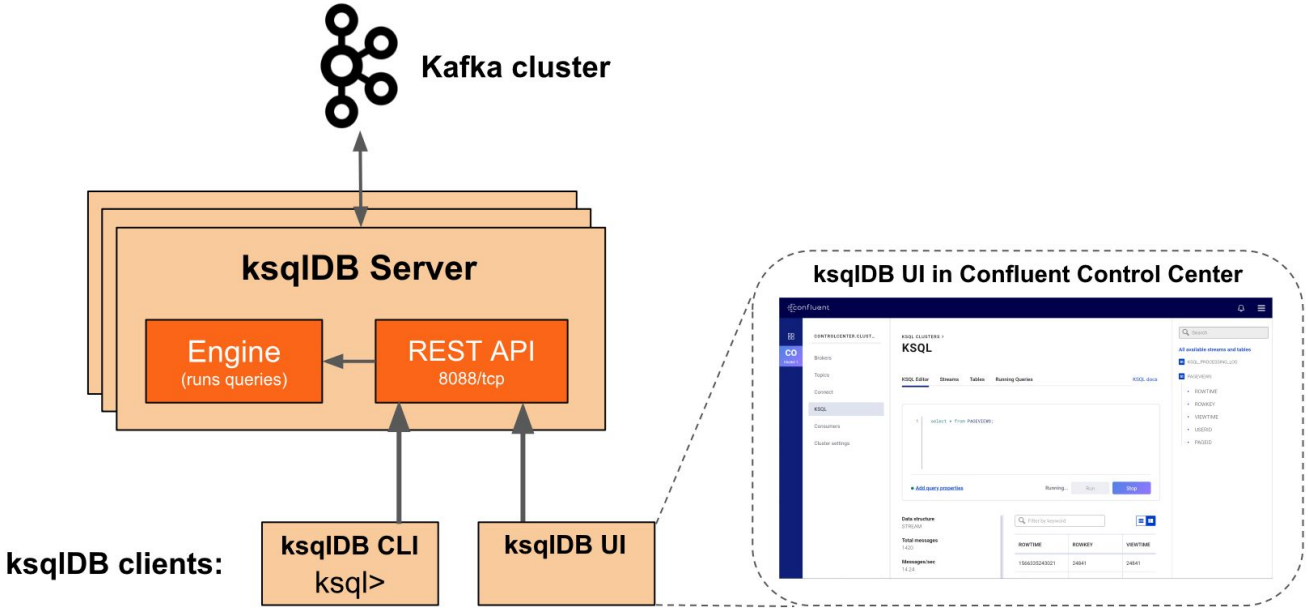
Since 2019 KSQL => ksqlDB

It stores replicated, fault-tolerant tables of data // in kafka

It allows queries and processing in SQL

You can work interactively against a cluster via a (RESTful) network API

ksqlDB architecture and components



ksqlDB components

ksqlDB Engine: The ksqlDB engine executes SQL statements and queries. You define your application logic by writing SQL statements, and the engine builds and runs the application on available ksqlDB servers. Each ksqlDB Server instance runs a ksqlDB engine. Under the hood, the engine parses your SQL statements and builds corresponding Kafka Streams topologies. The ksqlDB engine is implemented in the KsqlEngine.java class.

ksqlDB CLI: The ksqlDB CLI provides a console with a command-line interface for the ksqlDB engine. Use the ksqlDB CLI to interact with ksqlDB Server instances and develop your streaming applications. The ksqlDB CLI is designed to be familiar to users of MySQL, Postgres, and similar applications. The ksqlDB CLI is implemented in the io.confluent.ksql.cli package.

REST Interface: The REST server interface enables communicating with the ksqlDB engine from the CLI, Confluent Control Center, or from any other REST client. The ksqlDB REST server is implemented in the KsqlRestApplication.java class.

When you deploy your ksqlDB application, it runs on ksqlDB Server instances that are independent of one another, are fault-tolerant, and can be scaled.

ksqlDB vs Kafka Streams

Differences	**ksqlDB**	**Kafka Streams**
You write:	SQL statements	JVM applications
Graphical UI	Yes, in Confluent Control Center and Confluent Cloud	No
Console	Yes	No
Data formats	Avro, Protobuf, JSON, JSON_SR, CSV	Any data format, including Avro, JSON, CSV, Protobuf, XML
REST API included	Yes	No, but you can implement your own
Runtime included	Yes, the ksqlDB server	Applications run as standard JVM processes
Queryable state	No	Yes

Developer Workflows - API vs ksIDB

There are different workflows for ksqlDB and Kafka Streams when you develop streaming applications:

ksqlDB: You write SQL queries interactively and view the results in real-time, either in the ksqlDB CLI or in Confluent Control Center.

Kafka Streams: You write code in Java or Scala, recompile, and run and test the application in an IDE, like IntelliJ. You deploy the application to production as a jar file that runs in a Kafka cluster.

ksqlDB - SQL syntax

https://docs.ksqldb.io/en/latest/developer-guide/ksqldb-reference/quick-reference/

Terminate SQL statements with a semicolon character (;).

Statements can span multiple lines.

The hyphen character (-) isn't supported in names for streams, tables, topics, and columns.

Don't use quotes around stream names or table names when you CREATE them.

Escape single-quote characters (') inside string literals by using two successive single quotes ('). For example, to escape 'T', write ''T''.

