
StreamReactor Documentation

Release 0.1

Andrew Stevenson

Jan 05, 2018

Contents

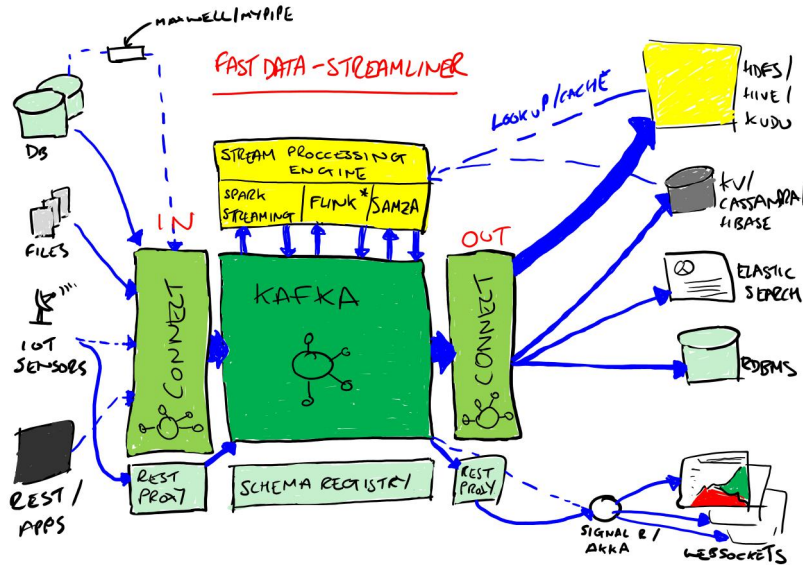
1	Components
----------	-------------------

3

The Stream Reactor is a set of components to build a reference architecture for streaming data platforms. At its core is Kafka, with Kafka Connect providing a unified way to stream data in and out of the system.

The actual processing is left to streaming engines and libraries such as Spark Streaming, Apache Flink, Storm and Kafka Streams.

DataMountaineer provides a range of supporting components to the main technologies, mainly Kafka, Kafka Connect and the Confluent Platform.



Download [here](#).

1.1 Install

The Stream Reactor components are built around The Confluent Platform. They rely on the Kafka Brokers, Zookeepers and optionally the Schema Registry provided by this distribution.

The following releases are available:

- 0.3.0
- 0.2.5
- 0.2.4
- 0.2.3
- 0.2.2

Kafka Version	Confluent Version	Stream reactor version
0.11.0.0	3.3	0.3.0
0.10.2.0	3.2.2	0.2.6
0.10.2.0	3.2	0.2.5
0.10.0.1	3.1	0.2.4
0.10.0.1	3.0.1	0.2.3
0.10.0.1	3.0.1	0.2.2

1.1.1 Docker Install

All the Stream Reactor Connectors, Confluent and UI's for Connect, Schema Registry and topic browsing are available in Docker. The Docker images are available in [DockerHub](#) and maintained by our partner [Landoop](#)

Pull the latest images:

```
docker pull landoop/fast-data-dev
docker pull landoop/fast-data-dev-connect-cluster

#UI's
docker pull landoop/kafka-topics-ui
docker pull landoop/schema-registry-ui
```

Individual docker images are available at DataMountaineers [DockerHub](#). We base our Docker images of Confluents base connector image. This contains a script that uses the environment variables starting with *CONNECT_* to create the Kafka Connect Worker property files. A second script uses the environment variables starting with *CONNECTOR_* to create a properties files for the actual connector we want to start.

Set the *CONNECT_* and *CONNECTOR_* environment variables accordingly when running the images.

On start, the Docker will launch Kafka Connect and the *Connect CLI* will push the the Connector configuration, created from the environment variables to Kafka Connectors once the rest api is up.

Important: We strongly recommend using Landoop's Fast Data Dev dockers. The stream reactor is prepackaged and UI's are included.

Helm Charts



Helm is a package manager for Kubernetes, Helm charts are available for Connectors [here](#) and targeted toward use with the [Landscaper](#).

Microservice architectures are all the rage and for good reason. Small, lightweight, business focused and independently deployable services provide scalability and isolation which it hard to achieve in monolithic systems.

However, adding and scaling lots of tiny processes, either in containers or not can pose challenges even with Kafka as a central data hub of your organisation. At DataMountaineer we are big fans of KStreams and Kafka Connect but as the numbers deployed grow;

1. How do you integrate with your CI/CD street?
2. How do you ensure your design time (provenance) topology is deployed and running?
3. How do you monitor and attach your lineage to this topology?
4. How do you attach different monitoring and alerting criteria?
5. How do you promote different flows to production independently?



Even if you are cool and use Dockers your landscape can still be complex...handling multi tenancy, inspecting and managing docker files, handling service discovery, environment variables and promotion to production.

Kafka Connect and KStreams play well in containers, all state is stored or backed up in Kafka so Eneco started moving off virtual machines onto Kubernetes. When doing this we set out with some goals in mind about how to manage the dataflows that were to be deployed;

1. Have a blueprint of what the landscape (apps in the cluster) looks like
2. Keep track of changes: when, what, why and by who;
3. Allow others to review changes before applying them;
4. Let the changes be promoted to specific environments.

This resulted in the Landscaper which takes a repository containing a desired state description of the landscape and eliminates difference between desired and actual state of releases in a Kubernetes cluster.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

1.1.2 Stream Reactor Install

Download the latest release from [here](#).

Unpack the archive:

```
#Stream reactor release
mkdir stream-reactor
tar xvf stream-reactor-0.3.0-3.3.0.tar.gz -C stream-reactor
```

Within the unpacked directory you will find the following structure:

```
.
|-- LICENSE
|-- README.md
|-- bin
|   |-- connect-cli
|   |-- sr-cli-linux
|   |-- sr-cli-osx
|   `-- start-connect
|-- conf
|   |-- azure-docdb-sink.properties
|   |-- blockchain-source.properties
|   |-- bloomberg-source.properties
|   |-- cassandra-sink.properties
|   |-- cassandra-source-incr.properties
|   |-- cassandra-source.properties
|   |-- coap-hazelcast-sink.properties
|   |-- coap-hazelcast-source.properties
|   |-- coap-sink.properties
|   |-- coap-source.properties
|   |-- druid-sink.properties
|   |-- elastic-sink.properties
|   |-- elastic5-sink.properties
|   |-- ftp-source.properties
|   |-- hazelcast-sink.properties
|   |-- hbase-sink.properties
```

```
| |-- influxdb-sink.properties
| |-- jms-sink.properties
| |-- jms-source.properties
| |-- kudu-sink.properties
| |-- mongodb-sink.properties
| |-- mqtt-source.properties
| |-- mqtt-sink.properties
| |-- redis-sink.properties
| |-- rethink-sink.properties
| |-- rethink-source.properties
| |-- voltdb-sink.properties
| `-- yahoo-source.properties
|-- libs
| |-- kafka-connect-azure-documentdb-0.3.0-3.3.0-all.jar
| |-- kafka-connect-blockchain-0.3.0-3.3.0-all.jar
| |-- kafka-connect-bloomberg-0.3.0-3.3.0-all.jar
| |-- kafka-connect-cassandra-0.3.0-3.3.0-all.jar
| |-- kafka-connect-coap-0.3.0-3.3.0-all.jar
| |-- kafka-connect-druid-0.3.0-3.3.0-all.jar
| |-- kafka-connect-elastic-0.3.0-3.3.0-all.jar
| |-- kafka-connect-elastic5-0.3.0-3.3.0-all.jar
| |-- kafka-connect-ftp-0.3.0-3.3.0-all.jar
| |-- kafka-connect-hazelcast-0.3.0-3.3.0-all.jar
| |-- kafka-connect-hbase-0.3.0-3.3.0-all.jar
| |-- kafka-connect-influxdb-0.3.0-3.3.0-all.jar
| |-- kafka-connect-jms-0.3.0-3.3.0-all.jar
| |-- kafka-connect-kudu-0.3.0-3.3.0-all.jar
| |-- kafka-connect-mongodb-0.3.0-3.3.0-all.jar
| |-- kafka-connect-mqtt-0.3.0-3.3.0-all.jar
| |-- kafka-connect-redis-0.3.0-3.3.0-all.jar
| |-- kafka-connect-rethink-0.3.0-3.3.0-all.jar
| |-- kafka-connect-voltdb-0.3.0-3.3.0-all.jar
| `-- kafka-connect-yahoo-0.3.0-3.3.0-all.jar
```

The `libs` folder contains all the Stream Reactor Connector jars.

1.1.3 Install Confluent

Confluent can be downloaded for [here](#)

```
#make confluent home folder
mkdir confluent

#download confluent
wget http://packages.confluent.io/archive/3.3/confluent-3.3.0-2.11.tar.gz

#extract archive to confluent folder
tar -xvf confluent-3.3.0-2.11.tar.gz -C confluent

#setup variables
export CONFLUENT_HOME=~/.confluent/confluent-3.3.0
```

Start the Confluent platform. Confluent have introduced a new CLI to start the platform, in addition a new `plugins.path` has been added to Kafka Connect. This provides classloader isolation for all Connectors found under this location, improving many dependency issues that are seen at runtime.

Edit the `$CONFLUENT_HOME/etc/schema-registry/connect-avro-distributed.properties`

and set the `plugin.path` to the location you unzipped the location of `$STREAMREACTOR_HOME` you set earlier.

Now start the Confluent Platform

```
#start the whole platform
$CONFLUENT_HOME/bin/confluent start
```

Examine the help menu of the *confluent* cli to see other options. For example:

```
# Stop the platform
confluent stop

# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

1.1.4 Release Notes

3.0.0

Features

- Upgrade CoAP to 2.0.0-M4
- Upgrade to Confluent 3.3 and Kafka 0.11.0.0.
- Added MQTT Sink.
- Add MQTT wildcard support.
- Upgrade CoAP to 2.0.0-M4.
- Added WITHCONVERTERS and WITHTYPE to JMS and MQTT connectors in KCQL to simplify configuration.
- Add flush mode to Kudu sink with a PR from @patsak. Thanks

0.2.6 (Pending)

Features

- Upgrade to Confluent 3.2.2
- Upgrade to KCQL 2x
- Add CQL generator to Cassandra source
- Add KCQL INCREMENTALMODE support to the Cassandra source, bulk mode and the timestamp column type is now take from KCQL
- Support for setting key and truststore type on Cassandra connectors
- Added token based paging support for Cassandra source
- Added default bytes converter to JMS Source
- Added default connection factory to JMS Source
- Added support for SharedDurableConsumers to JMS Connectors

- Upgraded JMS Connector to JMS 2.0
- Moved to Elastic4s 2.4
- Added Elastic5s with TCP, TCP+XPACK and HTTP client support
- Upgrade Azure Documentdb to 1.11.0
- Added optional progress counter to all connectors, it can be enabled with `connect.progress.enabled` which will periodically report log messages processed
- Added authentication and TLS to ReThink Connectors
- Added TLS support for ReThinkDB, add batch size option to source for draining the internal queues.
- Upgrade Kudu Client to 1.4.0
- Support for dates in Elastic Indexes and custom document types
- Upgrade Connect CLI to 1.0.2 (Renamed to connect-cli)

Bug Fixes

- Fixes for high CPU on CoAP source
- Fixes for high CPU on Cassandra source
- Fixed Avro double fields mapping to Kudu columns
- Fixes on JMS properties converter, Invalid schema when extracting properties

Misc

- Refactored Cassandra Tests to use only one embedded instance
- Removed unused batch size and bucket size options from Kudu, they are taken from KCQL
- Removed unused batch size option from DocumentDb
- Rename Azure DocumentDb `connect.documentdb.db` to `connect.documentdb.db`
- Rename Azure DocumentDb `connect.documentdb.database.create` to `connect.documentdb.db.create`
- Rename Cassandra Source `connect.cassandra.source.kcql` to `connect.cassandra.kcql`
- Rename Cassandra Source `connect.cassandra.source.timestamp.type` to `connect.cassandra.timestamp.type`
- Rename Cassandra Source `connect.cassandra.source.import.poll.interval` to `connect.cassandra.import.poll.interval`
- Rename Cassandra Source `connect.cassandra.source.error.policy` to `connect.cassandra.error.policy`
- Rename Cassandra Source `connect.cassandra.source.max.retries` to `connect.cassandra.max.retries`
- Rename Cassandra Sink `connect.cassandra.source.retry.interval` to `connect.cassandra.retry.interval`
- Rename Cassandra Sink `connect.cassandra.sink.kcql` to `connect.cassandra.kcql`
- Rename Cassandra Sink `connect.cassandra.sink.error.policy` to `connect.cassandra.error.policy`
- Rename Cassandra Sink `connect.cassandra.sink.max.retries` to `connect.cassandra.max.retries`
- Rename Cassandra Sink Sink `connect.cassandra.sink.retry.interval` to `connect.cassandra.retry.interval`
- Rename Coap Source `connect.coap.bind.port` to `connect.coap.port`
- Rename Coap Sink `connect.coap.bind.port` to `connect.coap.port`
- Rename Coap Source `connect.coap.bind.host` to `connect.coap.host`
- Rename Coap Sink `connect.coap.bind.host` to `connect.coap.host`

- Rename MongoDB *connect.mongo.database* to *connect.mongo.db*
- Rename MongoDB *connect.mongo.sink.batch.size* to *connect.mongo.batch.size*
- Rename Druid *connect.druid.sink.kcql* to *connect.druid.kcql*
- Rename Druid *connect.druid.sink.conf.file* to *connect.druid.kcql*
- Rename Druid *connect.druid.sink.write.timeout* to *connect.druid.write.timeout*
- Rename Elastic *connect.elastic.sink.kcql* to *connect.elastic.kcql*
- Rename HBase *connect.hbase.sink.column.family* to *connect.hbase.column.family*
- Rename HBase *connect.hbase.sink.kcql* to *connect.hbase.kcql*
- Rename HBase *connect.hbase.sink.error.policy* to *connect.hbase.error.policy*
- Rename HBase *connect.hbase.sink.max.retries* to *connect.hbase.max.retries*
- Rename HBase *connect.hbase.sink.retry.interval* to *connect.hbase.retry.interval*
- Rename Influx *connect.influx.sink.kcql* to *connect.influx.kcql*
- Rename Influx *connect.influx.connection.user* to *connect.influx.username*
- Rename Influx *connect.influx.connection.password* to *connect.influx.password*
- Rename Influx *connect.influx.connection.database* to *connect.influx.db*
- Rename Influx *connect.influx.connection.url* to *connect.influx.url*
- Rename Kudu *connect.kudu.sink.kcql* to *connect.kudu.kcql*
- Rename Kudu *connect.kudu.sink.error.policy* to *connect.kudu.error.policy*
- Rename Kudu *connect.kudu.sink.retry.interval* to *connect.kudu.retry.interval*
- Rename Kudu *connect.kudu.sink.max.retries* to *connect.kudu.max.retries*
- Rename Kudu *connect.kudu.sink.schema.registry.url* to *connect.kudu.schema.registry.url*
- Rename Redis *connect.redis.connection.password* to *connect.redis.password*
- Rename Redis *connect.redis.sink.kcql* to *connect.redis.kcql*
- Rename Redis *connect.redis.connection.host* to *connect.redis.host*
- Rename Redis *connect.redis.connection.port* to *connect.redis.port*
- Rename ReThink *connect.rethink.source.host* to *connect.rethink.host*
- Rename ReThink *connect.rethink.source.port* to *connect.rethink.port*
- Rename ReThink *connect.rethink.source.db* to *connect.rethink.db*
- Rename ReThink *connect.rethink.source.kcql* to *connect.rethink.kcql*
- Rename ReThink Sink *connect.rethink.sink.host* to *connect.rethink.host*
- Rename ReThink Sink *connect.rethink.sink.port* to *connect.rethink.port*
- Rename ReThink Sink *connect.rethink.sink.db* to *connect.rethink.db*
- Rename ReThink Sink *connect.rethink.sink.kcql* to *connect.rethink.kcql*
- Rename JMS *connect.jms.user* to *connect.jms.username*
- Rename JMS *connect.jms.converters.source* to *connect.jms.converters*
- Remove JMS *connect.jms.converters* and replace my *kcql* with *Converters*

- Remove JMS *connect.jms.queues* and replace my kcql *withType=QUEUE*
- Remove JMS *connect.jms.topics* and replace my kcql *withType=TOPIC*
- Rename Mqtt *connect.mqtt.source.kcql* to *connect.mqtt.kcql*
- Rename Mqtt *connect.mqtt.user* to *connect.mqtt.username*
- Rename Mqtt *connect.mqtt.hosts* to *connect.mqtt.connection.hosts*
- Remove Mqtt *connect.mqtt.converters* and replace my kcql *withConverters*
- Remove Mqtt *connect.mqtt.queues* and replace my kcql *withType=QUEUE*
- Remove Mqtt *connect.mqtt.topics* and replace my kcql *withType=TOPIC*
- Rename Hazelcast *connect.hazelcast.sink.kcql* to *connect.hazelcast.kcql*
- Rename Hazelcast *connect.hazelcast.sink.group.name* to *connect.hazelcast.group.name*
- Rename Hazelcast *connect.hazelcast.sink.group.password* to *connect.hazelcast.group.password*
- Rename Hazelcast *connect.hazelcast.sink.cluster.members* to *connect.hazelcast.cluster.members*
- Rename Hazelcast *connect.hazelcast.sink.batch.size* to *connect.hazelcast.batch.size*
- Rename Hazelcast *connect.hazelcast.sink.error.policy* to *connect.hazelcast.error.policy*
- Rename Hazelcast *connect.hazelcast.sink.max.retries* to *connect.hazelcast.max.retries*
- Rename Hazelcast *connect.hazelcast.sink.retry.interval* to *connect.hazelcast.retry.interval*
- Rename VoltDB *connect.volt.sink.kcql* to *connect.volt.kcql*
- Rename VoltDB *connect.volt.sink.connection.servers* to *connect.volt.servers*
- Rename VoltDB *connect.volt.sink.connection.user* to *connect.volt.username*
- Rename VoltDB *connect.volt.sink.connection.password* to *connect.volt.password*
- Rename VoltDB *connect.volt.sink.error.policy* to *connect.volt.error.policy*
- Rename VoltDB *connect.volt.sink.max.retries* to *connect.volt.max.retries*
- Rename VoltDB *connect.volt.sink.retry.interval* to *connect.volt.retry.interval*

0.2.5

- Adding Azure DocumentDb Sink
- Adding UPSERT to Elastic Search
- Cassandra improvements *withunwrap*
- Upgrade to Kudu 1.0 and CLI 1.0
- Add ingest_time to CoAP Source
- Support Confluent 3.2 and Kafka 0.10.2.
- Added Azure DocumentDB.
- Added JMS Source.
- Added Schemaless Json and Json with schema support to JMS Sink.
- InfluxDB bug fixes for tags and field selection.
- Support for Cassandra data type of `timestamp` in the Cassandra Source for timestamp tracking.

0.2.4 (26 Jan 2017)

- Added FTP and HTTP Source.
- Added InfluxDB tag support. KCQL: INSERT INTO target dimension SELECT * FROM influx-topic WITHTIMESTAMP sys_time() WITHTAG(field1, CONSTANT_KEY1=CONSTANT_VALUE1, field2,CONSTANT_KEY2=CONSTANT_VALUE1)
- Added InfluxDb consistency level. Default is ALL. Use connect.influx.consistency.level to set it to ONE/QUORUM/ALL/ANY.
- InfluxDb connect.influx.sink.route.query was renamed to connect.influx.sink.kcql.
- Added support for multiple contact points in Cassandra.

0.2.3 (5 Jan 2017)

- Added CoAP Source and Sink.
- Added MongoDB Sink.
- Added MQTT Source.
- Hazelcast support for ring buffers, maps, sets, lists and cache.
- Redis support for Sorted Sets.
- Added start scripts.
- Added Kafka Connect and Schema Registry CLI.
- Kafka Connect CLI now supports pause/restart/resume; checking connectors on the classpath and validating configuration of connectors.
- Support for Struct, Schema.STRING and Json with schema in the Cassandra, ReThinkDB, InfluxDB and MongoDB sinks.
- Rename export.query.route to sink.kcql.
- Rename import.query.route to source.kcql.
- Upgrade to KCQL 0.9.5 - Add support for STOREAS so specify target sink types, e.g. Redis Sorted Sets, Hazelcast map, queues, ringbuffers.

Fast Data Dev

This is Docker image for development.

If you need

1. Kafka Broker
2. ZooKeeper
3. Schema Registry
4. Kafka REST Proxy
5. Kafka Connect Distributed
6. Certified DataMountaineer Connectors (ElasticSearch, Cassandra, Redis ..)
7. Landoop's Fast Data Web UIs : schema-registry , kafka-topics , kafka-connect and

8. Embedded integration tests with examples

Run with:

```
docker run --rm -it --net=host landoop/fast-data-dev
```

On Mac OSX run:

```
docker run --rm -it \
  -p 2181:2181 -p 3030:3030 -p 8081:8081 \
  -p 8082:8082 -p 8083:8083 -p 9092:9092 \
  -e ADV_HOST=127.0.0.1 \
  landoop/fast-data-dev
```

That's it. Your Broker is at localhost:9092, your Kafka REST Proxy at localhost:8082, your Schema Registry at localhost:8081, your Connect Distributed at localhost:8083, your ZooKeeper at localhost:2181 and at <http://localhost:3030> you will find Landoop's Web UIs for Kafka Topics and Schema Registry, as well as a Coyote test report.

The screenshot displays the 'Kafka Development Environment' dashboard, a web interface for managing a Kafka cluster. The dashboard is titled 'Kafka Development Environment' and 'docker container powered by Landoop'. It features four main sections: 'SCHEMAS' (4), 'TOPICS' (11), 'CONNECTORS' (0), and 'BROKERS' (1). Each section has a corresponding UI link and a description. Below these are 'RUNNING SERVICES' (fast-data-dev) and 'COYOTE HEALTH CHECKS' (100% passed). The 'SERVICES PORTS' section lists the ports for Kafka Broker, Schema Registry, Kafka REST Proxy, Kafka Connect Distributed, ZooKeeper, and Web Server. The dashboard is powered by Landoop and includes a 'Report Issues & Stars!' button.

Section	Count	UI Link	Description
SCHEMAS	4	SCHEMA REGISTRY UI	Manage Avro schemas of your Kafka cluster
TOPICS	11	KAFKA TOPICS UI	Browse topics and download data
CONNECTORS	0	KAFKA CONNECT UI	Easy configure and manage connectors
BROKERS	1	KAFKA BROKERS	Kafka management and monitoring

RUNNING SERVICES
fast-data-dev

- ✓ Confluent v3.0.1 - Kafka v0.10.0.1
1 broker, including kafka connect distributed, schema registry, kafka rest, etc
- ✓ Datamountaineer Stream Reactor v0.2.2
Source & Sink connectors supporting KSQL, Elasticsearch, Cassandra, Redis etc
- ✓ Landoop Fast Data platform v0.7
Manage schemas, view history, browse topics & configure your cluster from your browser.

COYOTE HEALTH CHECKS
Your set up is being verified using the Coyote integration testing tool and generates working examples.

100%

Passed: 39 | Failed: 0

[VIEW RESULTS](#)

Find out more about Coyote awesome tool [here](#).

SERVICES PORTS

- 9092 : Kafka Broker
- 8081 : Schema Registry
- 8082 : Kafka REST Proxy
- 8083 : Kafka Connect Distributed
- 2181 : ZooKeeper
- 3030 : Web Server

[Report Issues & Stars!](#)

[Issue](#) 2 [Star](#) 6

powered by Landoop

Fast Data Dev Connect

This docker is targeted to more advanced users and is a special case since it doesn't set-up a Kafka cluster, instead it expects to find a Kafka Cluster with Schema Registry up and running.

The developer can then use this docker image to setup a connect-distributed cluster by just spawning a couple containers.

```
docker run -d --net=host \
  -e ID=01 \
  -e BS=broker1:9092,broker2:9092 \
  -e ZK=zk1:2181,zk2:2181 \
  -e SC=http://schema-registry:8081 \
  -e HOST=<IP OR FQDN> \
  landoop/fast-data-dev-connect-cluster
```

Things to look out for in configuration options:

1. It is important to give a full URL (including schema —<http://>) for schema registry.
2. ID should be unique to the Connect cluster you setup, for current and old instances. This is because Connect stores data in Brokers and Schema Registry. Thus even if you destroyed a Connect cluster, its data remain in your Kafka setup.
3. HOST should be set to an IP address or domain name that other connect instances and clients can use to reach the current instance. We chose not to try to autodetect this IP because such a feat would fail more often than not. Good choices are your local network ip (e.g 10.240.0.2) if you work inside a local network, your public ip (if you have one and want to use it) or a domain name that is resolvable by all the hosts you will use to talk to Connect.

If you don't want to run with `--net=host` you have to expose Connect's port which at default settings is 8083. There a `PORT` option, that allows you to set Connect's port explicitly if you can't use the default 8083. Please remember that it is important to expose Connect's port on the same port at the host. This is a choice we had to make for simplicity's sake.

```
docker run -d \
  -e ID=01 \
  -e BS=broker1:9092,broker2:9092 \
  -e ZK=zk1:2181,zk2:2181 \
  -e SC=http://schema-registry:8081 \
  -e HOST=<IP OR FQDN> \
  -e PORT=8085 \
  -p 8085:8085 \
  landoop/fast-data-dev-connect-cluster
```

Advanced

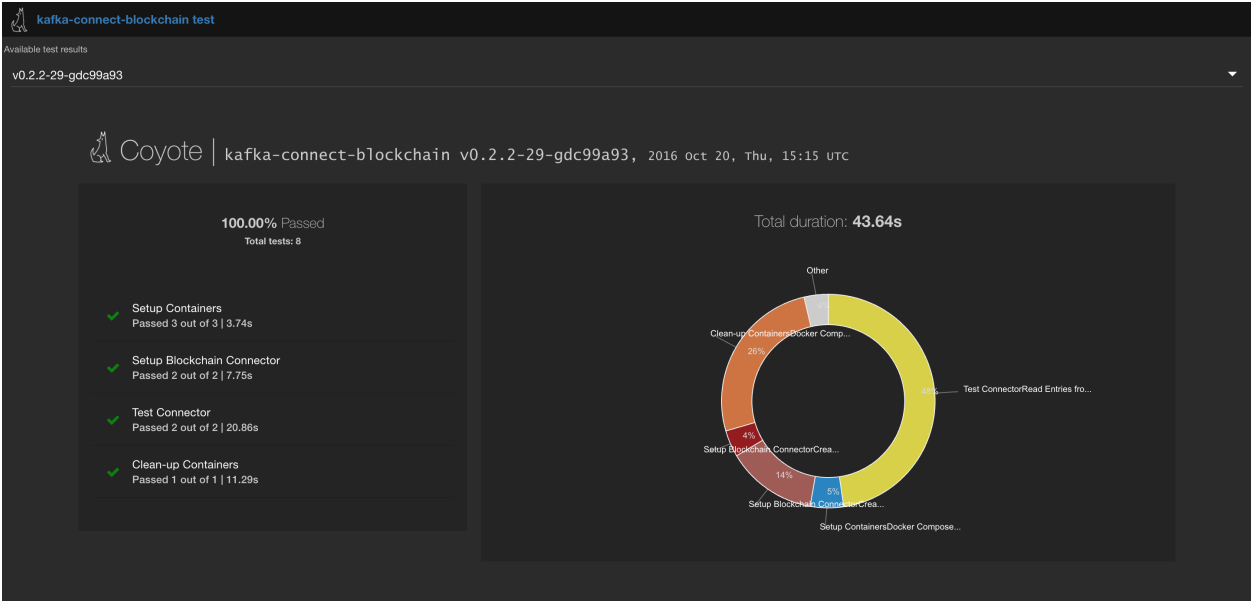
The container does not exit with CTRL+C. This is because we chose to pass control directly to Connect, so you check your logs via docker logs. You can stop it or kill it from another terminal.

Whilst the `PORT` variable sets the rest.port, the `HOST` variable sets the advertised host. This is the hostname that Connect will send to other Connect instances. By default Connect listens to all interfaces, so you don't have to worry as long as other instances can reach each instance via the advertised host.

Latest Test Results

To see the latest tests for the Connectors, in a docker, please visit Landoop's test github [here](#) Test results can be found [here](#).

An example for BlockChain is:



1.2 Kafka Connect

Kafka Connect is a tool to rapidly stream events in and out of Kafka. It has a narrow focus on data ingress in and egress out of the central nervous system of modern streaming frameworks. It is not an ETL and this separation of concerns allows developers to quickly build robust, durable and scalable pipelines in and out of Kafka.

Kafka connect forms an integral component in an ETL pipeline when combined with Kafka and a stream processing framework.

1.2.1 Modes

Kafka Connect can run either as a standalone process for testing and one-off jobs, or as a distributed, scalable, fault tolerant service supporting an entire organisations. This allows it to scale down to development, testing, and small production deployments with a low barrier to entry and low operational overhead, and to scale up to support a large organisations data pipeline.

Usually you'd run in distributed mode to get fault tolerance and better performance. In distributed mode you start Connect on multiple hosts and they join together to form a cluster. Connectors which are then submitted are distributed across the cluster.

For workers to join a Connect cluster, set the `group.id` in the `$CONFLUENT_HOME/etc/schema-registry/connect-avro-distributed.properties` file.

```
# The group ID is a unique identifier for the set of workers that form a single Kafka_
↪Connect
# cluster
group.id=connect-cluster
```

1.2.2 Schema Registry Support

DataMountaineer recommends all payloads in Kafka are Avro. Schema Registry provides a serving layer for your metadata. It provides a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas, provides multiple compatibility settings and allows evolution of schemas according to the configured compatibility setting. It provides serializers that plug into Kafka clients that handle schema storage and retrieval for Kafka messages that are sent in the Avro format.

All our Connectors support Avro and use the Confluent provided converters to translate the Avro into Kafka Connects internal `Struct` type to determine the schema and how to map onto the target sink store.

We have found some of the clients have already an infrastructure where they publish pure json on the topic and obviously the jump to follow the best practice and use schema registry is quite an ask. So we offer support for them as well for the following Sinks:

- ReThinkDB
- MongoDB
- InfluxDB
- DSE Cassandra Sink
- JMS - TextMessages only
- CoAP Sink
- MQTT Sink
- Elastic2x and 5x Sinks

We are upgrading the remaining Connectors. This allows plain text payloads with a json string.

1.2.3 Connectors

Kafka Connect Query Language

The Kafka Connect Query Language is implemented in antlr4 grammar files.

Why ?

While working on our sink/sources we ended up producing quite complex configuration in order to support the functionality required. Imagine a Sink where you Source from different topics and from each topic you want to cherry pick the payload fields or even rename them. Furthermore you might want the storage structure to be automatically created and/or even evolve or you might add new support for the likes of bucketing (Riak TS has one such scenario). Imagine the JDBC sink with a table which needs to be linked to two different topics and the fields in there need to be aligned with the table column names and the complex configuration involved ... or you can just write this

```
routes.query = "INSERT INTO transactions SELECT field1 as column1, field2 as column2, ↵
↵field3 FROM topic_A;
                INSERT INTO transactions SELECT fieldA1 as column1, fieldA2 as column2, ↵
↵fieldC FROM topic_B;"
```

Kafka Connect Query Language

There are two paths supported by this DSL. One is the INSERT that takes the following form (not all grammar is shown):

```
INSERT INTO $TARGET
SELECT *|columns
FROM $TOPIC_NAME
    [IGNORE columns]
    [AUTOCREATE]
    [PK columns]
    [AUTOEVOLVE]
    [BATCH = N]
    [CAPITALIZE]
    [PARTITIONBY cola[,colb]]
    [DISTRIBUTEBY cola[,colb]]
    [CLUSTERBY cola[,colb]]
    [WITHTIMESTAMP cola|sys_time()]
    [WITHFORMAT TEXT|JSON|AVRO|BINARY|OBJECT|MAP]
    [STOREAS $YOUR_TYPE([key=value, .....])]
```

and a *select* only:

```
SELECT *|columns
FROM $TOPIC_NAME
    [IGNORE columns]
    [WITHFORMAT TEXT|JSON|AVRO|BINARY]
    [WITHGROUP $YOUR_CONSUMER_GROUP]
    [WITHPARTITION (partition),[(partition, offset)]]
    [SAMPLE $RECORDS_NUMBER EVERY $SLIDE_WINDOW]
```

Examples

```

SELECT field1 FROM mytopic           // Project one avro field named field1
SELECT field1 AS newName             // Project and renames a field
SELECT * FROM mytopic                // Select everything - perfect for avro
↳ evolution
SELECT *, field1 AS newName FROM mytopic // Select all & rename a field -
↳ excellent for avro evolution
SELECT * FROM mytopic IGNORE badField // Select all & ignore a field -
↳ excellent for avro evolution
SELECT * FROM mytopic PK field1,field2 // Select all & with primary keys (for
↳ the sources where primary keys are required)
SELECT * FROM mytopic AUTOCREATE      // Select all and create the target
↳ Source (table for databases)
SELECT * FROM mytopic AUTOEVOLVE      // Select all & reflect the new fields
↳ added to the avro payload into the target

```

Source Connectors

Source connectors load or stream data from external systems into Kafka.

Kafka Connect Blockchain

A Connector to hook into the live streaming providing a real time feed for new bitcoin blocks and transactions provided by www.blockchain.info. The connector subscribe to notification on blocks, transactions or an address and receive JSON objects describing a transaction or block when an event occurs. This json is then pushed via kafka connect to a kafka topic and therefore can be consumed either by a Sink or have a live stream processing using for example Kafka Streams.

Since is a direct websocket connection the Source will only ever use one connector task at any point. There is no point spawning more and then have duplicate data.

One thing to remember is the subscription API from blockchain doesn't offer an option to start from a given timestamp. This means if the connect worker is down then you will miss some data.

The Sink connects to unconfirmed transaction!! Read more about the live data [here](#)

Prerequisites

- Confluent 3.3
- Java 1.8
- Scala 2.11

Confluent Setup

Follow the instructions [here](#).

Source Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for BlockChain.

```
bin/connect-cli create blockchain-source < conf/blockchain-source.properties

#Connector `blockchain-source`:
name=blockchain-source
connector.class=com.datamountaineer.streamreactor.connect.blockchain.source.
↳BlockchainSourceConnector
max.tasks=1
connect.blockchain.kafka.topic = blockchain-test
max.tasks=1
#task ids:
```

The `blockchain-source.properties` file defines:

1. The name of the source.
2. The Source class.
3. The max number of tasks the connector is allowed to created (1 task only).
4. The topics to write to.

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

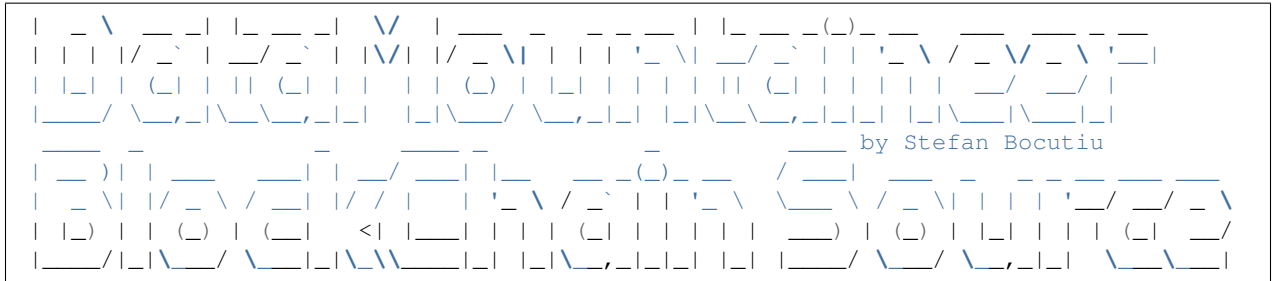
# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
blockchain-source
```

```
# Get connects logs
connect log connect

[2016-08-21 20:31:36,398] INFO Finished starting connectors and tasks (org.apache.
↳kafka.connect.runtime.distributed.DistributedHerder:769)
[2016-08-21 20:31:36,406] INFO
```



Test Records

Now we need to see records pushed on the topic. We can use the `kafka-avro-console-producer` to do this.

```
$ ./bin/kafka-avro-console-consumer --topic blockchain-test \
  --zookeeper localhost:2181 \
  --from-beginning
```

Now the console is reading blockchain transaction data which would print on the terminal.

Configurations

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Bloomberg

Kafka Connect Bloomberg is a Source connector to subscribe to Bloomberg feeds via the Bloomberg labs open API and write to Kafka.

Prerequisites

- Bloomberg subscription
- Confluent 3.3
- Java 1.8
- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

Source Connector QuickStart

We you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the kafka-connect-tools *cli* to post in our distributed properties file for Redis. If you are using the *dockers* you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create bloomberg-source < conf/bloomberg-source.properties
#Connector name=name=`bloomberg-source`
connector.class=com.datamountaineer.streamreactor.connect.bloomberg.
↳BloombergSourceConnector
tasks.max=1
connect.bloomberg.server.host=localhost
connect.bloomberg.server.port=8194
connect.bloomberg.service.uri=//blp/mkdata
connect.bloomberg.subscriptions=AAPL US Equity:LAST_PRICE,BID,ASK;IBM US Equity:BID,
↳ASK,HIGH,LOW,OPEN
kafka.topic=bloomberg
connect.bloomberg.buffer.size=4096
connect.bloomberg.authentication.mode=USER_AND_APPLICATION
#task ids: 0
```

The bloomberg-source.properties file defines:

1. The connector name.
2. The class containing the connector.
3. The number of tasks the connector is allowed to start.
4. The Bloomberg server host.
5. The Bloomberg server port.
6. The Bloomberg service uri.
7. The subscription keys to subscribe to.
8. The topic to write to.
9. The buffer size for the Bloomberg API to buffer events in.
10. The authentication mode.

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
bloomberg-source
```

Test Records

Now we need to see records pushed on the topic. We can use the `kafka-avro-console-producer` to do this.

```
$ ./bin/kafka-avro-console-consumer --topic bloomberg \
  --zookeeper localhost:2181 \
  --from-beginning
```

Now the console is reading bloomberg transaction data which would print on the terminal.

Features

The Source Connector allows subscriptions to BPIPE mkdata and refdata endpoints to feed data into Kafka.

Configurations

`connect.bloomberg.server.host`

The bloomberg endpoint to connect to.

- Data type : string
- Optional : no

`connect.bloomberg.server.port`

The Bloomberg endpoint to connect to.

