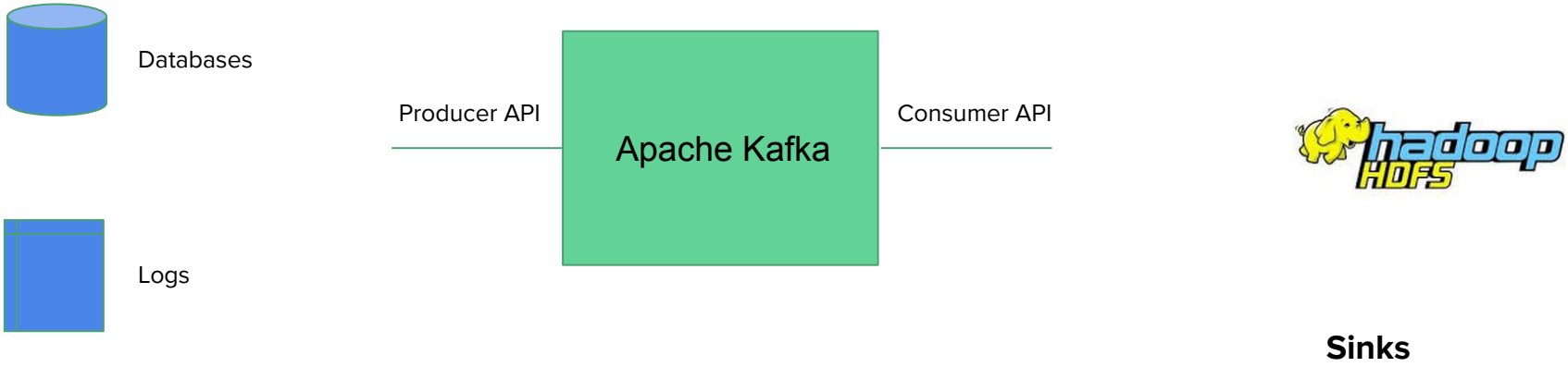
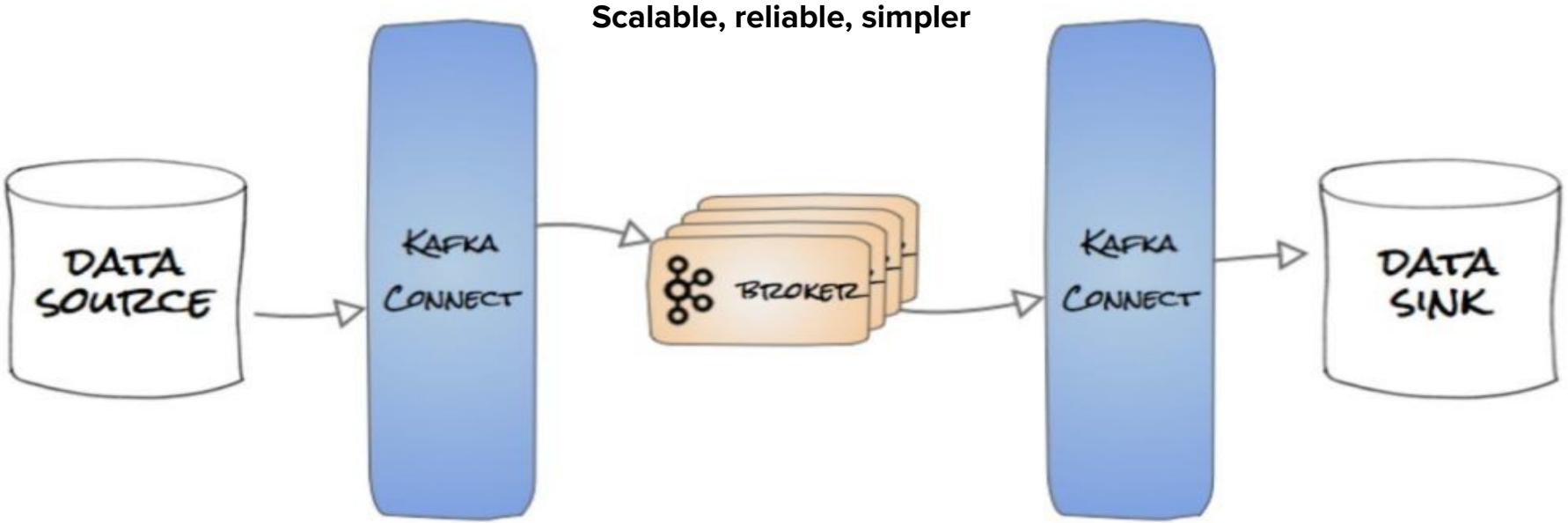
Kafka Connect