Use backticks around column and source names with characters that are unparseable by ksqlDB or when you want to control case. **ksqlDB uppercases all identifiers by default,** you need to use backticks to preserve the desired casing. For more information, see How to control the case of identifiers.

Streams & Tables

Every query started by a **CREATE STREAM AS SELECT** or **CREATE TABLE AS SELEC**T statement writes its results to an output topic. The created topic is configured with the following properties:

Name: By default, ksqlDB creates the output topic with the same name as the stream or table created by the statement. You can specify a custom name in the KAFKA_TOPIC property of the statement's WITH clause.

Partitions: By default ksqlDB creates an output topic with 4 partitions. You can specify a custom partition count in the PARTITIONS property of the statement's WITH clause.

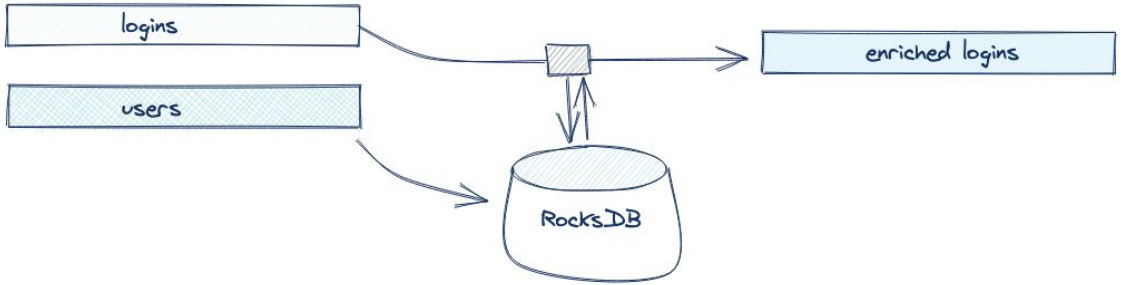
Replication Factor: By default, ksqlDB creates the output topic with a replication factor of 1. You can specify a custom replication factor in the REPLICAS property of the statement's WITH clause.

ksqlDB concepts: Streams & Tables

A **Kafka topic** in ksqlDB can be interpreted as either a **STREAM or TABLE**.

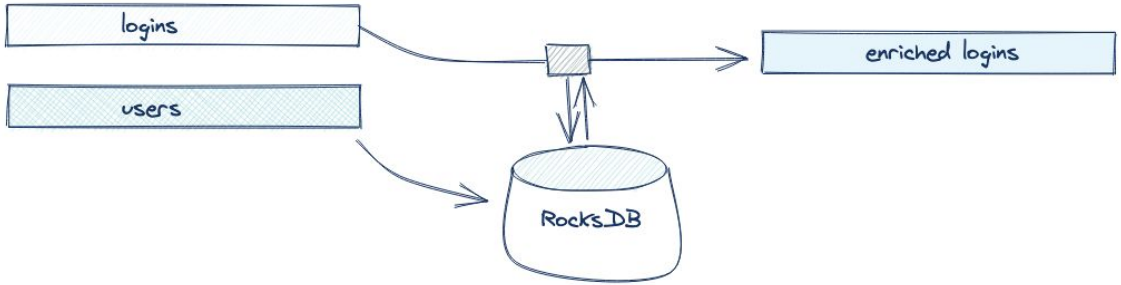
A **TABLE needs to be materialized locally on the computing node** in order to efficiently look up the most recent entry for a key without scanning the Apache Kafka® topic.

In ksqlDB, tables are materialized into RocksDB stores on the computing node to provide fast and memory-efficient key lookups. Each time a query processes an event from the source table, it will serialize the event into the format required and upserts it into RocksDB. Next, when it receives an event from the source stream, it looks for the value in the materialized store instead of scanning the topic.



ksqlDB concepts: Streams & Tables

The RocksDB store is replicated by ksqlDB using a mechanism called **changelog topics**. These topics (stored in Kafka) are used to restore the local state in the event that a new node comes online (or an old one is physically relocated). **A changelog topic is a log compacted, internal Kafka topic to ksqlDB that contains every change to the local state store**; the key and value for events in the changelog topic are byte for byte the same as the most recent matching key and value in the RocksDB store. Leveraging this topic makes recovery simple: You can scan the changelog topic and put each Kafka message directly into the state store without any additional processing.



ksqlDB terms: Push and pull queries

Push queries: the ones that push out a continual stream of changes. These queries run forever and produce an ongoing feed of results that updates as new changes occur.

Pull queries: the ones that allow you to pull data at a point in time. In other contexts, this kind of functionality is sometimes referred to as point-in-time queries or queryable state.

As an example, consider a ride sharing app. It needs to get a continuous feed of the current position of the driver (a push query), as well as look up the current value of several things such as the price of the ride (a pull query).

Push and pull queries

By default, a query is considered to be a pull query, just as in a normal database. For example, consider the following query:

```
SELECT ride_id, current_latitude, current_longitude
FROM ride_locations
WHERE ROWKEY = ‘6fd0fcdb’; // Note that this is treated as a pull query and returns the current location at the time the query is executed. However, if we append EMIT CHANGES, this will be transformed into a push query that produces a continuously updating stream of current driver position coordinates, not just the current state:

SELECT ride_id, current_latitude, current_longitude
FROM ride_locations
WHERE ROWKEY = ‘6fd0fcdb’
EMIT CHANGES;
```

Joins

Join multiple streams to create a new stream.