- Data type : string
- Optional : no

`connect.bloomberg.service.uri`

Which Bloomberg service to connect to. Can be `//blp/mkdata` or `//blp/refdata`.

- Data type : string
- Optional : no

`connect.bloomberg.authentication.mode`

The mode to authentication against the Bloomberg server. Either `APPLICATION_ONLY` or `USER_AND_APPLICATION`.

- Data type : string
- Optional : no

`connect.bloomberg.subscriptions`

- Data type : string
- Optional : no

Specifies which ticker subscription to make. The format is `TICKER:FIELD,FIELD,..`; e.g. `AAPL US Equity:LAST_PRICE;IBM US Equity:BID`

`connect.bloomberg.buffer.size`

- Data type : int
- Optional : yes

- Default : 2048

The buffer accumulating the data updates received from Bloomberg. If not provided it will default to 2048. If the buffer is full and a new update will be received it won't be added to the buffer until it is first drained.

`connect.bloomberg.kafka.topic`

The topic to write to.

- Data type : string
- Optional : no

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=bloomberg-source
connector.class=com.datamountaineer.streamreactor.connect.bloomberg.
↳BloombergSourceConnector
tasks.max=1
connect.bloomberg.server.host=localhost
connect.bloomberg.server.port=8194
connect.bloomberg.service.uri=/blp/mkdata
connect.bloomberg.subscriptions=AAPL US Equity:LAST_PRICE,BID,ASK;IBM US Equity:BID,
↳ASK,HIGH,LOW,OPEN
kafka.topic=bloomberg
connect.bloomberg.buffer.size=4096
```

Schema Evolution

TODO

Deployment Guidelines

TODO

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Cassandra Source

Kafka Connect Cassandra is a Source Connector for reading data from Cassandra and writing to Kafka.

The Source supports:

1. *The KCQL routing querying* - Allows for table to topic routing.
2. Incremental mode with timestamp, timeuuid and tokens support via kcql.
3. Bulk mode
4. Error policies for handling failures.

Prerequisites

- Cassandra 3.0.9
- Confluent 3.2
- Java 1.8
- Scala 2.11

Setup

Before we can do anything, including the QuickStart we need to install Cassandra and the Confluent platform.

Cassandra Setup

First download and install Cassandra if you don't have a compatible cluster available.

```
#make a folder for cassandra
mkdir cassandra

#Download Cassandra
wget http://apache.cs.uu.nl/cassandra/3.5/apache-cassandra-3.5-bin.tar.gz

#extract archive to cassandra folder
tar -xvf apache-cassandra-3.5-bin.tar.gz -C cassandra

#Set up environment variables
export CASSANDRA_HOME=~/.cassandra/apache-cassandra-3.5-bin
export PATH=$PATH:$CASSANDRA_HOME/bin

#Start Cassandra
sudo sh ~/.cassandra/bin/cassandra
```

Confluent Setup

Follow the instructions [here](#).

Source Connector

The Cassandra Source connector allows you to extract entries from Cassandra with the CQL driver and write them into a Kafka topic.

Each table specified in the configuration is polled periodically and each record from the result is converted to a Kafka Connect record. These records are then written to Kafka by the Kafka Connect framework.

The Source connector operates in two modes:

1. Bulk - Each table is selected in full each time it is polled.
2. Incremental - Each table is querying with lower and upper bounds to extract deltas.

In incremental mode the column used to identify new or delta rows has to be provided. Due to Cassandra's and CQL restrictions this should be a primary key or part of a composite primary keys. `ALLOW_FILTERING` can also be supplied as an configuration.

Note: TimeUUIDs are converted to strings. Use the [UUIDs](#) helpers to convert to Dates.

Only TimeUUID and Timestamp Cassandra data types are supported for tracking new rows in incremental mode. It is also possible to use TOKENS. When the connector is set with incremental mode as TOKEN, Cassandra's token functionality is used in the CQL statement that is generated.

The incremental mode is set in the via the `connect.cassandra.kcql` option. Allowed options are `TIMESTAMP`, `TIMEUUID` and `TOKEN`. For example:

```
INSERT INTO sink_test SELECT id, string_field FROM $TABLE5 PK id INCREMENTALMODE=TOKEN
```

Source Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Test data

Once you have installed and started Cassandra create a table to extract records from. This snippet creates a table called `orders` and inserts 3 rows representing fictional orders or some options and futures on a trading platform.

Start the Cassandra cql shell

```
bin ./cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.0.2 | CQL spec 3.3.1 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Execute the following:

```
CREATE KEYSPACE demo WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_
↪factor' : 3};
use demo;
```

```

create table orders (id int, created timeuuid, product text, qty int, price float,
↳PRIMARY KEY (id, created))
WITH CLUSTERING ORDER BY (created asc);

INSERT INTO orders (id, created, product, qty, price) VALUES (1, now(), 'OP-DAX-P-
↳20150201-95.7', 100, 94.2);
INSERT INTO orders (id, created, product, qty, price) VALUES (2, now(), 'OP-DAX-C-
↳20150201-100', 100, 99.5);
INSERT INTO orders (id, created, product, qty, price) VALUES (3, now(), 'FU-KOSPI-C-
↳20150201-100', 200, 150);

SELECT * FROM orders;

```

id	created	price	product	qty
1	17fa1050-137e-11e6-ab60-c9fbe0223a8f	94.2	OP-DAX-P-20150201-95.7	100
2	17fb6fe0-137e-11e6-ab60-c9fbe0223a8f	99.5	OP-DAX-C-20150201-100	100
3	17fbbe00-137e-11e6-ab60-c9fbe0223a8f	150	FU-KOSPI-C-20150201-100	200

```

(3 rows)

(3 rows)

```

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for Cassandra. If you are using the `dockers` you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```

bin/connect-cli create cassandra-source-orders < conf/cassandra-source-incr.
↳properties

#Connector `cassandra-source-orders`:
name=cassandra-source-orders
connector.class=com.datamountaineer.streamreactor.connect.cassandra.source.
↳CassandraSourceConnector
connect.cassandra.key.space=demo
connect.cassandra.kcql=INSERT INTO orders-topic SELECT * FROM orders PK created_
↳INCREMENTALMODE=TIMEUUID
connect.cassandra.contact.points=localhost
connect.cassandra.username=cassandra
connect.cassandra.password=cassandra
#task ids: 0

```

The `cassandra-source-incr.properties` file defines:

1. The name of the connector, must be unique.
2. The name of the connector class.
3. The keyspace (demo) we are connecting to.

4. The KCQL statement.
5. The ip or host name of the nodes in the Cassandra cluster to connect to.
6. Username and password, ignored unless you have set Cassandra to use the PasswordAuthenticator.

Use the Confluent CLI to view Connects logs.

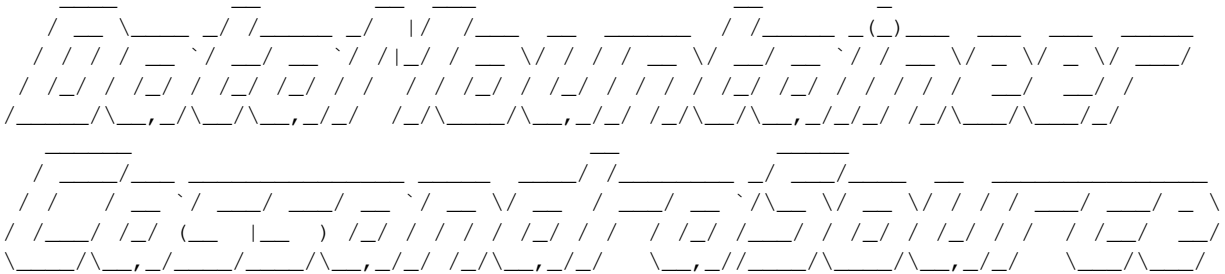
```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
cassandra-source
```

INFO



```
By Andrew Stevenson. (com.datamountaineer.streamreactor.connect.cassandra.source.
↳CassandraSourceTask:64)
[2016-05-06 13:34:41,193] INFO Attempting to connect to Cassandra cluster at
↳localhost and create keyspace demo. (com.datamountaineer.streamreactor.connect.
↳cassandra.CassandraConnection$:49)
[2016-05-06 13:34:41,263] INFO Using username_password. (com.datamountaineer.
↳streamreactor.connect.cassandra.CassandraConnection$:83)
[2016-05-06 13:34:41,459] INFO Did not find Netty's native epoll transport in the
↳classpath, defaulting to NIO. (com.datastax.driver.core.NettyUtil:83)
[2016-05-06 13:34:41,823] INFO Using data-center name 'datacenter1' for
↳DCAwareRoundRobinPolicy (if this is incorrect, please provide the correct
↳datacenter name with DCAwareRoundRobinPolicy constructor) (com.datastax.driver.core.
↳policies.DCAwareRoundRobinPolicy:95)
[2016-05-06 13:34:41,824] INFO New Cassandra host localhost/127.0.0.1:9042 added (com.
↳datastax.driver.core.Cluster:1475)
[2016-05-06 13:34:41,868] INFO Connection to Cassandra established. (com.
↳datamountaineer.streamreactor.connect.cassandra.source.CassandraSourceTask:87)
```

If you switch back to the terminal you started the Connector in you should see the Cassandra Source being accepted and the task starting and processing the 3 existing rows.

```
[2016-05-06 13:44:33,132] INFO Source task Thread[WorkerSourceTask-cassandra-source-
↳orders-0,5,main] finished initialization and start (org.apache.kafka.connect.
↳runtime.WorkerSourceTask:342)
[2016-05-06 13:44:33,137] INFO Query SELECT * FROM demo.orders WHERE created >
↳maxTimeuuid(?) AND created <= minTimeuuid(?) ALLOW FILTERING executing with
↳bindings (2016-05-06 09:23:28+0200, 2016-05-06 13:44:33+0200). (com.datamountaineer.
↳streamreactor.connect.cassandra.source.CassandraTableReader:156)
```

```
[2016-05-06 13:44:33,151] INFO Querying returning results for demo.orders. (com.
↳ datamountaineer.streamreactor.connect.cassandra.source.CassandraTableReader:185)
[2016-05-06 13:44:33,160] INFO Processed 3 rows for table orders-topic.orders (com.
↳ datamountaineer.streamreactor.connect.cassandra.source.CassandraTableReader:206)
[2016-05-06 13:44:33,160] INFO Found 3. Draining entries to batchSize 100. (com.
↳ datamountaineer.streamreactor.connect.queues.QueueHelpers$:45)
[2016-05-06 13:44:33,197] WARN Error while fetching metadata with correlation id 0 :
↳ {orders-topic=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient:582)
[2016-05-06 13:44:33,406] INFO Found 0. Draining entries to batchSize 100. (com.
↳ datamountaineer.streamreactor.connect.queues.QueueHelpers$:45)
```

Check Kafka, 3 rows as before.

```
$CONFLUENT_HOME/bin/kafka-avro-console-consumer \
--zookeeper localhost:2181 \
--topic orders-topic \
--from-beginning
{"id":{"int":1},"created":{"string":"Thu May 05 13:24:22 CEST 2016"},"price":{"float
↳ :94.2},"product":{"string":"DAX-P-20150201-95.7"},"qty":{"int":100}}
{"id":{"int":2},"created":{"string":"Thu May 05 13:26:21 CEST 2016"},"price":{"float
↳ :99.5},"product":{"string":"OP-DAX-C-20150201-100"},"qty":{"int":100}}
{"id":{"int":3},"created":{"string":"Thu May 05 13:26:44 CEST 2016"},"price":{"float
↳ :150.0},"product":{"string":"FU-KOSPI-C-20150201-100"},"qty":{"int":200}}
```

The Source tasks will continue to poll but not pick up any new rows yet.

Inserting new data

Now lets insert a row into the Cassandra table. Start the CQL shell and execute the following:

```
use demo;

INSERT INTO orders (id, created, product, qty, price) VALUES (4, now(), 'FU-
↳ DATAMOUNTAINEER-C-20150201-100', 500, 10000);

SELECT * FROM orders;

id | created | price | product
↳ | qty
-----+-----+-----+-----
↳ +-----
1 | 17fa1050-137e-11e6-ab60-c9fbe0223a8f | 94.2 | OP-DAX-P-20150201-95.
↳ 7 | 100
2 | 17fb6fe0-137e-11e6-ab60-c9fbe0223a8f | 99.5 | OP-DAX-C-20150201-
↳ 100 | 100
4 | 02acf5d0-1380-11e6-ab60-c9fbe0223a8f | 10000 | FU-DATAMOUNTAINEER-C-20150201-
↳ 100 | 500
3 | 17fbbe00-137e-11e6-ab60-c9fbe0223a8f | 150 | FU-KOSPI-C-20150201-
↳ 100 | 200

(4 rows)
cqlsh:demo>
```

Check the logs.


```
[2016-05-06 13:45:33,134] INFO Query SELECT * FROM demo.orders WHERE created >_
↳maxTimeuuid(?) AND created <= minTimeuuid(?) ALLOW FILTERING executing with_
↳bindings (2016-05-06 13:31:37+0200, 2016-05-06 13:45:33+0200). (com.datamountaineer.
↳streamreactor.connect.cassandra.source.CassandraTableReader:156)
[2016-05-06 13:45:33,137] INFO Querying returning results for demo.orders. (com.
↳datamountaineer.streamreactor.connect.cassandra.source.CassandraTableReader:185)
[2016-05-06 13:45:33,138] INFO Processed 1 rows for table orders-topic.orders (com.
↳datamountaineer.streamreactor.connect.cassandra.source.CassandraTableReader:206)
[2016-05-06 13:45:33,138] INFO Found 0. Draining entries to batchSize 100. (com.
↳datamountaineer.streamreactor.connect.queues.QueueHelpers$:45)
```

Check Kafka.

```
$CONFLUENT_HOME/bin/kafka-avro-console-consumer \
--zookeeper localhost:2181 \
--topic orders-topic \
--from-beginning

{"id":{"int":1},"created":{"string":"17fa1050-137e-11e6-ab60-c9fbe0223a8f"},"price":{"
↳float":94.2},"product":{"string":"OP-DAX-P-20150201-95.7"},"qty":{"int":100}}
{"id":{"int":2},"created":{"string":"17fb6fe0-137e-11e6-ab60-c9fbe0223a8f"},"price":{"
↳float":99.5},"product":{"string":"OP-DAX-C-20150201-100"},"qty":{"int":100}}
{"id":{"int":3},"created":{"string":"17fbbe00-137e-11e6-ab60-c9fbe0223a8f"},"price":{"
↳float":150.0},"product":{"string":"FU-KOSPI-C-20150201-100"},"qty":{"int":200}}
{"id":{"int":4},"created":{"string":"02acf5d0-1380-11e6-ab60-c9fbe0223a8f"},"price":{"
↳float":10000.0},"product":{"string":"FU-DATAMOUNTAINEER-C-20150201-100"},"qty":{"
↳int":500}}
```

Bingo, we have our extra row.

Features

Kafka Connect Query Language

Both connectors support **K**afka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

```
INSERT INTO <topic> SELECT * FROM <TABLE> PK <TRACKER_COLUMN>
↳<INCREMENTALMODE=TIMESTAMP|TIMEUUID|TOKEN>

#Select all columns from table orders and insert into a topic called orders-topic,
↳use column created to track new rows.
#Incremental mode set to TIMEUUID
INSERT INTO orders-topic SELECT * FROM orders PK created INCREMENTALMODE=TIMEUUID

#Select created, product, price from table orders and insert into a topic called
↳orders-topic, use column created to track new rows.
INSERT INTO orders-topic SELECT created, product, price FROM orders PK created.
```

The `PK` **key** word identifies the **column** used to track deltas in the target tables. **If**
↳the incremental **mode is set to** **TOKEN** this **column**
value **is** wrapped inside **Cassandras `token` function**.

Data Types

The Source connector supports copying tables in bulk and incrementally to Kafka.

The following CQL data types are supported:

CQL Type	Connect Data Type
TimeUUID	Optional String
UUID	Optional String
Inet	Optional String
Ascii	Optional String
Text	Optional String
Timestamp	Optional String
Date	Optional String
Tuple	Optional String
UDT	Optional String
Boolean	Optional Boolean
TinyInt	Optional Int8
SmallInt	Optional Int16
Int	Optional Int32
Decimal	Optional String
Float	Optional Float32
Counter	Optional Int64
BigInt	Optional Int64
VarInt	Optional Int64
Double	Optional Int64
Time	Optional Int64
Blob	Optional Bytes
Map	Optional String
List	Optional String
Set	Optional String

Note: For Map, List and Set the value is extracted from the Cassandra Row and inserted as a JSON string representation.

Modes

Incremental

In `incremental` mode the connector supports querying based on a column in the tables with CQL data type of `Timestamp` or `TimeUUID`.

Incremental mode is set by specifying `INCREMENTALMODE` in the `kcql` statement as either `TIMESTAMP`, `TIMEUUID` or `TOKEN`.

Kafka Connect tracks the latest record it retrieved from each table, so it can start at the correct location on the next iteration (or in case of a crash). In this case the maximum value of the records returned by the result-set is tracked and stored in Kafka by the framework. If no offset is found for the table at startup a default timestamp of 1900-01-01 is used. This is then passed to a prepared statement containing a range query.

Specifying TOKEN causes the connector to wrap the values in the *token* function. Only on PRIMARY KEY field of type token is supported. Your Cassandra cluster must use the Byte Ordered partitioner but this it is generally not recommended due to the creation of hotspots in the cluster. However, if Byte Ordered Partitioner is not used, the KC connector will miss all of the new rows whose token(PK column) falls “behind” the token recorded as the offset. This is because Cassandra’s other partitioners don’t order the tokens.

Warning: You must use the Byte Order Partitioner for the TOKEN mode to work correctly. Only one PRIMARY KEY field is supported for TOKENS.

For example:

```
#for timestamp type `timeuuid`
SELECT * FROM demo.orders WHERE created > maxTimeuuid(?) AND created <= minTimeuuid(?)

#for timestamp type as `timestamp`
SELECT * FROM demo.orders WHERE created > ? AND created <= ?

#for token
SELECT * FROM demo.orders WHERE created > token(?) and created <= token(?)
```

Bulk

In bulk mode the connector extracts the full table, no where clause is attached to the query. Bulk mode is set when no incremental mode is present in the KCQL statement.

Warning: Watch out with the poll interval. After each interval the bulk query will be executed again.

Topic Routing

The Sink supports topic routing that allows mapping the messages from topics to a specific table. For example map a topic called “bloomberg_prices” to a table called “prices”. This mapping is set in the `connect.cassandra.kcql` option.

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.cassandra.max.retries` and the `connect.cassandra.retry.interval`.

Configurations

`connect.cassandra.contact.points`

Contact points (hosts) in Cassandra cluster.

- Data type: string
- Optional : no

`connect.cassandra.key.space`

Key space the tables to write belong to.

- Data type: string
- Optional : no

`connect.cassandra.port`

Port for the native Java driver.

- Data type: int
- Optional : yes
- Default : 9042

`connect.cassandra.username`

Username to connect to Cassandra with.

- Data type: string
- Optional : yes

`connect.cassandra.password`

Password to connect to Cassandra with.

- Data type: string
- Optional : yes

`connect.cassandra.ssl.enabled`

Enables SSL communication against SSL enable Cassandra cluster.

- Data type: boolean
- Optional : yes

- Default : false

```
connect.cassandra.trust.store.password
```

Password for truststore.

- Data type: string
- Optional : yes

```
connect.cassandra.key.store.path
```

Path to truststore.

- Data type: string
- Optional : yes

```
connect.cassandra.key.store.password
```

Password for key store.

- Data type: string
- Optional : yes

```
connect.cassandra.ssl.client.cert.auth
```

Path to keystore.

- Data type: string
- Optional : yes

```
connect.cassandra.import.poll.interval
```

The polling interval between queries against tables in milliseconds. Default is 1 minute.

- Data type: int
- Optional : yes
- Default : 60000

Warning: WATCH OUT WITH BULK MODE AS MAY REPEATEDLY PULL IN THE SAME DATE.

```
connect.cassandra.import.mode
```

Either bulk or incremental.

- Data type : string
- Optional : no

```
connect.cassandra.kcql
```

Kafka connect query language expression. Allows for expressive table to topic routing, field selection and renaming. In incremental mode the timestampColumn can be specified by PK colName.

Examples:

```
INSERT INTO TOPIC1 SELECT * FROM TOPIC1 PK myTimeUUICol
```

- Data type : string
- Optional : no

Warning: The timestamp column must be of CQL Type TimeUUID.

`connect.cassandra.task.buffer.size`

The size of the queue for buffering resultset records before write to Kafka.

- Data type : int
- Optional : yes
- Default : 10000

`connect.cassandra.task.batch.size`

The number of records the Source task should drain from the reader queue.

- Data type : int
- Optional : yes
- Default : 1000

`connect.cassandra.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.cassandra.max.retries` option. The `connect.cassandra.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Default: throw

`connect.cassandra.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.cassandra.error.policy` is set to **retry**.

- Type: string
- Importance: high
- Default: 10

`connect.cassandra.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.cassandra.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Default : 60000 (1 minute)

`connect.cassandra.fetch.size`

The max number of rows the Cassandra driver will fetch at one time.

- Type: int
- Importance: medium

- Default : 5000

`connect.cassandra.slice.duration`

- Type: long
- Importance: medium
- Default : 10000

Duration in milliseconds to query for in target Cassandra table. Used to restrict query timestamp span.

`connect.cassandra.initial.offset`

- Type: string
- Importance: medium
- Default : 1900-01-01 00:00:00.0000000Z

The initial timestamp to start querying in Cassandra from (yyyy-MM-dd HH:mm:ss.SSS'Z')."

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Bulk Example

```
name=cassandra-source-orders-bulk
connector.class=com.datamountaineer.streamreactor.connect.cassandra.source.
↪CassandraSourceConnector
connect.cassandra.key.space=demo
connect.cassandra.kcql=INSERT INTO TABLE_X SELECT * FROM TOPIC_Y
connect.cassandra.contact.points=localhost
connect.cassandra.username=cassandra
connect.cassandra.password=cassandra
```

Incremental Example

```
name=cassandra-source-orders-incremental
connector.class=com.datamountaineer.streamreactor.connect.cassandra.source.
↪CassandraSourceConnector
connect.cassandra.key.space=demo
connect.cassandra.kcql=INSERT INTO TABLE_X SELECT * FROM TOPIC_Y PK created_
↪INCREMENTALMODE=TIMEUUID
connect.cassandra.contact.points=localhost
connect.cassandra.username=cassandra
connect.cassandra.password=cassandra
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

For the Source connector, at present no column selection is handled, every column from the table is queried to column additions and deletions are handled in accordance with the compatibility mode of the Schema Registry.

Future releases will support auto creation of tables and adding columns on changes to the topic schema.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

Troubleshooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Cassandra CDC

Note: This is still beta version

Kafka Connect Cassandra is a Source Connector for reading Change Data Capture from Cassandra and write the mutations to Kafka.

Why

We already provide a Kafka Connect Cassandra Source and people might ask us why another source. The main reason is performance. We have noticed our users with large data in a Cassandra table (column family) the performance drops. And this is not on the connector but rather how long it takes to pull the records. We have worked around the issue by introducing a limit on the records queried on Cassandra but even so we are not that much better.

With Apache Cassandra 3.0 the support for CDC(Change Data Capture) has been added and this source implementation is making use of that to capture the changes made to your column families.

Cassandra CDC

Change data capture is designed to capture insert, update, and delete activity applied to tables(column families), and to make the details of the changes available in an easily consumed format.

Cassandra CDC logging is configured per table, with limits on the amount of disk space to consume for storing the CDC logs. CDC logs use the same binary format as the commit log. Cassandra tables can be created or altered with a table property to use CDC logging.

```
CREATE TABLE foo (a int, b text, PRIMARY KEY(a)) WITH cdc=true;

ALTER TABLE foo WITH cdc=true;

ALTER TABLE foo WITH cdc=false;
```

CDC logging must be enabled in the cassandra.yaml file to begin logging. You should make sure your yaml file has the following:

```
cdc_enabled: true
```

You can enable the CDC logging on per node basis.

The Kafka Connect Source consumes the CDC log information and pushes it to Kafka before it deletes it.

Warning: Upon flushing the memtable to disk, CommitLogSegments containing data for CDC-enabled tables are moved to the configured cdc_raw directory. Once the disk space limit is reached, writes to CDC enabled tables will be rejected until space is freed.

Four CDC settings are configured in the cassandra.yaml

cdc_enabled

Enables/Disables CDC logging per node

cdc_raw_directory

The directory where the CDC log is stored.

- Package installations(default): \$CASSANDRA_HOME/cdc_raw.

- Tarball installations: `install_location/data/cdc_raw`.

`cdc_total_space_in_mb`

Total space available for storing CDC data. The default is 4096MB and 1/8th of the total space of the drive where the `cdc_raw_directory` resides. If space gets above this value, Cassandra will throw **WriteTimeoutException** on Mutations including tables with CDC enabled.

`cdc_free_space_check_interval_ms`

When the `cdc_raw` limit is hit and the Consumer is either running behind or experiencing back pressure, this interval is checked to see if any new space for cdc-tracked tables has been made available.

Warning: After changing properties in the `cassandra.yaml` file, you must restart the node for the changes to take effect.

Prerequisites

- Cassandra **3.0.9+**
- Confluent **3.2+**
- Java **1.8**
- Scala **2.11**

Setup

Before we can do anything, including the QuickStart we need to install Cassandra and the Confluent platform.

Cassandra Setup

First download and install Cassandra if you don't have a compatible cluster available.

```
#make a folder for cassandra
mkdir cassandra

#Download Cassandra
wget http://apache.mirror.anlx.net/cassandra/3.11.0/apache-cassandra-3.11.0-bin.tar.gz

#extract archive to cassandra folder
tar xvf apache-cassandra-3.11.0-bin.tar.gz -C cassandra --strip-components=1

#enable the CDC in the yaml configuration
sed -i -- 's/cdc_enabled: false/cdc_enabled: true/g' conf/cassandra.yaml

#set CASSANDRA_HOME
export CASSANDRA_HOME=$(pwd)/cassandra

#Start Cassandra
cd cassandra
sudo sh /bin/cassandra
```

Note: There can be only one instance of Apache Cassandra node per machine for the connector to run properly. All nodes should have the same path for the `cdc_raw` folder

Confluent Setup

Follow the instructions [here](#).

Source Connector

The Cassandra CDC Source connector will read the commit log mutations from the CDC logs and will push them to the target topic.

Note: messages sent to Kafka are in AVRO format.

The record pushed to Kafka populates both the key and the value part. Key will contain the metadata of the change while the value will contain the actual data change.

Record Key

The key data structure follows this layout:

```
{
  "keyspace": //Cassandra Keyspace name
  "table"    : //The Cassandra Column Family name
  "changeType": //The type of change in Cassandra
  "keys": {
    "key1":
    "key2":
    ..
  }
  "timestamp" : //the timestamp of when the change was made in Cassandra
  "deleted_columns": //which columns have been deleted. We will expand on the_
↪ details
}
```

Based on the mutation information we can identify the following types of changes:

INSERT

A record has been inserted/a record columns have been updated. There is no real solution for identifying an UPDATE unless the connector keeps track of all the keys seen.

```
INSERT INTO keyspace.orders (id, created, product, qty, price) VALUES (1, now(), 'OP-
↪DAX-P-20150201-95.7', 100, 94.2)
```

DELETE

An entire record has been deleted (tombstoned)

```
DELETE FROM datamountaineer.orders where id = 1
```

DELETE_COLUMN

Specific columns have been deleted (non PK columns).

```
DELETE product FROM datamountaineer.orders where id = 1
DELETE name.firstname FROM datamountaineer.users WHERE id=62c36092-82a1-3a00-93d1-
↪46196ee77204;
```

In this case the `deleted_columns` entry will contain “product/name.firstname”. If more than one column is deleted we will retain that information.

Value Key

The Kafka message value part contains the actual mutation data. Apart from the primary keys columns all the other columns have an optional schema in avro. The reason for that is because one can set the values on a subset of them during a CQL insert/update. In the QuickStart section we make use of the `users` table. The value AVRO schema associated with it looks like this

```
{
  "type" : "record",
  "name" : "users",
  "fields" : [ {
    "name" : "name",
    "type" : [ "null", {
      "type" : "record",
      "name" : "fullname",
      "fields" : [ {
        "name" : "firstname",
        "type" : [ "null", "string" ],
        "default" : null
      }, {
        "name" : "lastname",
        "type" : [ "null", "string" ],
        "default" : null
      } ],
      "connect.name" : "fullname"
    } ],
    "default" : null
  }, {
    "name" : "addresses",
    "type" : [ "null", {
      "type" : "array",
      "items" : {
        "type" : "record",
        "name" : "MapEntry",
        "namespace" : "io.confluent.connect.avro",
        "fields" : [ {
          "name" : "key",
          "type" : [ "null", "string" ],
          "default" : null
        }, {
          "name" : "value",
          "type" : [ "null", {
            "type" : "record",
            "name" : "address",
            "namespace" : "",
            "fields" : [ {
```

```

        "name" : "street",
        "type" : [ "null", "string" ],
        "default" : null
    }, {
        "name" : "city",
        "type" : [ "null", "string" ],
        "default" : null
    }, {
        "name" : "zip_code",
        "type" : [ "null", "int" ],
        "default" : null
    }, {
        "name" : "phones",
        "type" : [ "null", {
            "type" : "array",
            "items" : [ "null", "string" ]
        } ],
        "default" : null
    } ],
    "connect.name" : "address"
} ],
"default" : null
} ]
}
} ],
"default" : null
}, {
    "name" : "direct_reports",
    "type" : [ "null", {
        "type" : "array",
        "items" : [ "null", "fullname" ]
    } ],
    "default" : null
}, {
    "name" : "id",
    "type" : [ "null", "string" ],
    "default" : null
}, {
    "name" : "other_reports",
    "type" : [ "null", {
        "type" : "array",
        "items" : [ "null", "fullname" ]
    } ],
    "default" : null
} ],
"connect.name" : "users"
}

```

And a json representation of the actual value looks like this:

```

{
  "id" : "UUID-String",
  "name": {
    "firstname":"String"
    "lastname":"String"
  },
  "direct_reports":[
    {

```

```
        "firstname": "String"
        "lastname": "String"
    },
    ...
],
"other_reports": [
    {
        "firstname": "String"
        "lastname": "String"
    },
    ...
],
"addresses" : {
    "home" : {
        "street": "String",
        "city": "String",
        "zip_code": "int",
        "phones": [
            "+33 ...",
            "+33 ..."
        ]
    },
    "work" : {
        "street": "String",
        "city": "String",
        "zip_code": "int",
        "phones": [
            "+33 ...",
            "+33 ..."
        ]
    }
},
...
}
```

Data Types

The Source connector needs to map Apache Cassandra types to Kafka Connect Schema types. For the ones not so familiar with connect here is the list of supported Connect Types:

- INT8,
- INT16
- INT32
- INT64
- FLOAT32
- FLOAT64
- BOOLEAN
- STRING
- BYTES
- ARRAY

- MAP
- STRUCT

Along these primitive types there are the logical types for :

- Date
- Decimal
- Time
- Timestamp

As a result for most Apache Cassandra Types we have an equivalent type for Connect Schema. A Connect Source Record will be marshaled as AVRO when sent to Kafka.

CQL Type	Connect Data Type
AsciiType	OPTIONAL STRING
LongType	OPTIONAL INT64
BytesType	OPTIONAL BYTES
BooleanType	OPTIONAL BOOLEAN
CounterColumnType	OPTIONAL INT64
SimpleDateType	OPTIONAL Kafka Connect Date
DoubleType	OPTIONAL FLOAT64
DecimalType	OPTIONAL Kafka Connect Decimal
DurationType	OPTIONAL STRING
EmptyType	OPTIONAL STRING
FloatType	OPTIONAL FLOAT32
InetAddressType	OPTIONAL STRING
Int32Type	OPTIONAL INT32
ShortType	OPTIONAL INT16
UTF8Type	OPTIONAL STRING
TimeType	OPTIONAL KAFKA CONNECT Time
TimestampType	OPTIONAL KAFKA CONNECT Timestamp
TimeUUIDType	OPTIONAL STRING
ByteType	OPTIONAL INT8
UUIDType	OPTIONAL STRING
IntegerType	OPTIONAL INT32
ListType	OPTIONAL ARRAY of the inner type
MapType	OPTIONAL MAP of the inner types
SetType	OPTIONAL ARRAY of the inner type
UserType	OPTIONAL STRUCT for the user type

Please note we default to String for the these CQL types: DurationType, InetAddressType, TimeUUIDType, UUID-Type.

How does it work

It is expected that Kafka Connect worker will run on the same node as the Apache Cassandra node.

Important: Only one Apache Cassandra Node should run per machine to have the Connector work properly The **cdc_raw** folder location should be the same on all nodes running Apache Cassandra Node There should be only one Connector Worker instance per machine. Any more won't have any effect

Cassandra supports a master-less “ring” architecture. Each of the node in the Cassandra ring cluster will be responsible for storing the table records. The partition key hash and number of rings in the cluster determines the node where each record is stored. (we leave aside replication from this discussion).

Upon flushing the memtable to disk, all the commit log segments containing data for CDC-enabled tables are moved to the configured `cdc_raw` directory. It is only at this point the connector will pick up the changes.

Important: Changes in Cassandra are not picked up immediately. The memtables need to be flushed for the CDC commit logs to be available. You can use `nodetool` to flush the tables

Once a file lands in the CDC folder the Connector will pick it up and read the mutations. A CDC file can contain mutations for more than one table. Each mutation for the subscribed tables will be translated into a Kafka Connect Source Record which will be sent by the Connect framework to the topic configured in the connector properties.

The Connect source will process the files in the order they were created and one by one. This ensures the change sequence is retained. Once the records have been pushed to Kafka the CDC file is deleted.

The connector will only be able to read the mutations for the subscribed tables. Via configuration you can express which tables to consider and what topic should receive those mutations information.

```
INSERT INTO ordersTopic SELECT * FROM datamountaineer.orders
```

Important: Enabling CDC on a new table means you need to restart the connector for the changes to be picked up. The connector is driven by the configurations and not by the list of all the tables with CDC enabled. (Might be a feature, change to do)

Below you can find a flow diagram describing the process mentioned above.

Source Connector QuickStart

We will start the connector in distributed mode. Each connector exposes a rest endpoint for stopping, starting and updating the configuration. We have developed a Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Once you have installed and started Cassandra create a table to capture the mutations. We will use a bit more complex column family structure to show case what we support so far.

Let's start the `cql` shell tool to create our table

```
./bin/cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.0 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Now let's create a keyspace and a users column family(table)

```
CREATE KEYSPACE datamountaineer WITH REPLICATION = {'class' : 'SimpleStrategy',
↪ 'replication_factor' : 3};

CREATE TYPE datamountaineer.address(
  street text,
  city text,
```



```

    zip_code int,
    phones set<text>
);

CREATE TYPE datamountaineer.fullname (
    firstname text,
    lastname text
);

CREATE TABLE datamountaineer.users(
    id uuid PRIMARY KEY,
    name fullname,
    direct_reports set<frozen <fullname>>,
    other_reports list<frozen <fullname>>,
    addresses map<text, frozen <address>>);
ALTER TABLE datamountaineer.users WITH cdc=true;

```

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Start Kafka Connect in distributed mode by running:

```

#make sure you have $CASSANDRA_HOME variable setup (see Cassandra setup)
$CONFLUENT_HOME/bin/connect-distributed.sh $CONFLUENT_HOME/etc/schema-registry/
↪connect-avro-distributed.properties

```

```

#make sure you have $CASSANDRA_HOME variable setup (see Cassandra setup)
$ cat <<EOF > cassandra-cdc-source.json
{
  "name": "cassandra-connect-cdc",
  "config":{
    "name":"cassandra-connect-cdc",
    "tasks": 1,
    "connector.class":"com.datamountaineer.streamreactor.connect.cassandra.cdc.
↪CassandraCdcSourceConnector",
    "connect.cassandra.kcql":"INSERT INTO users-topic SELECT * FROM datamountaineer.
↪users",
    "connect.cassandra.yaml.path.url": "$CASSANDRA_HOME/conf/cassandra.yaml",
    "connect.cassandra.port": "9042",
    "connect.cassandra.contact.points":"localhost"
  }
}
EOF

```

If you visualize the file it should print something like (I used the .. because you might have a different path to where you stored Cassandra)

```

{
  "name": "cassandra-connect-cdc",
  "config":{
    "name":"cassandra-connect-cdc",
    "tasks": 1,
    "connector.class":"com.datamountaineer.streamreactor.connect.cassandra.cdc.
↪CassandraCdcSourceConnector",

```

```

    "connect.cassandra.kcql": "INSERT INTO orders-topic SELECT * FROM datamountaineer.
→orders",
    "connect.cassandra.yaml.path.url": "../cassandra/conf/cassandra.yaml",
    "connect.cassandra.port": "9042",
    "connect.cassandra.contact.points": "localhost"
  }
}

```

Next step is to spin up the connector. And for that we run the following bash script:

```
curl -X POST -H "Content-Type: application/json" --data @cassandra-cdc-source.json   
↪ http://localhost:8083/connectors
```

The output from the connect-distributed should read something similar to this:

```
[2017-08-01 22:34:35,653] INFO Acquiring port 64101 to enforce single instance being_
→run on Stepi (com.datamountaineer.streamreactor.connect.cassandra.cdc.
→CassandraCdcSourceTask:53)
[2017-08-01 22:34:35,660] INFO

      _ _ _ _ _ 
     /   \   _ _ _ _ _ 
    /_____\ /_____ \ 
   /         \       \ 
  /           \       \ 
 /             \       \ 
/               \       \ 
\               /       / 
 \             /       / 
  \           /       / 
   \         /       / 
    \_____/   _ _ _ _ _ 
     \___/   /_____ \ 
      _ _ _ _ _ 
     /   \   _ _ _ _ _ 
    /_____\ /_____ \ 
   /         \       \ 
  /           \       \ 
 /             \       \ 
/               \       \ 
\               /       / 
 \             /       / 
  \           /       / 
   \         /       / 
    \_____/   _ _ _ _ _ 

by Stefan Bocutiu

      _ _ _ _ _ 
     /   \   _ _ _ _ _ 
    /_____\ /_____ \ 
   /         \       \ 
  /           \       \ 
 /             \       \ 
/               \       \ 
\               /       / 
 \             /       / 
  \           /       / 
   \         /       / 
    \_____/   _ _ _ _ _ 

(com.datamountaineer.streamreactor.connect.cassandra.cdc.CassandraCdcSourceTask:65)
[2017-08-01 22:34:36,088] INFO CDC path is not set in Yaml. Using the default_
→location (com.datamountaineer.streamreactor.connect.cassandra.cdc.logs.
→CdcCassandra:55)
[2017-08-01 22:34:36,411] INFO Detected Guava >= 19 in the classpath, using modern_
→compatibility layer (com.datastax.driver.core.GuavaCompatibility:132)
```

Let's go back to the `cqlsh` terminal and insert some records into the `users` table and then perform some updates and deletes.

