Kafka Connect Concepts

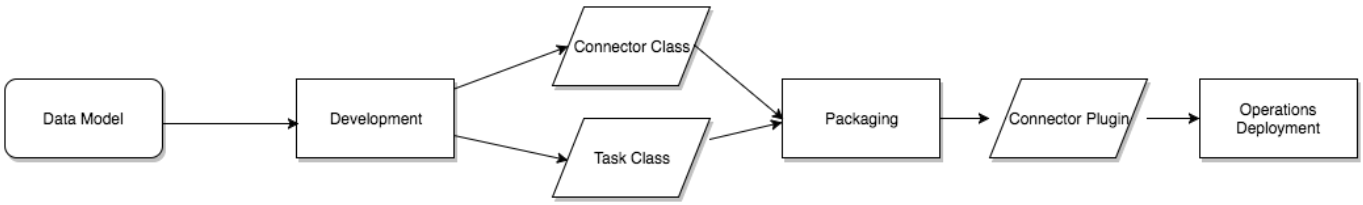
Kafka Connect is a framework to stream data into and out of Apache Kafka®. The Confluent Platform ships with several built-in connectors that can be used to stream data to or from commonly used systems such as relational databases or HDFS. In order to efficiently discuss the inner workings of Kafka Connect, it is helpful to establish a few major concepts.

- Connectors -- the high level abstraction that coordinates data streaming by managing tasks
- Tasks -- the implementation of how data is copied to or from Kafka
- Workers -- the running processes that execute connectors and tasks
- Converters -- the code used to translate data between Connect and the system sending or receiving data
- Transforms -- simple logic to alter each message produced by or sent to a connector

Connectors

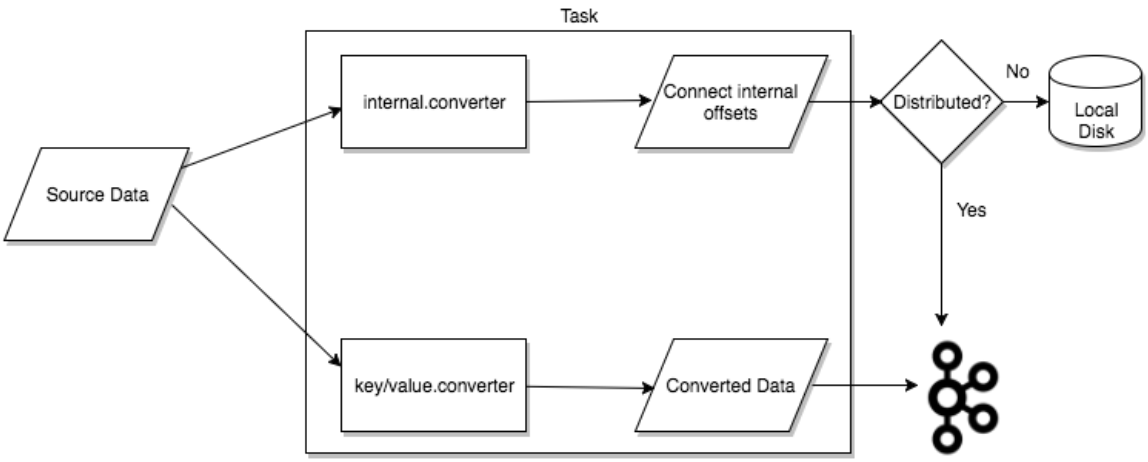
Connectors in Kafka Connect define where data should be copied to and from. A **connector instance** is a logical job that is responsible for managing the copying of data between Kafka and another system. All of the classes that implement or are used by a connector are defined in a **connector plugin**. Both connector instances and connector plugins may be referred to as "connectors", but it should always be clear from the context which is being referred to (e.g., "install a connector" refers to the plugin, and "check the status of a connector" refers to a connector instance).

We encourage users to leverage existing connectors. However, it is possible to write a new connector plugin from scratch. At a high level, a developer who wishes to write a new connector plugin follows the workflow below. Further information is available in the developer guide.