- When you join two streams, you must specify a WITHIN clause for matching records that both occur within a specified time interval.

Join multiple tables to create a new table.

Join multiple streams and tables to create a new stream.

Your ksqlDB applications must meet specific requirements for joins to be successful.

Co-partitioning data for joins

When you use KSQL to join streaming data, **you must ensure that your streams and tables are co-partitioned**, which means that input records on both sides of the join have the same configuration settings for partitions.

Co-partitioning requirements

- The input records for the join must have the same keying scheme
    - For example, you can join a stream of user clicks that's keyed by a VARCHAR userId field with a table of user profiles that's keyed by a VARCHAR userId field. The join won't match if the key fields don't have the same name and type.
- The input records must have the same number of partitions on both sides.
- Both sides of the join must have the same partitioning strategy.

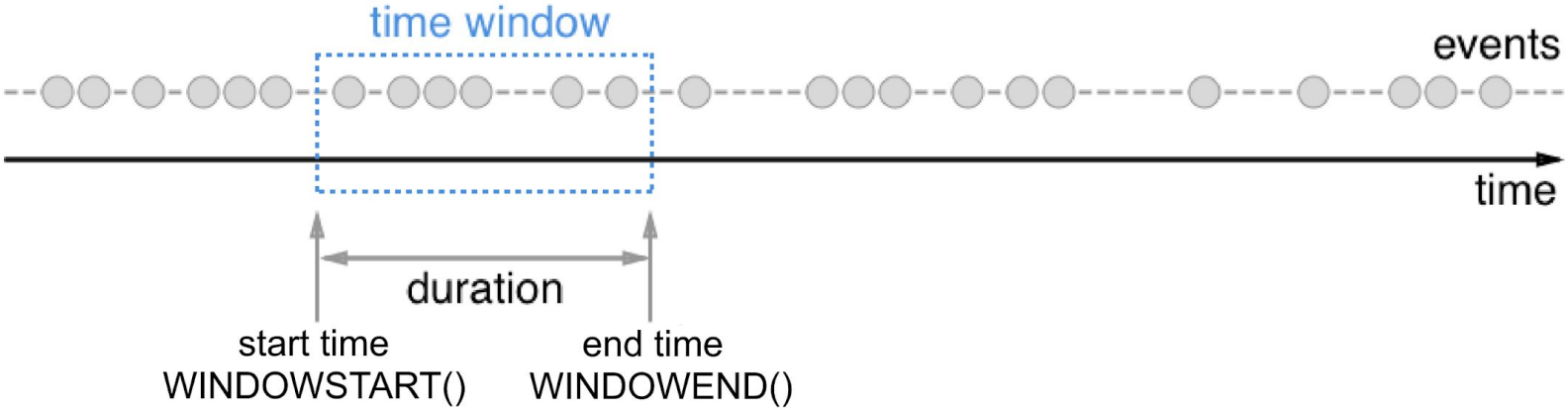
Co-partitioning data for joins

When your inputs are co-partitioned, records with the same key, from both sides of the join, are delivered to the same stream task during processing. If your inputs aren't co-partitioned, you need to re-key one of the them by using the PARTITION BY clause.

For example, in a stream-table join, if a stream of user clicks is keyed by pageId, but a table of user profiles is keyed by userId, one of the two inputs must be **re-keyed (re-partitioned).** Which of the two should be re-keyed depends on the situation. For example, if you need to re-partition a stream to be keyed by a product_id field, and keys need to be distributed over 6 partitions to make a join work, use the following KSQL statement:

CREATE STREAM products_rekeyed WITH (PARTITIONS=6) AS SELECT * FROM products PARTITION BY product_id;

Windowed Aggregation



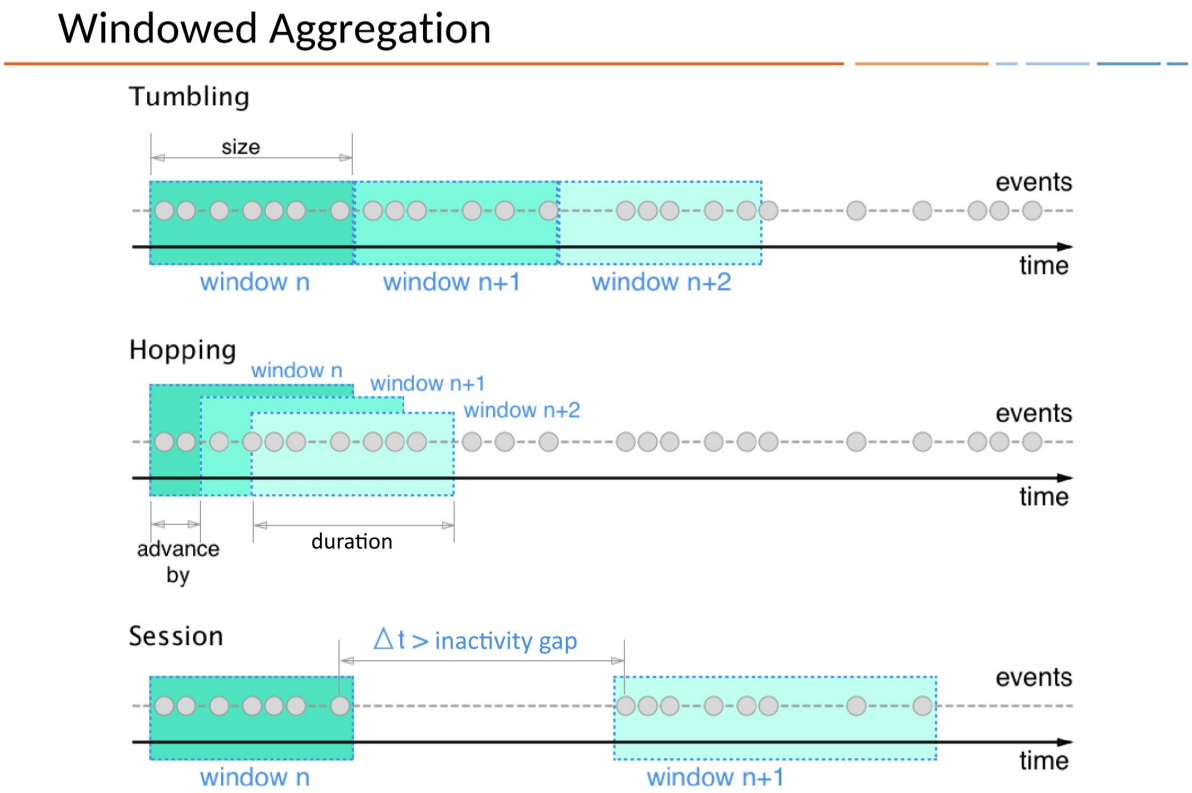
ksqlDB enables grouping records that have the same key for stateful operations, like joins, into windows. You specify a retention period for the window, and this retention period controls how long ksqlDB waits for out-of-order records. If a record arrives after the window's retention period has passed, the record is discarded and isn't processed in that window.