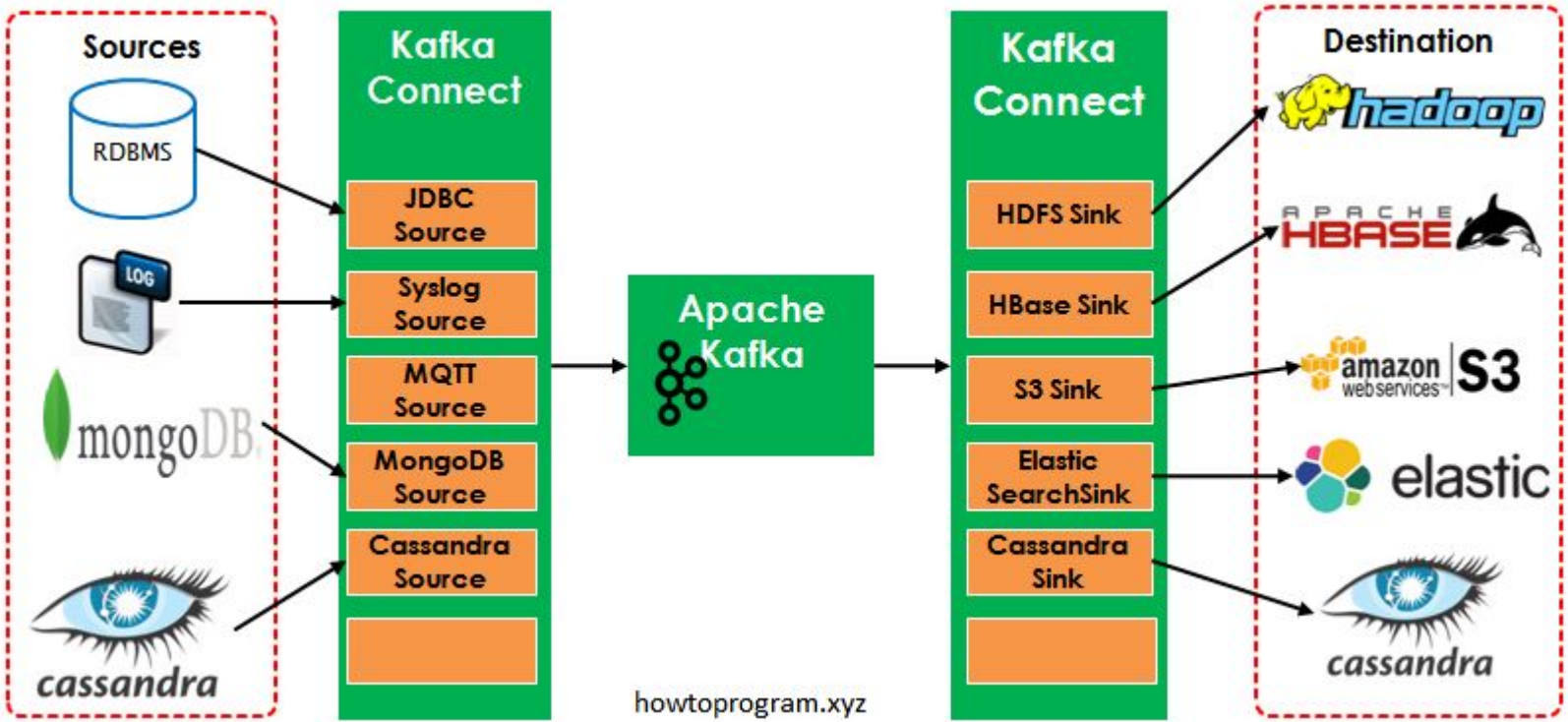
How to connect easier different sources/sinks?



Kafka Connect: open source Apache Kafka component that helps to move the data IN or OUT of Kafka easily.



How to connect easier different sources/sinks?



What is Kafka Connect?

Kafka has a **built-in framework** called Kafka Connect **for writing sources and sinks** that either continuously ingest data into Kafka or continuously ingest data in Kafka into external systems. An externally hosted list of connectors is maintained by Confluent at the Confluent Hub.

So what Kafka Connect provides is that rather than writing our own Consumer or Producer code, we can use a Connector that takes care of all the implementation details such as **fault tolerance, delivery semantics, ordering etc. and get the data moved**.

For developers Kafka Connect has a rich API that can be used to build up new connectors: https://docs.confluent.io/platform/current/connect/javadocs/index.html

Also, it has a REST API for configuration and management of connectors.

Main Concepts (Connectors, Tasks, Workers)

Connectors – the high level abstraction that coordinates data streaming by managing tasks. It decides how to split the data-copying work between the tasks. Connectors divide the actual job into smaller pieces as tasks in order to have the ability to scalability and fault tolerance.

