Connect to Database

Objective: to insert streaming data (movie ticket sales events) into database
The stream has three fields:

1. `title`, the name of the movie;
2. `sale_ts`, the time at which the ticket was sold; and
3. `ticket_total_value`, the price paid for the ticket.

Example of streaming data

```
INSERT INTO MOVIE_TICKET_SALES (title, sale_ts, ticket_total_value) VALUES
('Aliens', '2019-07-18T10:00:00Z', 10);

INSERT INTO MOVIE_TICKET_SALES (title, sale_ts, ticket_total_value) VALUES
('Die Hard', '2019-07-18T10:00:00Z', 12);

INSERT INTO MOVIE_TICKET_SALES (title, sale_ts, ticket_total_value) VALUES
('Die Hard', '2019-07-18T10:01:00Z', 12);
```

Part 1: Initial process (Produce and Consume with Python)

1. **Create a Kafka cluster via Docker**
    a. Three brokers and one zoo keeper

2. **Create a topic (movie)**
    a. Two partitions and three replicas

3. **Create a producer (movie_producer.py)**
    a. **pip install pymongo #This is for bson usage**

    To collect and send individual movie ticket sales events

```
from confluent_kafka import Producer
import json
import random
import time
from datetime import datetime
from bson import json_util

def acked(err, msg):
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err)))
    else:
        print("Message produced: %s" % (msg.value().decode()))

p = Producer({'bootstrap.servers':'localhost:8097'})

titles = ["ET", "Hulk", "Spiderman"]
prices=[12,24,36]

for i in range(10):

    current_time = datetime.now().strftime("%Y/%m/%d %H:%M:%S")

    data = {
        'title': random.choice(titles),
        'sale_ts': current_time,
        'ticket_total_value' : random.choice(prices)
    }

    p.produce('movie',
              key="key",
```

```
            value=json.dumps(data, default=json_util.default).encode('utf-8'),
            callback=acked)

    p.poll(1)
    time.sleep(random.randint(3,6))
```

4. **Create a consumer (movie_consumer.py)**

```
from confluent_kafka import Consumer
import json
import pandas as pd

running = True

def basic_consume_loop(c, topics):
    try:
        c.subscribe(topics)

        while running:
            msg = c.poll(timeout=1.0)

            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                results = json.loads(msg.value().decode())
                print(results) #print(results[“sale_ts”])

    finally:
        # Close down consumer to commit final offsets.
        c.close()

def shutdown():
    running = False

c = Consumer({'bootstrap.servers':'localhost:8097',
                    'group.id':'group1',
                    'auto.offset.reset':'earliest'})
```

```
basic_consume_loop(c,['movie'])
```

5. **Start running**
    a. python movie_consumer.py (Terminal 1)
    b. python movie_producer.py (Terminal 2)

Todo

- **Run movie_consumer2.py**
- **Run movie_producer.py again**
- **Check data in mysql using “select …**

Part 2: Consumer (Python) insert data into MySQL

1. Create a MySQL database via Docker

c. Download and unzip file "docker-mysql.zip"
d. Use CMD for accessing the "docker-mysql" folder
e. Start mysql docker by typing
   i. docker-compose up -d (**This command will only work if the 'docker-compose.yml' file exists in the same path)

2. Create the table "movie_tb" in mysql db

a. Enter the mysql terminal (How?)

```
mysql -uroot -p

enter password confluent2

show databases;

use connect_test2;

create table movie_tb (
    title varchar(100) not null,
    sale_ts varchar(100) not null,
    ticket_total_value int not null
   );
```

3. Test the DB by inserting a sample data and then select it.

```
INSERT INTO movie_tb (title, sale_ts, ticket_total_value) VALUES
('Aliens', '2019-07-18T10:00:00Z', 10);
```