Window types

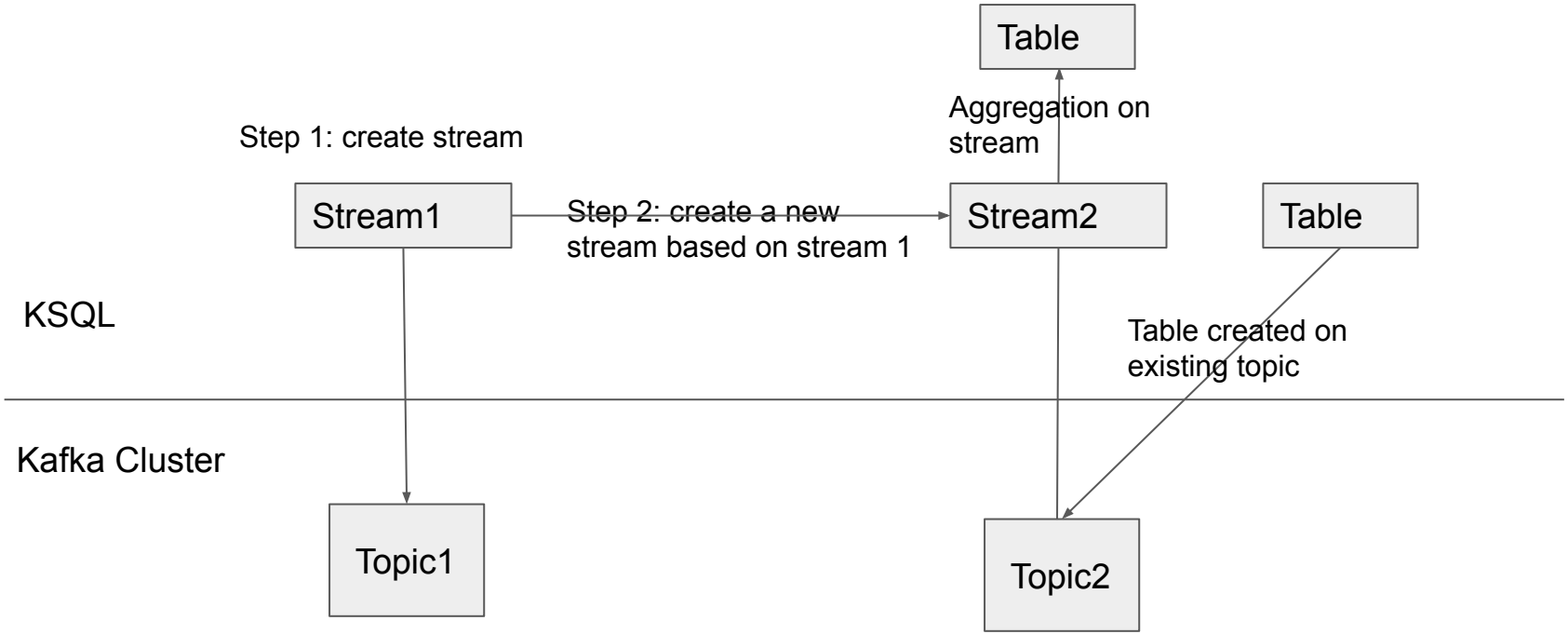
Hopping and tumbling windows are time windows, because they're defined by fixed durations they you specify. Session windows are dynamically sized based on incoming data and defined by periods of activity separated by gaps of inactivity.