Tasks

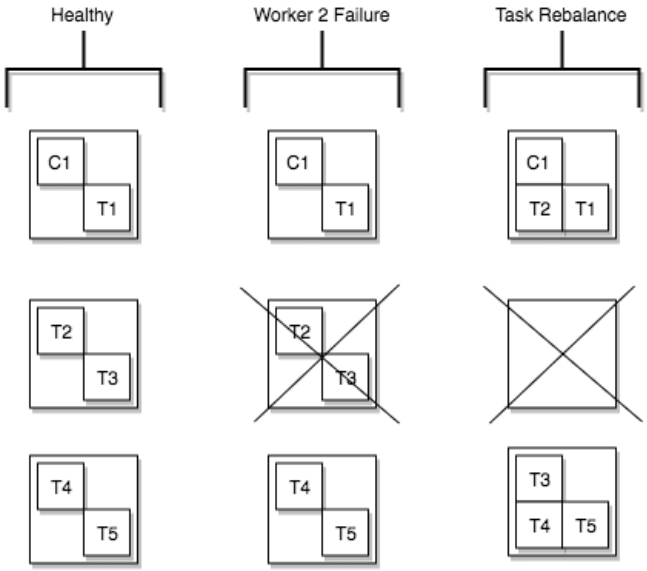
Tasks are the main actor in the data model for Connect. Each connector instance coordinates a set of **tasks** that actually copy the data. By allowing the connector to break a single job into many tasks, Kafka Connect provides built-in support for parallelism and scalable data copying with very little configuration. These tasks have no state stored within them. Task state is stored in Kafka in special topics `config.storage.topic` and `status.storage.topic` and managed by the associated connector. As such, tasks may be started, stopped, or restarted at any time in order to provide a resilient, scalable data pipeline.



High level representation of data passing through a Connect source task into Kafka. Note that internal offsets are stored either in Kafka or on disk rather than within the task itself.

Task Rebalancing

When a connector is first submitted to the cluster, the workers rebalance the full set of connectors in the cluster and their tasks so that each worker has approximately the same amount of work. This same rebalancing procedure is also used when connectors increase or decrease the number of tasks they require, or when a connector's configuration is changed. When a worker fails, tasks are rebalanced across the active workers. When a task fails, no rebalance is triggered as a task failure is considered an exceptional case. As such, failed tasks are not automatically restarted by the framework and should be restarted via the REST API.



Task failover example showing how tasks rebalance in the event of a worker failure.

Workers

Connectors and tasks are logical units of work and must be scheduled to execute in a process. Kafka Connect calls these processes **workers** and has two types of workers: standalone and distributed.

Standalone Workers

Standalone mode is the simplest mode, where a single process is responsible for executing all connectors and tasks.

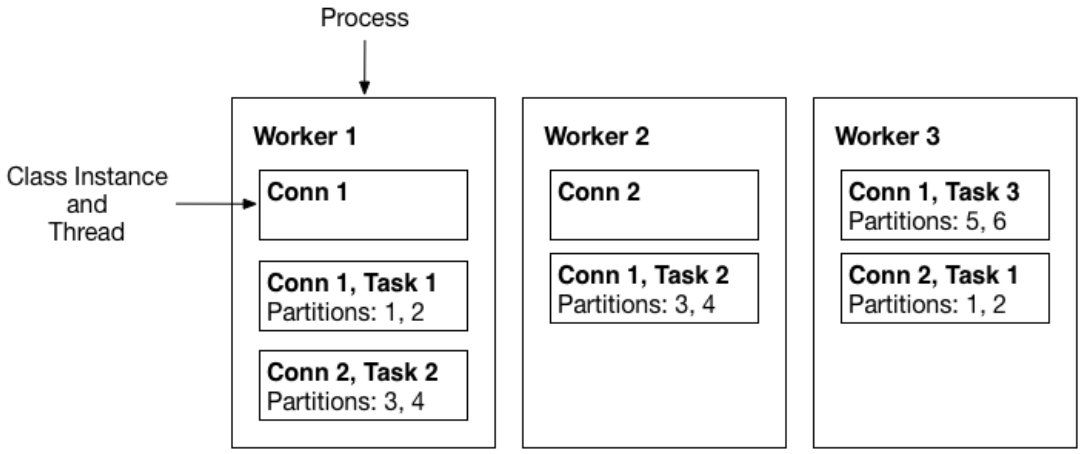
Since it is a single process, it requires minimal configuration. Standalone mode is convenient for getting started, during development, and in certain situations where only one process makes sense, such as collecting logs from a host. However, because there is only a single process, it also has more limited functionality: scalability is limited to the single process and there is no fault tolerance beyond any monitoring you add to the single process.

Distributed 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. If you add a worker, shut down a worker, or a worker fails unexpectedly, the rest of the workers detect this and automatically coordinate to redistribute connectors and tasks across the updated set of available workers. Note the similarity to consumer group rebalance. Under the covers, connect workers are using consumer groups to coordinate and rebalance.

Important

All workers with the same `group.id` will be in the same connect cluster. For example, if worker-a has `group.id=connect-cluster-a` and worker-b has the same `group.id`, worker-a and worker-b will form a cluster called `connect-cluster-a`.



A three-node Kafka Connect distributed mode cluster. Connectors (monitoring the source or sink system for changes that require reconfiguring tasks) and tasks (copying a subset of a connector's data) are automatically balanced across the active workers. The division of work between tasks is shown by the partitions that each task is assigned.