4. Modify the consumer (movie_consumer.py) and named it as (movie_consumer2.py) and pip install mysql.connector

```
from confluent_kafka import Consumer
import json
import pandas as pd

running = True

import mysql.connector
```

```
connection =
mysql.connector.connect(host="localhost",database="connect_test2",user="confluent2",
password="confluent2", port=3307)

def basic_consume_loop(c, topics):
    try:
        c.subscribe(topics)

        while running:
            msg = c.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                results = json.loads(msg.value().decode())
                print(results)

                insert_cmd = f"insert into movie_tb (title, sale_ts, ticket_total_value) values
                             (          '{results['title']}',
                                        '{results[‘sale_ts’]}',
                                        ‘{results[‘ticket_total_value’]}’)"

                print(insert_cmd)
                cursor = connection.cursor() #Should this line be called once?
                cursor.execute(insert_cmd)
                connection.commit()
    finally:
        # Close down consumer to commit final offsets.
        c.close()

def shutdown():
    running = False

c = Consumer({'bootstrap.servers':'localhost:8097',
                    'group.id':'group1',
                    'auto.offset.reset':'earliest'})

basic_consume_loop(c,['movie'])
```

5. **Start running**
    a. movie_consumer2.py
    b. movie_producer.py

Part 3: Use sink connector (Kafka Connect)

To use connector to insert data into mysql instead of Python code (How to do it?)

3.1 sink.json

```
{
  "name": "sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "tasks.max": "1",
    "topics": "movie",
    "connection.url": "jdbc:mysql://quickstart-mysql:3306/connect_test?useSSL=false",
    "connection.user": "confluent",
    "connection.password": "confluent",
    "table.name.format": "movie_tb",
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url": "http://schema-registry:8081",
    "value.converter.schemas.enable": "true",
    "insert.mode": "insert",
    "pk.mode": "none",
    "errors.tolerance": "all",
    "errors.log.enable": "true",
    "errors.log.include.messages": "true"
  }
}
```

Reference

1. https://docs.confluent.io/platform/current/connect/index.html

Kafka Connect

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka® and other data systems. It makes it simple to quickly define connectors that move large data sets in and out of Kafka. Kafka Connect can ingest entire databases or collect metrics from all your application servers into Kafka topics, making the data available for stream processing with low latency. An export connector can deliver data from Kafka topics into secondary indexes like Elasticsearch, or into batch systems–such as Hadoop for offline analysis.

Try Kafka Connect in Confluent Cloud

Quickly move data in and out of Kafka with fully managed connectors. Sign up for Confluent Cloud with your cloud marketplace account and unlock $1000 in free credits: AWS Marketplace, Google Cloud Marketplace, or Microsoft Azure Marketplace.

This page describes how Kafka Connect works, and includes important Kafka Connect terms and key concepts. You'll learn what Kafka Connect is–including its benefits and framework–and gain the understanding you need to put your data in motion.

What is Kafka Connect?

Kafka Connect is a free, open-source component of Apache Kafka® that serves as a centralized data hub for simple data integration between databases, key-value stores, search indexes, and file systems. You can use Kafka Connect to stream data between Apache Kafka® and other data systems and quickly create connectors that move large data sets in and out of Kafka.

Benefits of Kafka Connect

Kafka Connect provides the following benefits:

- **Data-centric pipeline**: Connect uses meaningful data abstractions to pull or push data to Kafka.
- **Flexibility and scalability**: Connect runs with streaming and batch-oriented systems on a single node (standalone) or scaled to an organization-wide service (distributed).
- **Reusability and extensibility**: Connect leverages existing connectors or extends them to fit your needs and provides lower time to production.

Kafka Connect is focused on streaming data to and from Kafka, making it simpler for you to write high quality, reliable, and high performance connector plugins. Kafka Connect also enables the framework to make guarantees that are difficult to achieve using other frameworks. It is an integral component of an ETL pipeline, when combined with Kafka and a stream processing framework.