Window type	Behavior	Description
Hopping Window	Time-based	Fixed-duration, overlapping windows
Tumbling Window	Time-based	Fixed-duration, non-overlapping, gap-less windows
Session Window	Session-based	Dynamically-sized, non-overlapping, data-driven windows

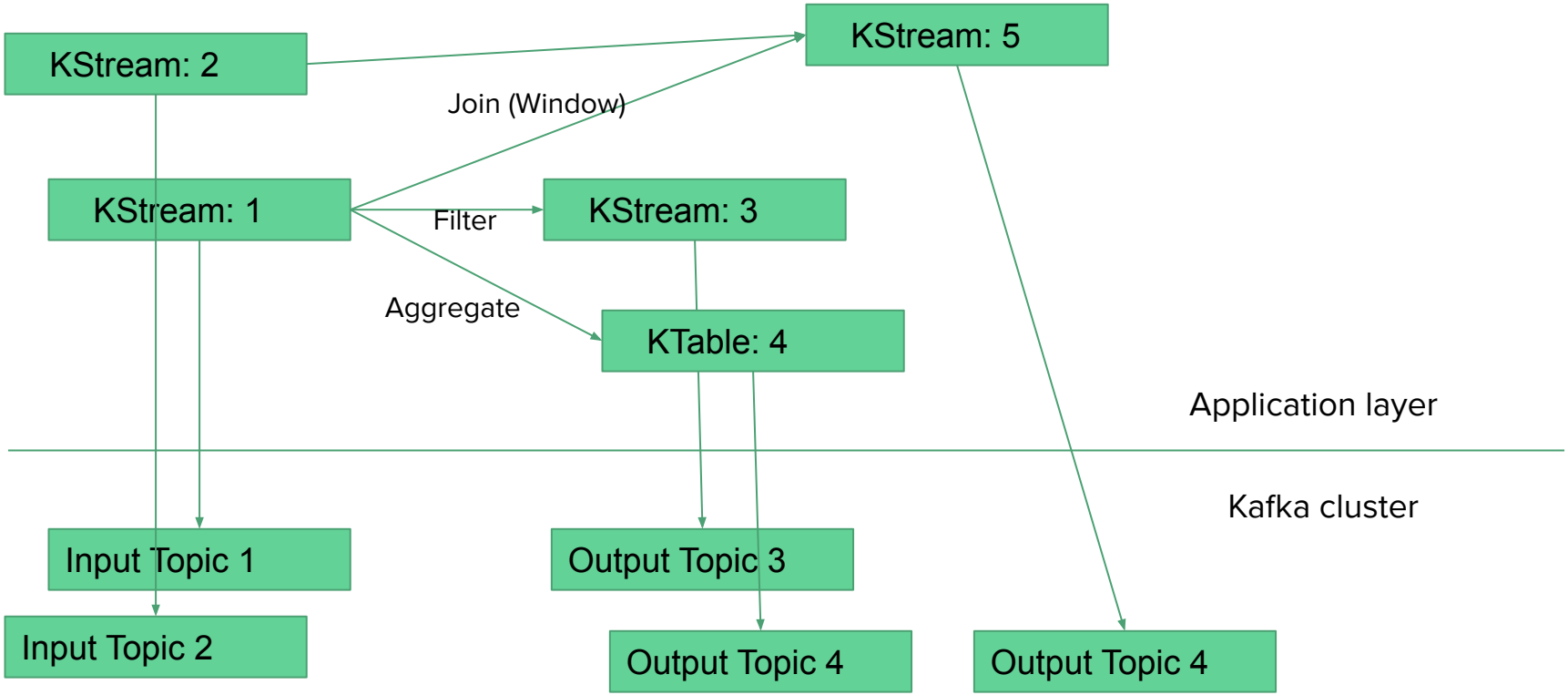
Window types



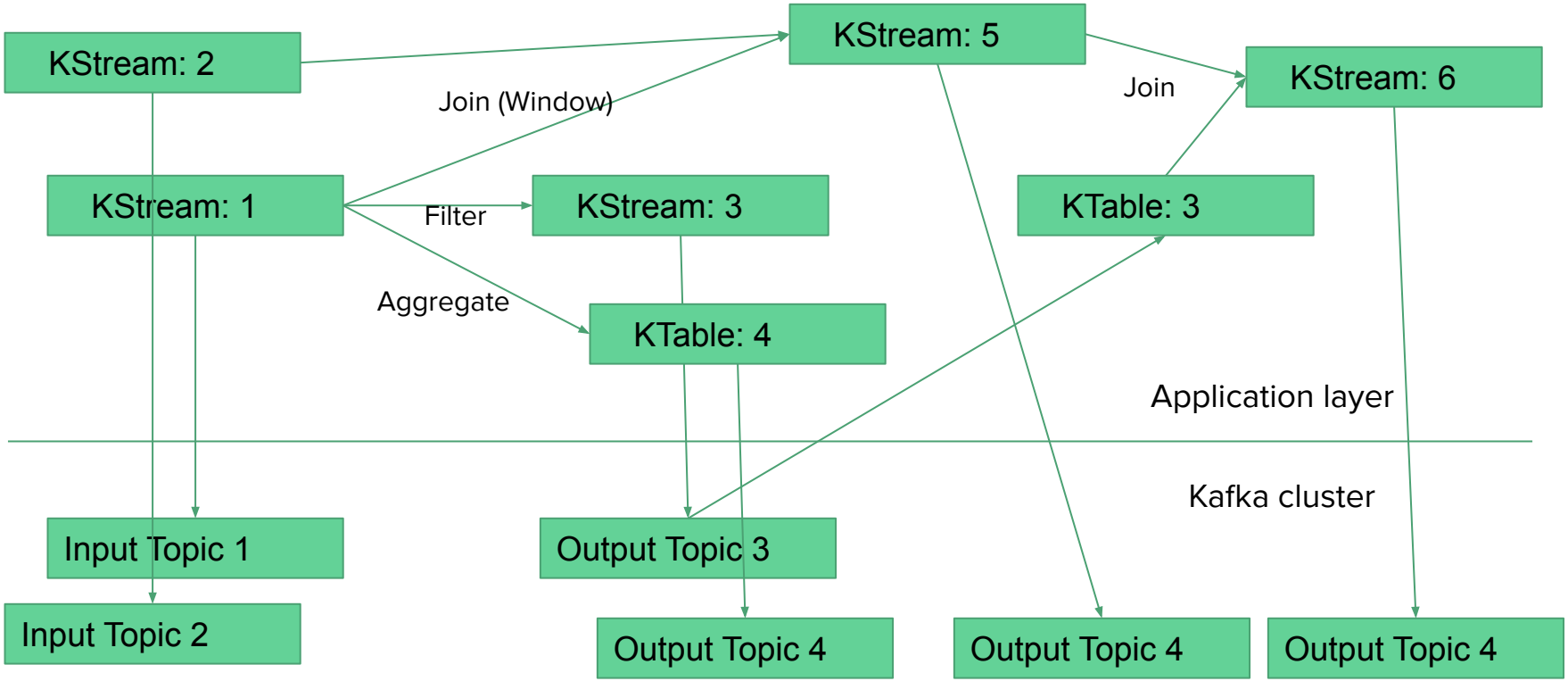
Hands on KSQL



Simplified :-) view of transformations



Simplified :-) view of transformations



ksqlDB components

ksqlDB server that contains:

- ksqlDB engine -- processes SQL statements and queries
- REST interface -- enables client access to the engine