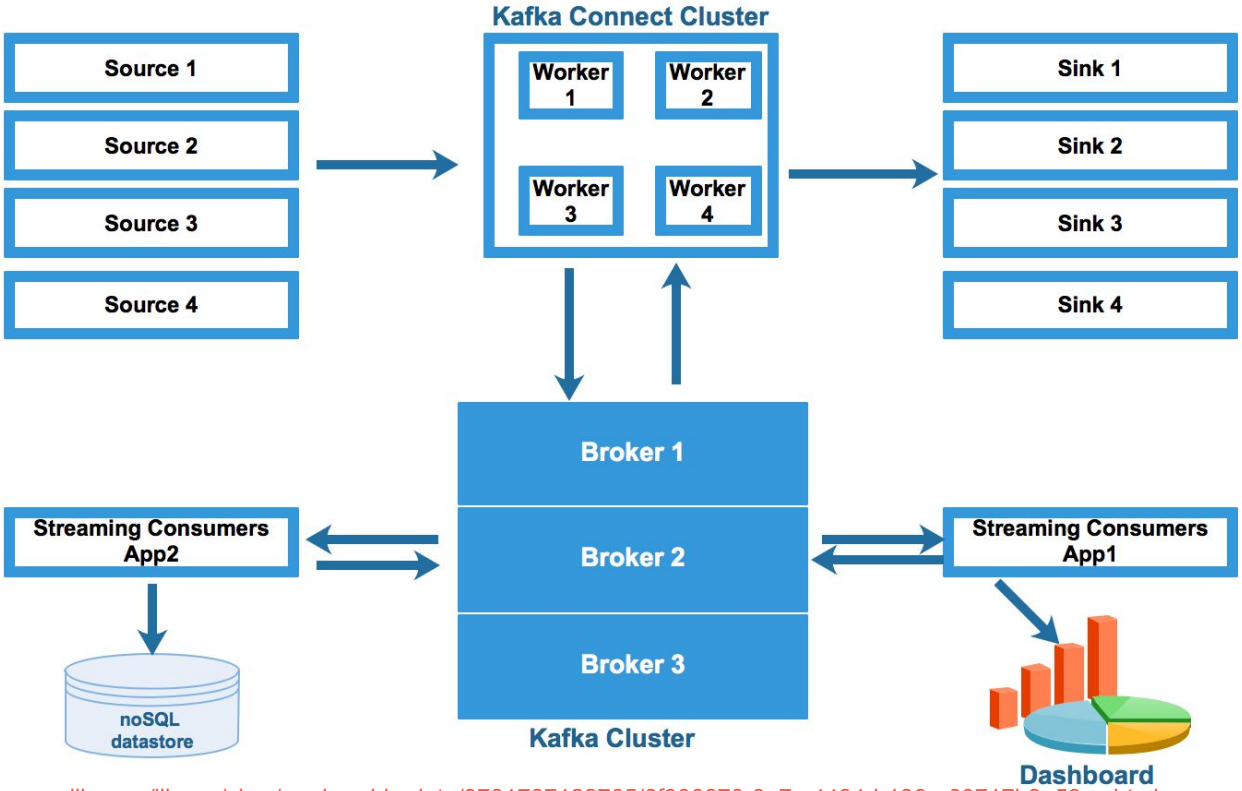
Source connector – Ingests entire databases and streams table updates to Kafka topics. A source connector can also collect metrics from all your application servers and store these in Kafka topics, making the data available for stream processing with low latency.

Sink connector – Delivers data from Kafka topics into secondary indexes such as Elasticsearch, or batch systems such as Hadoop for offline analysis.

Tasks – the implementation of how data is copied to or from Kafka

Workers – the running processes that execute connectors and tasks

Kafka Connect Architecture



Source: https://www.oreilly.com/library/view/modern-big-data/9781787122765/6f906670-8e7c-4494-b186-a60717b9e59a.xhtml

Kafka Connect Cluster
set of workers on which connectors are running

A Kafka Connect Cluster has one (**standalone**) or more (**distributed) workers running on one or multiple servers. The workers manage connectors and tasks, distributing work among the available worker processes.**

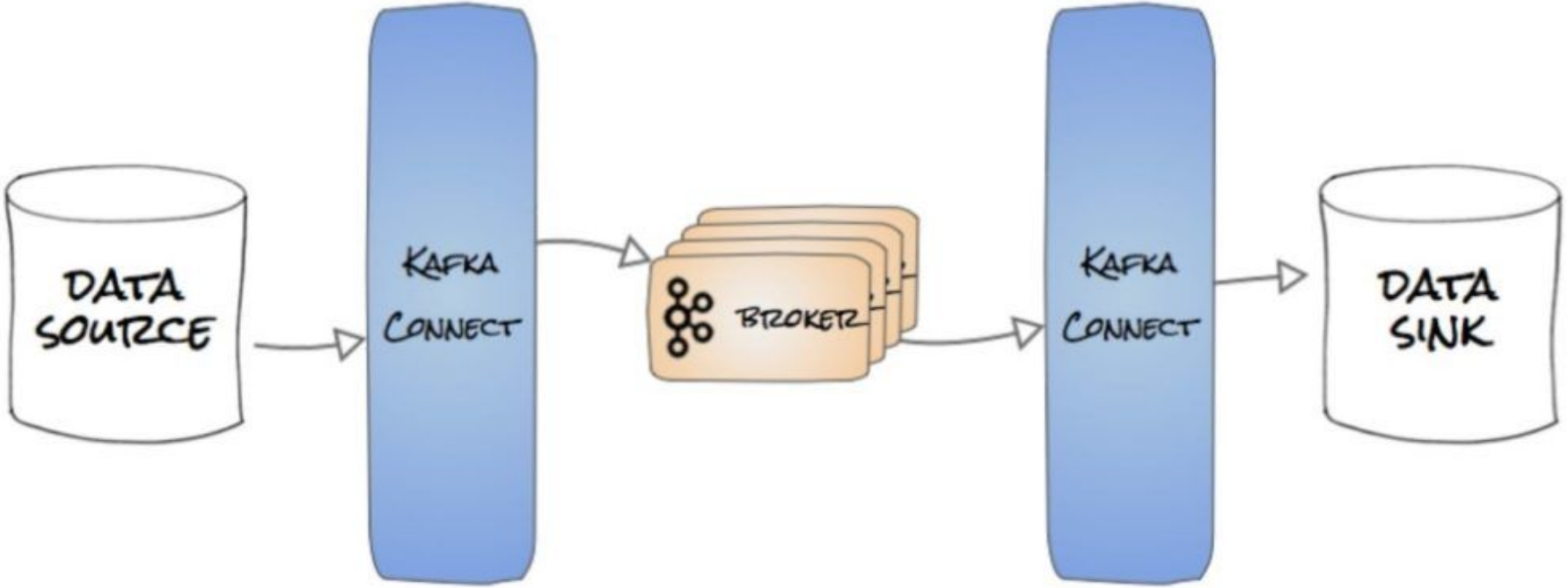
Standalone mode: All work is performed in a single worker as single process. It is easier to setup and configure and can be useful where using single worker makes sense. However it does not provide fault tolerance or scalability. // not to be used in production

Distributed mode: Multiple workers are in a cluster. Configured by REST API. Provides scalability and fault tolerance. When one connector dies, its tasks are redistributed by rebalance mechanism among other workers.