How Kafka Connect Works

The Kafka Connect framework allows you to ingest entire databases or collect metrics from all your application servers into Kafka topics, making the data available for stream processing with low latency. An export connector, for example, can deliver data from Kafka topics into secondary indexes like Elasticsearch, or into batch systems–such as Hadoop for offline analysis.

You can deploy Kafka Connect as a standalone process that runs jobs on a single machine (for example, log collection), or as a distributed, scalable, fault-tolerant service supporting an entire organization. Kafka Connect provides a low barrier to entry and low operational overhead. You can start small with a standalone environment for development and testing, and then scale up to a full production environment to support the data pipeline of a large organization.

To deploy Kafka Connect in your environment, see How to Use Kafka Connect - Get Started.

Kafka Connect Concepts

This section describes the following Kafka Connect 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
- Dead Letter Queue: How Connect handles connector errors

Connectors

Kafka Connect includes two types of connectors:

- **Source connector**: Source connectors ingest entire databases and stream table updates to Kafka topics. Source connectors can also collect metrics from all your application servers and

store the data in Kafka topics–making the data available for stream processing with low latency.

- **Sink connector**: Sink connectors deliver data from Kafka topics to secondary indexes, such as Elasticsearch, or batch systems such as Hadoop for offline analysis.

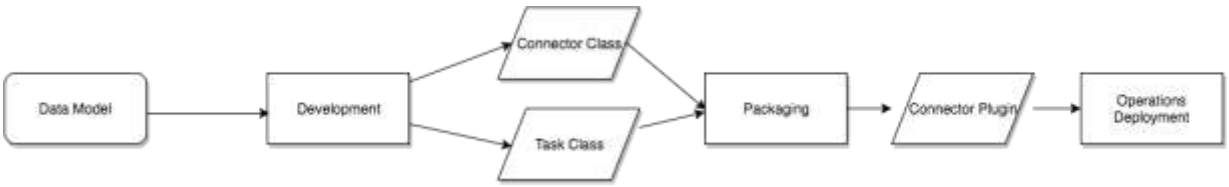
Confluent offers several pre-built connectors that can be used to stream data to or from commonly used systems, such as relational databases or HDFS. In order to efficiently discuss the inner workings of Kafka Connect, it is helpful to establish a few major concepts.

Tip

Confluent Cloud offers pre-built, fully managed, Kafka connectors that make it easy to instantly connect to popular data sources and sinks. With a simple GUI-based configuration and elastic scaling with no infrastructure to manage, Confluent Cloud connectors make moving data in and out of Kafka an effortless task, giving you more time to focus on application development. For information about Confluent Cloud connectors, see Connect to External Services.

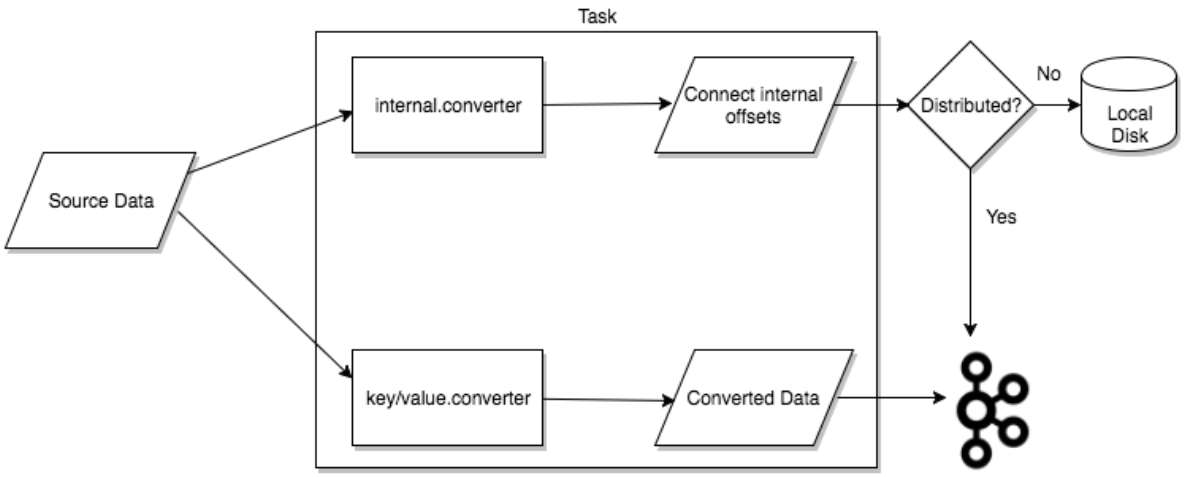
Connectors in Kafka Connect define where data should be copied to and from. A **connector instance** is a logical job that is responsible for managing the copying of data between Kafka and another system. All of the classes that implement or are used by a connector are defined in a **connector plugin**. 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 (for example, "install a connector" refers to the plugin, and "check the status of a connector" refers to a connector instance).