ksqlDB CLI -- console that provides a command-line interface (CLI) to the engine

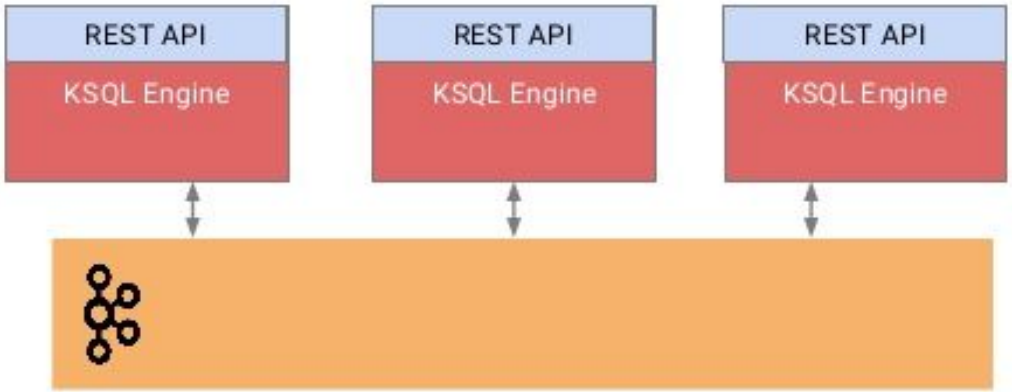
ksqlDB UI -- enables developing ksqlDB applications in Confluent Control Center and Confluent Cloud

Note: **The KSQL servers are run separately from the KSQL CLI client and Kafka brokers.** You can deploy servers on remote machines, VMs, or containers and then the CLI connects to these remote servers. Join ksqlDB engines to the same service pool by using the ksql.service.id property.

KSQL properties:
./confluent-5.5.1/etc/ksqldb/ksql-server.properties

KSQL Deployment

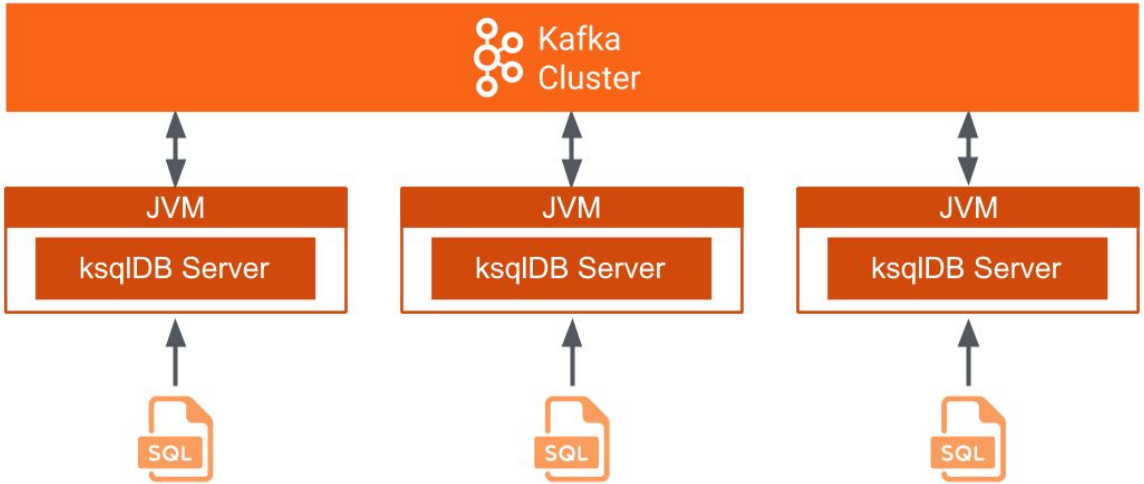
- KSQL Server
  - KSQL Engine
  - REST API