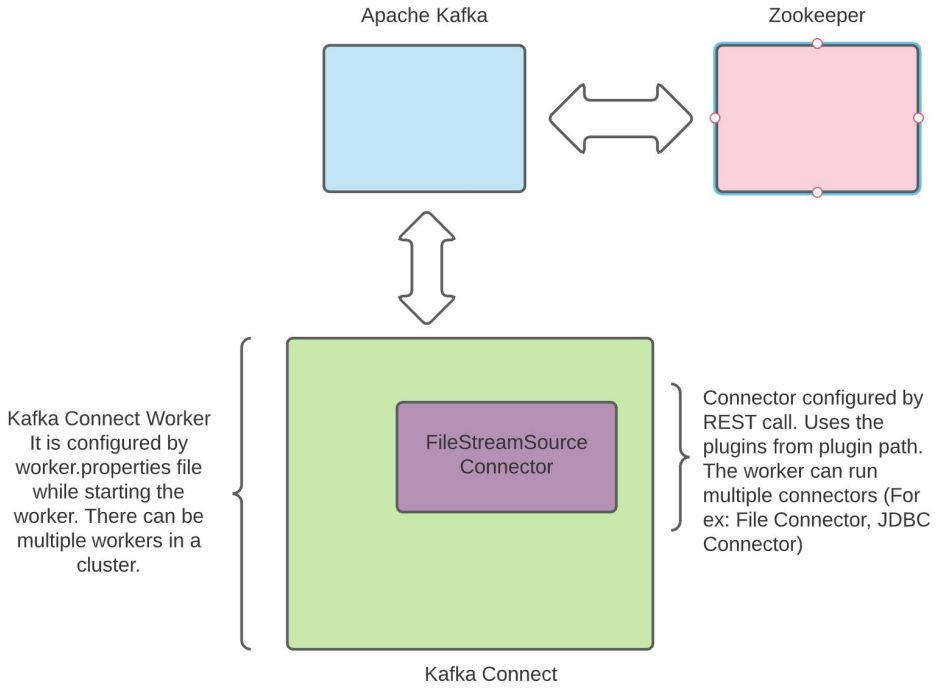
Note: that Kafka Connect does not automatically handle restarting or scaling of Workers, so this must be handled with some other solution.

Kafka Connect cluster/nodes <> Kafka Cluster

A Kafka connect cluster can be run on one or more servers (f**or production these will be separate to the servers that the Kafka Brokers are running on**), and one (but potentially more) workers on each server.



Example: FileStream connector



We can run the connector on either standalone or distributed workers.

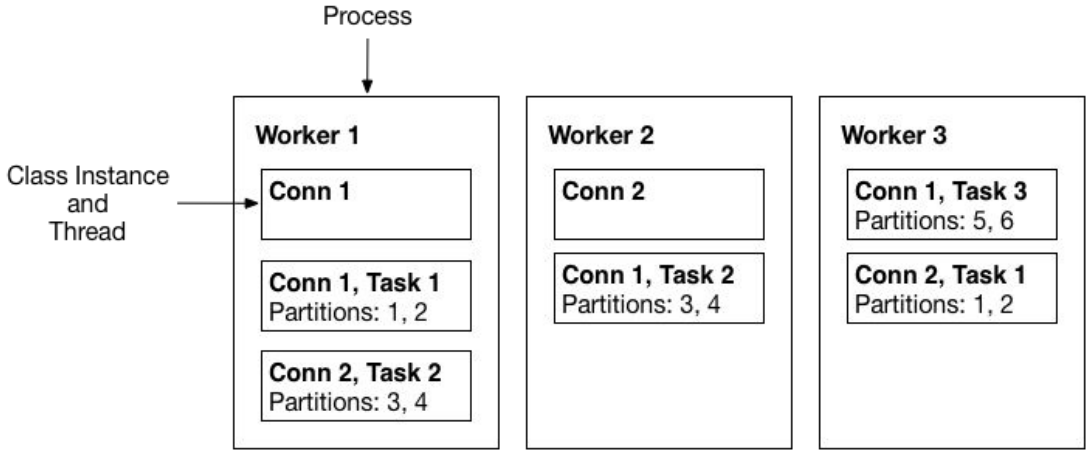
Step 1: get locally the connector code and configure the path in the connect properties
Step 1: start connect worker/workers - check the class
Step 3: create the connector config file
Step 4: load the connector