```
INSERT INTO datamountaineer.users(id, name) VALUES (62c36092-82a1-3a00-93d1-
↳ 46196ee77204, {firstname: 'Marie-Claude', lastname: 'Josset'});

UPDATE datamountaineer.users
SET
addresses = addresses + {
'home': {
street: '191 Rue St. Charles',
city: 'Paris',
zip_code: 75015,
phones: {'33 6 78 90 12 34'}
},
'work': {
street: '81 Rue de Paradis',
city: 'Paris',
zip_code: 7500,
phones: {'33 7 12 99 11 00'}
}
}
```

```

}
WHERE id=62c36092-82a1-3a00-93d1-46196ee77204;

INSERT INTO datamountaineer.users(id, direct_reports) VALUES (11c11111-82a1-3a00-93d1-
↳46196ee77204,{{firstname:'Jean-Claude',lastname:'Van Damme'}, {firstname:'Arnold',
↳lastname:'Schwarzenegger'}}});

INSERT INTO datamountaineer.users(id, other_reports) VALUES (22c11111-82a1-3a00-93d1-
↳46196ee77204, [{firstname:'Jean-Claude',lastname:'Van Damme'}, {firstname:'Arnold',
↳lastname:'Schwarzenegger'}]);

DELETE name.firstname FROM datamountaineer.users WHERE id=62c36092-82a1-3a00-93d1-
↳46196ee77204;

```

You will notice from the logs there are no new CDC files picked up and if you navigate to the CDC output folder you will see it is empty. The memtables needs to fill up to be flushed to disk. Let's use the tool provided by Apache Cassandra to flush the table: `nodetool`

```
$ $CASSANDRA_HOME/bin/nodetool drain
```

Once this completes your connect distributed log should print something along these lines:

```

[2017-08-01 23:10:42,842] INFO Reading mutations from the CDC file:/home/stepi/work/
↳programs/cassandra/data/cdc_raw/CommitLog-6-1501625205002.log. Checking file is
↳still being written... (com.datamountaineer.streamreactor.connect.cassandra.cdc.
↳logs.CdcCassandra:158)
[2017-08-01 23:10:43,352] INFO Global buffer pool is enabled, when pool is exhausted
↳(max is 0.000KiB) it will allocate on heap (org.apache.cassandra.utils.memory.
↳BufferPool:230)
[2017-08-01 23:10:43,355] INFO Maximum memory usage reached (0.000KiB), cannot
↳allocate chunk of 1.000MiB (org.apache.cassandra.utils.memory.BufferPool:91)
[2017-08-01 23:10:43,390] ERROR [Control connection] Cannot connect to any host,
↳scheduling retry in 4000 milliseconds (com.datastax.driver.core.
↳ControlConnection:153)
[2017-08-01 23:10:43,569] INFO 5 changes detected in /home/stepi/work/programs/
↳cassandra/data/cdc_raw/CommitLog-6-1501625205002.log (com.datamountaineer.
↳streamreactor.connect.cassandra.cdc.logs.CdcCassandra:173)

```

Let's see what was sent over to the users topic. We will run `kafka-avro-console-consumer` to read the records

```

$ ./bin/kafka-avro-console-consumer --zookeeper localhost:2181 --topic users-
↳topic --from-beginning --property print.key=true
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/stepi/work/programs/confluent-3.2.2/share/
↳java/kafka-serde-tools/slf4j-log4j12-1.7.6.jar!/org/slf4j/impl/StaticLoggerBinder.
↳class]
SLF4J: Found binding in [jar:file:/home/stepi/work/programs/confluent-3.2.2/share/
↳java/schema-registry/slf4j-log4j12-1.7.6.jar!/org/slf4j/impl/StaticLoggerBinder.
↳class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Using the ConsoleConsumer with old consumer is deprecated and will be removed in a
↳future major release. Consider using the new consumer by passing [bootstrap-server]
↳instead of [zookeeper].
{"keyspace":"datamountaineer","table":"users","changeType":"INSERT","deleted_columns
↳":null,"keys":{"id":{"string":"62c36092-82a1-3a00-93d1-46196ee77204"},"timestamp
↳":1501625394958965} {"name":{"fullname":{"firstname":{"string":"Marie-Claude"},
↳lastname":{"string":"Josset"}}},"addresses":null,"direct_reports":null,"id":{"
↳string":"62c36092-82a1-3a00-93d1-46196ee77204"},"other_reports":null}}

```

```
{
  "keyspace": "datamountaineer", "table": "users", "changeType": "INSERT", "deleted_columns": null, "keys": {
    "id": { "string": "62c36092-82a1-3a00-93d1-46196ee77204" }, "timestamp": 1501625395008930
  }, {
    "name": null, "addresses": {
      "array": [ { "key": { "string": "work" }, "value": { "address": { "street": { "string": "81 Rue de Paradis" }, "city": { "string": "Paris" }, "zip_code": { "int": 7500 }, "phones": { "array": [ { "string": "33 7 12 99 11 00" } ] } } } } ], {
      "key": { "string": "home" }, "value": { "address": { "street": { "string": "191 Rue St. Charles" }, "city": { "string": "Paris" }, "zip_code": { "int": 75015 }, "phones": { "array": [ { "string": "33 6 78 90 12 34" } ] } } } } ] } }, {
    "direct_reports": null, "id": { "string": "62c36092-82a1-3a00-93d1-46196ee77204" }, "other_reports": null
  }
}

{
  "keyspace": "datamountaineer", "table": "users", "changeType": "INSERT", "deleted_columns": null, "keys": {
    "id": { "string": "11c11111-82a1-3a00-93d1-46196ee77204" }, "timestamp": 1501625395013654
  }, {
    "name": null, "addresses": null, "direct_reports": { "array": [ { "fullname": { "firstname": { "string": "Arnold" }, "lastname": { "string": "Schwarzenegger" } } } ] }, {
    "fullname": { "firstname": { "string": "Jean-Claude" }, "lastname": { "string": "Van Damme" } } } ], {
    "id": { "string": "11c11111-82a1-3a00-93d1-46196ee77204" }, "other_reports": null
  }
}

{
  "keyspace": "datamountaineer", "table": "users", "changeType": "INSERT", "deleted_columns": null, "keys": {
    "id": { "string": "22c11111-82a1-3a00-93d1-46196ee77204" }, "timestamp": 1501625395015668
  }, {
    "name": null, "addresses": null, "direct_reports": null, "id": { "string": "22c11111-82a1-3a00-93d1-46196ee77204" }, "other_reports": { "array": [ { "fullname": { "firstname": { "string": "Jean-Claude" }, "lastname": { "string": "Van Damme" } } } ] }, {
    "fullname": { "firstname": { "string": "Arnold" }, "lastname": { "string": "Schwarzenegger" } } } ] } }

{
  "keyspace": "datamountaineer", "table": "users", "changeType": "DELETE_COLUMN", "deleted_columns": { "array": [ "name.firstname" ] }, "keys": {
    "id": { "string": "62c36092-82a1-3a00-93d1-46196ee77204" }, "timestamp": 1501625395018481
  }, {
    "name": null, "addresses": null, "direct_reports": null, "id": { "string": "62c36092-82a1-3a00-93d1-46196ee77204" }, "other_reports": null
  }
}
```

Exactly what was changed!!!

Features

Kafka Connect Query Language

Both connectors support **K**afka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

```
INSERT INTO <topic> SELECT * FROM <KEYSPACE>.<TABLE>

#Select all mutations for datamountaineer.orders
INSERT INTO ordersTopic SELECT * FROM datamountaineer.orders

#while KSQL allows for fields (column) addressing the CDC Source will not use them.
→ It pushes the entire Cassandra mutation information on the topic
```

Configurations

Here is a full list of configuration entries the connector knows about.

Name	Description	Data Type	Optional	Default
connect.cassandra.contact.points	Contact points (hosts) in	string	no	
connect.cassandra.port	Cassandra Node Client connection port	int	yes	9042
connect.cassandra.username	Username to connect to Cassandra with if connect.cassandra.authentication.mode is set to <i>username_password</i>	string	yes	
connect.cassandra.password	Password to connect to Cassandra with if connect.cassandra.authentication.mode is set to <i>username_password</i> .	string	yes	
connect.cassandra.ssl.enabled	Enables SSL communication against SSL enabled Cassandra cluster.	boolean	yes	false
connect.cassandra.trust.store.password	Password for truststore.	string	yes	
connect.cassandra.key.store.path	Path to truststore.	string	yes	
connect.cassandra.key.store.password	Password for key store.	string	yes	
connect.cassandra.ssl.client.cert.auth	Path to keystore.	string	yes	
connect.cassandra.query	Kafka connect query language expression. Allows for expressive table to topic mapping. It describes which CDC tables are monitored and the target Kafka topic for Cassandra CDC information.	string	no	
connect.cassandra.cdc.path	The location of the Cassandra Yaml file in URL format:file://path. The connector reads the file to get the cdc folder but also sets the internals of the Cassandra API allowing it to read the CDC files	string	no	
connect.cassandra.cdc.poll.interval	The delay time in milliseconds before the connector checks for Cassandra CDC files. We poll the CDC folder for new files.	long	yes	2000
connect.cassandra.cdc.filesize	The maximum number of Cassandra mutation to buffer. As it reads from the Cassandra CDC files the mutations are buffered before they are handed over to Kafka Connect.	int	yes	1000000
connect.cassandra.cdc.delete.on.read	The worker CDC thread will read a CDC file checking if any of the processed files are ready to be deleted (it means the records have been sent to kafka). Rather than waiting for a read to complete we can delete the files while reading a CDC file. a CDC file. You can disable it for faster faster reads by setting the value to false.	boolean	yes	false
connect.cassandra.cdc.single.worker.port	Kafka Connect framework doesn't allow yet configuration where you are running only one task per worker. If you allocate more tasks than workers then some will spin up more tasks. With Cassandra nodes we want one worker and one task - not more. To ensure this we allow the first task to grab a port subsequent calls to open the port will fail thus not allowing multiple instance running at once	int	yes	64101
connect.cassandra.cdc.decimal.scale	When reading the column family metadata we don't have details about decimal scale			

Deployment Guidelines

TODO

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect CoAP Source

A Connector and Source to stream messages from a CoAP server and write them to a Kafka topic.

The Source supports:

1. DTLS secure clients.
2. Observable resources.

The Source Connector automatically converts the CoAP response into a Kafka Connect `Struct` to be store in Kafka as Avro or Json dependent on the Converters used in Connect. The schema can found [here](#).

The key of the `Struct` message sent to Kafka is made from the source defined in the message, the resource on the CoAP server and the message id.

Prerequisites

- Confluent 3.3
- Java 1.8
- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

CoAP Setup

The connector uses [Californium](#) Java API under the hood. A resource is host at `coap://californium.eclipse.org:5683`. Copper, a [FireFox](#) browser addon is available so you can browse the server and resources.. This is an observable non confirmable resource. You can view messages via the browser addon by selecting the resource and clicking the `observe` button on the top menu.

Source Connector QuickStart

We you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the kafka-connect-tools [cli](#) to post in our distributed properties file for MQTT. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create coap-source < conf/coap-source.properties

#Connector name=`coap-source`
name = coap-source
tasks = 1
connector.class = com.datamountaineer.streamreactor.connect.coap.source.
↪CoapSourceConnector
connect.coap.uri = coap://californium.eclipse.org:5683
connect.coap.kcql = INSERT INTO coap-topic SELECT * FROM obs-pumping-non
#task ids: 0
```

The coap-source.properties file defines:

1. The name of the source.
2. The name number of tasks.
3. The class containing the connector.
4. The uri of the CoAP Server and port to connect to.
5. *The KCQL routing querying..* This specifies the target topic and source resource on the CoAP server.

Use the Confluent CLI to view Connects logs.

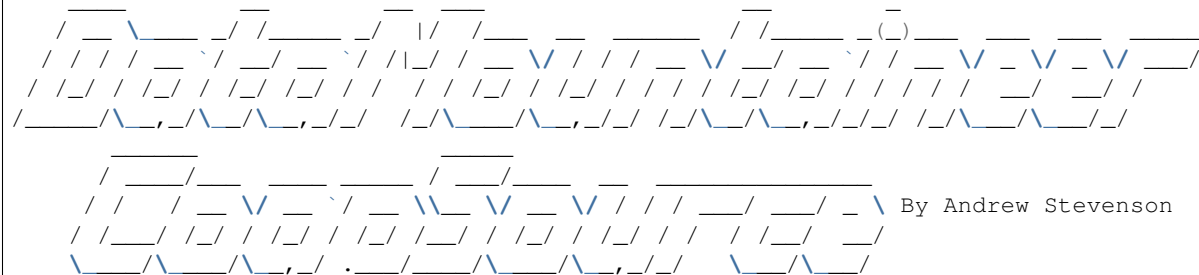
```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
coap-source
```

INFO



By Andrew Stevenson

```

    // (com.datamountaineer.streamreactor.connect.coap.source.
↪CoapSourceTask:54)
[2017-01-09 20:42:44,830] INFO CoapConfig values:
    connect.coap.uri = coap://californium.eclipse.org:5683
    connect.coap.port = 0
    connect.coap.truststore.pass = [hidden]
    connect.coap.cert.chain.key = client
    connect.coap.keystore.path =
    connect.coap.kcql = INSERT INTO coap-topic SELECT * FROM obs-pumping-non
    connect.coap.truststore.path =
    connect.coap.certs = []
    connect.coap.keystore.pass = [hidden]
    connect.coap.host = localhost
    (com.datamountaineer.streamreactor.connect.coap.configs.CoapConfig:178)
[2017-01-09 20:42:44,831] INFO Source task WorkerSourceTask{id=coap-source-0}
↪finished initialization and start (org.apache.kafka.connect.runtime.
↪WorkerSourceTask:138)
[2017-01-09 20:42:45,927] INFO Discovered resources /.well-known/core (com.
↪datamountaineer.streamreactor.connect.coap.source.CoapReader:60)
[2017-01-09 20:42:45,927] INFO Discovered resources /large (com.datamountaineer.
↪streamreactor.connect.coap.source.CoapReader:60)
[2017-01-09 20:42:45,928] INFO Discovered resources /large-create (com.
↪datamountaineer.streamreactor.connect.coap.source.CoapReader:60)
[2017-01-09 20:42:45,928] INFO Discovered resources /large-post (com.datamountaineer.
↪streamreactor.connect.coap.source.CoapReader:60)
[2017-01-09 20:42:45,928] INFO Discovered resources /large-separate (com.
↪datamountaineer.streamreactor.connect.coap.source.CoapReader:60)
[2017-01-09 20:42:45,928] INFO Discovered resources /large-update (com.
↪datamountaineer.streamreactor.connect.coap.source.CoapReader:60)

```

Check for records in Kafka

Check for records in Kafka with the console consumer.

```

bin/kafka-avro-console-consumer \
  --zookeeper localhost:2181 \
  --topic coap-topic \
  --from-beginning

```

```

{"message_id":{"int":4803},"type":{"string":"ACK"},"code":"4.04","raw_code":{"int
↪":132},"rtt":{"long":35},"is_last":{"boolean":true},"is_notification":{"boolean
↪":false},"source":{"string":"idvm-infk-mattern04.inf.ethz.ch:5683"},"destination":{"
↪string":"","timestamp":{"long":0},"token":{"string":"b24774e37c2314a4"},"is_
↪duplicate":{"boolean":false},"is_confirmable":{"boolean":false},"is_rejected":{"
↪boolean":false},"is_acknowledged":{"boolean":false},"is_canceled":{"boolean":false}
↪},"accept":{"int":-1},"block1":{"string":"","block2":{"string":"","content_format":
↪{"int":-1},"etags":[],"location_path":{"string":"","location_query":{"string":"","
↪max_age":{"long":60},"observe":null,"proxy_uri":null,"size_1":null,"size_2":null,
↪uri_host":null,"uri_port":null,"uri_path":{"string":"","uri_query":{"string":"","
↪payload":{"string":""}}
{"message_id":{"int":4804},"type":{"string":"ACK"},"code":"4.04","raw_code":{"int
↪":132},"rtt":{"long":34},"is_last":{"boolean":true},"is_notification":{"boolean
↪":false},"source":{"string":"idvm-infk-mattern04.inf.ethz.ch:5683"},"destination":{"
↪string":"","timestamp":{"long":0},"token":{"string":"b24774e37c2314a4"},"is_
↪duplicate":{"boolean":false},"is_confirmable":{"boolean":false},"is_rejected":{"
↪boolean":false},"is_acknowledged":{"boolean":false},"is_canceled":{"boolean":false}
↪},"accept":{"int":-1},"block1":{"string":"","block2":{"string":"","content_format":
↪{"int":-1},"etags":[],"location_path":{"string":"","location_query":{"string":"","
↪max_age":{"long":60},"observe":null,"proxy_uri":null,"size_1":null,"size_2":null,
↪uri_host":null,"uri_port":null,"uri_path":{"string":"","uri_query":{"string":"","
↪payload":{"string":""}}

```



```
{
  "message_id": {"int": 4805}, "type": {"string": "ACK"}, "code": "4.04", "raw_code": {"int": 132},
  "rtt": {"long": 35}, "is_last": {"boolean": true}, "is_notification": {"boolean": false},
  "source": {"string": "idvm-infk-mattern04.inf.ethz.ch:5683"}, "destination": {"string": ""},
  "timestamp": {"long": 0}, "token": {"string": "b24774e37c2314a4"}, "is_duplicate": {"boolean": false},
  "is_confirmable": {"boolean": false}, "is_rejected": {"boolean": false}, "is_acknowledged": {"boolean": false},
  "is_canceled": {"boolean": false}, "accept": {"int": -1}, "block1": {"string": ""}, "block2": {"string": ""},
  "content_format": {"int": -1}, "etags": [], "location_path": {"string": ""}, "location_query": {"string": ""},
  "max_age": {"long": 60}, "observe": null, "proxy_uri": null, "size_1": null, "size_2": null,
  "uri_host": null, "uri_port": null, "uri_path": {"string": ""}, "uri_query": {"string": ""},
  "payload": {"string": ""}
}
{
  "message_id": {"int": 4806}, "type": {"string": "ACK"}, "code": "4.04", "raw_code": {"int": 132},
  "rtt": {"long": 35}, "is_last": {"boolean": true}, "is_notification": {"boolean": false},
  "source": {"string": "idvm-infk-mattern04.inf.ethz.ch:5683"}, "destination": {"string": ""},
  "timestamp": {"long": 0}, "token": {"string": "b24774e37c2314a4"}, "is_duplicate": {"boolean": false},
  "is_confirmable": {"boolean": false}, "is_rejected": {"boolean": false}, "is_acknowledged": {"boolean": false},
  "is_canceled": {"boolean": false}, "accept": {"int": -1}, "block1": {"string": ""}, "block2": {"string": ""},
  "content_format": {"int": -1}, "etags": [], "location_path": {"string": ""}, "location_query": {"string": ""},
  "max_age": {"long": 60}, "observe": null, "proxy_uri": null, "size_1": null, "size_2": null,
  "uri_host": null, "uri_port": null, "uri_path": {"string": ""}, "uri_query": {"string": ""},
  "payload": {"string": ""}
}
```

Features

1. Secure DTLS client connection.
2. Supports Observable resources to stream changes on a resource to Kafka.
3. Routing of data via KCQL to topics.
4. Automatic conversion of CoAP Response messages to Connect Structs.

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The CoAP Source supports the following:

```
INSERT INTO <topic> SELECT * FROM <resource>
```

No selection of fields on the CoAP message is support. All the message attributes are mapped to predefined Struct representing the CoAP response message.

DTLS Secure connections

The Connector use the [Californium](#) Java API and for secure connections use the Scandium security module provided by Californium. Scandium (Sc) is an implementation of Datagram Transport Layer Security 1.2, also known as [RFC 6347](#).

Please refer to the Californium [certification](#) repo page for more information.

The connector supports:

1. SSL trust and key stores

2. Public/Private PEM keys and PSK client/identity

3. PSK Client Identity

The Sink will attempt secure connections in the following order if the URI schema of `connect.coap.uri` set to secure, i.e. “coaps”. If `connect.coap.username` is set PSK client identity authentication is used, if additional `connect.coap.private.key.path` Public/Private keys authentication will also be attempt. Otherwise SSL trust and key store.

```
`openssl pkcs8 -in privatekey.pem -topk8 -nocrypt -out privatekey-pkcs8.pem`
```

Only cipher suites `TLS_PSK_WITH_AES_128_CCM_8` and `TLS_PSK_WITH_AES_128_CBC_SHA256` are ↪ currently supported.

Warning: The keystore, truststore, public and private files must be available on the local disk of the worker task.

Loading specific certificates can be achieved by providing a comma separated list for the `connect.coap.certs` configuration option. The certificate chain can be set by the `connect.coap.cert.chain.key` configuration option.

Configurations

`connect.coap.uri`

Uri of the CoAP server.

- Data Type : string
- Importance: high
- Optional : no

`connect.coap.kcql`

The KCQL statement to select and route resources to topics.

- Data Type : string
- Importance: high
- Optional : no

`connect.coap.port`

The port the DTLS connector will bind to on the Connector host.

- Data Type : int
- Importance: medium
- Optional : yes
- Default : 0

`connect.coap.host`

The hostname the DTLS connector will bind to on the Connector host.

- Data Type : string
- Importance: medium
- Optional : yes

- Default : localhost

`connect.coap.username`

CoAP PSK identity.

- Data Type : string
- Importance: medium
- Optional : yes

`connect.coap.password`

CoAP PSK secret.

- Data Type : password
- Importance: medium
- Optional : yes

`connect.coap.public.key.file`

Path to the public key file for use in with PSK credentials.

- Data Type : string
- Importance: medium
- Optional : yes

`connect.coap.private.key.file`

Path to the private key file for use in with PSK credentials in PKCS8 rather than PKCS1 Use open SSL to convert.

```
`openssl pkcs8 -in privatekey.pem -topk8 -nocrypt -out privatekey-pkcs8.pem`
```

Only cipher suites TLS_PSK_WITH_AES_128_CCM_8 and TLS_PSK_WITH_AES_128_CBC_SHA256 are currently supported.

- Data Type : string
- Importance: medium
- Optional : yes

`connect.coap.keystore.pass`

The password of the key store

- Data Type : string
- Importance: medium
- Optional : yes
- Default : rootPass

`connect.coap.keystore.path`

The path to the keystore.

- Data Type : string
- Importance: medium
- Optional : yes

- Default :

`connect.coap.truststore.pass`

The password of the trust store

- Data Type : string
- Importance: medium
- Optional : yes
- Default : rootPass

`connect.coap.truststore.path`

The path to the truststore.

- Data Type : string
- Importance: medium
- Optional : yes
- Default :

`connect.coap.certs`

The certificates to load from the trust store.

- Data Type : list
- Importance: medium
- Optional : yes
- Default :

`connect.coap.cert.chain.key`

The key to use to get the certificate chain.

- Data Type : string
- Importance: medium
- Optional : yes
- Default : client

`connect.coap.batch.size`

The number of events to take from the internal queue to batch together to send to Kafka.

- Data Type : int
- Importance: medium
- Optional : yes
- Default : 100

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes

- Default : false

Schema Evolution

The schema is fixed.

The following schema is used for the key:

Name	Type
source	Optional string
source_resource	Optional String
message_id	Optional int32

The following schema is used for the payload:

Name	Type
message_id	Optional int32
type	Optional String
code	Optional String
raw_code	Optional int32
rtt	Optional int64
is_last	Optional boolean
is_notification	Optional boolean
source	Optional String
destination	Optional String
timestamp	Optional int64
token	Optional String
is_duplicate	Optional boolean
is_confirmable	Optional boolean
is_rejected	Optional boolean
is_acknowledged	Optional boolean
is_canceled	Optional boolean
accept	Optional int32
block1	Optional String
block2	Optional String
content_format	Optional int32
etags	Array of Optional Strings
location_path	Optional String
location_query	Optional String
max_age	Optional int64
observe	Optional int32
proxy_uri	Optional String
size_1	Optional String
size_2	Optional String
uri_host	Optional String
uri_port	Optional int32
uri_path	Optional String
uri_query	Optional String
payload	Optional String

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect, bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect FTP

Note: All credits for this connector go to Eneco's team, where this connector was forked from!

Monitors files on an FTP server and feeds changes into Kafka.

Provide the remote directories and on specified intervals, the list of files in the directories is refreshed. Files are downloaded when they were not known before, or when their timestamp or size are changed. Only files with a timestamp younger than the specified maximum age are considered. Hashes of the files are maintained and used to check for content changes. Changed files are then fed into Kafka, either as a whole (update) or only the appended part (tail), depending on the configuration. Optionally, file bodies can be transformed through a pluggable system prior to putting it into Kafka.

Data Types

Each Kafka record represents a file, and has the following types.

- The format of the keys is configurable through `connect.ftp.keystyle=string|struct`. It can be a *string* with the file name, or a *FileInfo* structure with *name: string* and *offset: long*. The offset is always 0 for files that are updated as a whole, and hence only relevant for tailed files.
- The values of the records contain the body of the file as *bytes*.

Configuration

In addition to the general configuration for Kafka connectors (e.g. name, connector.class, etc.) the following options are available.

`connect.ftp.address`

host[:port] of the ftp server.

- Type: string
- Importance: high
- Optional: no

`connect.ftp.user`

Username to connect with.

- Type: string
- Importance: high
- Optional: no

`connect.ftp.password`

Password to connect with.

- Type: string
- Importance: high
- Optional: no

`connect.ftp.refresh`

iso8601 duration that the server is polled.

- Type: string
- Importance: high
- Optional: no

`connect.ftp.file.maxage`

iso8601 duration for how old files can be.

- Type: string
- Importance: high
- Optional: no

`connect.ftp.keystyle`

SourceRecord keystyle, *string* or *struct*, see above.

- Type: string
- Importance: high
- Optional: no

`connect.ftp.monitor.tail`

Comma separated list of path:destinationtopic to tail.

- Type: string
- Importance: high
- Optional: yes

`connect.ftp.monitor.update`

Comma separated list of path:destinationtopic to monitor for updates.

- Type: string
- Importance: high
- Optional: yes

`connect.ftp.sourcerecordconverter`

Source Record converter class name.

- Type: string
- Importance: high
- Optional: yes

An example file:

```
name=ftp-source
connector.class=com.datamountaineer.streamreactor.connect.connect.ftp.source.
↳FtpSourceConnector
tasks.max=1

#server settings
connect.ftp.address=localhost:21
connect.ftp.user=ftp
connect.ftp.password=ftp

#refresh rate, every minute
connect.ftp.refresh=PT1M

#ignore files older than 14 days.
connect.ftp.file.maxage=P14D

#monitor /forecasts/weather/ and /logs/ for appends to files.
#any updates go to the topics `weather` and `error-logs` respectively.
connect.ftp.monitor.tail=/forecasts/weather/:weather,/logs/:error-logs

#keep an eye on /statuses/, files are retrieved as a whole and sent to topic `status`
connect.ftp.monitor.update=/statuses/:status

#keystyle controls the format of the key and can be string or struct.
```



```
#string only provides the file name
#struct provides a structure with the filename and offset
connect.ftp.keystyle=struct
```

Tailing Versus Update as a Whole

The following rules are used.

- *Tailed* files are *only* allowed to grow. Bytes that have been appended to it since a last inspection are yielded. Preceding bytes are not allowed to change;
- *Updated* files can grow, shrink and change anywhere. The entire contents are yielded.

Data Converters

Instead of dumping whole file bodies (and the danger of exceeding Kafka's *message.max.bytes*), one might want to give an interpretation to the data contained in the files before putting it into Kafka. For example, if the files that are fetched from the FTP are comma-separated values (CSVs), one might prefer to have a stream of CSV records instead. To allow to do so, the connector provides a pluggable conversion of *SourceRecords*. Right before sending a *SourceRecord* to the Connect framework, it is run through an object that implements:

```
package com.datamountaineer.streamreactor.connect.ftp

trait SourceRecordConverter extends Configurable {
  def convert(in: SourceRecord) : java.util.List[SourceRecord]
}
```

(for the Java people, read: *interface* instead of *trait*).

The default object that is used is a pass-through converter, an instance of:

```
class NopSourceRecordConverter extends SourceRecordConverter{
  override def configure(props: util.Map[String, _]): Unit = {}
  override def convert(in: SourceRecord) : util.List[SourceRecord] = Seq(in).asJava
}
```

To override it, create your own implementation of *SourceRecordConverter*, put the jar into your *\$CLASSPATH* and instruct the connector to use it via the *.properties*:

```
connect.ftp.sourcerecordconverter=your.name.space.YourConverter
```

Kafka Connect JMS Source

The JMS Source connector allows subscribe to messages on JMS queues and topics.

The Source supports:

1. Pluggable converters of JMS payloads. If no converters are specified a Avro message is created representing the JMS Message, the payload from the message is stored as a byte array in the *payload* field of the Avro.
2. *Out of the box* converters for *Json/Avro and Binary*
3. *The KCQL routing querying* - JMS Destination to Kafka topic mapping.

Prerequisites

- Confluent 3.3
- Java 1.8
- Scala 2.11
- A JMS framework (ActiveMQ for example)

Setup

Before we can do anything, including the QuickStart we need to install the Confluent platform. For ActiveMQ follow <http://activemq.apache.org/getting-started.html> for the instruction of setting it up.

Confluent Setup

Follow the instructions [here](#).

Source Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for JMS. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create jms-sink < conf/jms-source.properties
```

The `jms-source.properties` file defines:

```
name=jms-source
connector.class=com.datamountaineer.streamreactor.connect.jms.source.
↪JMSSourceConnector
tasks.max=1
connect.jms.kcql=INSERT INTO topic SELECT * FROM jms-queue
connect.jms.queues=jms-queue
connect.jms.initial.context.factory=org.apache.activemq.jndi.
↪ActiveMQInitialContextFactory
connect.jms.url=tcp://localhost:61616
connect.jms.connection.factory=ConnectionFactory
```

1. The source connector name.

2. The JMS Source Connector class name.
3. The number of tasks to start.
4. *The KCQL routing querying.*
5. A comma separated list of queues destination types on the target JMS, must match the *from* element in KCQL.
6. The JMS initial context factory.
7. The url of the JMS broker.
8. The JMS connection factory.

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
jms-source

#Connector `jms-source`:
name=jms-source
connect.jms.kcql=INSERT INTO topic SELECT * FROM jms-queue
tasks.max=1
connector.class=com.datamountaineer.streamreactor.connect.jms.source.
↳JMSSourceConnector
connect.jms.queues=jms-queue
connect.jms.initial.context.factory=org.apache.activemq.jndi.
↳ActiveMQInitialContextFactory
connect.jms.url=tcp://localhost:61616
connect.jms.connection.factory=ConnectionFactory
#task ids: 0
```

```
INFO Kafka version : 0.10.2.0-cp1 (org.apache.kafka.common.utils.AppInfoParser:83)
INFO Kafka commitId : 64c9b42f3319cdc9 (org.apache.kafka.common.utils.
↪AppInfoParser:84)
INFO
      / _ \      / _ \      / _ \ | / _ \      _ _ _ _ _ / _ \      ( ) _ _ _ _ _
↪ _
 / / / / _ ` _ / _ ` _ / | _ / _ _ V / / / _ V _ / _ ` / / _ V _ V _ V
↪ _/
 / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ /
 / _ _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ /
      / / | / / _ / _ \ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
      / _ \ / | _ / _ \ V _ V _ V _ V / / _ _ / _ / _ \ _ \      By Andrew Stevenson
      \ _ \ / / / _ _ \ / _ \ / _ \ _ _ \ _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \
      \ _ \ / / / _ _ \ / _ \ / _ \ _ _ \ _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \
(com.datamountaineer.streamreactor.connect.jms.source.JMSSourceTask:22)
INFO JMSConfig values:
connect.jms.batch.size = 100
connect.jms.connection.factory = ConnectionFactory
connect.jms.converter.throw.on.error = false
```

```

connect.jms.destination.selector = CDI
connect.jms.error.policy = THROW
connect.jms.initial.context.extra.params = []
connect.jms.initial.context.factory = org.apache.activemq.jndi.
↪ActiveMQInitialContextFactory
connect.jms.kcql = INSERT INTO topic SELECT * FROM jms-queue
connect.jms.max.retries = 20
connect.jms.password = null
connect.jms.queues = [jms-queue]
connect.jms.retry.interval = 60000
connect.jms.default.converters =
connect.jms.topics = []
connect.jms.url = tcp://localhost:61616
connect.jms.username = null
(com.datamountaineer.streamreactor.connect.jms.config.JMSConfig:180)
INFO Instantiated connector jms-source with version null of type class com.
↪datamountaineer.streamreactor.connect.jms.source.JMSSourceConnector (org.apache.
↪kafka.connect.runtime.Worker:181)
INFO Finished creating connector jms-source (org.apache.kafka.connect.runtime.
↪Worker:194)
INFO SourceConnectorConfig values:
    connector.class = com.datamountaineer.streamreactor.connect.jms.source.
↪JMSSourceConnector
    key.converter = null
    name = jms-source
    tasks.max = 1
    transforms = null
    value.converter = null
(org.apache.kafka.connect.runtime.SourceConnectorConfig:180)

```

Test Records

Now we need to send some records into the ActiveMQ broker for the Source Connector to pick up. We can do this with the ActiveMQ command line producer. In the bin folder of the Active MQ location run the following to insert 1000 messages into a queue called *jms-queue*.

```
activemq producer --destination queue://jms-queue --message "hello DataMountaineer"
```

We should immediately see the records coming through the sink and into our Kafka topic:

```

${CONFLUENT_HOME}/bin/kafka-avro-console-consumer \
--zookeeper localhost:2181 \
--topic topic \
--from-beginning

```

```

{"message_timestamp":{"long":1490799748984},"correlation_id":null,"redelivered":{"
↪"boolean":false},"reply_to":null,"destination":{"string":"queue://jms-queue"},
↪"message_id":{"string":"ID:Andrews-MacBook-Pro.local-49870-1490799747943-1:1:1:1:997
↪"},"mode":{"int":2},"type":null,"priority":{"int":4},"bytes_payload":{"bytes":"hello
↪"},"properties":null}
{"message_timestamp":{"long":1490799748985},"correlation_id":null,"redelivered":{"
↪"boolean":false},"reply_to":null,"destination":{"string":"queue://jms-queue"},
↪"message_id":{"string":"ID:Andrews-MacBook-Pro.local-49870-1490799747943-1:1:1:1:998
↪"},"mode":{"int":2},"type":null,"priority":{"int":4},"bytes_payload":{"bytes":"hello
↪"},"properties":null}
{"message_timestamp":{"long":1490799748986},"correlation_id":null,"redelivered":{"
↪"boolean":false},"reply_to":null,"destination":{"string":"queue://jms-queue"},
↪"message_id":{"string":"ID:Andrews-MacBook-Pro.local-49870-1490799747943-1:1:1:1:999
64"},"mode":{"int":2},"type":null,"priority":{"int":4},"bytes_payload":{"bytes":"hell
↪"},"properties":null}

```

```
{ "message_timestamp": { "long": 1490799748987 }, "correlation_id": null, "redelivered": {
  ↪ "boolean": false }, "reply_to": null, "destination": { "string": "queue://jms-queue" },
  ↪ "message_id": { "string": "ID:Andrews-MacBook-Pro.local-49870-1490799747943-"
  ↪ "1:1:1:1:1000" }, "mode": { "int": 2 }, "type": null, "priority": { "int": 4 }, "bytes_payload": {
  ↪ "bytes": "hello" }, "properties": null }
```

Features

The Source supports:

1. KCQL routing of JMS destination messages to Kafka topics.
2. Pluggable converters.
3. Default conversion of JMS Messages to Avro with the payload as a Byte array.
4. Extra connection properties for specialized connections such as SOLACE_VPN.

Converters

We provide four converters out of the box but you can plug your own. See an example [here](#). which and be set in `connect.jms.kcql` statement.

AvroConverter

```
com.datamountaineer.streamreactor.connect.converters.source.AvroConverter
```

The payload of the JMS message is an Avro message. In this case you need to provide a path for the Avro schema file to be able to decode it.

JsonSimpleConverter

```
com.datamountaineer.streamreactor.connect.converters.source.
JsonSimpleConverter
```

The payload for the JMS message is a Json message. This converter will parse the json and create an Avro record for it which will be sent over to Kafka.

JsonConverterWithSchemaEvolution

An experimental converter for converting Json messages to Avro. The resulting Avro schema is fully compatible as new fields are added as the JMS json payload evolves.

BytesConverter

```
com.datamountaineer.streamreactor.connect.converters.source.BytesConverter
```

This is the default implementation. The JMS payload is taken as is: an array of bytes and sent over Kafka as an avro record with `Schema.BYTES`. You don't have to provide a mapping for the source to get this converter!!

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The JMS Source supports the following:

```
INSERT INTO <kafka target> SELECT * FROM <jms destination> WITHTYPE <TOPIC|QUEUE>_
↪ [WITHCONVERTER=myclass]
```

Example:

```
#select from a JMS queue and write to a kafka topic
INSERT INTO topicA SELECT * FROM jms_queue WITHTYPE QUEUE

#select from a JMS topic and write to a kafka topic
INSERT INTO topicA SELECT * FROM jms_queue WITHTYPE TOPIC
```

Configurations

`connect.jms.url`

Provides the JMS broker url

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.username`

Provides the user for the JMS connection.

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.password`

Provides the password for the JMS connection.

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.initial.context.factory`

- Data Type: string
- Importance: high
- Optional: no

Initial Context Factory, e.g: `org.apache.activemq.jndi.ActiveMQInitialContextFactory`.

`connect.jms.connection.factory`

The ConnectionFactory implementation to use.

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.destination.selector`

- Data Type: String

- Importance: high
- Optional: no
- Default: CDI

Selector to use for destination lookup. Either CDI or JNDI.

`connect.jms.initial.context.extra.params`

- Data Type: String
- Importance: high
- Optional: yes

List (comma separated) of extra properties as key/value pairs with a colon delimiter to supply to the initial context e.g. SOLACE_JMS_VPN:my_solace_vp.