2 modes of deployment:

- Interactive
- non-interactive

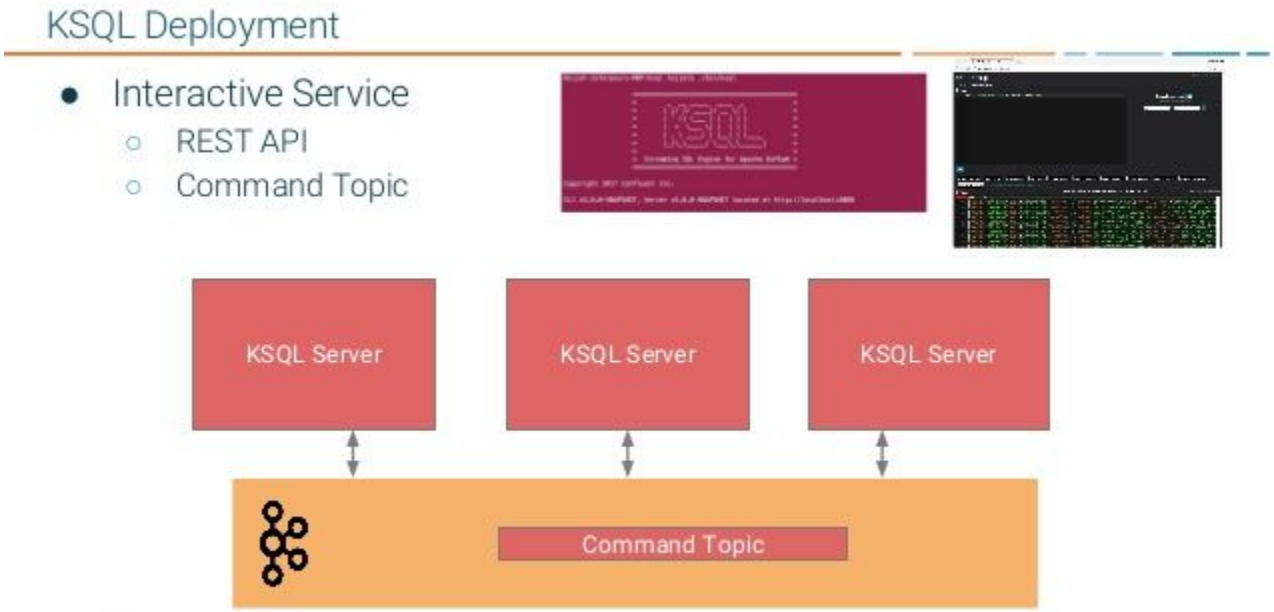
ksqlDB Standalone Application (Headless Mode)



Non-interactive deployment:

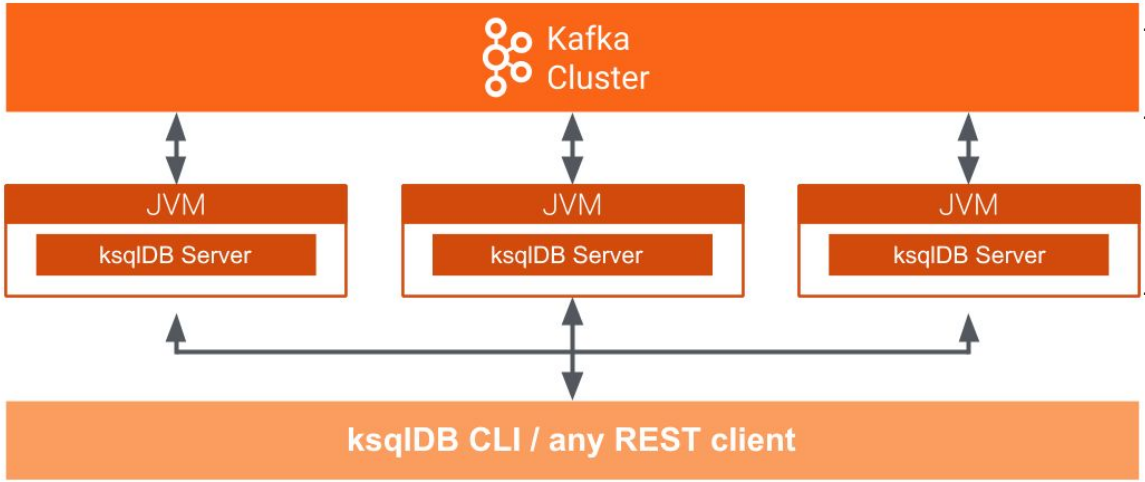
Start any number of server nodes
Pass a SQL file with SQL statements to execute:
<path-to-confluent>bin/ksql-node query-file=path/to/myquery.sql
Ensure resource isolation
Leave resource management to dedicated systems, like Kubernetes

Interactive mode



Unlike the headless mode, where each server already knows all the queries it's supposed to run at the time of startup (via the SQL file), interactive ksqlDB servers in the same ksqlDB cluster share the ksqlDB statements being executed using a special Kafka topic called the ksqlDB cluster's command topic.

ksqlDB Client/Server (Interactive Mode)



- Write statements and queries on the fly
- Start any number of server nodes dynamically: <path-to-confluent>/bin/ksql-server-start
- Start one or more CLIs or REST Clients and point them to a server: <path-to-confluent>/bin/ksql https://<ksql-server-ip-address>:8090