Connect Distributed mode

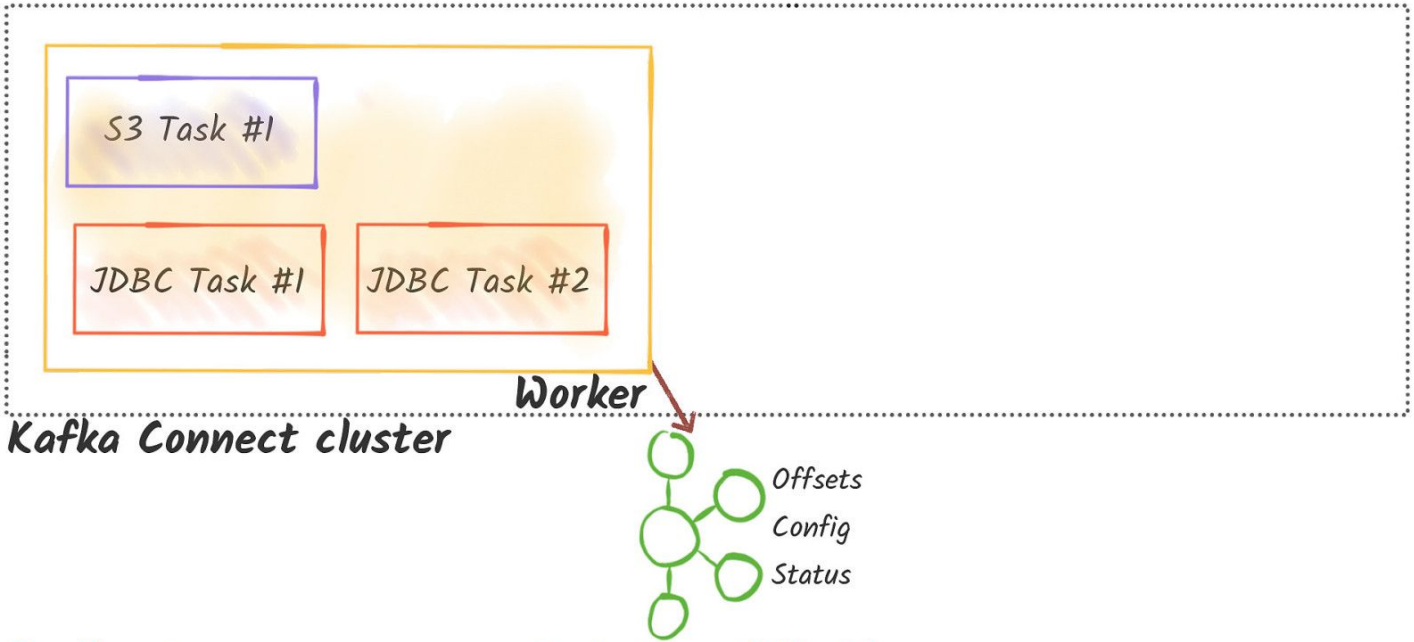
This model allows Kafka Connect to scale to the application. It can run scaled down to a single worker process that also acts as its own coordinator, or in clustered mode where connectors and tasks are dynamically scheduled on workers.

Distributed mode provides scalability and automatic fault tolerance for Kafka Connect. In distributed mode, you start many worker processes using the same group.id and they automatically coordinate to schedule execution of connectors and tasks across all available workers.

-

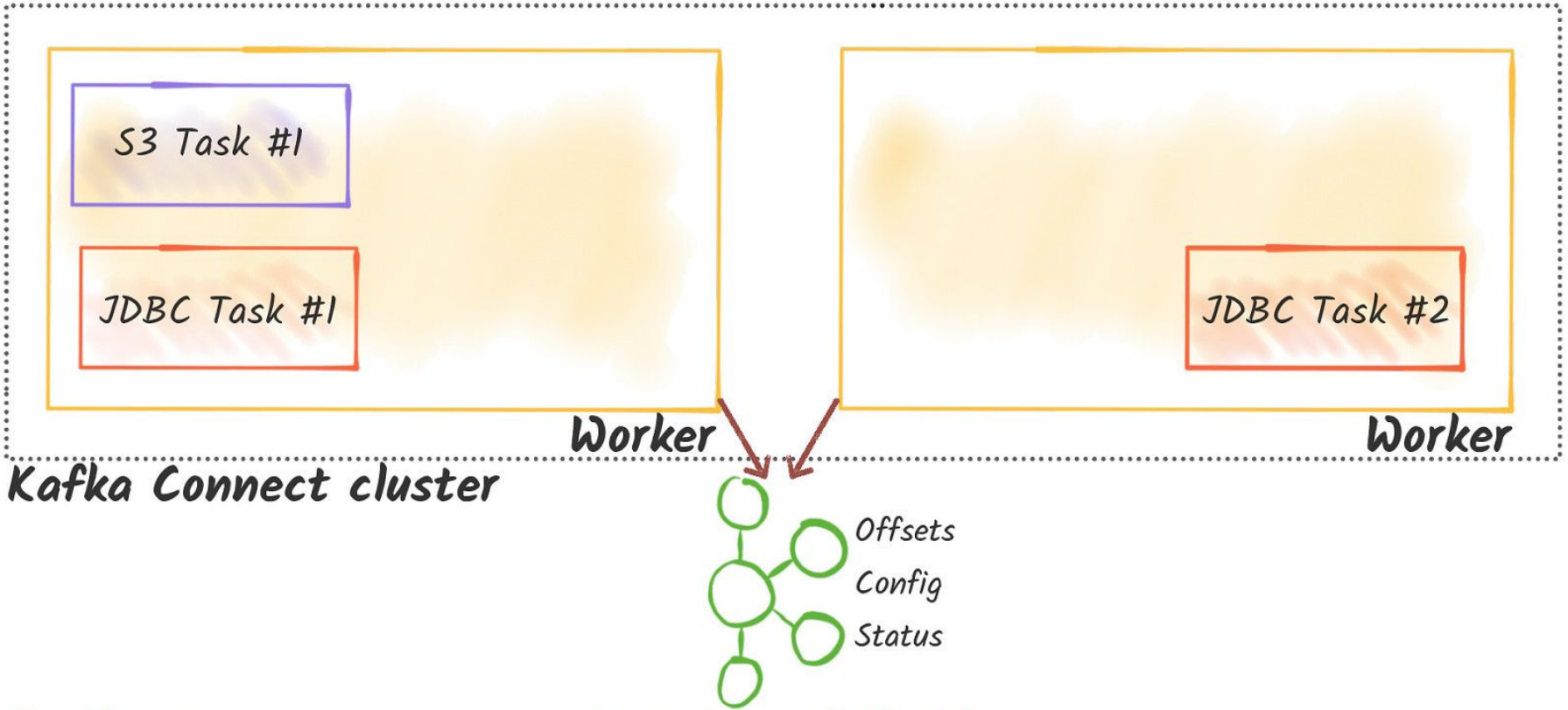


Kafka Connect Distributed Worker



Fault-tolerant? Yeah!