Confluent encourages 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 should keep to the following workflow. Further information is available in the developer guide.



Tasks

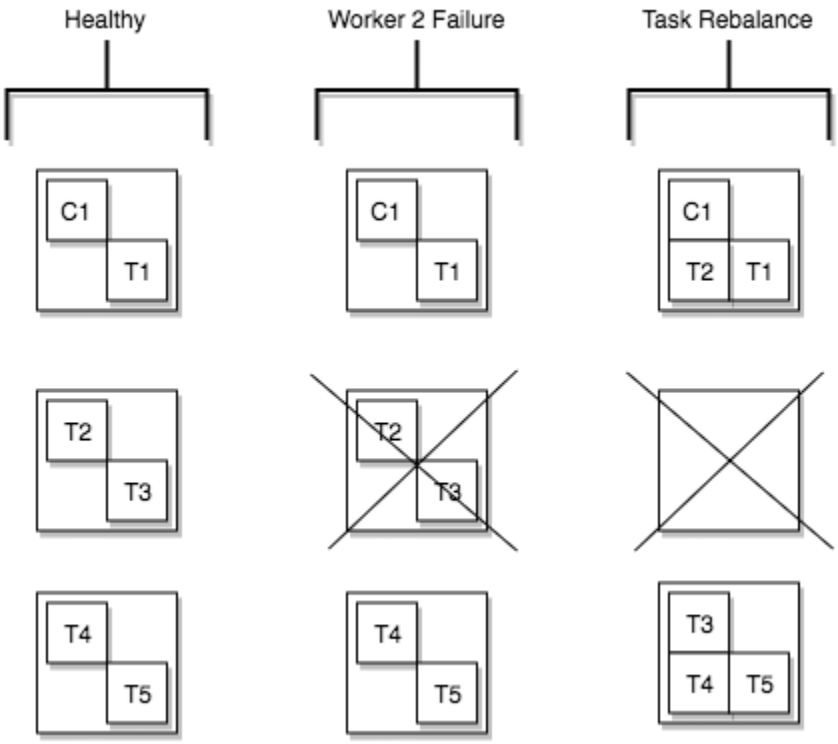
Tasks are the main actor in the data model for Connect. Each connector instance coordinates a set of tasks that copy data. By allowing the connector to break a single job into many tasks, Kafka Connect provides built-in support for parallelism and scalable data copying with minimal configuration. Tasks themselves have no state stored within them. Rather a task's state is stored in special topics in Kafka, `config.storage.topic` and `status.storage.topic`, and managed by the associated connector. Tasks may be started, stopped, or restarted at any time to provide a resilient and 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 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 restarted by the framework and should be restarted using 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 distrubuted.