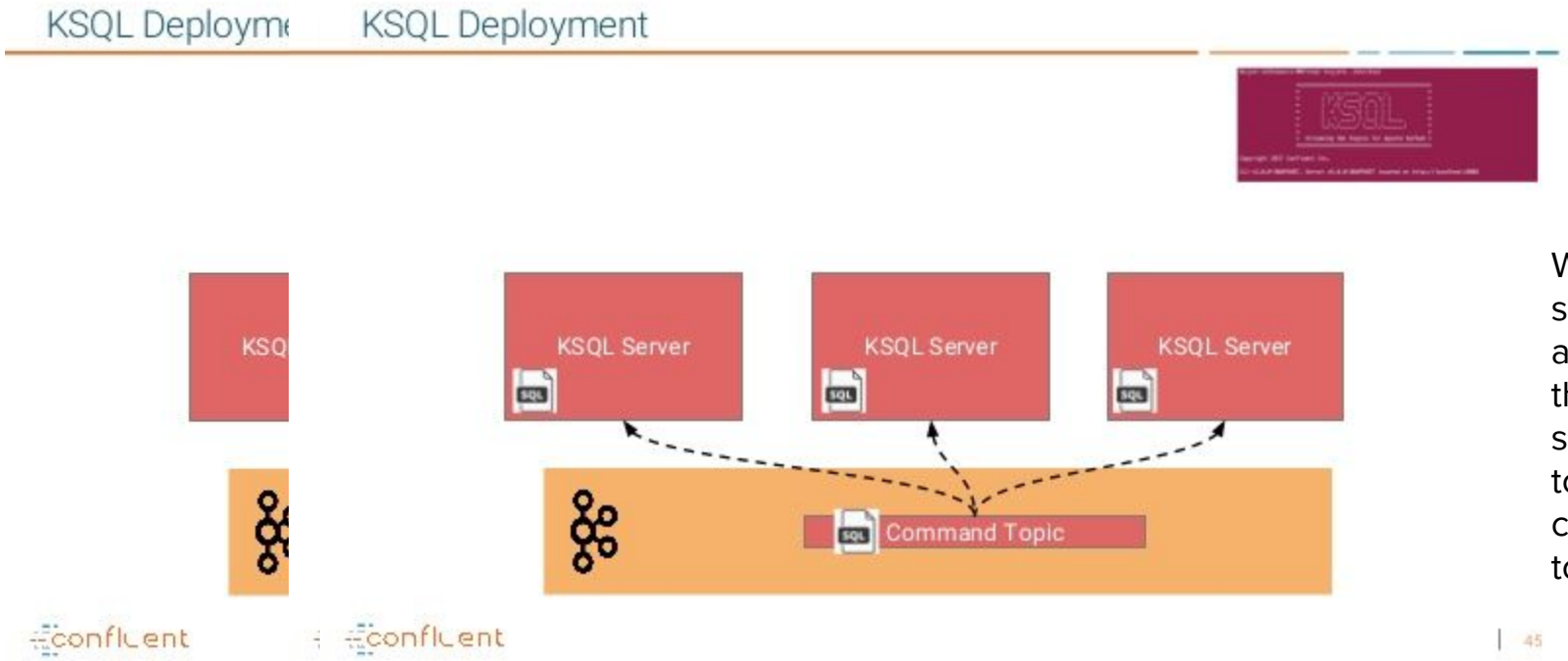
Command topic

In interactive mode, **ksqlDB shares statements with servers in the cluster over the *command topic***. The command topic stores every SQL statement, along with some metadata that ensures the statements are built compatible across ksqlDB restarts and upgrades. ksqlDB names the command topic _confluent-ksql-<service id>command_topic, where <service id> is the value in the ksql.service.id property.

By convention, the ksql.service.id property should end with a separator character of some form, for example a dash or underscore, as this makes the topic name easier to read.

ksqlDB writes DDL and DML statements to the command topic. Each ksqlDB Server reads the statement from the command topic, parsing and analyzing it.

What happens when a query is run in interactive



When ksql servers are added all of them subscribe to the command topic

What happens when the KSQL server restarts

If you only have a single KSQL server, then stopping that server will of course stop all the queries. Once the server is running again, all queries will continue from the points they stopped processing. No data is lost.

If you have multiple KSQL servers running, then stopping one (or some) of them will cause the remaining servers to take over any query processing tasks from the stopped servers. Once the stopped servers have been restarted the query processing workload will be shared again across all servers.

/Query, /ksql

curl -X "POST" "http://localhost:8088/ksql" -H {} -d $'{"ksql": "LIST STREAMS;","streamsProperties": {}}'

curl -X "POST" "http://localhost:8088/query" -H "Content-Type: application/vnd.ksql.v1+json; charset=utf-8" -d $'{"ksql": "SELECT * FROM test EMIT CHANGES LIMIT 4;","streamsProperties": {}}'

The /ksql resource runs a sequence of KSQL statements. All statements, except those starting with SELECT, can be run on this endpoint. To run SELECT statements use the /query endpoint.

The /query resource lets you stream the output records of a SELECT statement via a chunked transfer encoding. The response is streamed back until the LIMIT specified in the statement is reached, or the client closes the connection. If no LIMIT is specified in the statement, then the response is streamed until the client closes the connection.

Link: https://rmoff.net/2019/01/17/ksql-rest-api-cheatsheet/