Scaling the Distributed Worker



Fault-tolerant? **Yeah!**

Distributed workers

Distributed mode handles automatic balancing of work, allows you to scale up (or down) dynamically, and offers fault tolerance both in the active tasks and for configuration and offset commit data. Execution is very similar to standalone mode:

> bin/**connect-distributed.sh** config/connect-distributed.properties

The following configuration parameters are critical to set before starting your cluster:
group.id (default connect-cluster) - unique name for the cluster, used in forming the Connect cluster group; note that this must not conflict with consumer group IDs
Rest.advertised.host.name = internal name of the machine (default localhost)
is how a Connect worker communicates with other workers in the cluster. If you set it to localhost then each worker in the cluster will only ever be able to contact itself when you use the REST interface, e.g. to send configuration updates (see details here and here)
config.storage.topic (default connect-configs) - topic to use for storing connector and task configurations; note that this should be a single partition, highly replicated topic

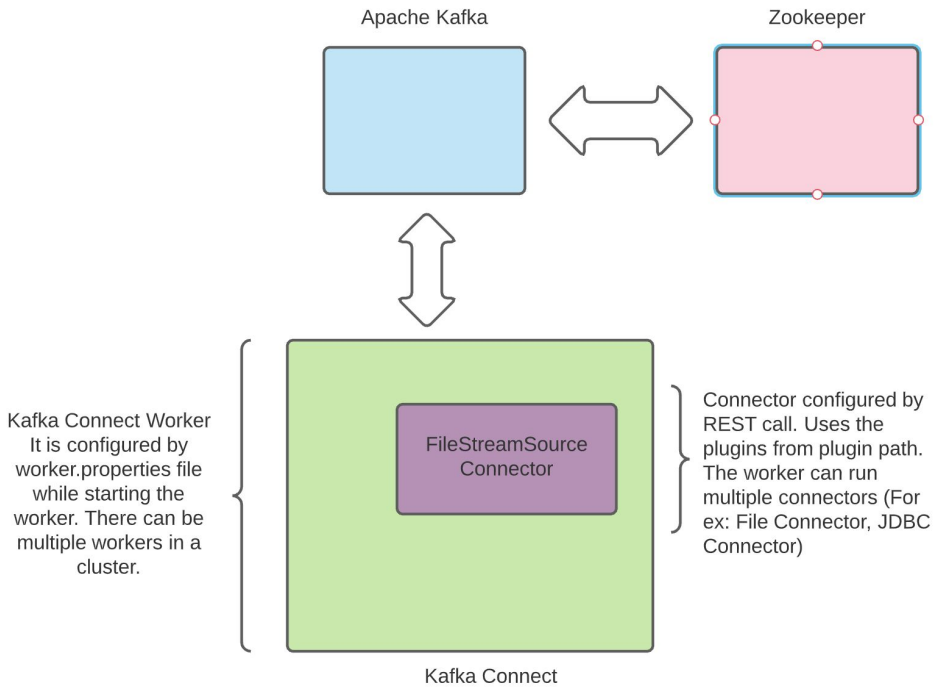
Configuring connectors

Connector configurations are simple key-value mappings. For standalone mode these are defined in a properties file and passed to the Connect process on the command line. In distributed mode, they will be included in the JSON payload for the request that creates (or modifies) the connector. Most configurations are connector dependent. However, there are a few common options:

name - Unique name for the connector. Attempting to register again with the same name will fail.
connector.class - The Java class for the connector
tasks.max - The maximum number of tasks that should be created for this connector. The connector may create fewer tasks if it cannot achieve this level of parallelism.

Sink connectors also have one additional option to control their input:
topics - A list of topics to use as input for this connector

Example: FileStream connector



In order to scale up the worker cluster, you need to follow the same steps of running Kafka Connect and starting Connector on each worker (All workers should have same group id). The high level overview of the architecture looks like as follows:

REST API

Since Kafka Connect is intended to be run as a service, it also supports a REST API for managing connectors. By default this service runs on port 8083. The following are the currently supported endpoints:
GET /connectors - return a list of active connectors
POST /connectors - create a new connector; the request body should be a JSON object containing a string name field and a object config field with the connector configuration parameters
GET /connectors/{name} - get information about a specific connector
GET /connectors/{name}/config - get the configuration parameters for a specific connector
PUT /connectors/{name}/config - update the configuration parameters for a specific connector
GET /connectors/{name}/tasks - get a list of tasks currently running for a connector
DELETE /connectors/{name} - delete a connector, halting all tasks and deleting its configuration

Let’s add FileStream connectors

See in Day 2: document “Starting Kafka Connect on Apache cluster”

Confluent JDBC connector

Allows you to import data from any relational database with a JDBC driver into Apache Kafka® topics (almost all relational databases that provide a JDBC driver, including Oracle, Microsoft SQL Server, DB2, MySQL and Postgres) . By using JDBC, this connector can support a wide variety of databases without requiring custom code for each one.