`connect.jms.kcql`

KCQL expression describing field selection and routes. The kcql expression also handles setting the JMS destination type, i.e. TOPIC or QUEUE via the `withtype` keyword and additionally the converter via the `withconverter` keyword. If no converter is specified the sink will default to the BytesConverter. This will send an avro message over Kafka using Schema.BYTES

- Data Type: string
- Importance: high
- Optional : no

`connect.converter.avro.schemas`

If the AvroConverter is used you need to provide an avro Schema to be able to read and translate the raw bytes to an avro record. The format is `$JMS_TOPIC=$PATH_TO_AVRO_SCHEMA_FILE`

- Data type: bool
- Importance: medium
- Optional: yes
- Default: null

`connect.jms.batch.size`

- Type: int
- Importance: medium
- Optional: yes
- Default: 100

The batch size to take from the JMS destination on each poll of Kafka Connect.

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Provide your own Converter

You can always provide your own logic for converting the JMS message to your an avro record. If you have messages coming in in Protobuf format you can deserialize the message based on the schema and create the avro record. All you have to do is create a new project and add our dependency:

Gradle:

```
compile "com.datamountaineer:kafka-connect-common:0.7.1"
```

Maven:

```
<dependency>
  <groupId>com.datamountaineer</groupId>
  <artifactId>kafka-connect-common</artifactId>
  <version>0.7.1</version>
</dependency>
```

Then all you have to do is implement `com.datamountaineer.streamreactor.connect.converters.source.Converter`.

Here is our BytesConverter class code:

```
class BytesConverter extends Converter {
  override def convert(kafkaTopic: String, sourceTopic: String, messageId: String, _
    bytes: Array[Byte]): SourceRecord = {
    new SourceRecord(Collections.singletonMap(Converter.TopicKey, sourceTopic),
      null,
      kafkaTopic,
      MsgKey.schema,
      MsgKey.getStruct(sourceTopic, messageId),
      Schema.BYTES_SCHEMA,
      bytes)
  }
}
```

Schema Evolution

Not applicable.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.


```
5. Start Connect, bin/connect-distributed etc/schema-registry/  
connect-avro-distributed.properties
```

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect ReThink

A Connector and Source to write events from ReThinkDB to Kafka. The connector subscribes to changefeeds on tables and streams the records to Kafka.

The Source supports:

1. *The KCQL routing querying* - Table to topic routing
2. Initialization (Read feed from start) via KCQL.
3. ReThinkDB type (add, delete, update).
4. ReThinkDB initial states.

Prerequisites

- Confluent 3.3
- RethinkDb 2.3.3
- Java 1.8
- Scala 2.11

Setup

Rethink Setup

Download and install RethinkDb. Follow the instruction [here](#) dependent on your operating system.

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for ReThinkDB. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create rethink-source < conf/rethink-source.properties
#Connector name=`rethink-source`
name=rethink-source
connect.rethink.host=localhost
connect.rethink.port=28015
connector.class=com.datamountaineer.streamreactor.connect.rethink.source.
↪ReThinkSourceConnector
tasks.max=1
connect.rethink.db=test
connect.rethink.kcql=INSERT INTO rethink-topic SELECT * FROM source-test
#task ids: 0
```

The `rethink-source.properties` file defines:

1. The name of the source.
2. The name of the rethink host to connect to.
3. The rethink port to connect to.
4. The Source class.
5. The max number of tasks the connector is allowed to created. The connector splits and groups the `connect.rethink.kcql` by the number of tasks to ensure a distribution based on allowed number of tasks and Source tables.
6. The ReThinkDB database to connect to.
7. *The KCQL routing querying.*

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect
```



```
{ "state": { "string": "ready" }, "old_val": null, "new_val": null, "type": { "string": "state" }  
↪ }  
{ "state": null, "old_val": null, "new_val": { "string": "{tv_show=Battlestar Galactica,  
↪ name=datamountaineers-rule, id=ec9d337e-ee07-4128-a830-22e4f055ce64, posts=[  
↪ {title=Decommissioning speech3, content=The Cylon War is long over...}, {title=We  
↪ are at war, content=Moments ago, this ship received word...}, {title=The new Earth,  
↪ content=The discoveries of the past few days...}]]}" }, "type": { "string": "add" } }
```

Features

The ReThinkDb Source writes change feed records from RethinkDb to Kafka.

The Source supports:

1. Table to topic routing
2. Initialization (Read feed from start)
3. ReThinkDB type (add, delete, update)
4. ReThinkDB initial states

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The ReThink Source supports the following:

```
INSERT INTO <target table> SELECT <fields> FROM <source topic> <INITIALIZE> <BATCH N>
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA  
INSERT INTO tableA SELECT * FROM topicA  
  
#Insert mode, select all fields from topicA and write to tableA, read from start  
INSERT INTO tableA SELECT * FROM topicA INITIALIZE  
  
#Insert mode, select all fields from topicA and write to tableA, read from start and  
↪ batch 100 rows to send to kafka  
INSERT INTO tableA SELECT * FROM topicA INITIALIZE BATCH = 100
```

Configurations

connect.rethink.kcql

Kafka connect query language expression. Allows for expressive topic to table routing, field selection and renaming. Fields to be used as the row key can be set by specifying the PK. The below example uses field1 as the primary key.

- Data type : string
- Importance: high
- Optional : no

Examples:

```
INSERT INTO TOPIC1 SELECT * FROM TABLE1; INSERT INTO TOPIC2 SELECT * FROM TABLE2
```

`connect.rethink.host`

Specifies the rethink server.

- Data type : string
- Importance: high
- Optional : no

`connect.rethink.port`

Specifies the rethink server port number.

- Data type : int
- Importance: high
- Optional : yes
- Default : 28015

`connect.rethink.db`

Specifies the rethink database to connect to.

- Data type : string
- Importance: high
- Optional : yes
- Default : `connect_rethink_sink`

`connect.rethink.batch.size`

The number of records to drain from the internal queue on each poll.

- Data type : int
- Importance: medium
- Optional : yes
- Default : 1000

`connect.rethink.linger.ms`

The number of milliseconds to wait before flushing the received messages to Kafka. The records will be flushed if the batch size is reached before the linger period has expired.

- Data type : int
- Importance: medium
- Optional : yes
- Default : 5000

`connect.rethink.cert.file`

Certificate file to connect to a TLS enabled ReThink cluster. **Cannot** be used in conjunction with username/password. `connect.rethink.auth.key` must be set.

- Data type: string

- Optional : yes

`connect.rethink.auth.key`

Authentication key to connect to a TLS enabled ReThink cluster. **Cannot** be used in conjunction with `username/password`. `connect.rethink.cert.file` must be set.

- Data type: string
- Optional : yes

`connect.rethink.username`

Username to connect to ReThink with.

- Data type: string
- Optional : yes

`connect.rethink.password`

Password to connect to ReThink with.

- Data type: string
- Optional : yes

`connect.rethink.ssl.enabled`

Enables SSL communication against an SSL enabled Rethink cluster.

- Data type: boolean
- Optional : yes
- Default : false

`connect.rethink.trust.store.password`

Password for truststore.

- Data type: string
- Optional : yes

`connect.rethink.key.store.path`

Path to truststore.

- Data type: string
- Optional : yes

`connect.rethink.key.store.password`

Password for key store.

- Data type: string
- Optional : yes

`connect.rethink.ssl.client.cert.auth`

Path to keystore.

- Data type: string
- Optional : yes

```
connect.progress.enabled
```

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=rethink-source
connect.rethink.host=localhost
connect.rethink.port=28015
connector.class=com.datamountaineer.streamreactor.connect.rethink.source.
↳ReThinkSourceConnector
tasks.max=1
connect.rethink.kcql=INSERT INTO rethink-topic SELECT * FROM source-test
```

Schema Evolution

The schema is fixed. The following schema is used:

Name	Type	Optional
state	string	yes
new_val	string	yes
old_val	string	yes
type	string	yes

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Yahoo

A Connector and Source to write events from the Yahoo Finance API to Kafka.

DOCS WIP!

Kafka Connect Mqtt Source

A Connector to read events from Mqtt and push them to Kafka. The connector subscribes to the specified topics and streams the records to Kafka.

The Source supports:

1. Pluggable converters of MQTT payloads.
2. *Out of the box converters for Json/Avro and Binary*
3. *The KCQL routing querying* - Topic to Topic mapping and Field selection.

Prerequisites

- Confluent 3.3
- Mqtt server
- Java 1.8
- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

Mqtt Setup

For testing we will use a simple application spinning up an mqtt server using Moquette. Download and unzip [this](#).

Once you have unpacked the archiver you should start the server.

```
bin/mqtt-server
```

You should see the following outcome:

```
log4j:WARN No appenders could be found for logger (io.moquette.server.Server).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Starting mqtt service on port 11883
Hit Enter to start publishing messages on topic: /mjson and /mavro.
```

The server has started but no records have been published yet. More on this later once we start the source.

Source Connector QuickStart

We you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for MQTT. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create mqtt-source < conf/source.kcql/mqtt-source.properties

#Connector name=`mqtt-source`
name=mqtt-source
tasks.max=1
connect.mqtt.connection.clean=true
connect.mqtt.connection.timeout=1000
connect.mqtt.kcql=INSERT INTO kjson SELECT * FROM /mjson WITHCONVERTER=`com.
↳ datamountaineer.streamreactor.connect.converters.source.JsonSimpleConverter`
connect.mqtt.connection.keep.alive=1000
connect.mqtt.client.id=dm_source_id
connect.mqtt.converter.throw.on.error=true
connect.mqtt.hosts=tcp://127.0.0.1:11883
connect.mqtt.service.quality=1
connector.class=com.datamountaineer.streamreactor.connect.mqtt.source.
↳ MqttSourceConnector
#task ids: 0
```

The `mqtt-source.properties` file defines:

1. The name of the source.
2. The name number of tasks.
3. Clean the mqtt connection.
4. The Kafka Connect Query statements to read from json and avro topics and insert into Kafka kjson and kavro topics.
5. Setting the time window to emit keep alive pings
6. The mqtt client identifier.
7. If a conversion can't happen it will throw an exception.
8. The connection to the Mqtt server.
9. The quality of service for the messages.
10. Set the connector source class.

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
mqtt-source
```

```
[2016-12-20 16:51:08,058] INFO
| _ _ \ _ _ | _ _ _ | _ _ | _ _ _ _ _ | _ _ _ _ _ | _ _ _ _ _ | | | | | | | | | | | | | | |
| | | / _ ' _ / _ ' | | | / _ _ | | | ' _ \ _ / _ ' _ \ _ \ _ |
| | | ( | | | ( | | | ( ) | | | | | | | ( | | | | _ / _ / |
| _ _ / \ _ / \ \ _ / \ _ / \ _ / \ _ / \ _ / \ _ / \ _ |
| _ \ | _ _ | _ _ | _ / _ | _ _ _ _ _ |
| | \ | / _ ' _ _ _ \ _ \ / _ \ | | | ' _ / _ / _ \
| | | | ( _ _ | _ _ ) | ( ) | _ _ | | ( _ /
| _ | _ \ _ / \ \ _ / \ _ / \ _ / \ _ | by Stefan Bocutiu
    | _ |
    (com.datamountaineer.streamreactor.connect.mqtt.source.MqttSourceTask:37)
[2016-12-20 16:51:08,090] INFO MqttSourceConfig values:
    connect.mqtt.kcql = INSERT INTO kjson SELECT * FROM /mjson WITHCONVERTER=`com.
    ↪ datamountaineer.streamreactor.connect.converters.source.JsonSimpleConverter`
    connect.mqtt.service.quality = 1
    connect.mqtt.connection.ssl.cert = null
    connect.mqtt.connection.keep.alive = 1000
    connect.mqtt.hosts = tcp://127.0.0.1:11883
    connect.mqtt.converter.throw.on.error = true
    connect.mqtt.connection.timeout = 1000
    connect.mqtt.username = null
    connect.mqtt.connection.clean = true
    connect.mqtt.connection.ssl.ca.cert = null
    connect.mqtt.connection.ssl.key = null
    connect.mqtt.password = null
```

```
connect.mqtt.client.id = dm_source_id
(com.datamountaineer.streamreactor.connect.mqtt.config.MqttSourceConfig:178)
```

Test Records

Go to the mqtt-server application you downloaded and unzipped and execute:

```
./bin/mqtt-server
```

This will put the following records into the json Mqtt topic:

```
TemperatureMeasure(1, 31.1, "EMEA", System.currentTimeMillis())
TemperatureMeasure(2, 30.91, "EMEA", System.currentTimeMillis())
TemperatureMeasure(3, 30.991, "EMEA", System.currentTimeMillis())
TemperatureMeasure(4, 31.061, "EMEA", System.currentTimeMillis())
TemperatureMeasure(101, 27.001, "AMER", System.currentTimeMillis())
TemperatureMeasure(102, 38.001, "AMER", System.currentTimeMillis())
TemperatureMeasure(103, 26.991, "AMER", System.currentTimeMillis())
TemperatureMeasure(104, 34.17, "AMER", System.currentTimeMillis())
```

Check for records in Kafka

Check for records in Kafka with the console consumer. the topic for kjson (the Mqtt payload was a json and we translated that into a Kafka Connect Struct)

```
bin/kafka-avro-console-consumer --zookeeper localhost:2181 --topic kjson --from-
↪beginning
```

You should see the following output

```
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
{"deviceId":1,"value":31.1,"region":"EMEA","timestamp":1482236627236}
{"deviceId":2,"value":30.91,"region":"EMEA","timestamp":1482236627236}
{"deviceId":3,"value":30.991,"region":"EMEA","timestamp":1482236627236}
{"deviceId":4,"value":31.061,"region":"EMEA","timestamp":1482236627236}
{"deviceId":101,"value":27.001,"region":"AMER","timestamp":1482236627236}
{"deviceId":102,"value":38.001,"region":"AMER","timestamp":1482236627236}
{"deviceId":103,"value":26.991,"region":"AMER","timestamp":1482236627236}
{"deviceId":104,"value":34.17,"region":"AMER","timestamp":1482236627236}
```

Check for records in Kafka with the console consumer. the topic for kavro (the Mqtt payload was a avro and we translated that into a Kafka Connect Struct)

```
bin/kafka-avro-console-consumer --zookeeper localhost:2181 --topic kavro --from-
↪beginning
```

You should see the following output

```
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
{"deviceId":1,"value":31.1,"region":"EMEA","timestamp":1482236627236}
{"deviceId":2,"value":30.91,"region":"EMEA","timestamp":1482236627236}
{"deviceId":3,"value":30.991,"region":"EMEA","timestamp":1482236627236}
{"deviceId":4,"value":31.061,"region":"EMEA","timestamp":1482236627236}
```

```
{ "deviceId":101, "value":27.001, "region":"AMER", "timestamp":1482236627236 }
{ "deviceId":102, "value":38.001, "region":"AMER", "timestamp":1482236627236 }
{ "deviceId":103, "value":26.991, "region":"AMER", "timestamp":1482236627236 }
{ "deviceId":104, "value":34.17, "region":"AMER", "timestamp":1482236627236 }
```

Features

The Mqtt source allows you to plugin your own converter. Say you receive protobuf data, all you have to do is to write your own very specific converter that knows how to convert from protobuf to SourceRecord. All you have to do is set the `WITHCONVERTER=` in the KCQL statement.

Converters

We provide four converters out of the box but you can plug your own. See an example [here](#).

AvroConverter

```
com.datamountaineer.streamreactor.connect.converters.source.AvroConverter
```

The payload for the Mqtt message is an Avro message. In this case you need to provide a path for the Avro schema file to be able to decode it.

JsonSimpleConverter

```
com.datamountaineer.streamreactor.connect.converters.source.
JsonSimpleConverter
```

The payload for the Mqtt message is a Json message. This converter will parse the json and create an Avro record for it which will be sent over to Kafka.

JsonConverterWithSchemaEvolution

An experimental converter for converting Json messages to Avro. The resulting Avro schema is fully compatible as new fields are added as the MQTT json payload evolves.

BytesConverter

```
com.datamountaineer.streamreactor.connect.converters.source.BytesConverter
```

This is the default implementation. The Mqtt payload is taken as is: an array of bytes and sent over Kafka as an avro record with `Schema.BYTES`. You don't have to provide a mapping for the source to get this converter!!

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The Mqtt Source supports the following:

```
INSERT INTO <target topic> SELECT * FROM <mqtt source topic> [WITHCONVERTER=`myclass`]
```

Example:

```
#Insert mode, select all fields from topicA and write to topic kafkaTopic1 with
↳converter myclass
INSERT INTO kafkaTopic1 SELECT * FROM mqttTopicA [WITHCONVERTER=myclass]
```

```
#wildcard
INSERT INTO kafkaTopic1 SELECT * FROM mqttTopicA/+/sensors [WITHCONVERTER=`myclass`]
```

Note: Wildcard MQTT subscriptions are supported but require the same converter to be used for all.

Configurations

`connect.mqtt.kcql`

Kafka connect query language expression. Allows for expressive Mqtt topic to Kafka topic routing. Currently there is no support for filtering the fields from the incoming payload.

- Data type : string
- Importance: high
- Optional : no

`connect.mqtt.hosts`

Specifies the mqtt connection endpoints.

- Data type : string
- Importance: high
- Optional : no

Example:

```
tcp://broker.datamountaineer.com:1883
```

`connect.mqtt.service.quality`

The Quality of Service (QoS) level is an agreement between sender and receiver of a message regarding the guarantees of delivering a message. There are 3 QoS levels in MQTT: At most once (0); At least once (1); Exactly once (2).

- Data type : int
- Importance: high
- Optional : yes
- Default: 1

`connect.mqtt.username`

Contains the Mqtt connection user name

- Data type : string
- Importance: medium
- Optional : yes
- Default: null

`connect.mqtt.password`

Contains the Mqtt connection password

- Data type : string

- Importance: medium
- Optional : yes
- Default: null

`connect.mqtt.client.id`

Provides the client connection identifier. If is not provided the framework will generate one.

- Data type: string
- Importance: medium
- Optional: yes
- Default: generated

`connect.mqtt.connection.timeout`

Sets the timeout to wait for the broker connection to be established

- Data type: int
- Importance: medium
- Optional: yes
- Default: 3000 (ms)

`connect.mqtt.connection.clean`

The clean session flag indicates the broker, whether the client wants to establish a persistent session or not. A persistent session (the flag is false) means, that the broker will store all subscriptions for the client and also all missed messages, when subscribing with Quality of Service (QoS) 1 or 2. If clean session is set to true, the broker won't store anything for the client and will also purge all information from a previous persistent session.

- Data type: boolean
- Importance: medium
- Optional: yes
- Default: true

`connect.mqtt.connection.keep.alive`

The keep alive functionality assures that the connection is still open and both broker and client are connected to one another. Therefore the client specifies a time interval in seconds and communicates it to the broker during the establishment of the connection. The interval is the longest possible period of time, which broker and client can endure without sending a message.

- Data type: int
- Importance: medium
- Optional: yes
- Default: 5000

`connect.mqtt.connection.ssl.ca.cert`

Provides the path to the CA certificate file to use with the Mqtt connection

- Data type: string
- Importance: medium
- Optional: yes

- Default: null

`connect.mqtt.connection.ssl.cert`

Provides the path to the certificate file to use with the Mqtt connection

- Data type: string
- Importance: medium
- Optional: yes
- Default: null

`connect.mqtt.connection.ssl.key`

Certificate private key file path.

- Data type: string
- Importance: medium
- Optional: yes
- Default: null

`connect.mqtt.converter.throw.on.error`

If set to false the conversion exception will be swallowed and everything carries on BUT the message is lost!!; true will throw the exception.Default is false.”

- Data type: bool
- Importance: medium
- Optional: yes
- Default: false

`connect.converter.avro.schemas`

If the AvroConverter is used you need to provide an avro Schema to be able to read and translate the raw bytes to an avro record. The format is \$MQTT_TOPIC=\$PATH_TO_AVRO_SCHEMA_FILE

- Data type: bool
- Importance: medium
- Optional: yes
- Default: null

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=mqtt-source
tasks.max=1
connect.mqtt.connection.clean=true
connect.mqtt.connection.timeout=1000
connect.mqtt.kcql=INSERT INTO kjson SELECT * FROM /mjson WITHCONVERTER=`myclass`;
↪INSERT INTO kavro SELECT * FROM /mavro WITHCONVERTER=`myclass`
connect.mqtt.connection.keep.alive=1000
connect.converter.avro.schemas=/mavro=$PATH_TO/temperaturemeasure.avro
connect.mqtt.client.id=dm_source_id,
connect.mqtt.converter.throw.on.error=true
connect.mqtt.hosts=tcp://127.0.0.1:11883
connect.mqtt.service.quality=1
connector.class=com.datamountaineer.streamreactor.connect.mqtt.source.
↪MqttSourceConnector
```

Provide your own Converter

You can always provide your own logic for converting the raw Mqtt message bytes to your an avro record. If you have messages coming in Protobuf format you can deserialize the message based on the schema and create the avro record. All you have to do is create a new project and add our dependency:

Gradle:

```
compile "com.datamountaineer:kafka-connect-common:0.7.1"
```

Maven:

```
<dependency>
  <groupId>com.datamountaineer</groupId>
  <artifactId>kafka-connect-common</artifactId>
  <version>0.7.1</version>
</dependency>
```

Then all you have to do is implement `com.datamountaineer.streamreactor.connect.converters.source.Converter`.

Here is our BytesConverter class code:

```
class BytesConverter extends Converter {
  override def convert(kafkaTopic: String, sourceTopic: String, messageId: String, ↪
  ↪bytes: Array[Byte]): SourceRecord = {
    new SourceRecord(Collections.singletonMap(Converter.TopicKey, sourceTopic),
      null,
      kafkaTopic,
      MsgKey.schema,
      MsgKey.getStruct(sourceTopic, messageId),
      Schema.BYTES_SCHEMA,
      bytes)
  }
}
```

All our implementation will send a a MsgKey object as the Kafka message key. It contains the Mqtt source topic and the Mqtt message id

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Sink Connectors

Sink connectors stream data from Kafka into external systems.

Kafka Connect Azure DocumentDb Sink

The Azure DocumentDb Sink allows you to write events from Kafka to your DocumentDb instance. The connector converts the Kafka Connect SinkRecords to DocumentDb Documents and will do an insert or upsert, depending on the configuration you chose. If the database doesn't exist it can be created automatically - if the configuration flag is set to true (See Configurations section below). The targeted collections will be created if they don't already exist.

The Sink supports:

1. *The KCQL routing querying* - Topic to measure mapping and Field selection.
2. Schema registry support for Connect/Avro with a schema.

3. Schema registry support for Connect and no schema (schema set to Schema.String)
4. Json payload support, no Schema Registry.
5. Error policies.
6. Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema.

The Sink supports three Kafka payloads type:

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example [here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires SchemaRegistry which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should read `value.converter=org.apache.kafka.connect.json.JsonConverter`. The difference would be to point your serialization to `org.apache.kafka.connect.json.JsonSerializer`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a GenericRecord, to just send a message with Schema.String and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support

Prerequisites

- Azure DocumentDb instance
- Confluent 3.3
- Java 1.8
- Scala 2.11

Setup

Before we can do anything, including the QuickStart we need to install the Confluent platform. For DocumentDb instance you can either use the emulator provided by Microsoft or provision yourself an instance in Azure.

Confluent Setup

Follow the instructions [here](#).

DocumentDb Setup

If you already have an instance of Azure DocumentDb running you can skip this step. Otherwise, please follow [this](#) to get an Azure account or use the Emulator.

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line

Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

The important configuration for Connect is related to the key and value deserializer. In the first example we default to the best practice where the source sends Avro messages to a Kafka topic. It is not enough to just be Avro messages but also the producer must work with the Schema Registry to create the schema if it doesn't exist and set the schema id in the message. Every message sent will have a magic byte followed by the Avro schema id and then the actual Avro record in binary format.

Here are the entries in the config setting all the above. The are placed in the `connect-properties` file Kafka Connect is started with. Of course if your SchemaRegistry runs on a different machine or you have multiple instances of it you will have to amend the configuration.

```
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081
```

Test Database

The Sink can handle creating the database if is not present. All you have to do in this case is to set the following in the configuration

```
connect.documentdb.database.create=true
```

Starting the Connector

Download, unpack and install the Stream Reactor and Confluent.. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for Azure DocumentDB. If you are using the `docker`s you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create azure-docdb-sink < conf/source.kcql/azure-docdb-sink.
↪properties

#Connector `azure-docdb-sink`:
name=azure-docdb-sink
connector.class=com.datamountaineer.streamreactor.connect.azure.documentdb.sink.
↪DocumentDbSinkConnector
tasks.max=1
topics=orders-avro
connect.documentdb.kcql=INSERT INTO orders SELECT * FROM orders-avro
connect.documentdb.database.name=dm
connect.documentdb.endpoint=[YOUR_AZURE_ENDPOINT]
connect.documentdb.database.create=true
connect.documentdb.master.key=[YOUR_MASTER_KEY]
connect.documentdb.batch.size=10

#task ids: 0
```

If you switch back to the terminal you started Kafka Connect in you should see the DocumentDb Sink being accepted and the task starting.

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
azure-docdb-sink
```

[2017-02-28 21:34:09,922] INFO

[illegible]

Test Records

Hint: If your input topic doesn't match the target use [Kafka Streams](#) to transform in realtime the input. Also checkout the [Plumber](#), which allows you to inject a Lua script into [Kafka Streams](#) to do this, no Java or Scala required!

Now we need to put some records it to the orders-topic. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema matches the table created earlier.

```
bin/kafka-avro-console-producer \
  --broker-list localhost:9092 --topic orders-avro \
  --property value.schema='{ "type": "record", "name": "myrecord", "fields": [{ "name": "id",
↪ "type": "string"},
  { "name": "created", "type": "string"}, { "name": "product", "type": "string"}, { "name":
↪ "price", "type": "double"}]}'
```

Now the producer is waiting for input. Paste in the following (each on a line separately):

```
{
  "id": "1",
  "created": "2016-05-06 13:53:00",
  "product": "OP-DAX-P-20150201-95.7",
  "price": 94.2
},
{
  "id": "2",
  "created": "2016-05-06 13:54:00",
  "product": "OP-DAX-C-20150201-100",
  "price": 99.5
},
{
  "id": "3",
  "created": "2016-05-06 13:55:00",
  "product": "FU-DATAMOUNTAINEER-20150201-100",
  "price": 10000
},
{
  "id": "4",
  "created": "2016-05-06 13:56:00",
  "product": "FU-KOSPI-C-20150201-100",
  "price": 150
}
```

Now if we check the logs of the connector we should see 4 records being inserted to DocumentDB:

```
#From the Query Explorer in you Azure run
SELECT * FROM orders
```

#The query should return something along the lines

```
[
  {
    "product": "OP-DAX-P-20150201-95.7",
    "created": "2016-05-06 13:53:00",
    "price": 94.2,
    "id": "1",
    "_rid": "Rrg+APfcfwABAAAAAAAAAA==",
    "_self": "dbs/****/colls/****/docs/Rrg+APfcfwABAAAAAAAAAA==/",
    "_etag": "\"4000c5f0-0000-0000-0000-58b5ecd10000\"",
    "_attachments": "attachments/",
    "_ts": 1488317649
  },
  {
    "product": "OP-DAX-C-20150201-100",
    "created": "2016-05-06 13:54:00",
    "price": 99.5,
    "id": "2",
    "_rid": "Rrg+APfcfwACAAAAAAAAAA==",
    "_self": "dbs/****/colls/****/docs/Rrg+APfcfwACAAAAAAAAAA==/",
    "_etag": "\"4000c6f0-0000-0000-0000-58b5ecd10000\"",
    "_attachments": "attachments/",
    "_ts": 1488317649
  },
  {
    "product": "FU-DATAMOUNTAINEER-20150201-100",
    "created": "2016-05-06 13:55:00",
    "price": 10000,
    "id": "3",
    "_rid": "Rrg+APfcfwADAAAAAAAAAA==",
    "_self": "dbs/****/colls/****/docs/Rrg+APfcfwADAAAAAAAAAA==/",
    "_etag": "\"4000c7f0-0000-0000-0000-58b5ecd10000\"",
    "_attachments": "attachments/",
    "_ts": 1488317650
  },
  {
    "product": "FU-KOSPI-C-20150201-100",
    "created": "2016-05-06 13:56:00",
    "price": 150,
    "id": "4",
    "_rid": "Rrg+APfcfwAEAAAAAAAAAA==",
    "_self": "dbs/****/colls/****/docs/Rrg+APfcfwAEAAAAAAAAAA==/",
    "_etag": "\"4000c8f0-0000-0000-0000-58b5ecd10000\"",
    "_attachments": "attachments/",
    "_ts": 1488317650
  }
]
```

Bingo, our 4 documents!

Legacy topics (plain text payload with a json string)

We have found some of the clients have already an infrastructure where they publish pure json on the topic and obviously the jump to follow the best practice and use schema registry is quite an ask. So we offer support for them as well.

This time we need to start the connect with a different set of settings.

```
#create a new configuration for connect
cp etc/schema-registry/connect-avro-distributed.properties etc/schema-registry/
↪connect-avro-distributed-json.properties
vi vim etc/schema-registry/connect-avro-distributed.properties
```

Replace the following 4 entries in the config

```
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081
```

with the following

```
key.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=false
value.converter=org.apache.kafka.connect.json.JsonConverter
value.converter.schemas.enable=false
```

Now let's restart the connect instance:

```
#start a new instance of connect
$bin/start-connect.sh
```

Use the CLI to remove the old DocumentDb Sink:

```
bin/connect-cli rm azure-docdb-sink
```

and start the new sink with the json properties files to read from the a different topic with json as the payload.

```
#make a copy of azure-docdb-sink.properties
cp azure-docdb-sink.properties azure-docdb-sink-json.properties
```

```
#edit azure-docdb-sink-json.properties replace the following keys
topics=orders-topic-json
connect.documentdb.kcql=INSERT INTO orders_j SELECT * FROM orders-topic-json
```

```
#start the connector for DocumentDb
bin/connect-cli create azure-docdb-sink-json < azure-docdb-sink-json.properties
```

Check the logs of Connect.

```
# Get connects logs
connect log connect

[2017-02-28 21:55:52,192] INFO DocumentDbConfig values:
    connect.documentdb.db.name = dm
    connect.documentdb.endpoint = [hidden]
    connect.documentdb.error.policy = THROW
```

[illegible]

Now it's time to produce some records. This time we will use the simple kafka-consoler-consumer to put simple json on the topic:

```

${CONFLUENT_HOME}/bin/kafka-console-producer --broker-list localhost:9092 --topic orders-topic-json

{"id": "1", "created": "2016-05-06 13:53:00", "product": "OP-DAX-P-20150201-95.7",
↪"price": 94.2}
{"id": "2", "created": "2016-05-06 13:54:00", "product": "OP-DAX-C-20150201-100",
↪"price": 99.5}
{"id": "3", "created": "2016-05-06 13:55:00", "product": "FU-DATAMOUNTAINEER-20150201-
↪100", "price":10000}

```

Let's check the DocumentDb database for the new records:

```
#From the Query Explorer in you Azure run
SELECT * FROM orders
```

```
#The query should return something along the lines
```

```
{
  "product": "OP-DAX-P-20150201-95.7",
  "created": "2016-05-06 13:53:00",
  "price": 94.2,
  "id": "1",
  "_rid": "Rrg+AP5X3gABAAAAAAAAAA==",
  "_self": "dbs/**/colls/**/docs/Rrg+AP5X3gABAAAAAAAAAA==/",
  "_etag": "\"00007008-0000-0000-0000-58b5f3ff0000\"",
}
```

```
    "_attachments": "attachments/",
    "_ts": 1488319485
  },
  {
    "product": "OP-DAX-C-20150201-100",
    "created": "2016-05-06 13:54:00",
    "price": 99.5,
    "id": "2",
    "_rid": "Rrg+AP5X3gACAAAAAAAAAA==",
    "_self": "dbs/****/colls/****/docs/Rrg+AP5X3gACAAAAAAAAAA==/",
    "_etag": "\"00007108-0000-0000-0000-58b5f3ff0000\"",
    "_attachments": "attachments/",
    "_ts": 1488319485
  },
  {
    "product": "FU-DATAMOUNTAINEER-20150201-100",
    "created": "2016-05-06 13:55:00",
    "price": 10000,
    "id": "3",
    "_rid": "Rrg+AP5X3gADAAAAAAAAAA==",
    "_self": "dbs/****/colls/****/docs/Rrg+AP5X3gADAAAAAAAAAA==/",
    "_etag": "\"00007208-0000-0000-0000-58b5f3ff0000\"",
    "_attachments": "attachments/",
    "_ts": 1488319485
  }
]
```

Bingo, our 3 rows!

Features

The sink connector will translate the SinkRecords to json and will insert each one in the database. We support to insert modes: INSERT and UPSERT. All of this can be expressed via KCQL (our own SQL like syntax for configuration. Please see below the section for Kafka Connect Query Language)

The sink supports:

1. Field selection - Kafka topic payload field selection is supported, allowing you to have choose selection of fields or all fields written to DocumentDb.
2. Topic to table routing. Your sink instance can be configured to handle multiple topics and collections. All you have to do is to set your configuration appropriately. Below you will find an example

```
connect.documentdb.kcql = INSERT INTO orders SELECT * FROM orders-topic; UPSERT INTO _
↪customers SELECT * FROM customer-topic PK customer_id; UPSERT INTO invoiceid as _
↪invoice_id, customerid as customer_id, value a SELECT invoice_id, FROM invoice-topic
```

3. Error policies for handling failures.

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**, *KCQL* allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The sink supports the following:


```
INSERT INTO <database>.<target collection> SELECT <fields> FROM <source topic> <PK_
↪field name>
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA
INSERT INTO collectionA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to tableB with primary_
↪key as the field id from the topic
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB PK id
```

Error Policies

The sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the database is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the sink will retry can be controlled by using the `connect.documentdb.max.retires` and the `connect.documentdb.retry.interval`.

Topic Routing

The sink supports topic routing that maps the messages from topics to a specific collection. For example map a topic called "bloomberg_prices" to a collection called "prices". This mapping is set in the `connect.documentdb.kcql` option. You don't need to set up multiple sinks for each topic or collection. The same sink instance can be configured to handle multiple collections. For example your configuration in this case:

```
connect.documentdb.kcql = INSERT INTO orders SELECT * FROM orders-topic; UPSERT INTO_
↪customers SELECT * FROM customer-topic PK customer_id; UPSERT INTO invoiceid as_
↪invoice_id, customerid as customer_id, value a SELECT invoice_id, FROM invoice-topic
```

Field Selection

The sink supports selecting fields from the source topic or selecting all. There is an option to rename a field as well. All of this can be easily expressed with KCQL:

- Select all fields from topic `fx_prices` and insert into the `fx` collection: `INSERT INTO fx SELECT * FROM fx_prices.`
- Select all fields from topic `fx_prices` and upsert into the `fx` collection, The assumption is there will be a ticker field in the incoming json: `UPSERT INTO fx SELECT * FROM fx_prices PK ticker.`
- Select specific fields from the topic `sample_topic` and insert into the `sample` collection: `INSERT INTO sample SELECT field1,field2,field3 FROM sample_topic.`
- Select specific fields from the topic `sample_topic` and upsert into the `sample` collection: `UPSERT INTO sample SELECT field1,field2,field3 FROM sample_fopic PK field1.`
- Rename some fields while selecting all from the topic `sample_topic` and insert into the `sample` collection: `INSERT INTO sample SELECT *, field1 as new_name1,field2 as new_name2 FROM sample_topic.`
- Rename some fields while selecting all from the topic `sample_topic` and upsert into the `sample` collection: `UPSERT INTO sample SELECT *, field1 as new_name1,field2 as new_name2 FROM sample_topic PK new_name1.`
- Select specific fields and rename some of them from the topic `sample_topic` and insert into the `sample` collection: `INSERT INTO sample SELECT field1 as new_name1,field2, field3 as new_name3 FROM sample_topic.`
- Select specific fields and rename some of them from the topic `sample_topic` and upsert into the `sample` collection: `INSERT INTO sample SELECT field1 as new_name1,field2, field3 as new_name3 FROM sample_fopic PK new_name3.`

Configurations

Configurations parameters:

`connect.documentdb.db`

The Azure DocumentDb target database.

- Data type: string
- Optional : no

`connect.documentdb.endpoint`

The service endpoint to use to create the client.

- Data type: string
- Optional : no

`connect.documentdb.master.key`

The connection master key

- Data type: string
- Optional : no

`connect.documentdb.consistency.level`

Determines the write visibility. There are four possible values: Strong,BoundedStaleness,Session nbyor Eventual

- Data type: string
- Optional : yes
- Default : Session

```
connect.documentdb.db.create
```

If set to true it will create the database if it doesn't exist. If this is set to default(false) an exception will be raised

- Data type: Boolean
- Optional : true
- Default : false

```
connect.documentdb.proxy
```

Specifies the connection proxy details.

- Data type: String
- Optional : yes

```
connect.documentdb.batch.size
```

The number of records the sink would push to DocumentDb at once (improved performance)

- Data type: int
- Optional : yes
- Default: 100

```
connect.documentdb.kcql
```

Kafka connect query language expression. Allows for expressive topic to collectionrouting, field selection and renaming.

Examples:

```
INSERT INTO TABLE1 SELECT * FROM TOPIC1;INSERT INTO TABLE2 SELECT field1, field2, ↵
↵field3 as renamedField FROM TOPIC2
```

- Data Type: string
- Optional : no

```
connect.documentdb.error.policy
```

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **NOOP**, the error is swallowed, **THROW**, the error is allowed to propagate and retry. For **RETRY** the Kafka message is redelivered up to a maximum number of times specified by the `connect.documentdb.max.retires` option. The `connect.documentdb.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Default: throw

```
connect.documentdb.max.retires
```

The maximum number of times a message is retried. Only valid when the `connect.documentdb.error.policy` is set to TRHOW.

- Type: string
- Importance: high
- Default: 10

`connect.documentdb.retry.interval`

The interval, in milliseconds between retries if the sink is using `connect.documentdb.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=azure-docdb-sink
connector.class=com.datamountaineer.streamreactor.connect.azure.documentdb.sink.
↳DocumentDbSinkConnector
tasks.max=1
topics=orders-avro
connect.documentdb.kcql=INSERT INTO orders SELECT * FROM orders-avro
connect.documentdb.db=dm
connect.documentdb.endpoint=[YOUR_AZURE_ENDPOINT]
connect.documentdb.db.create=true
connect.documentdb.master.key=[YOUR_MASTER_KEY]
connect.documentdb.batch.size=10
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.