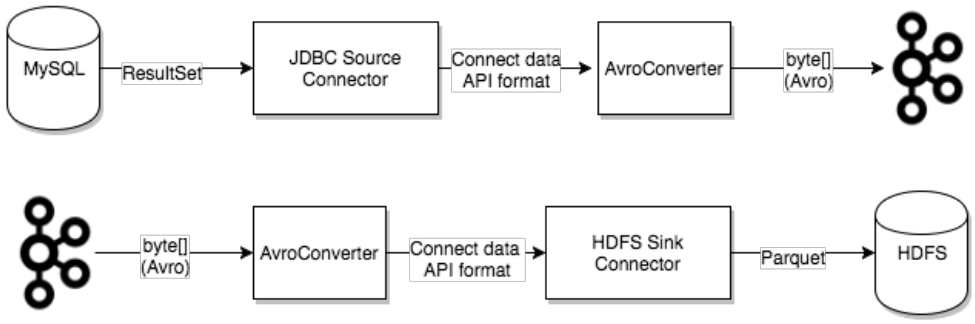
Converters

Converters are necessary to have a Kafka Connect deployment support a particular data format when writing to or reading from Kafka. Tasks use converters to change the format of data from bytes to Connect internal data format and vice versa.

By default, Confluent Platform provides the following converters:

- `AvroConverter` (recommended): use with Confluent Schema Registry
- `JsonConverter`: great for structured data
- `StringConverter`: simple string format
- `ByteArrayConverter`: provides a "pass-through" option that does no conversion

Converters are decoupled from connectors themselves to allow for reuse of converters between connectors naturally. For example, using the same Avro converter, the JDBC Source Connector can write Avro data to Kafka and the HDFS Sink Connector can read Avro data from Kafka. This means the same converter can be used even though, for example, the JDBC source returns a ResultSet that is eventually written to HDFS as a parquet file.



Example showing how converters are used when reading from a database using a JDBC Source Connector, writing to Kafka, and writing to HDFS with an HDFS Sink Connector.

Transforms

Connectors can be configured with transformations to make simple and lightweight modifications to individual messages. This can be convenient for minor data adjustments and event routing, and multiple transformations can be chained together in the connector configuration. However, more complex transformations and operations that apply to multiple messages are best implemented with KSQL and Kafka Streams.

A transform is a simple function that accepts one record as input and outputs a modified record. A number of transforms are provided by Kafka Connect, and these all perform simple but commonly useful modifications. However, you can implement the Transformation interface with your own custom logic, package them as a Kafka Connect plugin, and use them with any connectors.

When transforms are used with a source connector, Kafka Connect passes each source record produced by the connector through the first transformation, which makes its modifications and outputs a new source record. This updated source record is then passed to the next transform in the chain, which generates a new modified source record. This continues for the remaining transforms, and the final updated source record is then converted to the binary form and written to Kafka.

Transforms can also be used with sink connectors. Kafka Connect reads message from Kafka and converts the binary representation to a sink record. If there is a transform, Kafka Connect and passes the record through the first transformation, which makes its modifications and outputs a new, updated sink record. The updated sink record is then passed through the next transform in the chain, which generates a new sink record. This continues for the remaining transforms, and the final updated sink record is then passed to the sink connector for processing.

For more information, see Kafka Connect Transformations.

Transform	Description
Cast	Cast fields or the entire key or value to a specific type, e.g. to force an integer field to a smaller width.
Drop	Drop either a key or a value from a record and set it to null.
ExtractField	Extract the specified field from a Struct when schema present, or a Map in the case of schemaless data. Any null values are passed through unmodified.
ExtractTopic	Replace the record topic with a new topic derived from its key or value.
Flatten	Flatten a nested data structure. This generates names for each field by concatenating the field names at each level with a configurable delimiter character.
HoistField	Wrap data using the specified field name in a Struct when schema present, or a Map in the case of schemaless data.
InsertField	Insert field using attributes from the record metadata or a configured static value.
MaskField	Mask specified fields with a valid null value for the field type.
RegexRouter	Update the record topic using the configured regular expression and replacement string.
ReplaceField	Filter or rename fields.
SetSchemaMetadata	Set the schema name, version, or both on the record's key or value schema.
TimestampConverter	Convert timestamps between different formats such as Unix epoch, strings, and Connect Date and Timestamp types.
TimestampRouter	Update the record's topic field as a function of the original topic value and the record timestamp.
TombstoneHandler	Manage tombstone records. A tombstone record is defined as a record with the entire value field being null, whether or not it has ValueSchema.
ValueToKey	Replace the record key with a new key formed from a subset of fields in the record value.

Last updated on Oct 17, 2019.