Data is loaded by periodically executing a SQL query and creating an output record for each row in the result set. By default, all tables in a database are copied, each to its own output topic.

When copying data from a table, the connector can load only new or modified rows by specifying which columns should be used to detect new or modified data -bulk, incrementing, timestamp options.

Confluent JDBC connector - incrementing options

Incrementing Column: A single column containing a unique ID for each row, where newer rows are guaranteed to have larger IDs, i.e. an AUTOINCREMENT column. Note that this mode can only detect new rows. Updates to existing rows cannot be detected, so this mode should only be used for immutable data. One example where you might use this mode is when streaming fact tables in a data warehouse, since those are typically insert-only.

Timestamp Column: In this mode, a single column containing a modification timestamp is used to track the last time data was processed and to query only for rows that have been modified since that time.

Timestamp and Incrementing Columns: This is the most robust and accurate mode, combining an incrementing column with a timestamp column. By combining the two, as long as the timestamp is sufficiently granular, each (id, timestamp) tuple will uniquely identify an update to a row. Even if an update fails after partially completing, unprocessed updates will still be correctly detected and delivered when the system recovers.

Confluent JDBC connector - incrementing options

Custom Query: The source connector supports using custom queries instead of copying whole tables. With a custom query, one of the other update automatic update modes can be used as long as the necessary WHERE clause can be correctly appended to the query. Alternatively, the specified query may handle filtering to new updates itself; however, note that no offset tracking will be performed (unlike the automatic modes where incrementing and/or timestamp column values are recorded for each record), so the query must track offsets itself.

Bulk: This mode is unfiltered and therefore not incremental at all. It will load all rows from a table on each iteration. This can be useful if you want to periodically dump an entire table where entries are eventually deleted and the downstream system can safely handle duplicates.

MySQL - Kafka connector

Step 1: download the JDBC connector classes on the Kafka Connect worker node (https://d1i4a15mxbxib1.cloudfront.net/api/plugins/confluentinc/kafka-connect-jdbc/versions/10.2.0/confluentinc-kafka-connect-jdbc-10.2.0.zip)

Step 2: download mySQL jar: mySQL is not included by default in the connector classes, thus we need the specific mySQL jar (https://dev.mysql.com/downloads/connector/j/5.1.html) and place it in the connector library directory

Step 3: Make sure you add the location of the connector classes in the connect-distributed.properties file. Start Kafka connect

Step 4: verify the JDBC classes are visible for Kafka Connect

Step 5: configure the connector

Step 6: start the connector

What Connect contains besides Connectors

Kafka Connect is modular in nature, providing a very powerful way of handling integration requirements. Some key components include:

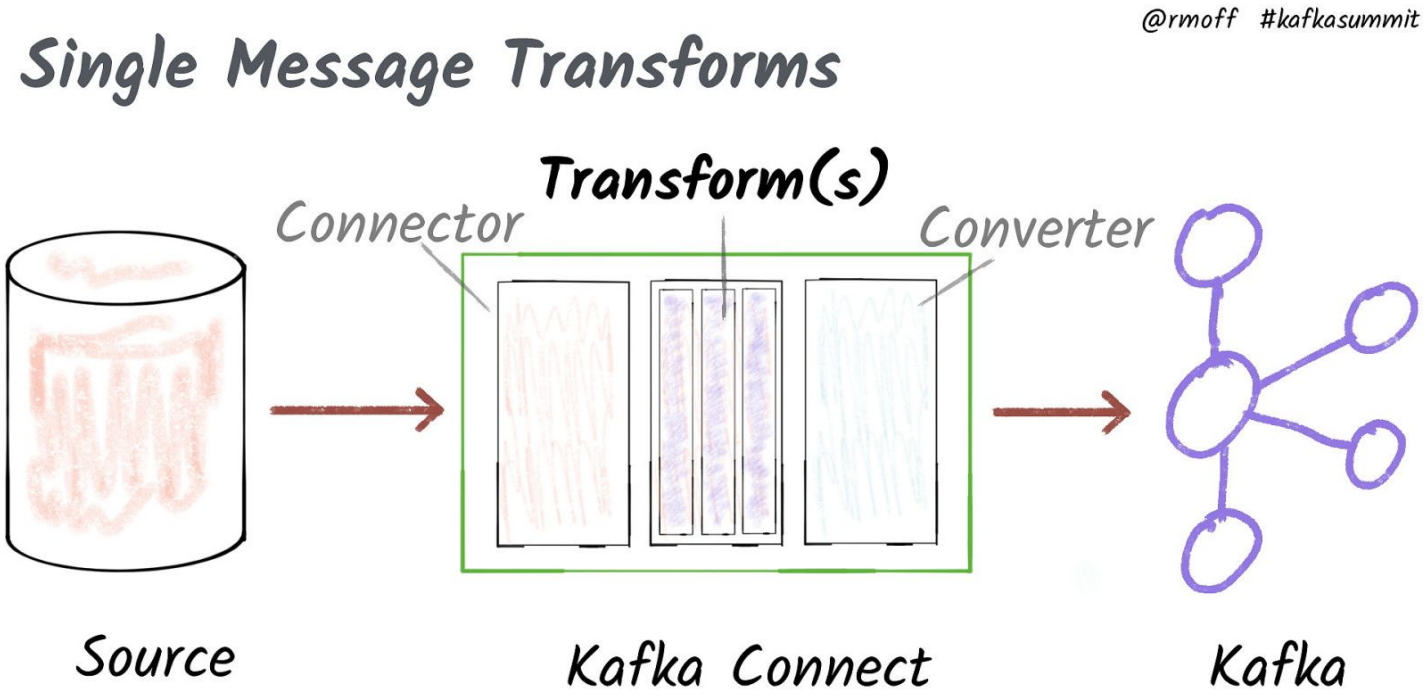
Connectors – the JAR files that define how to integrate with the data store itself

- A connector in Kafka Connect is responsible for taking the data from the source data store (for example, a database) and passing it as an internal representation of the data to the converter.