3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect,` `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Cassandra Sink

The Cassandra Sink allows you to write events from Kafka to Cassandra. The connector converts the value from the Kafka Connect SinkRecords to Json and uses Cassandra's JSON insert functionality to insert the rows. The task expects pre-created tables in Cassandra.

Note: The table and keyspace must be created before hand!

Note: If the target table has TimeUUID fields the payload string the corresponding field in Kafka must be a UUID.

The Sink supports:

1. *The KCQL routing querying* - Kafka topic payload field selection is supported, allowing you to have choose selection of fields or all fields written to Cassandra.
2. Topic to table routing via KCQL.
3. Error policies for handling failures.
4. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema.
5. Optional TTL, time to live on inserts. See Cassandras [documentation](#) for more information.
6. Delete for records in Cassandra for null payloads.

The Sink supports three Kafka payloads type:

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example [here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires the SchemaRegistry which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should be set to `value.converter=org.apache.kafka.connect.json.JsonConverter`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a `GenericRecord`, to just send a message with `Schema.String` and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support

Prerequisites!!

- Cassandra **2.2.4+** if you are on version 2.* or **3.0.1+** if you are on version 3.*
- Confluent 3.3
- Java 1.8
- Scala 2.11

Note: You must be using at least Cassandra 3.0.9 to have JSON support!

Setup

Before we can do anything, including the QuickStart we need to install Cassandra and the Confluent platform.

Cassandra Setup

First download and install Cassandra if you don't have a compatible cluster available.

```
#make a folder for cassandra
mkdir cassandra

#Download Cassandra
wget http://apache.cs.uu.nl/cassandra/3.5/apache-cassandra-3.5-bin.tar.gz

#extract archive to cassandra folder
tar -xvf apache-cassandra-3.5-bin.tar.gz -C cassandra

#Set up environment variables
export CASSANDRA_HOME=~/.cassandra/apache-cassandra-3.5-bin
export PATH=$PATH:$CASSANDRA_HOME/bin

#Start Cassandra
sudo sh ~/.cassandra/bin/cassandra
```

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Test data

The Sink currently expects precreated tables and keyspaces. So let's create a keyspace and table in Cassandra via the CQL shell first.

Once you have installed and started Cassandra create a table to write records to. This snippet creates a table called `orders` to hold fictional orders on a trading platform.

Start the Cassandra cql shell

```
bin ./cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.0.2 | CQL spec 3.3.1 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Execute the following to create the keyspace and table:

```
CREATE KEYSPACE demo WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_
↪factor' : 3};
use demo;

create table orders (id int, created varchar, product varchar, qty int, price float,
↪PRIMARY KEY (id, created))
WITH CLUSTERING ORDER BY (created asc);
```

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for Cassandra. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create cassandra-sink-orders < conf/cassandra-sink.properties

#Connector `cassandra-sink-orders`:
name=cassandra-sink-orders
connector.class=com.datamountaineer.streamreactor.connect.cassandra.sink.
↪CassandraSinkConnector
```

```
tasks.max=1
topics=orders-topic
connect.cassandra.kcql=INSERT INTO orders SELECT * FROM orders-topic
connect.cassandra.port=9042
connect.cassandra.key.space=demo
connect.cassandra.contact.points=localhost
connect.cassandra.username=cassandra
connect.cassandra.password=cassandra
#task ids: 0
```

The `cassandra-sink.properties` file defines:

1. The name of the sink.
2. The Sink class.
3. The max number of tasks the connector is allowed to created (1 task only).
4. The topics to read from.
5. *The KCQL routing querying.*
6. The Cassandra host.
7. The Cassandra port.
8. The Cassandra Keyspace.
9. The username.
10. The password.

Use the Confluent CLI to view Connects logs.

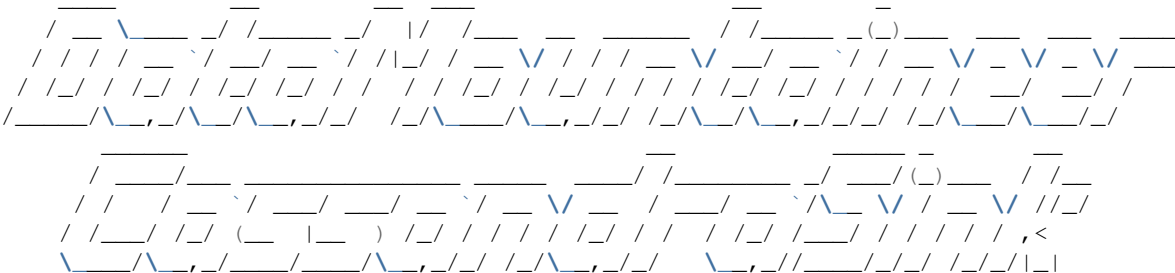
```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
cassandra-sink
```

```
[2016-05-06 13:52:28,178] INFO
```



```
By Andrew Stevenson. (com.datamountaineer.streamreactor.connect.cassandra.sink.  
↳CassandraSinkTask:50)  
[2016-05-06 13:52:28,179] INFO Attempting to connect to Cassandra cluster at_  
↳localhost and create keyspace demo. (com.datamountaineer.streamreactor.connect.  
↳cassandra.CassandraConnection$:49)
```



```
[2016-05-06 13:52:28,187] WARN You listed localhost/0:0:0:0:0:0:1:9042 in your
↳ contact points, but it wasn't found in the control host's system.peers at startup
↳ (com.datastax.driver.core.Cluster:2105)
[2016-05-06 13:52:28,211] INFO Using data-center name 'datacenter1' for
↳ DCAwareRoundRobinPolicy (if this is incorrect, please provide the correct
↳ datacenter name with DCAwareRoundRobinPolicy constructor) (com.datastax.driver.core.
↳ policies.DCAwareRoundRobinPolicy:95)
[2016-05-06 13:52:28,211] INFO New Cassandra host localhost/127.0.0.1:9042 added (com.
↳ datastax.driver.core.Cluster:1475)
[2016-05-06 13:52:28,290] INFO Initialising Cassandra writer. (com.datamountaineer.
↳ streamreactor.connect.cassandra.sink.CassandraJsonWriter:40)
[2016-05-06 13:52:28,295] INFO Preparing statements for orders-topic. (com.
↳ datamountaineer.streamreactor.connect.cassandra.sink.CassandraJsonWriter:62)
[2016-05-06 13:52:28,305] INFO Sink task org.apache.kafka.connect.runtime.
↳ WorkerSinkTask@37e65d57 finished initialization and start (org.apache.kafka.connect.
↳ runtime.WorkerSinkTask:155)
[2016-05-06 13:52:28,331] INFO Source task Thread[WorkerSourceTask-cassandra-source-
↳ orders-0,5,main] finished initialization and start (org.apache.kafka.connect.
↳ runtime.WorkerSourceTask:342)
```

Test Records

Now we need to put some records it to the orders-topic. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema matches the table created earlier.

Hint: If your input topic doesn't match the target use Kafka Streams to transform in realtime the input. Also checkout the [Plumber](#), which allows you to inject a Lua script into [Kafka Streams](#) to do this, no Java or Scala required!

```
${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic orders-topic \
--property value.schema='{ "type": "record", "name": "myrecord", "fields": [{ "name": "id",
↳ "type": "int" }, { "name": "created", "type": "string" }, { "name": "product", "type": "string" },
↳ { "name": "price", "type": "double" }, { "name": "qty", "type": "int" } ] }'
```

Now the producer is waiting for input. Paste in the following (each on a line separately):

```
{ "id": 1, "created": "2016-05-06 13:53:00", "product": "OP-DAX-P-20150201-95.7",
↳ "price": 94.2, "qty": 100 }
{ "id": 2, "created": "2016-05-06 13:54:00", "product": "OP-DAX-C-20150201-100", "price
↳ ": 99.5, "qty": 100 }
{ "id": 3, "created": "2016-05-06 13:55:00", "product": "FU-DATAMOUNTAINEER-20150201-
↳ 100", "price": 10000, "qty": 100 }
{ "id": 4, "created": "2016-05-06 13:56:00", "product": "FU-KOSPI-C-20150201-100",
↳ "price": 150, "qty": 100 }
```

Now if we check the logs of the connector we should see 2 records being inserted to Cassandra:

```
[2016-05-06 13:55:10,368] INFO Setting newly assigned partitions [orders-topic-0] for
↳ group connect-cassandra-sink-1 (org.apache.kafka.clients.consumer.internals.
↳ ConsumerCoordinator:219)
[2016-05-06 13:55:16,423] INFO Received 4 records. (com.datamountaineer.streamreactor.
↳ connect.cassandra.sink.CassandraJsonWriter:96)
```

```
[2016-05-06 13:55:16,484] INFO Processed 4 records. (com.datamountaineer.  
↳streamreactor.connect.cassandra.sink.CassandraJsonWriter:138)
```

```
use demo;  
SELECT * FROM orders;
```

id	created	price	product	qty
1	2016-05-06 13:53:00	94.2	OP-DAX-P-20150201-95.7	100
2	2016-05-06 13:54:00	99.5	OP-DAX-C-20150201-100	100
3	2016-05-06 13:55:00	10000	FU-DATAMOUNTAINEER-20150201-100	500
4	2016-05-06 13:56:00	150	FU-KOSPI-C-20150201-100	200

```
(4 rows)
```

Bingo, our 4 rows!

Features

The Sink connector uses Cassandra's [JSON](#) insert functionality. The SinkRecord from Kafka Connect is converted to JSON and feed into the prepared statements for inserting into Cassandra.

See Cassandra's [documentation](#) for type mapping.

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The Cassandra Sink supports the following:

```
INSERT INTO <target table> SELECT <fields> FROM <source topic> TTL=<TTL>
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA  
INSERT INTO tableA SELECT * FROM topicA  
  
#Insert mode, select 3 fields and rename from topicB and write to tableB  
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB  
  
#Insert mode, select 3 fields and rename from topicB and write to tableB with TTL  
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB TTL=100000
```

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other expectations thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.cassandra.max.retries` and the `connect.cassandra.retry.interval`.

Topic Routing

The Sink supports topic routing that allows mapping the messages from topics to a specific table. For example map a topic called "bloomberg_prices" to a table called "prices". This mapping is set in the `connect.cassandra.kcql` option.

Field Selection

The Sink supports selecting fields from the Source topic or selecting all fields and mapping of these fields to columns in the target table. For example, map a field called "qty" in a topic to a column called "quantity" in the target table.

All fields can be selected by using "*" in the field part of `connect.cassandra.kcql`.

Leaving the column name empty means trying to map to a column in the target table with the same name as the field in the source topic.

Deletion in Cassandra

Compacted topics in Kafka retain the last message per key. Deletion in Kafka occurs by tombstoning. If compaction is enabled on the topic and a message is sent with a null payload, Kafka flags this record for delete and is compacted/removed from the topic. For more information on compaction see [this](#).

The use case for this delete functionality would be, for example, when the source topic is a compacted topic, perhaps capturing data changes from an upstream source such as a CDC connector. Let's say a record is deleted from the upstream source and that delete operation is propagated to the kafka topic, with the key of the kafka message as the PK of the record in the targeted cassandra table - meaning the value of the kafka message is now null. This feature allows you to delete these records in Cassandra.

This functionality will be migrated to [KCQL](#) in future releases.

Deletion in Cassandra is supported based on fields in the key of messages with a empty/null payload. Deletion is enabled by settings the `connect.delete.enabled` option. A Cassandra delete statement must be provided, `connect.delete.statement` which specifies the Cassandra CQL delete statement with parameters to bind field values from the key to, for example, with the delete statement of:

```
DELETE FROM orders WHERE id = ? and product = ?
```

If a message was received with a empty/null value and key fields `key.id` and `key.product` the final bound Cassandra statement would be:

```
# connect.delete.enabled=true
# connect.delete.statement=DELETE FROM orders WHERE id = ? and product = ?
# connect.delete.struct_flds=id,product

# "{ \"key\": { \"id\" : 999, \"product\" : \"DATAMOUNTAINEER\" }, \"value\" : null }"
DELETE FROM orders WHERE id = 999 and product = \"DATAMOUNTAINEER\"
```

Note: Deletion will only occur if a message with an empty payload is recieved from Kafka.

Important: Ensure your ordinal position of the `connect.delete.struct_flds` matches the bind order in the Cassandra delete statement!

Legacy topics (plain text payload with a json string)

We have found some of the clients have already an infrastructure where they publish pure json on the topic and obviously the jump to follow the best practice and use schema registry is quite an ask. So we offer support for them as well.

This time we need to start the connect with a different set of settings.

```
#create a new configuration for connect
cp etc/schema-registry/connect-avro-distributed.properties etc/schema-registry/
↪connect-distributed-json.properties
vi etc/schema-registry/connect-distributed-json.properties
```

Replace the following 4 entries in the config

```
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081
```

with the following

```
key.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=false
value.converter=org.apache.kafka.connect.json.JsonConverter
value.converter.schemas.enable=false
```

Now let's restart the connect instance: .. sourcecode:: bash

```
#start a new instance of connect $CONFLUENT_HOME/bin/connect-distributed etc/schema-
registry/connect-distributed-json.properties
```

Configurations

Configurations common to both Sink and Source are:

`connect.cassandra.contact.points`

Contact points (hosts) in Cassandra cluster. This is a comma separated value. i.e: host-1, host-2

- Data type: string
- Optional : no

`connect.cassandra.key.space`

Key space the tables to write belong to.

- Data type: string
- Optional : no

`connect.cassandra.port`

Port for the native Java driver.

- Data type: int
- Optional : yes
- Default : 9042

`connect.cassandra.username`

Username to connect to Cassandra with.

- Data type: string
- Optional : yes

`connect.cassandra.password`

Password to connect to Cassandra with.

- Data type: string
- Optional : yes

`connect.cassandra.ssl.enabled`

Enables SSL communication against SSL enable Cassandra cluster.

- Data type: boolean
- Optional : yes
- Default : false

`connect.cassandra.trust.store.password`

Password for truststore.

- Data type: string
- Optional : yes

`connect.cassandra.key.store.path`

Path to truststore.

- Data type: string
- Optional : yes

`connect.cassandra.key.store.password`

Password for key store.

- Data type: string
- Optional : yes

`connect.cassandra.ssl.client.cert.auth`

Path to keystore.

- Data type: string
- Optional : yes

`connect.cassandra.kcql`

Kafka connect query language expression. Allows for expressive topic to table routing, field selection and renaming.

Examples:

```
INSERT INTO TABLE1 SELECT * FROM TOPIC1;INSERT INTO TABLE2 SELECT field1, field2, ↵
↵field3 as renamedField FROM TOPIC2
```

- Data Type: string
- Optional : no

`connect.cassandra.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.cassandra.max.retries` option. The `connect.cassandra.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Default: throw

`connect.cassandra.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.cassandra.error.policy` is set to `retry`.

- Type: string
- Importance: high
- Default: 10

`connect.cassandra.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.cassandra.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

`connect.delete.enabled`

Enables row deletion from cassandra.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

`connect.delete.statement`

Delete statement for cassandra. Required if `connect.delete.enabled` is set.

- Type: string
- Importance: medium
- Optional: yes
- Default :

`connect.delete.struct_flds`

Fields in the key struct data type used in there delete statement. Comma-separated in the order they are found in `connect.delete.statement`

- Type: string
- Importance: medium
- Optional: yes
- Default :

Example

```
name=cassandra-sink-orders
connector.class=com.datamountaineer.streamreactor.connect.cassandra.sink.
↳CassandraSinkConnector
tasks.max=1
topics=orders-topic
connect.cassandra.kcql = INSERT INTO TABLE1 SELECT * FROM TOPIC1;INSERT INTO TABLE2_
↳SELECT field1,
field2, field3 as renamedField FROM TOPIC2
connect.cassandra.contact.points=localhost
connect.cassandra.port=9042
connect.cassandra.key.space=demo
connect.cassandra.contact.points=localhost
connect.cassandra.username=cassandra
connect.cassandra.password=cassandra
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

For the Sink connector, if columns are added to the target Cassandra table and not present in the Source topic they will be set to null by Cassandra's JSON insert functionality. Columns which are omitted from the JSON value map are treated as a null insert (which results in an existing value being deleted, if one is present), if a record with the same key is inserted again.

Future releases will support auto creation of tables and adding columns on changes to the topic schema.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be pushed to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has its own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

Troubleshooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect CoAP Sink

A Connector and Sink to stream messages from Kafka to a CoAP server.

The Sink supports:

1. DTLS secure clients.
2. *The KCQL routing querying* - Topic to measure mapping and Field selection.
3. Schema registry support for Connect/Avro with a schema.
4. Schema registry support for Connect and no schema (schema set to Schema.String)
5. Json payload support, no Schema Registry.
6. Error policies.
7. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema

The Sink supports three Kafka payloads type:

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example [here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires the SchemaRegistry which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should be set to `value.converter=org.apache.kafka.connect.json.JsonConverter`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a GenericRecord, to just send a message with Schema.String and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support

The payload of the CoAP request sent to the CoAP server is sent as json.

Prerequisites

- Confluent 3.3
- Java 1.8
- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

CoAP Setup

The connector uses [Californium](#) Java API under the hood. Copper, a [FireFox](#) browser addon is available so you can browse the server and resources.

We will use a simple CoAP test server we have developed for testing. Download the CoAP test server from our github release page and start the server in a new terminal tab.

```
mkdir coap_server
cd coap_server
wget https://github.com/datamountaineer/coap-test-server/releases/download/v1.0/start-
server.sh
chmod +x start-server.sh
./start-server.sh
```

You will see the server start listening on port 5864 for secure DTLS connections and on port 5633 for insecure connections.

```
m.DTLSTLSConnector$Worker.java:-1) run() in thread DTLS-Receiver-0.0.0.0/0.0.0.0:5634 at
(2017-01-10 15:41:08)
1 INFO [CoapEndpoint]: Starting endpoint at localhost/127.0.0.1:5633 - (org.eclipse.
californium.core.network.CoapEndpoint.java:192) start() in thread main at (2017-01-
10 15:41:08)
1 CONFIG [UDPConnector]: UDPConnector starts up 1 sender threads and 1 receiver
threads - (org.eclipse.californium.elements.UDPConnector.java:261) start() in
thread main at (2017-01-10 15:41:08)
1 CONFIG [UDPConnector]: UDPConnector listening on /127.0.0.1:5633, recv buf = 65507,
send buf = 65507, recv packet size = 2048 - (org.eclipse.californium.elements.
UDPConnector.java:261) start() in thread main at (2017-01-10 15:41:08)
Secure CoAP server powered by Scandium (Sc) is listening on port 5634
UnSecure CoAP server powered by Scandium (Sc) is listening on port 5633
```

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for MQTT. If you are using the `dockers` you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create coap-source < conf/coap-source.properties

#Connector name=`coap-sink`
name = coap-sink
tasks = 1
connector.class = com.datamountaineer.streamreactor.connect.coap.sink.
CoapSinkConnector
connect.coap.uri = coap://localhost:5683
connect.coap.kcql = INSERT INTO unsecure SELECT * FROM coap_topic
```

```
topics = coap_topic
#task ids: 0
```

The `coap-source.properties` file defines:

1. The name of the sink.
2. The name number of tasks.
3. The class containing the connector.
4. The uri of the CoAP Server and port to connect to.
5. *The KCQL routing querying..* This specifies the target resources on the CoAP server and the source topic.
6. The topics to source (Required by Connect Framework).

Use the Confluent CLI to view Connects logs.

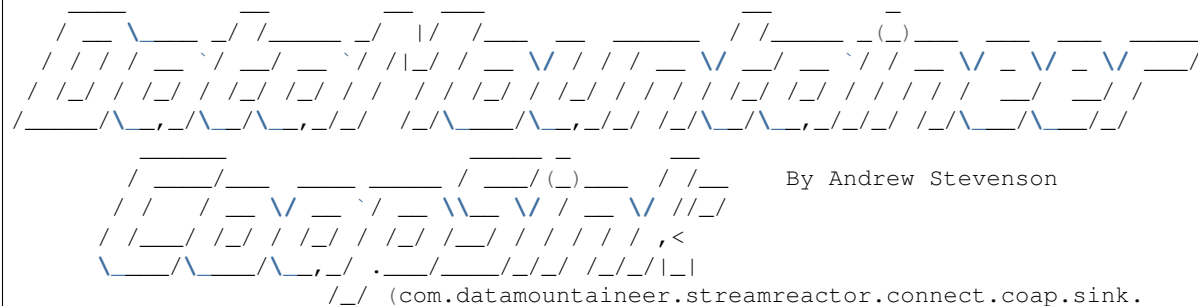
```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
coap-sink
```

INFO



By Andrew Stevenson

```

      /_/_ (com.datamountaineer.streamreactor.connect.coap.sink.
↪CoapSinkTask:52)
[2017-01-10 12:57:32,238] INFO CoapSinkConfig values:
  connect.coap.uri = coap://localhost:5683
  connect.coap.port = 0
  connect.coap.retry.interval = 60000
  connect.coap.truststore.pass = [hidden]
  connect.coap.cert.chain.key = client
  connect.coap.error.policy = THROW
  connect.coap.kcql = INSERT INTO unsecure SELECT * FROM coap_topic
  connect.coap.host = localhost
  connect.coap.certs = []
  connect.coap.max.retires = 20
  connect.coap.keystore.path =
  connect.coap.truststore.path =
  connect.coap.keystore.pass = [hidden]
  (com.datamountaineer.streamreactor.connect.coap.configs.CoapSinkConfig:178)
```

Test Records

Now we need to put some records it to the `coap_topic` topics. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a `firstname` field of type string, a `lastname` field of type string, an `age` field of type int and a `salary` field of type double.

```
${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
  --broker-list localhost:9092 --topic coap-topic \
  --property value.schema='{ "type": "record", "name": "User",
    "fields": [{ "name": "firstName", "type": "string"}, { "name": "lastName", "type": "string"}, {
    ↪ "name": "age", "type": "int"}, { "name": "salary", "type": "double"}] }'
```

Now the producer is waiting for input. Paste in the following:

```
{"firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830}
```

Check for Records in the CoAP server via Copper

Now check the logs of the connector you should see this:

```
[2017-01-10 13:47:36,525] INFO Delivered 1 records for coap-topic. (com.
  ↪ datamountaineer.streamreactor.connect.coap.sink.CoapSinkTask:47)
```

In Firefox go the following url. If you have not installed Copper do so [here](#).

```
coap://127.0.0.1:5633/insecure
```

Hit the get button and the records will be displayed in the bottom panel.

The screenshot shows the Copper web interface for a CoAP server at 127.0.0.1:5633. The interface is divided into several sections:

- Top Bar:** Shows the URL `coap://127.0.0.1:5633/insecure` and various navigation icons.
- Message List:** A table showing the received messages. The first message is highlighted:

Time	CoAP Message	MID	Token	Options	Payload
14:36:10	NON-GET	199	empty	Uri-Path: insecure, Block2: 0/0/512	
14:36:10	NON-2.05 Content	29712	empty	Content-Format: 0, Max-Age: 2, Block2: 0/0/512	<code>{"firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830.0}</code>
- Message Detail View:** A detailed view of the selected message (2.05 Content). It shows:
 - Header:** Type: NON, Code: 2.05 Content, MID: 29712, Token: empty.
 - Option:** Content-Format: text/plain (Info: 0), Max-Age: 2 (Info: 1 byte), Block2: 0 (512 B/block) (Info: 1 byte).
 - Payload (64):** `{"firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830.0}`
- Debug Control:** A panel on the right with a 'Reset' button and a 'Token' input field.

Features

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The CoAP Sink supports the following:

```
INSERT INTO <resource> SELECT <fields> FROM <source topic>
```

Example:

```
#Insert mode, select all fields from topicA and write to resourceA
INSERT INTO resourceA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to resourceA
INSERT INTO resourceA SELECT x AS a, y AS b and z AS c FROM topicB
```

This is set in the `connect.coap.kcql` option.

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers.

Retry

Any error on write to the target database causes the `RetryableException` to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.influx.max.retries` and the `connect.coap.retry.interval`.

DTLS Secure connections

The Connector use the [Californium](#) Java API and for secure connections use the Scandium security module provided by Californium. Scandium (Sc) is an implementation of Datagram Transport Layer Security 1.2, also known as [RFC 6347](#).

Please refer to the Californium [certification](#) repo page for more information.

The connector supports:

1. SSL trust and key stores
2. Public/Private PEM keys and PSK client/identity
3. PSK Client Identity

The Sink will attempt secure connections in the following order if the URI schema of `connect.coap.uri` set to secure, i.e. “coaps”. If `connect.coap.username` is set PSK client identity authentication is used, if additional `connect.coap.private.key.path` Public/Private keys authentication will also be attempt. Otherwise SSL trust and key store.

```
`openssl pkcs8 -in privatekey.pem -topk8 -nocrypt -out privatekey-pkcs8.pem`
```

Only cipher suites `TLS_PSK_WITH_AES_128_CCM_8` and `TLS_PSK_WITH_AES_128_CBC_SHA256` are ↪ currently supported.

Warning: The keystore, truststore, public and private files must be available on the local disk of the worker task.

Loading specific certificates can be achieved by providing a comma separated list for the `connect.coap.certs` configuration option. The certificate chain can be set by the `connect.coap.cert.chain.key` configuration option.

Configurations

`connect.coap.uri`

Uri of the CoAP server.

- Data Type : string
- Importance: high
- Optional : no

`connect.coap.kcql`

The KCQL statement to select and route resources to topics.

- Data Type : string
- Importance: high
- Optional : no

`connect.coap.port`

The port the DTLS connector will bind to on the Connector host.

- Data Type : int
- Importance: medium
- Optional : yes
- Default : 0

`connect.coap.host`

The hostname the DTLS connector will bind to on the Connector host.

- Data Type : string
- Importance: medium
- Optional : yes
- Default : localhost

`connect.coap.username`

CoAP PSK identity.

- Data Type : string
- Importance: medium
- Optional : yes

`connect.coap.password`

CoAP PSK secret.

- Data Type : password
- Importance: medium
- Optional : yes

`connect.coap.public.key.file`

Path to the public key file for use in with PSK credentials.

- Data Type : string
- Importance: medium
- Optional : yes

`connect.coap.private.key.file`

Path to the private key file for use in with PSK credentials in PKCS8 rather than PKCS1 Use open SSL to convert.

```
`openssl pkcs8 -in privatekey.pem -topk8 -nocrypt -out privatekey-pkcs8.pem`
```

Only cipher suites TLS_PSK_WITH_AES_128_CCM_8 and TLS_PSK_WITH_AES_128_CBC_SHA256 are currently supported.

- Data Type : string
- Importance: medium
- Optional : yes

`connect.coap.keystore.pass`

The password of the key store

- Data Type : password
- Importance: medium
- Optional : yes
- Default : rootPass

`connect.coap.keystore.path`

The path to the keystore.

- Data Type : string
- Importance: medium
- Optional : yes
- Default :

`connect.coap.truststore.pass`

The password of the trust store

- Data Type : password
- Importance: medium
- Optional : yes
- Default : rootPass

`connect.coap.truststore.path`

The path to the truststore.

- Data Type : string
- Importance: medium
- Optional : yes
- Default :

`connect.coap.certs`

The certificates to load from the trust store.

- Data Type : list
- Importance: medium
- Optional : yes
- Default :

`connect.coap.cert.chain.key`

The key to use to get the certificate chain.

- Data Type : string
- Importance: medium
- Optional : yes
- Default : client

`connect.coap.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.coap.max.retries` option. The `connect.coap.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: medium
- Optional: yes

- Default: RETRY

`connect.coap.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.coap.error.policy` is set to `retry`.

- Type: string
- Importance: high
- Optional: yes
- Default: 10

`connect.coap.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.coap.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional: yes
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Elastic

A Connector and Sink to write events from Kafka to Elastic Search using [Elastic4s](#) client. The connector converts the value from the Kafka Connect SinkRecords to Json and uses Elastic4s's JSON insert functionality to index.

The Sink creates an Index and Type corresponding to the topic name and uses the JSON insert functionality from Elastic4s.

The Sink supports:

1. Auto index creation at start up.
2. *The KCQL routing querying* - Topic to index mapping and Field selection.
3. Auto mapping of the Kafka topic schema to the index.
4. Tagging indexes with a document type.
5. Adding index name suffixes with a data pattern.
6. Upsert and insert records.

Prerequisites

- Confluent 3.3
- Elastic Search 2.2
- Java 1.8
- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

Elastic Setup

Download and start Elastic search.

```
curl -L -O https://download.elastic.co/elasticsearch/release/org/elasticsearch/
↪distribution/tar/elasticsearch/2.2.0/elasticsearch-2.2.0.tar.gz
tar -xvf elasticsearch-2.2.0.tar.gz
cd elasticsearch-2.2.0/bin
./elasticsearch --cluster.name elasticsearch
```

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for Elastic. If you are using the *dockers* you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create elastic-sink < conf/elastic-sink.properties

#Connector name=`elastic-sink`
connect.progress.enabled=true
name=elastic-sink
connector.class=com.datamountaineer.streamreactor.connect.elastic.ElasticSinkConnector
connect.elastic.url=localhost:9300
connect.elastic.cluster.name=elasticsearch
tasks.max=1
topics=orders-topic
connect.elastic.kcql=INSERT INTO index_1 SELECT * FROM orders-topic
#task ids: 0
```

The `elastic-sink.properties` file defines:

1. The name of the connector.
2. The class containing the connector.
3. The name of the cluster on the Elastic Search server to connect to.
4. The max number of task allowed for this connector.
5. The Source topic to get records from.
6. *The KCQL routing querying.*

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
elastic-sink
```

```
[2016-05-08 20:56:52,241] INFO
```



```
by Andrew Stevenson
    (com.datamountaineer.streamreactor.connect.elastic.ElasticSinkTask:33)
```

```
[2016-05-08 20:56:52,327] INFO [Hebe] loaded [], sites [] (org.elasticsearch.
↳ plugins:149)
[2016-05-08 20:56:52,765] INFO Initialising Elastic Json writer (com.datamountaineer.
↳ streamreactor.connect.elastic.ElasticJsonWriter:31)
[2016-05-08 20:56:52,777] INFO Assigned List(test_table) topics. (com.datamountaineer.
↳ streamreactor.connect.elastic.ElasticJsonWriter:33)
[2016-05-08 20:56:52,836] INFO Sink task org.apache.kafka.connect.runtime.
↳ WorkerSinkTask@69b6b39 finished initialization and start (org.apache.kafka.connect.
↳ runtime.WorkerSinkTask:155)
```

Test Records

Now we need to put some records it to the `test_table` topics. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a `id` field of type `int` and a `random_field` of type `string`.

```

${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic orders-topic \
--property value.schema='{"type": "record", "name": "myrecord", "fields": [{"name": "id",
↪ "type": "int"},
{"name": "random_field", "type": "string"}]}'

```

Now the producer is waiting for input. Paste in the following:

```
{ "id": 999, "random_field": "foo" }
{ "id": 888, "random_field": "bar" }
```

Check for records in Elastic Search

Now if we check the logs of the connector we should see 2 records being inserted to Elastic Search:

```
[2016-05-08 21:02:52,095] INFO Flushing Elastic Sink (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticSinkTask:73)
[2016-05-08 21:03:52,097] INFO No records received. (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticJsonWriter:63)
[2016-05-08 21:03:52,097] INFO org.apache.kafka.connect.runtime.
↳WorkerSinkTask@69b6b39 Committing offsets (org.apache.kafka.connect.runtime.
↳WorkerSinkTask:187)
[2016-05-08 21:03:52,097] INFO Flushing Elastic Sink (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticSinkTask:73)
[2016-05-08 21:04:20,613] INFO Elastic write successful for 2 records! (com.
↳datamountaineer.streamreactor.connect.elastic.ElasticJsonWriter:77)
```

If we query Elastic Search for id 999:

```
curl -XGET 'http://localhost:9200/INDEX_1/_search?q=id:999'

{
  "took": 45,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1.2231436,
    "hits": [{
      "_index": "INDEX_1",
      "_type": "INDEX_1",
      "_id": "AVMY4eZXFGuf2uMZyxjU",
      "_score": 1.2231436,
      "_source": {
        "id": 999,
        "random_field": "foo"
      }
    }]
  }
}
```

Features

1. Auto index creation at start up.
2. Topic to index mapping.
3. Auto mapping of the Kafka topic schema to the index.
4. Field selection

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The Elastic Sink supports insert/upsert, setting the document type and setting suffixes on the index with a data format:

```
#INSERT
INSERT INTO <index> SELECT <fields> FROM <source topic>
[WITHDOCTYPE=<your_document_type>]
[WITHINDEXSUFFIX=<your_suffix>]
[PK field]

#UPSERT
UPSERT INTO <index> SELECT <fields> FROM <source topic> [PK field]
```

WITHDOCTYPE allows you to associate a document type to the document inserted. *WITHINDEXSUFFIX* allows you to specify a suffix to your index and we support date format. All you have to say is ‘_suffix_{YYYY-MM-dd}’

Example:

```
#Insert mode, select all fields from topicA and write to indexA
INSERT INTO indexA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to indexB
INSERT INTO indexB SELECT x AS a, y AS b and z AS c FROM topicB PK y

# UPSERT
UPSERT INTO indexC SELECT id, string_field FROM topicC PK id
```

This is set in the `connect.elastic.kcql` option.

Auto Index Creation

The Sink will automatically create missing indexes at startup. The Sink use elastic4s, more details can be found [here](#)

Configurations

`connect.elastic.url`

Url of the Elastic cluster.

- Data Type : string
- Importance: high
- Optional : no

`connect.elastic.kcql`

Kafka connect query language expression. Allows for expressive table to topic routing, field selection and renaming.

Examples:

```
INSERT INTO INDEX_1 SELECT field1, field2 FROM TOPIC1
```

- Data type : string
- Importance: high

- Optional : no

`connect.elastic.write.timeout`

Specifies the wait time for pushing the records to ES.

- Data type : long
- Importance: low
- Optional : yes
- Default : 300000 (5mins)

`connect.elastic.throw.on.error`

Throws the exception on write failure. Default is 'true'

- Data type : long
- Importance: low
- Optional : yes
- Default: : true

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=elastic-sink
connector.class=com.datamountaineer.streamreactor.connect.elastic.ElasticSinkConnector
connect.elastic.url=localhost:9300
connect.elastic.cluster.name=elasticsearch
tasks.max=1
topics=test_table
connect.elastic.kcql=INSERT INTO INDEX_1 SELECT field1, field2 FROM TOPIC1
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

Elastic Search is very flexible about what is inserted. All documents in Elasticsearch are stored in an index. We do not need to tell Elasticsearch in advance what an index will look like (eg what fields it will contain) as Elasticsearch will adapt the index dynamically as more documents are added, but we must at least create the index first. The Sink connector automatically creates the index at start up if it doesn't exist.

The Elastic Search Sink will automatically index if new fields are added to the Source topic, if fields are removed the Kafka Connect framework will return the default value for this field, dependent of the compatibility settings of the Schema registry.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Elastic 5

A Connector and Sink to write events from Kafka to Elastic Search using [Elastic4s](#) client. The connector converts the value from the Kafka Connect SinkRecords to Json and uses Elastic4s's JSON insert functionality to index.

The Sink creates an Index and Type corresponding to the topic name and uses the JSON insert functionality from Elastic4s.

The Sink supports:

1. Auto index creation at start up.
2. *The KCQL routing querying* - Topic to index mapping and Field selection.
3. Auto mapping of the Kafka topic schema to the index.
4. Tagging indexes with a document type.

5. Adding index name suffixes with a data pattern.
6. Upsert and insert records.

Prerequisites

- Confluent 3.3
- Elastic Search 5.4
- Java 1.8
- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

Elastic Setup

Download and start Elastic search.

```
curl -L -O https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-5.4.0.
tar.gz
tar -xvf elasticsearch-5.4.0.tar.gz
cd elasticsearch-5.4.0/bin
./elasticsearch --cluster.name elasticsearch
```

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for Elastic. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create elastic-sink < conf/elastic-sink.properties

#Connector name='elastic-sink'
connect.progress.enabled=true
name=elastic-sink
connector.class=com.datamountaineer.streamreactor.connect.elastic5.
↳ElasticSinkConnector
connect.elastic.url=localhost:9300
connect.elastic.cluster.name=elasticsearch
tasks.max=1
topics=orders-topic
connect.elastic.kcql=INSERT INTO index_1 SELECT * FROM orders-topic
#task ids: 0
```

The `elastic-sink.properties` file defines:

1. The name of the connector.
2. The class containing the connector.
3. The name of the cluster on the Elastic Search server to connect to.
4. The max number of task allowed for this connector.
5. The Source topic to get records from.
6. *The KCQL routing querying.*

Use the Confluent CLI to view Connects logs.

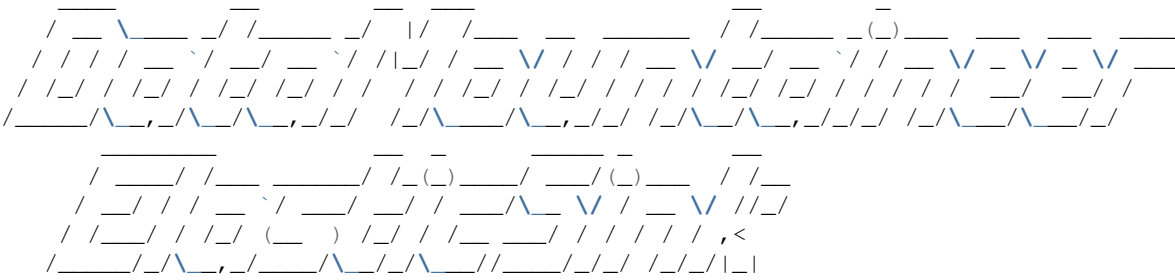
```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
elastic-sink
```

```
[2016-05-08 20:56:52,241] INFO
```



```
by Andrew Stevenson
    (com.datamountaineer.streamreactor.connect.elastic.ElasticSinkTask:33)
```

```
[2016-05-08 20:56:52,327] INFO [Hebe] loaded [], sites [] (org.elasticsearch.
plugins:149)
```

```
[2016-05-08 20:56:52,765] INFO Initialising Elastic Json writer (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticJsonWriter:31)
[2016-05-08 20:56:52,777] INFO Assigned List(test_table) topics. (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticJsonWriter:33)
[2016-05-08 20:56:52,836] INFO Sink task org.apache.kafka.connect.runtime.
↳WorkerSinkTask@69b6b39 finished initialization and start (org.apache.kafka.connect.
↳runtime.WorkerSinkTask:155)
```

Test Records

Now we need to put some records it to the test_table topics. We can use the kafka-avro-console-producer to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a id field of type int and a random_field of type string.