Standalone workers

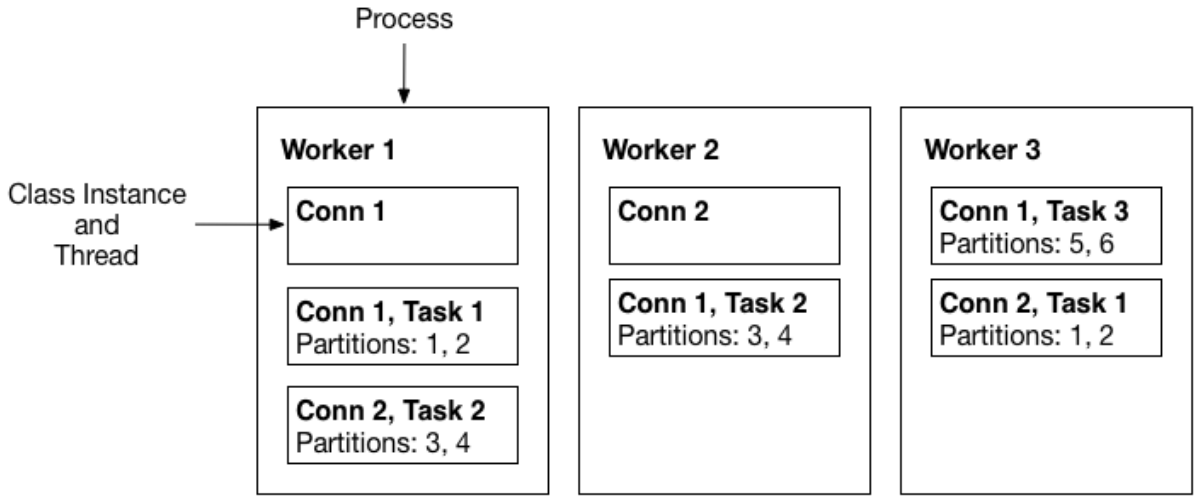
Standalone mode is the simplest mode, where a single process is responsible for executing all connectors and tasks. Since it is a single process, it requires minimal configuration. Standalone mode is convenient for getting started, during development, and in certain situations where only one process 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 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 acknowledge this and coordinate to redistribute connectors and tasks across the

updated set of available workers. Note the similarity to consumer group rebalance. Behind the scenes, connect workers use consumer groups to coordinate and rebalance.

Note that 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 balanced across the active workers. The division of work between tasks is shown by the partitions that each task is assigned.

Converters

Converters are required 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 a Connect internal data format and vice versa.

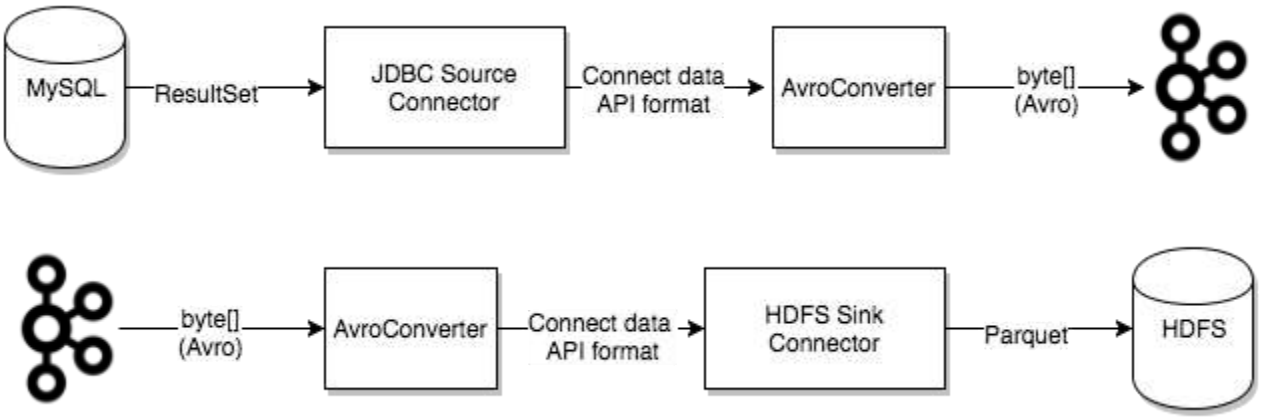
By default, Confluent Platform provides the following converters:

- **AvroConverter** `io.confluent.connect.avro.AvroConverter`: use with Schema Registry
- **ProtobufConverter** `io.confluent.connect.protobuf.ProtobufConverter`: use with Schema Registry
- **JsonSchemaConverter** `io.confluent.connect.json.JsonSchemaConverter`: use with Schema Registry
- **JsonConverter** `org.apache.kafka.connect.json.JsonConverter` (without Schema Registry): use with structured data
- **StringConverter** `org.apache.kafka.connect.storage.StringConverter`: simple string format

- **ByteArrayConverter** `org.apache.kafka.connect.converters.ByteArrayConverter`: provides a "pass-through" option that does no conversion

Converters are decoupled from connectors themselves to allow for the reuse of converters between connectors. 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.

The following graphic shows how converters are used to read from a database using a JDBC Source Connector, write to Kafka, and finally write to HDFS with an HDFS Sink Connector.



You can use the following built-in primitive converters with Connect:

- `org.apache.kafka.connect.converters.DoubleConverter`: Serializes to and deserializes from DOUBLE values. When converting from bytes to Connect format, the converter returns an optional FLOAT64 schema.
- `org.apache.kafka.connect.converters.FloatConverter`: Serializes to and deserializes from FLOAT values. When converting from bytes to Connect format, the converter returns an optional FLOAT32 schema.
- `org.apache.kafka.connect.converters.IntegerConverter`: Serializes to and deserializes from INTEGER values. When converting from bytes to Connect format, the converter returns an optional INT32 schema.
- `org.apache.kafka.connect.converters.LongConverter`: Serializes to and deserializes from LONG values. When converting from bytes to Connect format, the converter returns an optional INT64 schema.
- `org.apache.kafka.connect.converters.ShortConverter`: Serializes to and deserializes from SHORT values. When converting from bytes to Connect format, the converter returns an optional INT16 schema.

For detailed information about converters, see Configuring Key and Value Converters. For information about how converters and Schema Registry work, see Integrate Schemas

from Kafka Connect in Confluent Platform. You can also view Converters and Serialization Explained if you'd like to dive deeper into converters.

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 many transformations can be chained together in the connector configuration. However, more complex transformations and operations that apply to many messages are best implemented with ksqlDB for Confluent Platform and Kafka Streams for Confluent Platform.

A transform is a simple function that accepts one record as an input and outputs a modified record. All transforms provided by Kafka Connect perform simple but commonly useful modifications. Note that you can implement the Transformation interface with your own custom logic, package them as a Kafka Connect plugin, and use them with any connector.

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. The final updated source record is 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 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 Single Message Transform Reference for Confluent Platform.

Dead Letter Queue

Dead Letter Queues (DLQs) are only applicable for sink connectors. Note that for Confluent Cloud sink connectors a DLQ topic is autogenerated. For more information, see Confluent Cloud Dead Letter Queue.

An invalid record may occur for a number of reasons. One example is when a record arrives at a sink connector serialized in JSON format, but the sink connector configuration is expecting Avro format. When an invalid record can't be processed by the sink

connector, the error is handled based on the connector `errors.tolerance` configuration property.

There are two valid values for `errors.tolerance`:

- `none` (default)
- `all`

When `errors.tolerance` is set to `none`, an error or invalid record causes the connector task to immediately fail and the connector goes into a failed state. To resolve this issue, you must review the Kafka Connect Worker log and do the following:

1. Examine what caused the failure.
2. Fix the issue.
3. Restart the connector.

When `errors.tolerance` is set to `all`, all errors or invalid records are ignored and processing continues. No errors are written to the Connect Worker log. To determine if records are failing, you must use internal metrics, or count the number of records at the source and compare that with the number of records processed.