Converters serialize this source data object onto the topic. This means that you can have data on a topic in Avro (for example), and when you come to write it to HDFS (for example), you simply specify that you want the sink connector to use that format.

Transforms – optional in-flight manipulation of messages

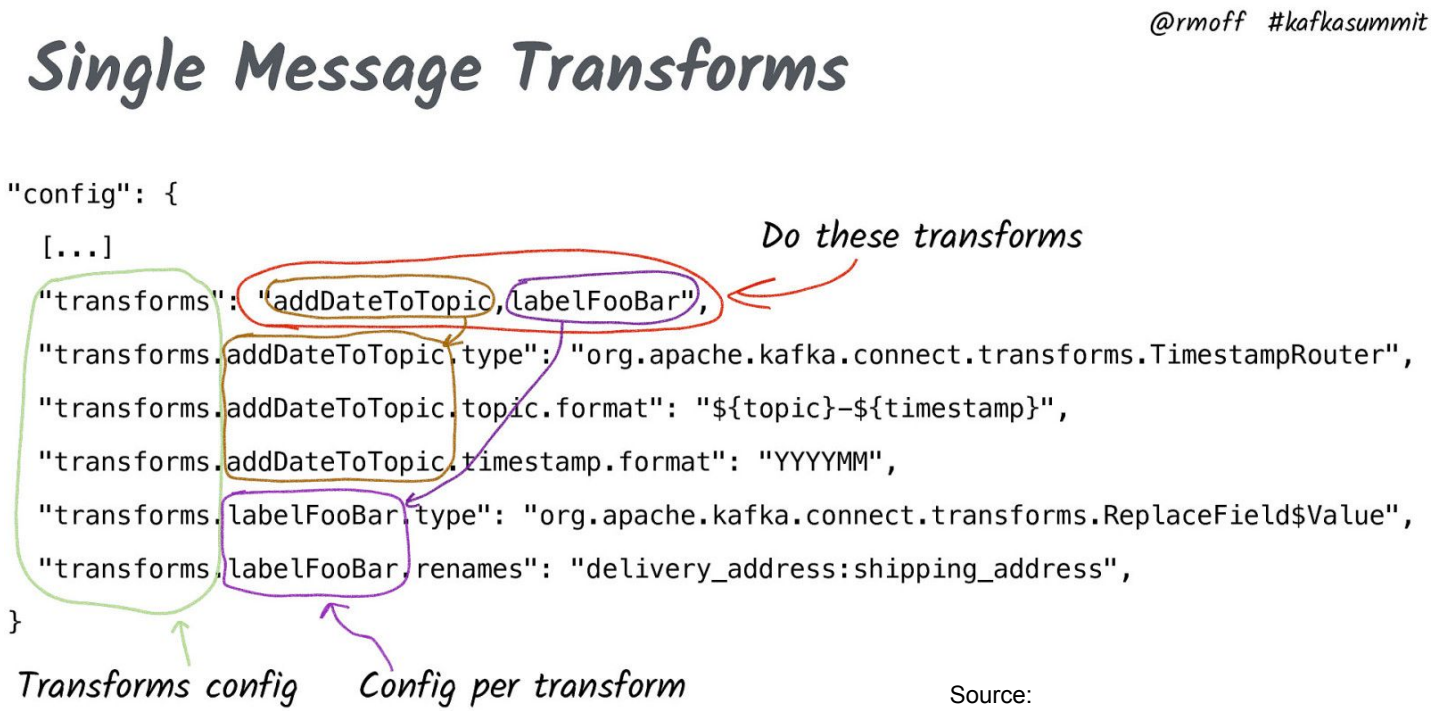
SMT


@rmoff #kafkasummit
Single Message Transforms
Transform(s)
Connector
Converter
Source
Kafka Connect
Kafka

Built-in Transformations

- **InsertField** – Add a field using either static data or record metadata
- **ReplaceField** – Filter or rename fields
- **MaskField** – Replace field with valid null value for the type (0, empty string, etc.)
- **ValueToKey** – Set the key to one of the value's fields
- **HoistField** – Wrap the entire event as a single field inside a Struct or a Map
- **ExtractField** – Extract a specific field from Struct and Map and include only this field in results
- **SetSchemaMetadata** – Modify the schema name or version
- **TimestampRouter** – Modify the topic of a record based on original topic and timestamp, which is useful when using a sink that needs to write to different tables or indexes based on timestamps
- **RegexpRouter** – Modify the topic of a record based on the original topic, replacement string, and a regular expression

SMT



Source:
https://talks.rmoff.net/QZ5nsS/from-zero-to-hero-with-kafka-connect#smoJuf4

SMT - add key

```
"transforms":"createKey.extractInt",
"transforms.createKey.type":"org.apache.kafka.connect.transforms.ValueToKey",
"transforms.createKey.fields":"id",
"transforms.extractInt.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
"transforms.extractInt.field":"id"
```

Source:
https://docs.confluent.io/5.2.0/connect/kafka-connect-jdbc/source-connector/index.html