```
${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic orders-topic \
--property value.schema='{ "type": "record", "name": "myrecord", "fields": [{ "name": "id",
↳"type": "int"},
{"name": "random_field", "type": "string"}] }'
```

Now the producer is waiting for input. Paste in the following:

```
{ "id": 999, "random_field": "foo" }
{ "id": 888, "random_field": "bar" }
```

Check for records in Elastic Search

Now if we check the logs of the connector we should see 2 records being inserted to Elastic Search:

```
[2016-05-08 21:02:52,095] INFO Flushing Elastic Sink (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticSinkTask:73)
[2016-05-08 21:03:52,097] INFO No records received. (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticJsonWriter:63)
[2016-05-08 21:03:52,097] INFO org.apache.kafka.connect.runtime.
↳WorkerSinkTask@69b6b39 Committing offsets (org.apache.kafka.connect.runtime.
↳WorkerSinkTask:187)
[2016-05-08 21:03:52,097] INFO Flushing Elastic Sink (com.datamountaineer.
↳streamreactor.connect.elastic.ElasticSinkTask:73)
[2016-05-08 21:04:20,613] INFO Elastic write successful for 2 records! (com.
↳datamountaineer.streamreactor.connect.elastic.ElasticJsonWriter:77)
```

If we query Elastic Search for id 999:

```
curl -XGET 'http://localhost:9200/INDEX_1/_search?q=id:999'

{
  "took": 45,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  }
}
```

```
    },
    "hits": {
      "total": 1,
      "max_score": 1.2231436,
      "hits": [{
        "_index": "INDEX_1",
        "_type": "INDEX_1",
        "_id": "AVMY4eZXFguf2uMZyxjU",
        "_score": 1.2231436,
        "_source": {
          "id": 999,
          "random_field": "foo"
        }
      }]
    }
  }
}
```

Features

1. Auto index creation at start up.
2. Topic to index mapping.
3. Auto mapping of the Kafka topic schema to the index.
4. Field selection

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The Elastic Sink supports insert/upsert, setting the document type and setting suffixes on the index with a data format:

```
INSERT INTO <index> SELECT <fields> FROM <source topic>
[WITHDOCTYPE=<your_document_type>]
[WITHINDEXSUFFIX=<your_suffix>]
[PK field]

#UPSERT
UPSERT INTO <index> SELECT <fields> FROM <source topic> [PK field]
```

WITHDOCTYPE allows you to associate a document type to the document inserted. *WITHINDEXSUFFIX* allows you to specify a suffix to your index and we support date format. All you have to say is ‘_suffix_{YYYY-MM-dd}’

Example:

```
#Insert mode, select all fields from topicA and write to indexA
INSERT INTO indexA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to indexB
INSERT INTO indexB SELECT x AS a, y AS b and z AS c FROM topicB PK y

# UPSERT
UPSERT INTO indexC SELECT id, string_field FROM topicC PK id
```

This is set in the `connect.elastic.kcql` option.

Auto Index Creation

The Sink will automatically create missing indexes at startup. The Sink use elastic4s, more details can be found [here](#)

Configurations

`connect.elastic.url`

Url of the Elastic cluster.

- Data Type : string
- Importance: high
- Optional : no

`connect.elastic.kcql`

Kafka connect query language expression. Allows for expressive table to topic routing, field selection and renaming.

Examples:

```
INSERT INTO INDEX_1 SELECT field1, field2 FROM TOPIC1
```

- Data type : string
- Importance: high
- Optional : no

`connect.elastic.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.elastic.max.retries` option. The `connect.elastic.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Default: throw

`connect.elastic.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.elastic.error.policy` is set to `retry`.

- Type: string
- Importance: high
- Default: 10

`connect.elastic.xpack.settings`

Enables secure connection. [here](#). By providing a value for the entry the sink will end up creating a secure connection. The entry is a ; separated list of key=value sequence

Example:

```
connect.elastic.xpack.settings=xpack.security.user=transport_client_user:changeme;  
↪ xpack.ssl.key=/path/to/client.key;xpack.ssl.certificate=/path/to/client.crt
```

- Type: string
- Importance: medium
- Default: null
- Optional: yes

`connect.elastic.xpack.plugins`

Provides the list of plugins to enable. The entry is a ; separated list of full class path (The classes need to derive from *org.elasticsearch.plugins.Plugin*)

- Type: string
- Importance: medium
- Default: null
- Optional: yes

`connect.elastic.write.timeout`

Specifies the wait time for pushing the records to ES.

- Data type : long
- Importance: low
- Optional : yes
- Default : 300000 (5mins)

`connect.elastic.url.prefix`

URL connection string prefix.

- Data type : string
- Importance: low
- Optional : yes
- Default : elasticsearch

`connect.elastic.cluster.name`

Name of the elastic search cluster, used in local mode for setting the connection

- Data type : string
- Importance: low
- Optional : yes
- Default : elasticsearch

`connect.elastic.use.http`

TCP or HTTP. Elastic4s client type to use, http or tcp, default is tcp.

Note: The HTTP Client is not support with Xpack.

- Data type : string

- Importance: low
- Optional : yes
- Default : TCP

`connect.elastic.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.elastic.max.retries` option. The `connect.elastic.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Optional : yes
- Default: RETRY

`connect.elastic.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.elastic.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional : yes
- Default: 10

`connect.elastic.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.elastic.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional : yes
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=elastic-sink
connector.class=com.datamountaineer.streamreactor.connect.elastic5.
↳ElasticSinkConnector
connect.elastic.url=localhost:9300
connect.elastic.cluster.name=elasticsearch
tasks.max=1
topics=test_table
connect.elastic.kcql=INSERT INTO INDEX_1 SELECT field1, field2 FROM TOPIC1
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

Elastic Search is very flexible about what is inserted. All documents in Elasticsearch are stored in an index. We do not need to tell Elasticsearch in advance what an index will look like (eg what fields it will contain) as Elasticsearch will adapt the index dynamically as more documents are added, but we must at least create the index first. The Sink connector automatically creates the index at start up if it doesn't exist.

The Elastic Search Sink will automatically index if new fields are added to the Source topic, if fields are removed the Kafka Connect framework will return the default value for this field, dependent of the compatibility settings of the Schema registry.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```


Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect HazelCast

A Connector and Sink to write events from Kafka to HazelCast. The connector takes the value from the Kafka Connect SinkRecords and inserts/update an entry in HazelCast. The Sink supports writing to a reliable topic, ring buffer, queue, set, list, imap, multi-map and icache.

The Sink supports:

1. *The [KCQL routing querying](#)* - Kafka topic payload field selection is supported, allowing you to have choose selection of fields or all fields written to Hazelcast.
2. Topic to table routing via KCQL.
3. Error policies for handling failures.
4. Encoding as JSON, TEXT or Avro in Hazelcast via KCQL.
5. Storing in a Hazelcast RELIABLE_TOPIC, RING_BUFFER, QUEUE, SET, LIST, IMAP, MULTI_MAP, ICACHE via KCQL.

Prerequisites

- Confluent 3.3
- Hazelcast 3.6.4 or higher
- Java 1.8
- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

HazelCast Setup

Download and install HazelCast from [here](#)

When you download and extract the Hazelcast ZIP or TAR.GZ package, you will see 3 scripts under the `/bin` folder which provide basic functionality for member and cluster management.

The following are the names and descriptions of each script:

- `start.sh` - Starts a Hazelcast member with default configuration in the working directory.
- `stop.sh` - Stops the Hazelcast member that was started in the current working directory.

Start HazelCast:

```
bin/start.sh

INFO: [10.128.137.102]:5701 [dev] [3.6.4] Address[10.128.137.102]:5701 is STARTING
Aug 16, 2016 2:43:04 PM com.hazelcast.nio.tcp.nonblocking.NonBlockingIOThreadingModel
INFO: [10.128.137.102]:5701 [dev] [3.6.4] TcpIpConnectionManager configured with Non-
Blocking IO-threading model: 3 input threads and 3 output threads
Aug 16, 2016 2:43:07 PM com.hazelcast.cluster.impl.MulticastJoiner
INFO: [10.128.137.102]:5701 [dev] [3.6.4]

Members [1] {
  Member [10.128.137.102]:5701 this
}

Aug 16, 2016 2:43:07 PM com.hazelcast.core.LifecycleService
INFO: [10.128.137.102]:5701 [dev] [3.6.4] Address[10.128.137.102]:5701 is STARTED
```

This will start Hazelcast with a default group called *dev* and password *dev-pass*

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for HazelCast. If you are using the *dockers* you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create hazelcast-sink < conf/hazelcast-sink.properties

#Connector name=`hazelcast-sink`
name=hazelcast-sink
```

[illegible]

Test Records

Now we need to put some records it to the `test_table` topics. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a `firstname` field of type `string` a `lastname` field of type `string`, an `age` field of type `int` and a `salary` field of type `double`.

```

${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic hazelcast-topic \
--property value.schema='{ "type": "record", "name": "User",
  "fields": [{ "name": "firstName", "type": "string" }, { "name": "lastName", "type": "string" }, {
  "name": "age", "type": "int" }, { "name": "salary", "type": "double" } ] }'

```

Now the producer is waiting for input. Paste in the following:

```
{"firstName": "John", "lastName": "Smith", "age":30, "salary": 4830}
```

Check for records in HazelCast

Now check the logs of the connector you should see this:

```
[2016-08-20 16:53:58,608] INFO Received 1 records. (com.datamountaineer.streamreactor.
↪connect.hazelcast.sink.HazelCastWriter:62)
[2016-08-20 16:53:58,644] INFO Written 1 (com.datamountaineer.streamreactor.connect.
↪hazelcast.sink.HazelCastWriter:71)
```

Now stop the connector.

Features

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The HazelCast Sink supports the following:

```
INSERT INTO <reliable topic> SELECT <fields> FROM <source topic> WITHFORMAT
<JSON|AVRO> STOREAS <RELIABLE_TOPIC|RING_BUFFER|QUEUE|SET|LIST|IMAP|MULTI_MAP|ICACHE>
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA
INSERT INTO tableA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to tableB, store as_
↪serialized avro encoded byte arrays
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB WITHFORMAT avro_
↪STOREAS RING_BUFFER
```

This is set in the `connect.hazelcast.kcql` option.

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other expectations thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.hazelcast.max.retries` and the `connect.hazelcast.retry.interval`.

With Format

Hazelcast requires that data stored in collections and topics is serializable. The Sink offers two modes to store data.

Avro In this mode the Sink converts the SinkRecords from Kafka to Avro encoded byte arrays. *Json* In this mode the Sink converts the SinkRecords from Kafka to Json strings.

This behaviour is controlled by the KCQL statement in the `connect.hazelcast.kcql` option. The default is JSON.

Stored As

The Hazelcast Sink supports storing data in RingBuffers, ReliableTopics, Queues, Sets, Lists, IMaps, Multi-maps and ICaches. This behaviour is controlled by the KCQL statement in the `connect.hazelcast.kcql` option. Note that IMaps, Multi-maps and ICaches support a key as well as a value.

```
#store into a ring buffer
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB WITHFORMAT avro
↪STOREAS RING_BUFFER
#store into a reliable topic
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB WITHFORMAT avro
↪STOREAS RELIABLE_TOPIC
#store into a queue
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB WITHFORMAT avro
↪STOREAS QUEUE
#store into a set
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB WITHFORMAT avro
↪STOREAS SET
#store into a list
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB WITHFORMAT avro
↪STOREAS LIST
#store into an i-map with field1 used as the map key
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB PK field1 WITHFORMAT
↪avro STOREAS IMAP
#store into a multi-map with field1 used as the map key
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB PK field1 WITHFORMAT
↪avro STOREAS MULTI_MAP
#store into an i-cache with field1 used as the cache key
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB PK field1 WITHFORMAT
↪avro STOREAS ICACHE
```

Parallel Writes

By default each task in the Sink will write the records it receives sequentially, the Sink optionally supports parallel writes where an `executorThreadPool` is started and records are written in parallel. While this results in better performance we can't guarantee the order of the writes.

To enable parallel writes set the `connect.hazelcast.parallel.write` configuration option to `true`.

Configurations

`connect.hazelcast.kcql`

KCQL expression describing field selection and routes.

- Data type : string
- Importance: high
- Optional : no

`connect.hazelcast.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.hazelcast.max.retries` option. The `connect.hazelcast.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Optional: yes
- Default: throw

`connect.hazelcast.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.hazelcast.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional: yes
- Default: 10

`connect.hazelcast.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.hazelcast.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional: yes
- Default : 60000 (1 minute)

`connect.hazelcast.batch.size`

Specifies how many records to insert together at one time. If the connect framework provides less records when it is calling the Sink it won't wait to fulfill this value but rather execute it.

- Type : int
- Importance : medium
- Optional: yes
- Defaults : 1000

`connect.hazelcast.cluster.members`

Address List is the initial list of cluster addresses to which the client will connect. The client uses this list to find an alive node. Although it may be enough to give only one address of a node in the cluster (since all nodes communicate with each other), it is recommended that you give the addresses for all the nodes.

- Data type : string
- Importance : high
- Optional: no
- Default: localhost

`connect.hazelcast.group.name`

The group name of the connector in the target Hazelcast cluster.

- Data type : string
- Importance : high
- Optional: no
- Default: dev

`connect.hazelcast.group.password`

The password for the group name.

- Data type : string
- Importance : high
- Optional : yes
- Default : dev-pass

`connect.hazelcast.timeout`

Connection timeout is the timeout value in milliseconds for nodes to accept client connection requests.

- Data type : int
- Importance : low
- Optional : yes
- Default : 5000

`connect.hazelcast.retries`

Number of times a client will retry the connection at startup.

- Data type : int
- Importance : low

- Optional : yes
- Default : 2

`connect.hazelcast.keep.alive`

Enables/disables the SO_KEEPALIVE socket option. The default value is true.

- Data type : boolean
- Importance : low
- Optional : yes
- Default : true

`connect.hazelcast.tcp.no.delay`

Enables/disables the SO_REUSEADDR socket option. The default value is true.

- Data type : boolean
- Importance : low
- Optional : yes
- Default : true

`connect.hazelcast.linger.seconds`

Enables/disables SO_LINGER with the specified linger time in seconds. The default value is 3.

- Data type : int
- Importance : low
- Optional : yes
- Default : 3

`connect.hazelcast.buffer.size`

Sets the SO_SNDBUF and SO_RCVBUF options to the specified value in KB for this Socket. The default value is 32.

- Data type : int
- Importance : low
- Optional : yes
- Default : 32

`connect.hazelcast.parallel.write`

All the sink to write in parallel the records received from Kafka on each poll. Order of writes in not guaranteed.

- Data type : boolean
- Importance : medium
- Optional : yes
- Default : false

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium

- Optional: yes
- Default : false

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

The Sink serializes either an Avro or Json representation of the Sink record to the target reliable topic in Hazelcast. Hazelcast is agnostic to the schema.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect, bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be pushed to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has its own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

Troubleshooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect HBase

A Connector and Sink to write events from Kafka to HBase. The connector takes the value from the Kafka Connect SinkRecords and inserts a new entry to HBase.

The Sink supports:

1. *The KCQL routing querying* - Kafka topic payload field selection is supported, allowing you to select fields written to HBase.
2. Topic to table routing via KCQL.
3. RowKey selection - Selection of fields to use as the row key, if none specified the topic name, partition and offset are used via KCQL.
4. Error policies.

Prerequisites

- Confluent 3.3
- HBase 1.2.0
- Java 1.8
- Scala 2.11

Setup

HBase Setup

Download and extract HBase:

```
wget https://www.apache.org/dist/hbase/1.2.1/hbase-1.2.1-bin.tar.gz
tar -xvf hbase-1.2.1-bin.tar.gz -C hbase
```

Edit conf/hbase-site.xml and add the following content:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///tmp/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/tmp/zookeeper</value>
  </property>
</configuration>
```

The `hbase.cluster.distributed` is required since when you start hbase it will try and start it's own Zookeeper, but in this case we want to use Confluents.

Now start HBase and check the logs to ensure it's up:

```
bin/start-hbase.sh
```

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

HBase Table

The Sink expects a precreated table in HBase. In the HBase shell create the test table, go to your HBase install location.

```
bin/hbase shell
hbase(main):001:0> create 'person',{NAME=>'d', VERSIONS=>1}

hbase(main):001:0> list
person
1 row(s) in 0.9530 seconds

hbase(main):002:0> describe 'person'
DESCRIPTION
'person', {NAME => 'd', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
↳KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0',
↳BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION
_SCOPE => '0'}
1 row(s) in 0.0810 seconds
```

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for HBase. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create hbase-sink < conf/hbase-sink.properties

#Connector name=`hbase-sink`
name=person-hbase-test
connector.class=com.datamountaineer.streamreactor.connect.hbase.HbaseSinkConnector
```

```
tasks.max=1
topics=hbase-topic
connect.hbase.column.family=d
connect.hbase.kcql=INSERT INTO person SELECT * FROM hbase-topic PK firstName, lastName
#task ids: 0
```

This `hbase-sink.properties` configuration defines:

1. The name of the sink.
2. The Sink class.
3. The max number of tasks the connector is allowed to created. Should not be greater than the number of partitions in the Source topics otherwise tasks will be idle.
4. The Source kafka topics to take events from.
5. The HBase column family to write to.
6. *The KCQL routing querying.*

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
hbase-sink
```

INFO

The diagram illustrates a sequence of operations on a tape. The tape is represented by a horizontal line with various symbols and markers. The operations are indicated by arrows and labels above the tape. The sequence starts with a series of 'V' markers, followed by a series of 'V' markers with a 'V' label, then a series of 'V' markers with a 'V' label, and finally a series of 'V' markers with a 'V' label. The diagram is divided into several sections by vertical lines, and the operations are performed in a specific order.

By Stefan Bocutiu (com.datamountaineer.streamreactor.connect.hbase.HbaseSinkTask:44)

Test Records

Now we need to put some records it to the `test_table` topics. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a `firstname` field of type string, a `lastname` field of type string, an `age` field of type int and a `salary` field of type double.

```
${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic hbase-topic \
```

```
--property value.schema='{ "type": "record", "name": "User",  
  "fields": [{ "name": "firstName", "type": "string"}, { "name": "lastName", "type": "string"}, {  
→ "name": "age", "type": "int"},  
  { "name": "salary", "type": "double"}] }'
```

Now the producer is waiting for input. Paste in the following:

```
{ "firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830 }  
{ "firstName": "Anna", "lastName": "Jones", "age": 28, "salary": 5430 }
```

Check for records in HBase

Now check the logs of the connector you should see this

```
INFO Sink task org.apache.kafka.connect.runtime.WorkerSinkTask@48ffb4dc finished_  
→ initialization and start (org.apache.kafka.connect.runtime.WorkerSinkTask:155)  
INFO Writing 2 rows to Hbase... (com.datamountaineer.streamreactor.connect.hbase.  
→ writers.HbaseWriter:83)
```

In HBase:

```
hbase(main):004:0> scan 'person'  
ROW                                COLUMN+CELL  
Anna\x0AJones                      column=d:age,   
→ timestamp=1463056888641, value=\x00\x00\x00\x1C  
Anna\x0AJones                      column=d:firstName,   
→ timestamp=1463056888641, value=Anna  
Anna\x0AJones                      column=d:income,   
→ timestamp=1463056888641, value=@\xB56\x00\x00\x00\x00  
Anna\x0AJones                      column=d:lastName,   
→ timestamp=1463056888641, value=Jones  
John\x0ASmith                      column=d:age,   
→ timestamp=1463056693877, value=\x00\x00\x00\x1E  
John\x0ASmith                      column=d:firstName,   
→ timestamp=1463056693877, value=John  
John\x0ASmith                      column=d:income,   
→ timestamp=1463056693877, value=@\xB2\xDE\x00\x00\x00\x00  
John\x0ASmith                      column=d:lastName,   
→ timestamp=1463056693877, value=Smith  
2 row(s) in 0.0260 seconds
```

Now stop the connector.

Features

The HBase Sink writes records from Kafka to HBase.

The Sink supports:

1. Field selection - Kafka topic payload field selection is supported, allowing you to select fields written to HBase.
2. Topic to table routing.
3. RowKey selection - Selection of fields to use as the row key, if none specified the topic name, partition and offset are used.

4. Error policies.

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The HBase Sink supports the following:

```
INSERT INTO <table> SELECT <fields> FROM <source topic> <PK> primary_key_cols
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA and use the default_
↪rowkey (topic name, partition, offset)
INSERT INTO tableA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to tableB, use field y_
↪from the topic as the row key
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB PK y
```

This is set in the `connect.hbase.kcql` option.

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers.

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.hbase.max.retries` and the `connect.hbase.retry.interval`.

Configurations

```
connect.hbase.column.family
```

The hbase column family.

- Type: string
- Importance: high
- Optional: no

`connect.hbase.kcql`

Kafka connect query language expression. Allows for expressive topic to table routing, field selection and renaming. Fields to be used as the row key can be set by specifying the PK. The below example uses field1 and field2 as the row key.

Examples:

```
INSERT INTO TABLE1 SELECT * FROM TOPIC1; INSERT INTO TABLE2 SELECT * FROM TOPIC2 PK_
↪field1, field2
```

If no primary keys are specified the topic name, partition and offset converted to bytes are used as the HBase rowkey.

- Type: string
- Importance: high
- Optional: no

`connect.hbase.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.hbase.max.retries` option. The `connect.hbase.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: medium
- Optional: yes
- Default: RETRY

`connect.hbase.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.hbase.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional: yes
- Default: 10

`connect.hbase.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.hbase.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional: yes
- Default : 60000 (1 minute)


```
connect.progress.enabled
```

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
connect.hbase.column.family=d
connect.hbase.kcql=INSERT INTO person SELECT * FROM TOPIC1
connector.class=com.datamountaineer.streamreactor.connect.hbase.HbaseSinkConnector
tasks.max=1
topics=TOPIC1
name=hbase-test
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

The HBase Sink will automatically write and update the HBase table if new fields are added to the Source topic, if fields are removed the Kafka Connect framework will return the default value for this field, dependent of the compatibility settings of the Schema registry. This value will be put into the HBase column family cell based on the `connect.hbase.kcql` value.

Deployment Guidelines

Ensure the `hbase-site.xml` is on the the classpath of the connector.

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Influx

A Connector and Sink to write events from Kafka to InfluxDB. The connector takes the value from the Kafka Connect SinkRecords and inserts a new entry to InfluxDB.

The Sink supports:

1. *The KCQL routing querying* - Topic to index mapping and Field selection.
2. Auto mapping of the Kafka topic schema to the index.
3. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema

The Sink supports three Kafka payloads type:

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example [here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires the SchemaRegistry which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should be set to `value.converter=org.apache.kafka.connect.json.JsonConverter`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a GenericRecord, to just send a message with Schema.String and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support.

Prerequisites

- Confluent 3.3
- Java 1.8

- Scala 2.11

Setup

Confluent Setup

Follow the instructions [here](#).

InfluxDB Setup

[Download](#) and start InfluxDB. Users of OS X 10.8 and higher can install InfluxDB using the Homebrew package manager. Once brew is installed, you can install InfluxDB by running:

```
brew update
brew install influxdb
# start influx
influxd
```

Versions Prior to 1.3

InfluxDB prior to version 1.3 starts an Admin web server listening on port 8083 by default. For this quickstart this will collide with Kafka Connects default port of 8083. Since we are running on a single node we will need to edit the InfluxDB config.

```
#create config dir
sudo mkdir /etc/influxdb
#dump the config
influxd config > /etc/influxdb/influxdb.generated.conf
```

Now change the following section to a port 8087 or any other free port.

```
[admin]
enabled = true
bind-address = ":8087"
https-enabled = false
https-certificate = "/etc/ssl/influxdb.pem"
```

Now start InfluxDB.

```
influxd
```

If you are running on a single node start InfluxDB with the new configuration file we generated.

```
influxd -config /etc/influxdb/influxdb.generated.conf
```

Sink Connector QuickStart

We you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Test data

The Sink expects a database to exist in InfluxDB. Use the InfluxDB CLI to create this:

```
~ influx
Visit https://enterprise.influxdata.com to register for updates, InfluxDB server_
↪management, and monitoring.
Connected to http://localhost:8086 version v1.0.2
InfluxDB shell version: v1.0.2
```

```
> CREATE DATABASE mydb
```

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the kafka-connect-tools [cli](#) to post in our distributed properties file for InfluxDB. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create influx-sink < conf/influxdb-sink.properties

#Connector name=`influx-sink`
name=influxdb-sink
connector.class=com.datamountaineer.streamreactor.connect.influx.InfluxSinkConnector
tasks.max=1
topics=influx-topic
connect.influx.kcql=INSERT INTO influxMeasure SELECT * FROM influx-topic_
↪WITHTIMESTAMP sys_time()
connect.influx.url=http://localhost:8086
connect.influx.db=mydb
#task ids: 0
```

The influx-sink.properties file defines:

1. The name of the connector.
2. The class containing the connector.
3. The max number of task allowed for this connector.
4. The Source topic to get records from.
5. *The KCQL routing querying.*
6. The InfluxDB connection URL.
7. The InfluxDB database.

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
influxdb-sink
```

```
[INFO]
com.datamountaineer.streamreactor.connect.influx.InfluxSinkTask:45)
[INFO InfluxSinkConfig values:
connect.influx.retention.policy = autogen
connect.influx.error.policy = THROW
connect.influx.username = root
connect.influx.db = mydb
connect.influx.password = [hidden]
connect.influx.url = http://localhost:8086
connect.influx.retry.interval = 60000
connect.influx.kcql = INSERT INTO influxMeasure SELECT * FROM influx-topic_
WITHTIMESTAMP sys_time()
connect.influx.max.retires = 20
(com.datamountaineer.streamreactor.connect.influx.config.InfluxSinkConfig:178)
```

Test Records

Now we need to put some records it to the `test_table` topics. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a `company` field of type `string` a `address` field of type `string`, an `latitude` field of type `int` and a `longitude` field of type `int`.

```

${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic influx-topic \
--property value.schema='{ "type": "record", "name": "User",
    "fields": [{ "name": "company", "type": "string", { "name": "address", "type": "string"}, {
    ← "name": "latitude", "type": "float"}, { "name": "longitude", "type": "float"}] }'

```

Now the producer is waiting for input. Paste in the following:

```
{"company": "DataMountaineer", "address": "MountainTop", "latitude": -49.817964,  
  ↪ "longitude": -141.645812}
```

Check for records in InfluxDB

Now check the logs of the connector you should see this:

```
INFO Setting newly assigned partitions [influx-topic-0] for group connect-influx-sink_
↳ (org.apache.kafka.clients.consumer.internals.ConsumerCoordinator:231)
INFO Received 1 record(-s) (com.datamountaineer.streamreactor.connect.influx.
↳ InfluxSinkTask:81)
INFO Writing 1 points to the database... (com.datamountaineer.streamreactor.connect.
↳ influx.writers.InfluxDbWriter:45)
INFO Records handled (com.datamountaineer.streamreactor.connect.influx.
↳ InfluxSinkTask:83)
```

Check in InfluxDB.

```
influx
Visit https://enterprise.influxdata.com to register for updates, InfluxDB server_
↳ management, and monitoring.
Connected to http://localhost:8086 version v1.0.2
InfluxDB shell version: v1.0.2
> use mydb;
Using database mydb
> show measurements;
name: measurements
-----
name
influxMeasure

> select * from influxMeasure;
name: influxMeasure
-----
time                address      async    company      latitude      _
↳      longitude
1478269679104000000 MontainTop    true     DataMountaineer -49.817962646484375 -
↳ 141.64581298828125
```

Features

1. Topic to index mapping. 3. Auto mapping of the Kafka topic schema to the index. 4. Field selection 5. Tagging the data points using constants or fields from the payload

Tag

InfluxDB allows via the client API to provide a set of tags (key-value) to each point added. The current connector version allows you to provide them via the KCQL

```
INSERT INTO <measure> SELECT <fields> FROM <source topic> WITHTIMESTAMP <field_name>
↳ |sys_time() WITHTAG(field| (constant_key=constant_value))
```

Example:

```
#Tagging using constants
INSERT INTO measureA SELECT * FROM topicA WITHTAG (DataMountaineer=awesome, _
↳ Influx=rulz!)

#Tagging using fields in the payload. Say we have a Payment structure with these_
↳ fields: amount, from, to, note
INSERT INTO measureA SELECT * FROM topicA WITHTAG (from, to)
```

```
#Tagging using a combination of fields in the payload and constants. Say we have a
↪Payment structure with these fields: amount, from, to, note
INSERT INTO measureA SELECT * FROM topicA WITHTAG (from, to,
↪provider=DataMountaineer)
```

Note: At the moment you can only reference the payload fields but if the structure is nested you can't address nested fields. Support for such functionality will be provided soon. You can't tag with fields present in the Kafka message key, or use the message metadata(partition, topic, index).

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The Influx Sink supports the following:

```
INSERT INTO <measure> SELECT <fields> FROM <source topic> WITHTIMESTAMP <field_name>
↪|sys_time()
```

Example:

```
#Insert mode, select all fields from topicA and write to indexA
INSERT INTO measureA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to indexB, use field Y
↪as the point measurement
INSERT INTO measureB SELECT x AS a, y AS b and z AS c FROM topicB WITHTIMESTAMP y

#Insert mode, select 3 fields and rename from topicB and write to indexB, use field Y
↪as the current system time for
#Point measurement
INSERT INTO measureB SELECT x AS a, y AS b and z AS c FROM topicB WITHTIMESTAMP sys_
↪time()
```

This is set in the `connect.influx.kcql` option.

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers.

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.influx.max.retries` and the `connect.influx.retry.interval`.

Configurations

`connect.influx.kcql`

Kafka connect query language expression. Allows for expressive topic to table routing, field selection and renaming. For InfluxDB it allows either setting a default or selecting a field from the topic as the Point measurement.

- Data type : string
- Importance: high
- Optional : no

`connect.influx.url`

The InfluxDB database url.

- Data type : string
- Importance: high
- Optional : no

`connect.influx.db`

The InfluxDB database.

- Data type : string
- Importance: high
- Optional : no

`connect.influx.username`

The InfluxDB username.

- Data type : string
- Importance: high
- Optional : yes

`connect.influx.password`

The InfluxDB password.

- Data type : string
- Importance: high

- Optional : yes

`connect.influx.consistency.level`

Specifies the write consistency. If any write operations do not meet the configured consistency guarantees, an error will occur and the data will not be indexed. The default consistency-level is `ALL`. Other available options are `ANY`, `ONE`, `QUORUM`

- Data type : string
- Importance: medium
- Optional : yes
- Default : `ALL`

`connect.influx.retention.policy`

Determines how long InfluxDB keeps the data - the options for specifying the duration of the retention policy are listed below. Note that the minimum retention period is one hour. `DURATION` determines how long InfluxDB keeps the data - the options for specifying the duration of the retention policy are listed below. Note that the minimum retention period is one hour.

m minutes h hours d days w weeks INF infinite

Default retention is *autogen* from 1.0 onwards or *default* for any previous version

- Data type : string
- Importance: medium
- Optional : yes

`connect.influx.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.influx.max.retries` option. The `connect.influx.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: medium
- Optional: yes
- Default: `RETRY`

`connect.influx.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.influx.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional: yes
- Default: 10

`connect.influx.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.influx.error.policy` set to **RETRY**.

- Type: int
- Importance: high
- Optional: no
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=influxdb-sink
connector.class=com.datamountaineer.streamreactor.connect.influx.InfluxSinkConnector
tasks.max=1
topics=influx-topic
connect.influx.kcql=INSERT INTO influxMeasure SELECT * FROM influx-topic_
↪WITHTIMESTAMP sys_time()
connect.influx.url=http://localhost:8086
connect.influx.db=mydb
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect JMS Sink

The JMS Sink connector allows you to extract entries from a Kafka topic with the CQL driver and pass them to a JMS topic/queue. The connector allows you to specify the payload type sent to the JMS target:

1. JSON
2. AVRO
3. MAP
4. OBJECT

The Sink supports:

1. *The KCQL routing querying*. Kafka topic payload field selection is supported, allowing you to select fields written to the queue or topic in JMS.
2. Topic to topic routing via KCQL.
3. Payload format selection via KCQL.
4. Error policies for handling failures.
5. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema.

The Sink supports three Kafka payloads type for TextMessage (Format JSON) only:

Note: Only support with used with KCQL format type JSON. This sends messages at TextMessages to the JMS destination.

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example [here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires the SchemaRegistry

which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should be set to `value.converter=org.apache.kafka.connect.json.JsonConverter`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a `GenericRecord`, to just send a message with `Schema.String` and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support.

Prerequisites

- Confluent 3.3
- Java 1.8
- Scala 2.11
- A JMS framework (ActiveMQ for example)

Setup

Before we can do anything, including the QuickStart we need to install the Confluent platform. For ActiveMQ follow <http://activemq.apache.org/getting-started.html> for the instruction of setting it up.

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

We you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for JMS. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create jms-sink < conf/jms-sink.properties
```

The `jms-sink.properties` file defines:

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
jms-sink
```

Test Records

Now we need to put some records it to the test_table topics. We can use the kafka-avro-console-producer to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a id field of type int and a random_field of type string.

```
${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic jms_test \
--property value.schema='{ "type": "record", "name": "User",
"fields": [{ "name": "firstName", "type": "string"}, { "name": "lastName", "type": "string"}, {
↪ "name": "age", "type": "int"}, { "name": "salary", "type": "double"}] }'
```

Now the producer is waiting for input. Paste in the following:

```
{ "firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830 }
{ "firstName": "Anna", "lastName": "Jones", "age": 28, "salary": 5430 }
```

Now check for records in ActiveMQ.

Now stop the connector.

Features

The Sink supports:

1. Field selection - Kafka topic payload field selection is supported, allowing you to select fields written to the queue or topic in JMS.
2. Topic to JMS Destination routing.
3. Payload format selection.
4. Error policies for handling failures.
5. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema. Only supported when storing as JSON

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The JMS Sink supports the following:

```
INSERT INTO <jms target> SELECT <fields> FROM <source topic> STOREAS  
↪<AVRO|JSON|MAP|OBJECT> WITHTYPE <TOPIC|QUEUE>
```

Example:

```
#select all fields from topicA and write to jmsA queue  
INSERT INTO jmsA SELECT * FROM topicA WITHTYPE QUEUE  
  
#select 3 fields and rename from topicB and write to jmsB topic as JSON in a_  
↪TextMessage  
INSERT INTO jmsB SELECT x AS a, y AS b and z AS c FROM topicB STOREAS JSON WITHTYPE_  
↪TOPIC
```

JMS Payload

When a message is sent to a JMS target it can be one of the following:

1. JSON - Send a TextMessage;
2. AVRO - Send a BytesMessage;
3. MAP - Send a MapMessage;
4. OBJECT - Send an ObjectMessage

Topic Routing

The Sink supports topic routing that allows mapping the messages from topics to a specific jms target. For example, map a topic called “bloomberg_prices” to a jms target named “prices”. This mapping is set in the `connect.jms.kcql` option.

Example:

```
//Select all  
INSERT INTO prices SELECT * FROM bloomberg_prices; INSERT INTO jms3 SELECT * FROM_  
↪topic2
```

Configurations

`connect.jms.url`

Provides the JMS broker url

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.username`

Provides the user for the JMS connection.

- Data Type: string
- Importance: high

- Optional : no

`connect.jms.password`

Provides the password for the JMS connection.

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.initial.context.factory`

- Data Type: string
- Importance: high
- Optional: no

Initial Context Factory, e.g: `org.apache.activemq.jndi.ActiveMQInitialContextFactory`.

`connect.jms.connection.factory`

The ConnectionFactory implementation to use.

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.destination.selector`

- Data Type: String
- Importance: high
- Optional: no
- Default: CDI

Selector to use for destination lookup. Either CDI or JNDI.

`connect.jms.initial.context.extra.params`

- Data Type: String
- Importance: high
- Optional: yes

List (comma separated) of extra properties as key/value pairs with a colon delimiter to supply to the initial context e.g. `SOLACE_JMS_VPN:my_solace_vp`.

`connect.jms.kcql`

KCQL expression describing field selection and routes. The kcql expression also handles setting the JMS destination type, i.e. TOPIC or QUEUE via the `withtype` keyword.

- Data Type: string
- Importance: high
- Optional : no

`connect.jms.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.jms.max.retries` option. The `connect.jms.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: medium
- Optional: yes
- Default: RETRY

`connect.jms.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.jms.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional: yes
- Default: 10

`connect.jms.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.jms.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional: yes
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Schema Evolution

Not applicable.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.

3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect, bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Kudu

A Connector and Sink to write events from Kafka to kudu. The connector takes the value from the Kafka Connect SinkRecords and inserts a new entry to Kudu.

The Sink supports:

1. *The [KCQL routing querying](#)* - Kafka topic payload field selection is supported, allowing you to select fields written to Kudu.
2. Topic to table routing via KCQL.
3. Auto table create with DISTRIBUTE BY partition strategy via KCQL.
4. Auto evolution of tables via KCQL.
5. Error policies for handling failures.

Prerequisites

- Confluent 3.3
- Kudu 0.8
- Java 1.8
- Scala 2.11

Setup

Kudu Setup

Download and check Kudu QuickStart VM starts up (VM password is demo).

```
curl -s https://raw.githubusercontent.com/cloudera/kudu-examples/master/demo-vm-setup/  
↪bootstrap.sh | bash
```

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Kudu Table

Let's create a table in Kudu via Impala. The Sink does support auto creation of tables but they are not sync'd yet with Impala.