An error-handling feature is available that will route all invalid records to a special topic and report the error. This topic contains a DLQ of records that could not be processed by the sink connector.

Create a Dead Letter Queue topic

To create a DLQ, add the following configuration properties to your sink connector configuration:

```
errors.tolerance = all
errors.deadletterqueue.topic.name = <dead-letter-topic-name>
```

The following example shows a GCS Sink connector configuration with DLQ enabled:

```
{
 "name": "gcs-sink-01",
 "config": {
   "connector.class": "io.confluent.connect.gcs.GcsSinkConnector",
   "tasks.max": "1",
   "topics": "gcs_topic",
   "gcs.bucket.name": "<my-gcs-bucket>",
   "gcs.part.size": "5242880",
   "flush.size": "3",
   "storage.class": "io.confluent.connect.gcs.storage.GcsStorage",
```

```
    "format.class": "io.confluent.connect.gcs.format.avro.AvroFormat",
    "partitioner.class": "io.confluent.connect.storage.partitioner.DefaultPartitioner
",
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url": "http://localhost:8081",
    "schema.compatibility": "NONE",
    "confluent.topic.bootstrap.servers": "localhost:9092",
    "confluent.topic.replication.factor": "1",
    "errors.tolerance": "all",
    "errors.deadletterqueue.topic.name": "dlq-gcs-sink-01"
  }
}
```

Even if the DLQ topic contains the records that failed, it does not show why. You can add the following configuration property to include failed record header information.

```
errors.deadletterqueue.context.headers.enable=true
```

Record headers are added to the DLQ when `errors.deadletterqueue.context.headers.enable` parameter is set to `true`–the default is `false`. You can then use the kafkacat to view the record header and determine why the record failed. Errors are also sent to Connect Reporter. To avoid conflicts with the original record header, the DLQ context header keys start with `_connect.errors`.

Here is the same example configuration with headers enabled:

```
 {
  "name": "gcs-sink-01",
  "config": {
    "connector.class": "io.confluent.connect.gcs.GcsSinkConnector",
    "tasks.max": "1",
    "topics": "gcs_topic",
    "gcs.bucket.name": "<my-gcs-bucket>",
    "gcs.part.size": "5242880",
    "flush.size": "3",
    "storage.class": "io.confluent.connect.gcs.storage.GcsStorage",
    "format.class": "io.confluent.connect.gcs.format.avro.AvroFormat",
    "partitioner.class": "io.confluent.connect.storage.partitioner.DefaultPartitioner
",
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url": "http://localhost:8081",
    "schema.compatibility": "NONE",
    "confluent.topic.bootstrap.servers": "localhost:9092",
    "confluent.topic.replication.factor": "1",
    "errors.tolerance": "all",
    "errors.deadletterqueue.topic.name": "dlq-gcs-sink-01",
    "errors.deadletterqueue.context.headers.enable":true
  }
}
```

For more information about DLQs, see Kafka Connect Deep Dive – Error Handling and Dead Letter Queues.

Use a Dead Letter Queue with security

When you use Confluent Platform with security enabled, the Confluent Platform Admin Client creates the Dead Letter Queue (DLQ) topic. Invalid records are first passed to an internal producer constructed to send these records, and then, the Admin Client creates the DLQ topic.

For the DLQ to work in a secure Confluent Platform environment, you must add additional Admin Client configuration properties (prefixed with `admin.*`) to the Connect Worker configuration. The following SASL/PLAIN example shows additional Connect Worker configuration properties:

```
admin.ssl.endpoint.identification.algorithm=https
admin.sasl.mechanism=PLAIN
admin.security.protocol=SASL_SSL
admin.request.timeout.ms=20000
admin.retry.backoff.ms=500
admin.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
  username="<user>" \
  password="<secret>";
```

For details about configuring your Connect worker, sink connector, and dead letter queue topic in a Role-Based Access Control (RBAC) environment, see Kafka Connect and RBAC.