```
#demo/demo  
ssh demo@quickstart -t impala-shell  
  
CREATE TABLE default.kudu_test (id INT,random_field STRING, PRIMARY KEY(id))  
PARTITION BY HASH PARTITIONS 16  
STORED AS KUDU
```

Note: The Sink will fail to start if the tables matching the topics do not already exist and the KQL statement is not in auto create mode.

When creating a new Kudu table using Impala, you can create the table as an internal table or an external table.

Internal

An internal table is managed by Impala, and when you drop it from Impala, the data and the table truly are dropped. When you create a new table using Impala, it is generally a internal table.

External

An external table (created by `CREATE EXTERNAL TABLE`) is not managed by Impala, and dropping such a table does not drop the table from its source location (here, Kudu). Instead, it only removes the mapping between Impala and Kudu. This is the mode used in the syntax provided by Kudu for mapping an existing table to Impala.

See the Impala documentation for more information about internal and external tables. Here's an example:

```
CREATE EXTERNAL TABLE default.kudu_test
STORED AS KUDU
TBLPROPERTIES ('kudu.table_name'='kudu_test');
```

Impala Databases and Kudu

Every Impala table is contained within a namespace called a database. The default database is called default, and users may create and drop additional databases as desired.

Note: When a managed Kudu table is created from within Impala, the corresponding Kudu table will be named `impala::my_database.table_name`

Tables created by the sink are not automatically visible to Impala. You must map the table in Impala.

```
CREATE EXTERNAL TABLE my_mapping_table
STORED AS KUDU
TBLPROPERTIES (
'kudu.table_name' = 'my_kudu_table'
);
```

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the kafka-connect-tools [cli](#) to post in our distributed properties file for Kudu. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create kudu-sink < conf/kudu-sink.properties
#Connector name=kudu-sink
connector.class=com.datamountaineer.streamreactor.connect.kudu.KuduSinkConnector
tasks.max=1
connect.kudu.master=quickstart
connect.kudu.kcql = INSERT INTO impala::default.kudu_test SELECT * FROM kudu-test
topics=kudu-test
#task ids: 0
```

The `kudu-sink.properties` file defines:

1. The name of the sink.
2. The Sink class.
3. The max number of tasks the connector is allowed to created. Should not be greater than the number of partitions in the Source topics otherwise tasks will be idle.
4. The Kudu master host.
5. *The KCQL routing querying.*

6. The Source kafka topics to take events from.

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
kudu-sink
```

```
[2016-05-08 22:00:20,823] INFO
```

by Andrew Stevenson

```
(com.datamountaineer.streamreactor.connect.kudu.KuduSinkTask:37)
[2016-05-08 22:00:20,823] INFO KuduSinkConfig values:
  connect.kudu.master = quickstart
  (com.datamountaineer.streamreactor.connect.kudu.KuduSinkConfig:165)
[2016-05-08 22:00:20,824] INFO Connecting to Kudu Master at quickstart (com.
↳ datamountaineer.streamreactor.connect.kudu.KuduWriter$:33)
[2016-05-08 22:00:20,875] INFO Initialising Kudu writer (com.datamountaineer.
↳ streamreactor.connect.kudu.KuduWriter:40)
[2016-05-08 22:00:20,892] INFO Assigned topics (com.datamountaineer.streamreactor.
↳ connect.kudu.KuduWriter:42)
[2016-05-08 22:00:20,904] INFO Sink task org.apache.kafka.connect.runtime.
↳ WorkerSinkTask@b60ba7b finished initialization and start (org.apache.kafka.connect.
↳ runtime.WorkerSinkTask:155)
```

Test Records

Now we need to put some records it to the `test_table` topics. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a `id` field of type `int` and a `random_field` of type `string`.

```

${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic kudu-test \
--property value.schema='{"type": "record", "name": "myrecord", "fields": [{"name": "id",
↪ "type": "int"},
{"name": "random_field", "type": "string"}]}'

```

Now the producer is waiting for input. Paste in the following:

```
{ "id": 999, "random_field": "foo" }
{ "id": 888, "random_field": "bar" }
```

Check for records in Kudu

Now check the logs of the connector you should see this:

[illegible]

by Andrew Stevenson

```
(com.datamountaineer.streamreactor.connect.kudu.KuduSinkTask:37)
[2016-05-08 22:09:22,065] INFO KuduSinkConfig values:
    connect.kudu.master = quickstart
    (com.datamountaineer.streamreactor.connect.kudu.KuduSinkConfig:165)
[2016-05-08 22:09:22,066] INFO Connecting to Kudu Master at quickstart (com.
→datamountaineer.streamreactor.connect.kudu.KuduWriter$:33)
[2016-05-08 22:09:22,116] INFO Initialising Kudu writer (com.datamountaineer.
→streamreactor.connect.kudu.KuduWriter:40)
[2016-05-08 22:09:22,134] INFO Assigned topics kudu_test (com.datamountaineer.
→streamreactor.connect.kudu.KuduWriter:42)
[2016-05-08 22:09:22,148] INFO Sink task org.apache.kafka.connect.runtime.
→WorkerSinkTask@68496440 finished initialization and start (org.apache.kafka.connect.
→runtime.WorkerSinkTask:155)
[2016-05-08 22:09:22,476] INFO Written 2 for kudu_test (com.datamountaineer.
→streamreactor.connect.kudu.KuduWriter:90)
```

In Kudu:

```
#demo/demo
ssh demo@quickstart -t impala-shell

SELECT * FROM kudu_test;

Query: select * FROM kudu_test
+-----+-----+
| id | random_field |
+-----+-----+
| 888 | bar          |
| 999 | foo          |
+-----+-----+
Fetched 2 row(s) in 0.14s
```

Now stop the connector.

Features

The Kudu Sink writes records from Kafka to Kudu.

The Sink supports:

1. Field selection - Kafka topic payload field selection is supported, allowing you to select fields written to Kudu.
2. Topic to table routing.
3. Auto table create with HASH partition strategy by using DISTRIBUTE BY with configurable buckets. Tables that are autocreated are not immediately visible in Impala. You must map them in Impala.
4. Auto evolution of tables.
5. Error policies for handling failures.

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

Note: If tables are created by Impala suffix `impala::database_name` to the target table.

The Kudu Sink supports the following:

```
<write mode> INTO [impala]:[database].<target table> SELECT <fields> FROM <source_  
→topic> <AUTOCREATE> <AUTOEVOLVE> <DISTRIBUTEBY> <PK_FIELDS> INTO <NBR_OF_BUCKETS>_  
→BUCKETS
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA  
INSERT INTO tableA SELECT * FROM topicA  
  
#Insert mode, select 3 fields and rename from topicB and write to tableB  
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB  
  
#Upsert mode, select all fields from topicC, auto create tableC and auto evolve, use_  
→field1 and field2 as the primary keys  
UPSERT INTO tableC SELECT * FROM topicC AUTOCREATE DISTRIBUTEBY field1, field2 INTO_  
→10 BUCKETS AUTOEVOLVE
```

Error Polices

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the Sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other expectations thrown by drivers.

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.kudu.max.retries` and the `connect.kudu.retry.interval`.

Auto conversion of Connect records to Kudu

The Sink automatically converts incoming Connect records to Kudu inserts or upserts.

Topic Routing

The Sink supports topic routing that allows mapping the messages from topics to a specific table. For example, map a topic called "bloomberg_prices" to a table called "prices". This mapping is set in the `connect.kudu.kcql` option.

Example:

```
//Select all
INSERT INTO table1 SELECT * FROM topic1; INSERT INTO tableA SELECT * FROM topicC
```

Field Selection

The Kudu Sink supports field selection and mapping. This mapping is set in the `connect.kudu.kcql` option.

Examples:

```
//Rename or map columns
INSERT INTO table1 SELECT lst_price AS price, qty AS quantity FROM topicA

//Select all
INSERT INTO table1 SELECT * FROM topic1
```

Tip: Check you mappings to ensure the target columns exist.

Warning: Field selection disables evolving the target table if the upstream schema in the Kafka topic changes. By specifying field mappings it is assumed the user is not interested in new upstream fields. For example they may be tapping into a pipeline for a Kafka stream job and not be intended as the final recipient of the stream.

If you chose field selection you must include the primary key fields otherwise the insert will fail.

Write Modes

The Sink supports both **insert** and **upsert** modes. This mapping is set in the `connect.kudu.kcql` option.

Insert

Insert is the default write mode of the sink.

Insert Idempotency

Kafka currently provides at least once delivery semantics. Therefore, this mode may produce errors if unique constraints have been implemented on the target tables. If the error policy has been set to NOOP then the Sink will discard the error and continue to process, however, it currently makes no attempt to distinguish violation of integrity constraints from other exceptions such as casting issues.

Upsert

The Sink support Kudu upserts which replaces the existing row if a match is found on the primary keys.

Upsert Idempotency

Kafka currently provides at least once delivery semantics and order is a guaranteed within partitions.

This mode will, if the same record is delivered twice to the sink, result in an idempotent write. The existing record will be updated with the values of the second which are the same.

If records are delivered with the same field or group of fields that are used as the primary key on the target table, but different values, the existing record in the target table will be updated.

Since records are delivered in the order they were written per partition the write is idempotent on failure or restart. Redelivery produces the same result.

Auto Create Tables

The Sink supports auto creation of tables for each topic. This mapping is set in the `connect.kudu.kcql` option.

Primary keys are set in the `DISTRIBUTE BY` clause of the `connect.kudu.kcql`.

Tables are created with the Kudu hash partition strategy. The number of buckets must be specified in the `kcql` statement.

```
#AutoCreate the target table
INSERT INTO table1 SELECT * FROM topic AUTOCREATE DISTRIBUTE BY field1, field2 INTO 10
↪BUCKETS
```

Note: The fields specified as the primary keys (`distributeby`) must be in the `SELECT` clause or all fields must be selected

The Sink will try and create the table at start up if a schema for the topic is found in the Schema Registry. If no schema is found the table is created when the first record is received for the topic.

Note: Tables that are created are not visible to Impala. You must map them in Impala yourself.

Auto Evolve Tables

The Sink supports auto evolution of tables for each topic. This mapping is set in the `connect.kudu.kcql` option. When set the Sink will identify new schemas for each topic based on the schema version from the Schema registry. New columns will be identified and an alter table DDL statement issued against Kudu.

Schema evolution can occur upstream, for example any new fields or change in data type in the schema of the topic, or downstream DDLs on the database.

Upstream changes must follow the schema evolution rules laid out in the Schema Registry. This Sink only supports BACKWARD and FULLY compatible schemas. If new fields are added the Sink will attempt to perform a ALTER table DDL statement against the target table to add columns. All columns added to the target table are set as nullable.

Fields cannot be deleted upstream. Fields should be of Avro union type `[null, <dataType>]` with a default set. This allows the Sink to either retrieve the default value or null. The Sink is not aware that the field has been deleted as a value is always supplied to it.

Warning: If a upstream field is removed and the topic is not following the Schema Registry's evolution rules, i.e. not full or backwards compatible, any errors will default to the error policy.

Downstream changes are handled by the sink. If columns are removed, the mapped fields from the topic are ignored. If columns are added, we attempt to find a matching field by name in the topic.

Error Polices

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.kudu.max.retries` and the `connect.cassandra.retry.interval`.

Data Type Mappings

Connect Type	Kudu Data Type
INT8	INT8
INT16	INT16
INT32	INT32
INT64	INT64
BOOLEAN	BOOLEAN
FLOAT32	FLOAT
FLOAT64	FLOAT
BYTES	BINARY

Configurations

`connect.kudu.master`

Specifies a Kudu server.

- Data type : string
- Importance: high
- Optional : no

`connect.kudu.kcql`

Kafka connect query language expression. Allows for expressive topic to table routing, field selection and renaming.

Examples:

```
INSERT INTO impala::default.TABLE1 SELECT * FROM TOPIC1; INSERT INTO TABLE2 SELECT ↵  
↵field1, field2, field3 as renamedField FROM TOPIC2
```

- Data Type: string
- Importance: high
- Optional : no

`connect.kudu.write.flush.mode`

Flush mode on write.

1. SYNC - flush each sink record. Batching is disabled.
 2. BATCH_BACKGROUND - flush batch of sink records in background thread.
 3. BATCH_SYNC - flush batch of sink records.
- Data Type: string
 - Importance: medium
 - Optional : yes
 - Default: SYNC

`connect.kudu.mutation.buffer.space`

Kudu Session mutation buffer space.

- Data Type: int

- Importance: low
- Optional : yes
- Default: 10000

`connect.kudu.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.kudu.max.retries` option. The `connect.kudu.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Optional : yes
- Default: RETRY

`connect.kudu.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.kudu.error.policy` is set to **retry**.

- Type: string
- Importance: medium
- Optional : yes
- Default: 10

`connect.kudu.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.kudu.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional : yes
- Default : 60000 (1 minute)

`connect.kudu.schema.registry.url`

The url for the Schema registry. This is used to retrieve the latest schema for table creation.

- Type : string
- Importance : high
- Optional : yes
- Default : <http://localhost:8081>

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes

- Default : false

Example

```
name=kudu-sink
connector.class=com.datamountaineer.streamreactor.connect.kudu.KuduSinkConnector
tasks.max=1
connect.kudu.master=quickstart
connect.kudu.kcql=INSERT INTO impala::default.kudu_test SELECT * FROM kudu_test_
↪AUTOCREATE DISTRIBUTE BY id INTO 5 BUCKETS
topics=kudu-test
connect.kudu.schema.registry.url=http://myhost:8081
```

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Mongo Sink

The Mongo Sink allows you to write events from Kafka to your MongoDB instance. The connector converts the value from the Kafka Connect SinkRecords to MongoDB Document and will do an insert or upsert depending on the configuration you chose. It is expected the database is created upfront; the targeted MongoDB collections will be created if they don't exist

Note: The database needs to be created upfront!

The Sink supports:

1. *The KCQL routing querying* - Topic to measure mapping and Field selection.
2. Schema registry support for Connect/Avro with a schema.
3. Schema registry support for Connect and no schema (schema set to Schema.String)
4. Json payload support, no Schema Registry.
5. Error policies.
6. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema

The Sink supports three Kafka payloads type:

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example [here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires the SchemaRegistry which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should be set to `value.converter=org.apache.kafka.connect.json.JsonConverter`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a GenericRecord, to just send a message with Schema.String and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support.

Prerequisites

- MongoDB 3.2.10
- Confluent 3.3
- Java 1.8
- Scala 2.11

Setup

Before we can do anything, including the QuickStart we need to install MongoDB and the Confluent platform.

Confluent Setup

Follow the instructions [here](#).

MongoDb Setup

If you already have an instance of Mongo running you can skip this step. First download and install MongoDB Community edition. This is the manual approach for installing on Ubuntu. You can follow the details <https://docs.mongodb.com/v3.2/administration/install-community/> for your OS.

```
#go to home folder
cd ~
#make a folder for mongo
mkdir mongodb

#Download Mongo
wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-ubuntu1604-3.2.10.
→tgz

#extract the archive
tar xvf mongodb-linux-x86_64-ubuntu1604-3.2.10.tgz -C mongodb
cd mongodb
mv mongodb-linux-x86_64-ubuntu1604-3.2.10/* .

#create the data folder
mkdir data
mkdir data/db

#Start MongoDB
bin/mongod --dbpath data/db
```

Sink Connector QuickStart

We you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

The important configuration for Connect is related to the key and value deserializer. In the first example we default to the best practice where the source sends Avro messages to a Kafka topic. It is not enough to just be Avro messages but also the producer must work with the Schema Registry to create the schema if it doesn't exist and set the schema id in the message. Every message sent will have a magic byte followed by the Avro schema id and then the actual Avro record in binary format.

Here are the entries in the config setting all the above. The are placed in the `connect-properties` file Kafka Connect is started with. Of course if your SchemaRegistry runs on a different machine or you have multiple instances of it you will have to amend the configuration.

```
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081
```

Test Database

The Sink requires that a database be precreated in MongoDB.

```
#from a new terminal
cd ~/mongodb/bin

#start the cli
./mongo

#list all dbs
show dbs

#create a new database named connect
use connect
#create a dummy collection and insert one document to actually create the database
db.dummy.insert({"name": "Kafka Rulz!"})

#list all dbs
show dbs
```

Starting the Connector

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the kafka-connect-tools [cli](#) to post in our distributed properties file for Kudu. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create mongo-sink < conf/source.kcql/mongo-sink.properties

#Connector `mongo-sink-orders`:
name=mongo-sink-orders
connector.class=com.datamountaineer.streamreactor.connect.mongodb.sink.
↳MongoSinkConnector
tasks.max=1
topics=orders-topic
connect.mongo.kcql=INSERT INTO orders SELECT * FROM orders-topic
connect.mongo.db=connect
connect.mongo.connection=mongodb://localhost:27017
connect.mongo.batch.size=10

#task ids: 0
```

If you switch back to the terminal you started Kafka Connect in you should see the Mongo Sink being accepted and the task starting.

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
mongo-sink
```

```
[2016-11-06 22:25:29,354] INFO MongoConfig values:
    connect.mongo.retry.interval = 60000
    connect.mongo.kcql = INSERT INTO orders SELECT * FROM orders-topic
    connect.mongo.connection = mongodb://localhost:27017
    connect.mongo.error.policy = THROW
    connect.mongo.db = connect
    connect.mongo.sink.batch.size = 10
    connect.mongo.max.retires = 20
(com.datamountaineer.streamreactor.connect.mongodb.config.MongoConfig:178)
[2016-11-06 22:25:29,399] INFO
```



_ by Stefan Bocutiu

```
. (com.datamountaineer.streamreactor.connect.mongodb.sink.MongoSinkTask:51)
[2016-11-06 22:25:29,990] INFO Initialising Mongo writer.Connection to mongodb://
→localhost:27017 (com.datamountaineer.streamreactor.connect.mongodb.sink.MongoWriter
→$:126)
```

Test Records

Hint: If your input topic doesn't match the target use [Kafka Streams](#) to transform in realtime the input. Also checkout the [Plumber](#), which allows you to inject a Lua script into [Kafka Streams](#) to do this, no Java or Scala required!

Now we need to put some records it to the orders-topic. We can use the `kafka-avro-console-producer` to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema matches the table created earlier.

```
bin/kafka-avro-console-producer \
  --broker-list localhost:9092 --topic orders-topic \
  --property value.schema='{ "type": "record", "name": "myrecord", "fields": [{ "name": "id",
↵ "type": "int"},
{ "name": "created", "type": "string"}, { "name": "product", "type": "string"}, { "name":
↵ "price", "type": "double"}]}'
```

Now the producer is waiting for input. Paste in the following (each on a line separately):

```
{
  "id": 1, "created": "2016-05-06 13:53:00", "product": "OP-DAX-P-20150201-95.7",
  "price": 94.2
},
{
  "id": 2, "created": "2016-05-06 13:54:00", "product": "OP-DAX-C-20150201-100", "price": 99.5
},
{
  "id": 3, "created": "2016-05-06 13:55:00", "product": "FU-DATAMOUNTAINEER-20150201-100", "price": 10000
},
{
  "id": 4, "created": "2016-05-06 13:56:00", "product": "FU-KOSPI-C-20150201-100", "price": 150
}
```

Now if we check the logs of the connector we should see 2 records being inserted to MongoDB:


```
[2016-11-06 22:30:30,473] INFO Setting newly assigned partitions [orders-topic-0] for
↳group connect-mongo-sink-orders (org.apache.kafka.clients.consumer.internals.
↳ConsumerCoordinator:231)
[2016-11-06 22:31:29,328] INFO WorkerSinkTask{id=mongo-sink-orders-0} Committing
↳offsets (org.apache.kafka.connect.runtime.WorkerSinkTask:261)
```

```
#Open a new terminal and navigate to the mongodb instalation folder
./bin/mongo
> show databases
  connect  0.000GB
  local    0.000GB
> use connect
  switched to db connect
> show collections
  dummy
  orders
> db.orders.find()
{ "_id" : ObjectId("581fb21b09690a24b63b35bd"), "id" : 1, "created" : "2016-05-06
↳13:53:00", "product" : "OP-DAX-P-20150201-95.7", "price" : 94.2 }
{ "_id" : ObjectId("581fb2f809690a24b63b35c2"), "id" : 2, "created" : "2016-05-06
↳13:54:00", "product" : "OP-DAX-C-20150201-100", "price" : 99.5 }
{ "_id" : ObjectId("581fb2f809690a24b63b35c3"), "id" : 3, "created" : "2016-05-06
↳13:55:00", "product" : "FU-DATAMOUNTAINEER-20150201-100", "price" : 10000 }
{ "_id" : ObjectId("581fb2f809690a24b63b35c4"), "id" : 4, "created" : "2016-05-06
↳13:56:00", "product" : "FU-KOSPI-C-20150201-100", "price" : 150 }
```

Bingo, our 4 rows!

Legacy topics (plain text payload with a json string)

We have found some of the clients have already an infrastructure where they publish pure json on the topic and obviously the jump to follow the best practice and use schema registry is quite an ask. So we offer support for them as well.

This time we need to start the connect with a different set of settings.

```
#create a new configuration for connect
cp etc/schema-registry/connect-avro-distributed.properties etc/schema-registry/
↳connect-avro-distributed-json.properties
vi etc/schema-registry/connect-avro-distributed-json.properties
```

Replace the following 4 entries in the config

```
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081
```

with the following

```
key.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=false
value.converter=org.apache.kafka.connect.json.JsonConverter
value.converter.schemas.enable=false
```

Now let's restart the connect instance:

```
#start a new instance of connect
$CONFLUENT_HOME/bin/confluent stop
$CONFLUENT_HOME/bin/confluent start
```

Use the CLI to remove the old MongoDB Sink:

```
bin/connect-cli rm mongo-sink
```

and start the new Sink with the json properties files to read from the a different topic with json as the payload.

```
#start the connector for mongo
bin/connect-cli create mongo-sink-orders-json < mongo-sink-orders-json.properties
```

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

```
[2016-11-06 23:53:09,881] INFO MongoConfig values:  
    connect.mongo.retry.interval = 60000  
    connect.mongo.kcql = UPSERT INTO orders_json SELECT id, product as product_name,  
→ price as value FROM orders-topic-json PK id  
    connect.mongo.connection = mongodb://localhost:27017  
    connect.mongo.error.policy = THROW  
    connect.mongo.db = connect  
    connect.mongo.batch.size = 10  
    connect.mongo.max.retires = 20  
  
(com.datamountaineer.streamreactor.connect.mongodb.config.MongoConfig:178)  
[2016-11-06 23:53:09,927] INFO  
  
_ _ _ _ _  
| | \ / _ | | _ | V | _ - - - | | _ _ ( ) _ _ _ _ _  
| | | / _ ' | / _' ||V|| / _\| | | | ' \ \| _ / _ ' \ \| _ \| '  
| | | ( | | | ( | | | ( ) | | | | | | | ( | | | | _ / _ / |  
|_| _ / \ , _ \ \ \ , _ | | _ \ _ / \ , _ | | _ \ \ \ , _ | | | _ \ \ _ |  
_ by Stefan Bocutiu  
| V | _ _ _ _ _ | _ \| | _ / ( ) _ _ | | _  
| \| | / _ \| ' \ / _' | / _ \| | | | ' \ \ _ \| | ' \| | / /  
| | | | ( ) | | | ( | ( ) | | | | _ ) | | | | <  
| | | _ \| _ / | _ \| , \| _ / _ _ / | _ _ / | | | | _ \|  
. (com.datamountaineer.streamreactor.connect.mongodb.sink.MongoSinkTask:51)  
[2016-11-06 23:53:10,270] INFO Initialising Mongo writer.Connection to mongodb://  
→ localhost:27017 (com.datamountaineer.streamreactor.connect.mongodb.sink.MongoWriter  
→ $:126)
```

Now it's time to produce some records. This time we will use the simple kafka-console-producer to put simple json on the topic:

```
$ {CONFLUENT_HOME}/bin/kafka-console-producer --broker-list localhost:9092 --topic_
↪ orders-topic-json

{"id": 1, "created": "2016-05-06 13:53:00", "product": "OP-DAX-P-20150201-95.7",
↪ "price": 94.2}
{"id": 2, "created": "2016-05-06 13:54:00", "product": "OP-DAX-C-20150201-100", "price
↪ ": 99.5}
```

```
{ "id": 3, "created": "2016-05-06 13:55:00", "product": "FU-DATAMOUNTAINEER-20150201-100", "price": 10000 }
```

Following the command you should have something similar to this in the logs for your connect:

```
[2016-11-07 00:08:30,200] INFO Setting newly assigned partitions [orders-topic-json-0] for group connect-mongo-sink-orders-json (org.apache.kafka.clients.consumer.internals.ConsumerCoordinator:231)
[2016-11-07 00:08:30,324] INFO Opened connection [connectionId{localValue:3,serverValue:9}] to localhost:27017 (org.mongodb.driver.connection:71)
```

Let's check the mongo db database for the new records:

```
#Open a new terminal and navigate to the mongodb installation folder
./bin/mongo
> show databases
  connect  0.000GB
  local    0.000GB
> use connect
  switched to db connect
> show collections
  dummy
  orders
  orders_json
> db.orders_json.find()
{ "_id" : ObjectId("581fc5fe53b2c9318a3c1004"), "created" : "2016-05-06 13:53:00", "id" : NumberLong(1), "product_name" : "OP-DAX-P-20150201-95.7", "value" : 94.2 }
{ "_id" : ObjectId("581fc5fe53b2c9318a3c1005"), "created" : "2016-05-06 13:54:00", "id" : NumberLong(2), "product_name" : "OP-DAX-C-20150201-100", "value" : 99.5 }
{ "_id" : ObjectId("581fc5fe53b2c9318a3c1006"), "created" : "2016-05-06 13:55:00", "id" : NumberLong(3), "product_name" : "FU-DATAMOUNTAINEER-20150201-100", "value" : 10000 }
```

Bingo, our 3 rows!

Features

The sink connector will translate the SinkRecords to json and will insert each one in the database. We support to insert modes: INSERT and UPSERT. All of this can be expressed via KCQL (our own SQL like syntax for configuration. Please see below the section for Kafka Connect Query Language)

The sink supports:

1. Field selection - Kafka topic payload field selection is supported, allowing you to have choose selection of fields or all fields written to MongoDB.
2. Topic to table routing. Your sink instance can be configured to handle multiple topics and collections. All you have to do is to set your configuration appropriately. Below you will find an example

```
connect.mongo.kcql = INSERT INTO orders SELECT * FROM orders-topic; UPSERT INTO customers SELECT * FROM customer-topic PK customer_id; UPSERT INTO invoiceid as invoice_id, customerid as customer_id, value a SELECT invoice_id, FROM invoice-topic
```

3. Error policies for handling failures.

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**, *KCQL* allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

MongoDb sink supports the following:

```
INSERT INTO <target collection> SELECT <fields> FROM <source topic> <PK field name>
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA
INSERT INTO collectionA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to tableB with primary
↪key as the field id from the topic
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB PK id
```

Error Policies

The sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the database is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the sink will retry can be controlled by using the `connect.mongo.max.retires` and the `connect.mongo.retry.interval`.

Topic Routing

The sink supports topic routing that maps the messages from topics to a specific collection. For example map a topic called "bloomberg_prices" to a collection called "prices". This mapping is set in the `connect.mongo.kcql` option. You don't need to set up multiple sinks for each topic or collection. The same sink instance can be configured to handle multiple collections. For example your configuration in this case:

```
connect.mongo.kcql = INSERT INTO orders SELECT * FROM orders-topic; UPSERT INTO
↳customers SELECT * FROM customer-topic PK customer_id; UPSERT INTO invoiceid as
↳invoice_id, customerid as customer_id, value a SELECT invoice_id, FROM invoice-topic
```

Field Selection

The sink supports selecting fields from the source topic or selecting all. There is an option to rename a field as well. All of this can be easily expressed with KCQL:

- Select all fields from topic `fx_prices` and insert into the `fx` collection: `INSERT INTO fx SELECT * FROM fx_prices.`
- Select all fields from topic `fx_prices` and upsert into the `fx` collection, The assumption is there will be a `ticker` field in the incoming json: `UPSERT INTO fx SELECT * FROM fx_prices PK ticker.`
- Select specific fields from the topic `sample_topic` and insert into the `sample` collection: `INSERT INTO sample SELECT field1,field2,field3 FROM sample_topic.`
- Select specific fields from the topic `sample_topic` and upsert into the `sample` collection: `UPSERT INTO sample SELECT field1,field2,field3 FROM sample_fopic PK field1.`
- Rename some fields while selecting all from the topic `sample_topic` and insert into the `sample` collection: `INSERT INTO sample SELECT *, field1 as new_name1,field2 as new_name2 FROM sample_topic.`
- Rename some fields while selecting all from the topic `sample_topic` and upsert into the `sample` collection: `UPSERT INTO sample SELECT *, field1 as new_name1,field2 as new_name2 FROM sample_topic PK new_name1.`
- Select specific fields and rename some of them from the topic `sample_topic` and insert into the `sample` collection: `INSERT INTO sample SELECT field1 as new_name1,field2, field3 as new_name3 FROM sample_topic.`
- Select specific fields and rename some of them from the topic `sample_topic` and upsert into the `sample` collection: `INSERT INTO sample SELECT field1 as new_name1,field2, field3 as new_name3 FROM sample_fopic PK new_name3.`

TLS/SSL

TLS/SSL is support by setting `?ssl=true` in the `connect.mongo.connection` option. The MongoDB driver will then load attempt to load the truststore and keystore using the JVM system properties.

You will need to set several JVM system properties to ensure that the client is able to validate the SSL certificate presented by the server:

```
javax.net.ssl.trustStore: the path to a trust store containing the certificate of the
↳signing authority
javax.net.ssl.trustStorePassword: the password to access this trust store
```

The trust store is typically created with the `keytool` command line program provided as part of the JDK. For example:

```
keytool -importcert -trustcacerts -file <path to certificate authority file> -
↳keystore <path to trust store> -storepass <password>
```

You will also need to set several JVM system properties to ensure that the client presents an SSL certificate to the MongoDB server:

```
javax.net.ssl.keyStore: the path to a key store containing the client's SSL
↳certificates
javax.net.ssl.keyStorePassword: the password to access this key store
```

The key store is typically created with the `keytool` or the `openssl` command line program.

Authentication Mechanism

All authentication methods are supported, X.509, LDAP Plain, Kerberos (GSSAPI), MongoDB-CR and SCRAM-SHA-1. The default as of MongoDB version 3.0 SCRAM-SHA-1. To set the authentication mechanism set the `authMechanism` in the `connect.mongo.connection` option.

Note: The mechanism can either be set in the connection string but this requires the password to be in plain text in the connection string or via the `connect.mongo.auth.mechanism` option.

If the username is set it overrides the username/password set in the connection string and the `connect.mongo.auth.mechanism` has precedence.

e.g.

```
# default of scram
mongodb://host1/?authSource=db1
# scram explicit
mongodb://host1/?authSource=db1&authMechanism=SCRAM-SHA-1
# mongo-cr
mongodb://host1/?authSource=db1&authMechanism=MONGODB-CR
# x.509
mongodb://host1/?authSource=db1&authMechanism=MONGODB-X509
# kerberos
mongodb://host1/?authSource=db1&authMechanism=GSSAPI
# ldap
mongodb://host1/?authSource=db1&authMechanism=PLAIN
```

Configurations

Configurations parameters:

`connect.mongo.db`

The target MongoDB database name.

- Data type: string
- Optional : no

`connect.mongo.connection`

The mongodb endpoints connections in the format `mongodb://host1[:port1][,host2[:port2],...[,hostN[:portN]]]/[database][?options]`

- Data type: string
- Optional : no

Note: Setting username and password in the endpoints is not secure, they will be pass to Connect as plain text before being given to the driver. Use the `connect.mongo.username` and `connect.mongo.password` options.

`connect.mongo.batch.size`

The number of records the sink would push to mongo at once (improved performance)

- Data type: int
- Optional : yes
- Default: 100

`connect.mongo.kcql`

Kafka connect query language expression. Allows for expressive topic to collectionrouting, field selection and renaming.

Examples:

```
INSERT INTO TABLE1 SELECT * FROM TOPIC1;INSERT INTO TABLE2 SELECT field1, field2, ↵
↵field3 as renamedField FROM TOPIC2
```

- Data Type: string
- Optional : no

`connect.mongo.username`

The username to use for authentication. If the username is set it overrides the username/password set in the connection string and the `connect.mongo.auth.mechanism` has precedence.

- Data Type: string
- Option: yes
- Default:

`connect.mongo.password`

The password to use for authentication.

- Data Type: string
- Optional: yes
- Default:

`connect.mongo.auth.mechanism`

The mechanism to use for authentication. GSSAPI (Kerberos), PLAIN (LDAP), X.509 or SCRAM-SHA-1.

- Data Type: string
- Optional: yes
- Default: SCRAM-SHA-1

`connect.mongo.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **NOOP**, the error is swallowed, **THROW**, the error is allowed to propagate and retry. For **RETRY** the Kafka message is redelivered up to a maximum number of times specified by the `connect.mongo.max.retires` option. The `connect.mongo.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Default: throw

`connect.mongo.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.mongo.error.policy` is set to `TRHOW`.

- Type: string
- Importance: high
- Default: 10

`connect.mongo.retry.interval`

The interval, in milliseconds between retries if the sink is using `connect.mongo.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=mongo-sink-orders
connector.class=com.datamountaineer.streamreactor.connect.mongodb.sink.
↳MongoSinkConnector
tasks.max=1
topics=orders-topic
connect.mongo.kcql=INSERT INTO orders SELECT * FROM orders-topic
connect.mongo.db=connect
connect.mongo.connection=mongodb://localhost:27017
connect.mongo.batch.size=10
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect, bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect MQTT Sink

A Connector and Sink to stream messages from Kafka to a MQTT brokers.

The Sink supports:

1. *The [KCQL routing querying](#)* - Topic to measure mapping and Field selection.
2. Error policies.
3. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema

The Sink supports three Kafka payloads type:

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example

[here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires the SchemaRegistry which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should be set to `value.converter=org.apache.kafka.connect.json.JsonConverter`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a GenericRecord, to just send a message with Schema.String and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support

The payload of the MQTT message sent to the MQTT broker is sent as json.

Prerequisites

- Confluent 3.3
- Java 1.8
- Scala 2.11
- Mqtt server

Setup

Confluent Setup

Follow the instructions [here](#).

MQTT Setup

HiveMQ, who provide enterprise MQTT brokers have a webclient we'll use for the quickstart. Go to this [link](#), connect and set up a subscription to `/datamountaineer/mqtt_sink_topic/+`` topic. You should see this.



Connection
● connected

Host
Port
ClientID

Disconnect

Username
Password
Keep Alive
SSL
☐
Clean Session
☒

Last-Will Topic
Last-Will QoS
Last-Will Retain
☐

Last-Will Message

Publish

Messages

Subscriptions

Add New Topic Subscription

Qos: 2
☒

Now we have a webclient listening for messages which will come from the MQTT Sink.

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our [GitHub releases](#) page.

Starting the Connector

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for MQTT. If you are using the `dockers` you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create mqtt-sink < conf/mqtt-sink.properties

#Connector name=`mqtt-sink`
name=mqtt-sink
tasks.max=1
```

```
connector.class=com.datamountaineer.streamreactor.connect.mqtt.sink.MqttSinkConnector
topics=kafka-topic
connect.mqtt.hosts=tcp://broker.mqttdashboard.com:1883
connect.mqtt.kcql=INSERT INTO /datamountaineer/mqtt_topic SELECT * FROM kafka-topic
connect.mqtt.clean=true
connect.mqtt.timeout=1000
connect.mqtt.keep.alive=1000
connect.mqtt.client.id=dm_sink_id,
connect.mqtt.service.quality=1
tasks.max=1
#task ids: 0
```

The `mqtt-sink.properties` file defines:

1. The name of the sink.
2. The name number of tasks.
3. The class containing the connector.
4. The url of the HiveMQ public Server and port to connect to.
5. *The KCQL routing querying*.. This specifies the target topic on the Mqtt server and the source kafka topics.
6. The topics to source (Required by Connect Framework).

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

```
INFO Kafka commitId : 5cdaa94d0a69e0d (org.apache.kafka.common.utils.  
↪AppInfoParser:84)  
INFO Setting task configurations for 1 workers. (com.datamountaineer.streamreactor.  
↪connect.mqtt.sink.MqttSinkConnector:52)  
INFO Finished starting connectors and tasks (org.apache.kafka.connect.runtime.  
↪distributed.DistributedHerder:825)  
INFO  
  
      _/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_  
    /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/  
   //__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//  
  //__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//_  
 /___//___//___//___//___//___//___//___//___//___//___//___//___//___//  
  
     _/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_  
    /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/  
   //__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//  
  //__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//__//_  
 /___//___//___//___//___//___//___//___//___//___//___//___//___//___//  
  
                               By Andrew Stevenson  
                              ,<  
                          /___//___//___//___//___//___//___//___//___//___//___//___//  
        (com.datamountaineer.streamreactor.connect.mqtt.sink.MqttSinkTask:41)  
INFO MqttSinkConfig values:  
  connect.mqtt.clean = true  
  connect.mqtt.client.id = dm_sink_id,  
  connect.mqtt.converter.throw.on.error = false  
  connect.mqtt.error.policy = THROW  
  connect.mqtt.hosts = tcp://broker.mqttpdashboard.com:1883  
  connect.mqtt.kcql = INSERT INTO /datamountaineer/mqtt_topic SELECT * FROM kafka-  
↪topic
```

```

connect.mqtt.keep.alive = 1000
connect.mqtt.max.retries = 20
connect.mqtt.password = null
connect.mqtt.retry.interval = 60000
connect.mqtt.service.quality = 1
connect.mqtt.ssl.ca.cert = null
connect.mqtt.ssl.cert = null
connect.mqtt.ssl.key = null
connect.mqtt.timeout = 1000
connect.mqtt.username = null
connect.progress.enabled = true
(com.datamountaineer.streamreactor.connect.mqtt.config.MqttSinkConfig:223)

```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```

#check for running connectors with the CLI
bin/connect-cli ps
mqtt-sink

```

Test Records

Now we need to put some records it to the kafka_topic topics. We can use the kafka-avro-console-producer to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a firstname field of type string, a lastname field of type string, an age field of type int and a salary field of type double.

```

${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
  --broker-list localhost:9092 --topic kafka-topic \
  --property value.schema='{ "type": "record", "name": "User", "namespace": "com.
  ↳ datamountaineer.streamreactor.connect.mqtt"
  , "fields": [{ "name": "firstName", "type": "string"}, { "name": "lastName", "type": "string"},
  ↳ { "name": "age", "type": "int"}, { "name": "salary", "type": "double"} ] }'

```

Now the producer is waiting for input. Paste in the following:


```

{"firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830}

```

Check for Records in the MQTT Broker

Go back to browser you started the HiveMQ webclient in. You should see the messages arrive in the messages section.


Websockets Client Showcase

Connection ● connected ⬆

Host

Port

ClientID

Disconnect

Username

Password

Keep Alive

SSL
☐

Clean Session
☒

Last-Will Topic

Last-Will QoS

Last-Will Retain
☐

Last-Will Message

Publish ⬇

Messages ⬆

2017-08-29 18:38:58 Topic: /datamountaineer/mqtt_topic Qos: 1
{"firstName":"John","lastName":"Smith","age":30,"salary":4830.0}

Subscriptions ⬆

Add New Topic Subscription

Qos: 2 ✕
/datamountaineer/m...

Features

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The CoAP Sink supports the following:

```
INSERT INTO <resource> SELECT <fields> FROM <source topic>
```

Example:

```
#Insert mode, select all fields from topicA and write to topicA
INSERT INTO topicA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to topicA
INSERT INTO topicA SELECT x AS a, y AS b and z AS c FROM topicB
```

This is set in the `connect.mqtt.kcql` option.

Error Polices

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more

information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers.

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.mqtt.max.retries` and the `connect.mqtt.retry.interval`.

Configurations

`connect.mqtt.kcql`

Kafka connect query language expression. Allows for expressive Mqtt topic to Kafka topic routing. Currently there is no support for filtering the fields from the incoming payload.

- Data type : string
- Importance: high
- Optional : no

`connect.mqtt.hosts`

Specifies the mqtt connection endpoints.

- Data type : string
- Importance: high
- Optional : no

Example:

```
tcp://broker.datamountaineer.com:1883
```

`connect.mqtt.service.quality`

The Quality of Service (QoS) level is an agreement between sender and receiver of a message regarding the guarantees of delivering a message. There are 3 QoS levels in MQTT: At most once (0); At least once (1); Exactly once (2).

- Data type : int
- Importance: high
- Optional : yes

- Default: 1

`connect.mqtt.username`

Contains the Mqtt connection user name

- Data type : string
- Importance: medium
- Optional : yes
- Default: null

`connect.mqtt.password`

Contains the Mqtt connection password

- Data type : string
- Importance: medium
- Optional : yes
- Default: null

`connect.mqtt.client.id`

Provides the client connection identifier. If is not provided the framework will generate one.

- Data type: string
- Importance: medium
- Optional: yes
- Default: generated

`connect.mqtt.connection.timeout`

Sets the timeout to wait for the broker connection to be established

- Data type: int
- Importance: medium
- Optional: yes
- Default: 3000 (ms)

`connect.mqtt.connection.clean`

The clean session flag indicates the broker, whether the client wants to establish a persistent session or not. A persistent session (the flag is false) means, that the broker will store all subscriptions for the client and also all missed messages, when subscribing with Quality of Service (QoS) 1 or 2. If clean session is set to true, the broker won't store anything for the client and will also purge all information from a previous persistent session.

- Data type: boolean
- Importance: medium
- Optional: yes
- Default: true

`connect.mqtt.connection.keep.alive`

The keep alive functionality assures that the connection is still open and both broker and client are connected to one another. Therefore the client specifies a time interval in seconds and communicates it to the broker during the

establishment of the connection. The interval is the longest possible period of time, which broker and client can endure without sending a message.

- Data type: int
- Importance: medium
- Optional: yes
- Default: 5000

`connect.mqtt.connection.ssl.ca.cert`

Provides the path to the CA certificate file to use with the Mqtt connection

- Data type: string
- Importance: medium
- Optional: yes
- Default: null

`connect.mqtt.connection.ssl.cert`

Provides the path to the certificate file to use with the Mqtt connection

- Data type: string
- Importance: medium
- Optional: yes
- Default: null

`connect.mqtt.connection.ssl.key`

Certificate private key file path.

- Data type: string
- Importance: medium
- Optional: yes
- Default: null

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.

3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect, bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect Redis

A Connector and Sink to write events from Kafka to Redis. The connector takes the value from the Kafka Connect SinkRecords and inserts a new entry to Redis.

The Sink supports:

1. *The **KCQL routing querying*** - Kafka topic payload field selection is supported, allowing you to select fields written to Redis.
2. Topic to table routing via KCQL.
3. RowKey selection - Selection of fields to use as the row key, if none specified the topic name, partition and offset are used via KCQL.
4. Error policies for handling failures.
5. Storing as one or more Stored Sets.

Prerequisites

- Confluent 3.3
- Jedis 2.8.1
- Java 1.8
- Scala 2.11

Setup

Redis Setup

Download and install Redis.

```
wget http://download.redis.io/redis-stable.tar.gz
tar xvzf redis-stable.tar.gz
cd redis-stable
sudo make install
```

Start Redis

```
bin/redis-server
```

Check Redis is running:

```
redis-cli ping
PONG
sudo service redis-server status
```

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for Redis. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create redis-sink < conf/redis-sink.properties
#Connector name=`redis-sink`
connect.redis.host=localhost
connect.redis.port=6379
connector.class=com.datamountaineer.streamreactor.connect.redis.sink.
↳RedisSinkConnector
tasks.max=1
topics=redis-topic
```

```
connect.redis.kcql=INSERT INTO TABLE1 SELECT * FROM redis-topic
#task ids: 0
```

The `redis-sink.properties` file defines:

1. The name of the sink.
2. The name of the redis host to connect to.
3. The redis port to connect to.
4. The Sink class.
5. The max number of tasks the connector is allowed to created. Should not be greater than the number of partitions in the Source topics otherwise tasks will be idle.
6. The Source kafka topics to take events from.
7. *The KCQL routing querying.*

Warning: If your redis server is requiring the connection to be authenticated you will need to provide an extra setting:

```
connect.redis.connection.password=$REDIS_PASSWORD
```

Don't set the value to empty if no password is required.

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
redis-sink
```

```
[2016-05-08 22:37:05,616] INFO
```

```
/ _ \_ / / _ _ / _ | / _ _ _ _ / / _ _ ( ) _ _ _ _ _  
/ / / _ _ \ _ _ \ / | / / _ V / / / _ V _ _ \ / _ V _ V _ V _  
/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/  
/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/  
  
/ _ \_ _ _ / ( ) _ _ _ / _ _ / ( ) _ _ / / _  
/ / / _ V _ _ / / _ _ / \ _ V / _ V // _/  
/_ , _ / _ / / / ( ) _ _ / / / / / , <  
/_ / | | \ _ / \ _ , _ / _ _ // _ _ / _ / / _ / _ / | |
```

```
(com.datamountaineer.streamreactor.connect.redis.sink.config.RedisSinkConfig:165)
```

```
[2016-05-08 22:37:05,641] INFO Settings:  
RedisSinkSettings(RedisConnectionInfo(localhost, 6379, None), RedisKey(FIELDS,  
↳ WrappedArray(firstName, lastName)), PayloadFields(false, Map(firstName -> firstName, _  
↳ lastName -> lastName, age -> age, salary -> income)))
```

```
(com.datamountaineer.streamreactor.connect.redis.sink.RedisSinkTask:65)
[2016-05-08 22:37:05,687] INFO Sink task org.apache.kafka.connect.runtime.
↪WorkerSinkTask@44b24eaa finished initialization and start (org.apache.kafka.connect.
↪runtime.WorkerSinkTask:155)
```

Test Records

Now we need to put some records it to the test_table topics. We can use the kafka-avro-console-producer to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a firstname field of type string, a lastname field of type string, an age field of type int and a salary field of type double.

```
${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic redis-topic \
--property value.schema='{ "type": "record", "name": "User",
"fields": [{ "name": "firstName", "type": "string"}, { "name": "lastName", "type": "string"}, {
↪"name": "age", "type": "int"}, { "name": "salary", "type": "double"} ] }'
```

Now the producer is waiting for input. Paste in the following:

```
{ "firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830 }
```

Check for records in Redis

Now check the logs of the connector you should see this:

```
INFO Received record from topic:redis-topic partition:0 and offset:0 (com.
↪datamountaineer.streamreactor.connect.redis.sink.writer.RedisDbWriter:48)
INFO Empty list of records received. (com.datamountaineer.streamreactor.connect.redis.
↪sink.RedisSinkTask:75)
```

Check in Redis.

```
redis-cli

127.0.0.1:6379> keys *
1) "John.Smith"
2) "11"
3) "10"
127.0.0.1:6379>
127.0.0.1:6379> get "John.Smith"
"{ \"firstName\": \"John\", \"lastName\": \"Smith\", \"age\": 30, \"income\": 4830.0 }"
```

Now stop the connector.

Features

The Redis Sink writes records from Kafka to Redis.

The Sink supports:

1. Field selection - Kafka topic payload field selection is supported, allowing you to select fields written to Redis.

2. Topic to table routing.
3. RowKey selection - Selection of fields to use as the row key, if none specified the topic name, partition and offset are used.
4. Error policies for handling failures.
5. Sorted sets/Cache modes

Cache Mode

The purpose of this mode is to **cache** in Redis [*Key-Value*] pairs. Imagine a Kafka topic with currency foreign exchange rate messages:

```
{ "symbol": "USDGBP" , "price": 0.7943 }  
{ "symbol": "EURGBP" , "price": 0.8597 }
```

You may want to store in Redis: the **symbol** as the *Key* and the **price** as the *Value*. This will effectively make Redis a **caching** system, which multiple other application can access to get the (*latest*) value. To achieve that using this particular Kafka Redis Sink Connector, you need to specify the **KCQL** as:

```
SELECT price FROM yahoo-fx PK symbol
```

This will update the keys *USDGBP* , *EURGBP* with the relevant price using the (default) Json format:

```
Key=EURGBP Value={ "price": 0.7943 }
```

We can prefix the name of the *Key* using the INSERT statement for Multiple SortedSets:

```
INSERT INTO FX- SELECT price FROM yahoo-fx PK symbol STOREAS_  
↳ SortedSet (score=timestamp)
```

This will create key with names *FX-USDGBP* , *FX-EURGBP* etc.

Sorted Sets

To **insert** messages from a Kafka topic into 1 Sorted Set (SS) use the following **KCQL** syntax:

```
INSERT INTO cpu_stats SELECT * FROM cpuTopic STOREAS SortedSet(score=timestamp)
```

This will create and add entries into the (sorted set) named **cpu_stats**. The entries will be ordered in the Redis set based on the *score* that we define it to be the value of the *timestamp* field of the Avro message from Kafka. In the above example we are selecting and storing all the fields of the Kafka message.

Multiple Sorted Sets

You can create multiple sorted sets by promoting each value of **one field** from the Kafka message into one Sorted Set (SS) and selecting which values to store into the sorted-sets. You can achieve that by using the KCQL syntax and defining with the field using **PK** (primary key)

```
INSERT INTO cpu_stats SELECT temperature, humidity FROM sensorsTopic PK_  
↳ sensorID STOREAS SortedSet (score=timestamp)
```

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option. This is set in the `connect.redis.kcql` option.

The Redis Sink supports the following:

```
INSERT INTO cache|sortedSet SELECT * FROM TOPIC [PK FIELD] [STOREAS ↵
↵SortedSet (key=FIELD) ]
```

```
#insert messages from a Kafka topic into 1 Sorted Set (SS) named cpuTopic and ordered ↵
↵by score with the value of the timestamp field in the message
INSERT INTO cpu_stats SELECT * from cpuTopic STOREAS SortedSet (score=timestamp)

#insert into multiple sorted sets by setting the PK key word to select a field from ↵
↵the message as a primary key
INSERT INTO cpu_stats SELECT temperature, humidity FROM sensorsTopic PK sensorID ↵
↵STOREAS SortedSet (score=timestamp)
```

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other expectations thrown by drivers.

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.redis.max.retries` and the `connect.redis.retry.interval`.

Configurations

`connect.redis.kcql`

Kafka connect query language expression. Allows for expressive topic to table routing, field selection and renaming. Fields to be used as the row key can be set by specifying the PK. The below example uses `field1` as the primary key.

- Data type : string

- Importance: high
- Optional : no

Examples:

```
INSERT INTO TABLE1 SELECT * FROM TOPIC1;INSERT INTO TABLE2 SELECT * FROM TOPIC2 PK_
↪field1
```

Examples:

```
INSERT INTO TABLE1 SELECT * FROM TOPIC1;INSERT INTO TABLE2 SELECT * FROM TOPIC2 PK_
↪field1, field2
```

`connect.redis.host`

Specifies the Redis server.

- Data type : string
- Importance: high
- Optional : no

`connect.redis.port`

Specifies the Redis server port number.

- Data type : int
- Importance: high
- Optional : no

`connect.redis.password`

Specifies the authorization password.

- Data type : string
- Importance: high
- Optional : yes
- Description: If you don't have a password set up on the redis server don't provide the value or you will see this error: "ERR Client sent AUTH, but no password is set"

`connect.redis.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.redis.max.retries` option. The `connect.redis.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: medium
- Optional: yes
- Default: RETRY

`connect.redis.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.redis.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional: yes
- Default: 10

`connect.redis.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.redis.error.policy` set to **RETRY**.

- Type: int
- Importance: high
- Optional: no
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=redis-sink
connect.redis.host=localhost
connect.redis.port=6379
connector.class=com.datamountaineer.streamreactor.connect.redis.sink.
↳RedisSinkConnector
tasks.max=1
topics=redis-topic
connect.redis.kcql=INSERT INTO TABLE1 SELECT * FROM redis-topic
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

The Redis Sink will automatically write and update the Redis table if new fields are added to the Source topic, if fields are removed the Kafka Connect framework will return the default value for this field, dependent of the compatibility settings of the Schema registry. This value will be put into the Redis column family cell based on the `connect.redis.kcql` mappings.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect`, `bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect ReThink

A Connector and Sink to write events from Kafka to RethinkDb. The connector takes the value from the Kafka Connect SinkRecords and inserts a new entry to RethinkDb.

The Sink supports:

1. *The KCQL routing querying* - Kafka topic payload field selection is supported, allowing you to select fields written to RethinkDb.
2. Topic to table routing via KCQL.
3. RowKey selection - Selection of fields to use as the row key, if none specified the topic name, partition and offset are used via KCQL.
4. RethinkDB write modes via KCQL.
5. Error policies for handling failures.

6. Payload support for Schema.Struct and payload Struct, Schema.String and Json payload and Json payload with no schema

The Sink supports three Kafka payloads type:

Connect entry with Schema.Struct and payload Struct. If you follow the best practice while producing the events, each message should carry its schema information. Best option is to send Avro. Your connect configurations should be set to `value.converter=io.confluent.connect.avro.AvroConverter`. You can find an example [here](#). To see how easy is to have your producer serialize to Avro have a look at [this](#). This requires the SchemaRegistry which is open source thanks to Confluent! Alternatively you can send Json + Schema. In this case your connect configuration should be set to `value.converter=org.apache.kafka.connect.json.JsonConverter`. This doesn't require the SchemaRegistry.

Connect entry with Schema.String and payload json String. Sometimes the producer would find it easier, despite sending Avro to produce a GenericRecord, to just send a message with Schema.String and the json string.

Connect entry without a schema and the payload json String. There are many existing systems which are publishing json over Kafka and bringing them in line with best practices is quite a challenge. Hence we added the support.

Prerequisites

- Confluent 3.3
- RethinkDb 2.3.3
- Java 1.8
- Scala 2.11

Setup

Rethink Setup

Download and install RethinkDb. Follow the instruction [here](#) dependent on your operating system.

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

We you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools` *cli* to post in our distributed properties file for ReThinkDB. If you are using the *dockers* you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create rethink-sink < rethink-sink.properties
#Connector name=`rethink-sink`
name=rethink-sink
connect.rethink.db=dbname
connect.rethink.host=localhost
connect.rethink.port=28015
connector.class=com.datamountaineer.streamreactor.connect.rethink.sink.
↳ ReThinkSinkConnector
tasks.max=1
topics=rethink-topic
connect.rethink.kcql=INSERT INTO TABLE1 SELECT * FROM rethink_topic
#task ids: 0
```

The `rethink-sink.properties` file defines:

1. The name of the sink.
2. The name of the rethink database.
3. The name of the rethink host to connect to.
4. The rethink port to connect to.
5. The Sink class.
6. The max number of tasks the connector is allowed to created. Should not be greater than the number of partitions in the Source topics otherwise tasks will be idle.
7. The Source kafka topics to take events from.
8. *The KCQL routing querying.*

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
rethink-sink
```

[2016-05-08 22:37:05,616] INFO

Now we need to put some records it to the `test_table` topics. We can use the `kafka-avro-console-producer` to do this.

```
$_(CONFLUENT_HOME)/bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic rethink_topic \
--property value.schema='{ "type": "record", "name": "User", "namespace": "com.
↳ datamountaineer.streamreactor.connect.rethink"
, "fields": [{ "name": "firstName", "type": "string" }, { "name": "lastName", "type": "string" },
↳ { "name": "age", "type": "int" }, { "name": "salary", "type": "double" } ] }'
```

```
{"firstName": "John", "lastName": "Smith", "age":30, "salary": 4830}
```

Now check the logs of the connector you should see this:

```
INFO Received record from topic:person_rethink partition:0 and offset:0 (com.
↳ datamountaineer.streamreactor.connect.rethink.sink.writer.rethinkDbWriter:48)
INFO Empty list of records received. (com.datamountaineer.streamreactor.connect.
↳ rethink.sink.RethinkSinkTask:75)
```

Now stop the connector.

The ReThinkDb Sink writes records from Kafka to RethinkDb.

1. Field selection - Kafka topic payload field selection is supported, allowing you to select fields written to RethinkDB.
2. Topic to table routing.
3. RowKey selection - Selection of fields to use as the row key, if none specified the topic name, partition and offset are used.
4. RethinkDB write modes.
5. Error policies for handling failures.

Kafka Connect Query Language

Kafka Connect Query Language found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The ReThink Sink supports the following:

```
<write mode> INTO <target table> SELECT <fields> FROM <source topic> <AUTOCREATE> <PK_  
↪FIELD>
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA  
INSERT INTO tableA SELECT * FROM topicA  
  
#Insert mode, select 3 fields and rename from topicB and write to tableB  
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB  
  
#Upsert mode, select all fields from topicC, auto create tableC and auto evolve, use_  
↪field1 as the primary key  
UPSERT INTO tableC SELECT * FROM topicC AUTOCREATE PK field1
```

Write Modes

The Sink support two write modes **insert** and **upsert** which map to RethinkDb's conflict policies, **insert** to **ERROR** and **upsert** to **REPLACE**.

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers.

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.rethink.max.retries` and the `connect.rethink.retry.interval`.

Topic Routing

The Sink supports topic routing that allows mapping the messages from topics to a specific table. For example, map a topic called “bloomberg_prices” to a table called “prices”. This mapping is set in the `connect.rethink.kcql` option.

Example:

```
//Select all
INSERT INTO table1 SELECT * FROM topic1; INSERT INTO tableA SELECT * FROM topicC
```

Field Selection

The ReThink Sink supports field selection and mapping. This mapping is set in the `connect.rethink.kcql` option.

Examples:

```
//Rename or map columns
INSERT INTO table1 SELECT lst_price AS price, qty AS quantity FROM topicA

//Select all
INSERT INTO table1 SELECT * FROM topic1
```

Tip: Check you mappings to ensure the target columns exist.

Auto Create Tables

The Sink supports auto creation of tables for each topic. This mapping is set in the `connect.rethink.kcql` option.

A user specified primary can be set in the PK clause for the `connect.rethink.kcql` option. Only one key is supported. If more than one is set only the first is used. If no primary keys are set the default primary key called `id` is used. The value for the default key is the topic name, partition and offset of the records.

```
#AutoCreate the target table
INSERT INTO table1 SELECT * FROM topic AUTOCREATE PK field1
```

Note: The fields specified as the primary keys must be in the SELECT clause or all fields must be selected

The Sink will try and create the table at start up if a schema for the topic is found in the Schema Registry. If no schema is found the table is created when the first record is received for the topic.

Error Policies

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.rethink.max.retries` and the `connect.cassandra.retry.interval`.

Configurations

`connect.rethink.kcql`

Kafka connect query language expression. Allows for expressive topic to table routing, field selection and renaming. Fields to be used as the row key can be set by specifying the PK. The below example uses `field1` as the primary key.

- Data type : string
- Importance: high
- Optional : no

Examples:

```
INSERT INTO TABLE1 SELECT * FROM TOPIC1; INSERT INTO TABLE2 SELECT * FROM TOPIC2 PK_
↪field1
```

`connect.rethink.host`

Specifies the rethink server.

- Data type : string
- Importance: high
- Optional : no

`connect.rethink.port`

Specifies the rethink server port number.

- Data type : int
- Importance: high
- Optional : yes

`connect.rethink.db`

Specifies the rethink database to connect to.

- Data type : string
- Importance: high
- Optional : yes
- Default : connect_rethink_sink

`connect.rethink.cert.file`

Certificate file to connect to a TLS enabled ReThink cluster. **Cannot** be used in conjunction with username/password. `connect.rethink.auth.key` must be set.

- Data type: string
- Optional : yes

`connect.rethink.auth.key`

Authentication key to connect to a TLS enabled ReThink cluster. **Cannot** be used in conjunction with username/password. `connect.rethink.cert.file` must be set.

- Data type: string
- Optional : yes

`connect.rethink.username`

Username to connect to ReThink with.

- Data type: string
- Optional : yes

`connect.rethink.password`

Password to connect to ReThink with.

- Data type: string
- Optional : yes

`connect.rethink.ssl.enabled`

Enables SSL communication against an SSL enabled Rethink cluster.

- Data type: boolean
- Optional : yes
- Default : false

`connect.rethink.trust.store.password`

Password for truststore.

- Data type: string
- Optional : yes

`connect.rethink.key.store.path`

Path to truststore.

- Data type: string
- Optional : yes

`connect.rethink.key.store.password`

Password for key store.

- Data type: string
- Optional : yes

`connect.rethink.ssl.client.cert.auth`

Path to keystore.

- Data type: string
- Optional : yes

`connect.rethink.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.rethink.max.retries` option. The `connect.rethink.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Optional : yes
- Default: RETRY

`connect.rethink.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.rethink.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional : yes
- Default: 10

`connect.rethink.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.rethink.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional : yes
- Default : 60000 (1 minute)

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Example

```
name=rethink-sink
connect.rethink.db=dbname
connect.rethink.host=localhost
connect.rethink.port=28015
connector.class=com.datamountaineer.streamreactor.connect.rethink.sink.
↳ ReThinkSinkConnector
tasks.max=1
topics=person_rethink
connect.rethink.kcql=INSERT INTO TABLE1 SELECT * FROM person_rethink
```

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

The rethink Sink will automatically write and update the rethink table if new fields are added to the Source topic, if fields are removed the Kafka Connect framework will return the default value for this field, dependent of the compatibility settings of the Schema registry.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect, bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

Kafka Connect VoltDB

A Connector and Sink to write events from Kafka to VoltDB. The connector used the built in stored procedures for inserts and upserts but requires the tables to be pre-created.

The Sink supports:

1. *The [KCQL routing querying](#)* - Kafka topic payload field selection is supported, allowing you to select fields written to VoltDB.
2. Topic to table routing via KCQL.
3. Voltdb write modes, upsert and insert via KCQL.
4. Error policies for handling failures.

Prerequisites

- Confluent 3.3
- VoltDB 6.4
- Java 1.8
- Scala 2.11

Setup

VoltDB Setup

Download VoltDB from [here](#)

Unzip the archive

```
tar -xzf voltdb-ent-*.tar.gz
```

Start VoltDB:

```
cd voltdb-ent-*
bin/voltdb create

Build: 6.5 voltdb-6.5-0-gd1fe3fa-local Enterprise Edition
Initializing VoltDB...

  _ _ _ _ _
 | | / / _ _ / / / _ _ V / _ _ )
 | | / / _ _ V / _ _ / / / _ _ |
 | | / / _ _ / / _ _ / / _ _ /
```

```
|___/\___/\___/\___/\___/\___/\
-----

Connecting to VoltDB cluster as the leader...
Host id of this node is: 0
Starting VoltDB with trial license. License expires on Sep 11, 2016.
Initializing the database and command logs. This may take a moment...
WARN: This is not a highly available cluster. K-Safety is set to 0.
```

Confluent Setup

Follow the instructions [here](#).

Sink Connector QuickStart

When you start the Confluent Platform, Kafka Connect is started in distributed mode (`confluent start`). In this mode a Rest Endpoint on port 8083 is exposed to accept connector configurations. We developed Command Line Interface to make interacting with the Connect Rest API easier. The CLI can be found in the Stream Reactor download under the `bin` folder. Alternatively the Jar can be pulled from our GitHub [releases](#) page.

Create Voltdb Table

At present the Sink doesn't support auto creation of tables so we need to login to VoltDb to create one. In the directory you extracted Voltdb start the `sqlcmd` shell and enter the following DDL statement. This creates a table called `person`.

```
create table person(firstname varchar(128), lastname varchar(128), age int, salary_
↪float, primary key (firstname, lastname));
```

```
bin ./sqlcmd
SQL Command :: localhost:21212
1> create table person(firstname varchar(128), lastname varchar(128), age int, salary_
↪float, primary key (firstname, lastname));
Command succeeded.
2>
```

Starting the Connector (Distributed)

Download, unpack and install the Stream Reactor and Confluent. Follow the instructions [here](#) if you haven't already done so. All paths in the quickstart are based in the location you installed the Stream Reactor.

Once the Connect has started we can now use the `kafka-connect-tools cli` to post in our distributed properties file for VoltDB. If you are using the [dockers](#) you will have to set the following environment variable to for the CLI to connect to the Rest API of Kafka Connect of your container.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

```
bin/connect-cli create voltdb-sink < conf/voltdb-sink.properties

#Connector `voltdb-sink`:
```

```
name=voltddb-sink
connector.class=com.datamountaineer.streamreactor.connect.voltddb.VoltSinkConnector
max.tasks=1
topics=sink-test
connect.volt.servers=localhost:21212
connect.volt.kcql=INSERT INTO person SELECT * FROM sink-test
connect.volt.password=
connect.volt.username=
#task ids:
```

The `volt-db-sink.properties` file defines:

1. The name of the sink.
2. The Sink class.
3. The max number of tasks the connector is allowed to created.
4. The topics to read from (Required by framework)
5. The name of the voltdb host to connect to.
6. Username to connect as.
7. The password for the username.
8. *The KQQL routing querying.*

Use the Confluent CLI to view Connects logs.

```
# Get the logs from Connect
confluent log connect

# Follow logs from Connect
confluent log connect -f
```

We can use the CLI to check if the connector is up but you should be able to see this in logs as-well.

```
#check for running connectors with the CLI
bin/connect-cli ps
voltdb-sink
```

```
[2016-08-21 20:31:36,398] INFO Finished starting connectors and tasks (org.apache.  
↳kafka.connect.runtime.distributed.DistributedHerder:769)  
[2016-08-21 20:31:36,406] INFO  
  
by Stefan Bocutiu  
  
(com.datamountaineer.streamreactor.connect.voltddb.VoltSinkTask:44)  
[2016-08-21 20:31:36,407] INFO VoltSinkConfig values:  
connect.volt.error.policy = THROW
```

```

connect.volt.retry.interval = 60000
connect.volt.kcql = INSERT INTO person SELECT * FROM sink-test
connect.volt.max.retires = 20
connect.volt.servers = localhost:21212
connect.volt.username =
connect.volt.password =
(com.datamountaineer.streamreactor.connect.voltdb.config.VoltSinkConfig:178)
[2016-08-21 20:31:36,501] INFO Settings:com.datamountaineer.streamreactor.connect.
↪voltdb.config.VoltSettings$@34c34c3e (com.datamountaineer.streamreactor.connect.
↪voltdb.VoltSinkTask:71)
[2016-08-21 20:31:36,565] INFO Connecting to VoltDB... (com.datamountaineer.
↪streamreactor.connect.voltdb.writers.VoltConnectionConnectFn$:28)
[2016-08-21 20:31:36,636] INFO Connected to VoltDB node at: localhost:21212 (com.
↪datamountaineer.streamreactor.connect.voltdb.writers.VoltConnectionConnectFn$:46)

```

Test Records

Now we need to put some records it to the test_table topics. We can use the kafka-avro-console-producer to do this.

Start the producer and pass in a schema to register in the Schema Registry. The schema has a firstname field of type string a lastname field of type string, an age field of type int and a salary field of type double.

```

${CONFLUENT_HOME}/bin/kafka-avro-console-producer \
  --broker-list localhost:9092 --topic sink-test \
  --property value.schema='{ "type": "record", "name": "User", "namespace": "com.
↪datamountaineer.streamreactor.connect.voltdb"
  , "fields": [{"name": "firstName", "type": "string"}, {"name": "lastName", "type": "string"},
↪{"name": "age", "type": "int"}, {"name": "salary", "type": "double"}] } '

```

Now the producer is waiting for input. Paste in the following:

```
{ "firstName": "John", "lastName": "Smith", "age": 30, "salary": 4830 }
```

Check for records in VoltDb

Now check the logs of the connector you should see this:

```

[2016-08-21 20:41:25,361] INFO Writing complete (com.datamountaineer.streamreactor.
↪connect.voltdb.writers.VoltDbWriter:61)
[2016-08-21 20:41:25,362] INFO Records handled (com.datamountaineer.streamreactor.
↪connect.voltdb.VoltSinkTask:86)

```

In VoltDb sqlcmd terminal

```

SELECT * FROM PERSON;

FIRSTNAME  LASTNAME  AGE  SALARY
-----
John       Smith     30   4830.0

(Returned 1 rows in 0.01s)

```

Now stop the connector.

Features

The Sink supports:

1. Field selection - Kafka topic payload field selection is supported, allowing you to select fields written to VoltDB.
2. Topic to table routing.
3. Voltdb write modes, upsert and insert.
4. Error policies for handling failures.

Kafka Connect Query Language

Kafka **C**onnect **Q**uery **L**anguage found here [GitHub repo](#) allows for routing and mapping using a SQL like syntax, consolidating typically features in to one configuration option.

The Voltdb Sink supports the following:

```
INSERT INTO <table> SELECT <fields> FROM <source topic>
UPSERT INTO <table> SELECT <fields> FROM <source topic>
```

Example:

```
#Insert mode, select all fields from topicA and write to tableA
INSERT INTO tableA SELECT * FROM topicA

#Insert mode, select 3 fields and rename from topicB and write to tableB
INSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB

#Upsert mode, select 3 fields and rename from topicB and write to tableB
UPSERT INTO tableB SELECT x AS a, y AS b and z AS c FROM topicB
```

This is set in the `connect.volt.kcql` option.

Error Polices

The Sink has three error policies that determine how failed writes to the target database are handled. The error policies affect the behaviour of the schema evolution characteristics of the sink. See the schema evolution section for more information.

Throw

Any error on write to the target database will be propagated up and processing is stopped. This is the default behaviour.

Noop

Any error on write to the target database is ignored and processing continues.

Warning: This can lead to missed errors if you don't have adequate monitoring. Data is not lost as it's still in Kafka subject to Kafka's retention policy. The Sink currently does **not** distinguish between integrity constraint violations and or other exceptions thrown by drivers..

Retry

Any error on write to the target database causes the `RetryIterable` exception to be thrown. This causes the Kafka connect framework to pause and replay the message. Offsets are not committed. For example, if the table is offline it

will cause a write failure, the message can be replayed. With the Retry policy the issue can be fixed without stopping the sink.

The length of time the Sink will retry can be controlled by using the `connect.volt.max.retries` and the `connect.volt.retry.interval`.

Topic Routing

The Sink supports topic routing that allows mapping the messages from topics to a specific table. For example, map a topic called “bloomberg_prices” to a table called “prices”. This mapping is set in the `connect.volt.kcql` option.

Example:

```
//Select all
INSERT INTO table1 SELECT * FROM topic1; INSERT INTO tableA SELECT * FROM topicC
```

Write Modes

The Sink supports both **insert** and **upsert** modes. This mapping is set in the `connect.volt.kcql` option.

Insert

Insert is the default write mode of the sink.

Insert Idempotency

Kafka currently provides at least once delivery semantics. Therefore, this mode may produce errors if unique constraints have been implemented on the target tables. If the error policy has been set to NOOP then the Sink will discard the error and continue to process, however, it currently makes no attempt to distinguish violation of integrity constraints from other exceptions such as casting issues.

Upsert

The Sink support VoltDB upserts which replaces the existing row if a match is found on the primary keys.

Upsert Idempotency

Kafka currently provides at least once delivery semantics and order is a guaranteed within partitions.

This mode will, if the same record is delivered twice to the sink, result in an idempotent write. The existing record will be updated with the values of the second which are the same.

If records are delivered with the same field or group of fields that are used as the primary key on the target table, but different values, the existing record in the target table will be updated.

Since records are delivered in the order they were written per partition the write is idempotent on failure or restart. Redelivery produces the same result.

Configurations

`connect.volt.kcql`

KCQL expression describing field selection and routes.

- Data type : string
- Importance : high
- Optional : no

`connect.volt.servers`

Comma separated server[:port].

- Type : string
- Importance : high
- Optional : no

`connect.volt.username`

The user to connect to the volt database.

- Type : string
- Importance : high
- Optional : no

`connect.volt.password`

The password for the voltdb user.

- Type : string
- Importance : high
- Optional : no

`connect.volt.error.policy`

Specifies the action to be taken if an error occurs while inserting the data.

There are three available options, **noop**, the error is swallowed, **throw**, the error is allowed to propagate and retry. For **retry** the Kafka message is redelivered up to a maximum number of times specified by the `connect.volt.max.retries` option. The `connect.volt.retry.interval` option specifies the interval between retries.

The errors will be logged automatically.

- Type: string
- Importance: high
- Default: throw

`connect.volt.max.retries`

The maximum number of times a message is retried. Only valid when the `connect.volt.error.policy` is set to `retry`.

- Type: string
- Importance: medium
- Optional: yes
- Default: 10

`connect.volt.retry.interval`

The interval, in milliseconds between retries if the Sink is using `connect.volt.error.policy` set to **RETRY**.

- Type: int
- Importance: medium
- Optional: yes
- Default : 60000 (1 minute)

`connect.volt.batch.size`

Specifies how many records to insert together at one time. If the connect framework provides less records when it is calling the Sink it won't wait to fulfill this value but rather execute it.

- Type : int
- Importance : medium
- Optional: yes
- Defaults : 1000

`connect.progress.enabled`

Enables the output for how many records have been processed.

- Type: boolean
- Importance: medium
- Optional: yes
- Default : false

Schema Evolution

Upstream changes to schemas are handled by Schema registry which will validate the addition and removal of fields, data type changes and if defaults are set. The Schema Registry enforces Avro schema evolution rules. More information can be found [here](#).

No schema evolution is handled by the Sink yet on changes in the upstream topics.

Deployment Guidelines

Distributed Mode

Connect, in production should be run in distributed mode.

1. Install the Confluent Platform on each server that will form your Connect Cluster.
2. Create a folder on the server called `plugins/streamreactor/libs`.
3. Copy into the folder created in step 2 the required connector jars from the stream reactor download.
4. Edit `connect-avro-distributed.properties` in the `etc/schema-registry` folder where you installed Confluent and uncomment the `plugin.path` option. Set it to the path you deployed the stream reactor connector jars in step 2.
5. Start `Connect, bin/connect-distributed etc/schema-registry/connect-avro-distributed.properties`

Connect Workers are long running processes so set an `init.d` or `systemctl` service accordingly.

Connector configurations can then be push to any of the workers in the Cluster via the CLI or curl, if using the CLI remember to set the location of the Connect worker you are pushing to as it defaults to localhost.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Kubernetes

Helm Charts are provided at our [repo](#), add the repo to your Helm instance and install. We recommend using the Landscaper to manage Helm Values since typically each Connector instance has it's own deployment.

Add the Helm charts to your Helm instance:

```
helm repo add datamountaineer https://datamountaineer.github.io/helm-charts/
```

TroubleShooting

Please review the FAQs and join our [slack channel](#).

1.3 Tools

Helper tools and libraries for interacting with the components of the architecture, Kafka Connect and Schema Registry.

1.3.1 Kafka Connect CLI

The CLI is meant to behave as a good unix citizen: input from `stdin`; output to `stdout`; out of band info to `stderr` and non-zero exit status on error. Commands dealing with configuration expect or producedata in `.properties` style: `key=value` lines and comments start with a `#`.

```
kafka-connect-cli 1.0.2
Usage: kafka-connect-cli [ps|get|rm|create|run|status] [options] [<connector-name>]

  --help
    prints this usage text
  -e <value> | --endpoint <value>
    Kafka Connect REST URL, default is http://localhost:8083/

Command: ps
list active connectors names.

Command: get
get the configuration of the specified connector.

Command: rm
remove the specified connector.

Command: create
create the specified connector with the .properties from stdin; the connector cannot
↪ already exist.

Command: run
create or update the specified connector with the .properties from stdin.

Command: status
get connector and it's task(s) state(s).

  <connector-name>
    connector name
```

You can override the default endpoint by setting an environment variable `KAFKA_CONNECT_REST` i.e.

```
export KAFKA_CONNECT_REST="http://myserver:myport"
```

Requirements

- Java 1.8

To Build

```
gradle fatJar
```

Usage

Clone this repository, do a `mvn package` and run the jar in a way you prefer, for example with the provided `cli` shell script. The CLI can be used as follows.

Get Active Connectors

Command: `ps`

Example:

```
$ ./cli ps
twitter-source
```

Get Connector Information

Command: `get`

Example:

```
$ ./cli get twitter-source
#Connector `twitter-source`:
name=twitter-source
tasks.max=1

(snip)

track.terms=test
#task ids: 0
```

Delete a Connector

Command: `rm`

Example:

```
$ ./cli rm twitter-source
```

Create a New Connector

The connector cannot already exist.

Command: create

Example:

```
$ ./cli create twitter-source <twitter.properties
#Connector `twitter-source`:
name=twitter-source
tasks.max=1

(snip)

track.terms=test
#task ids: 0
```

Create or Update a Connector

Either starts a new connector if it did not exist, or update an existing connector.

Command: run

Example:

```
$ ./cli run twitter-source <twitter.properties
#Connector `twitter-source`:
name=twitter-source
tasks.max=1

(snip)

track.terms=test
#task ids: 0
```

1.3.2 Schema Registry CLI

This repository contains a CLI and Go client for the REST API of Confluent's Kafka Schema Registry.

CLI

To install the CLI, assuming a properly setup Go installation, do:

```
go get github.com/datamountaineer/schema-registry/schema-registry-cli
```

After that, the CLI is found in `$GOPATH/bin/schema-registry-cli`. Running `schema-registry-cli` without arguments gives:

```
A command line interface for the Confluent schema registry

Usage:
  schema-registry-cli [command]

Available Commands:
```

```

add          registers the schema provided through stdin
exists       checks if the schema provided through stdin exists for the subject
get          retrieves a schema specified by id or subject
subjects     lists all registered subjects
versions     lists all available versions

```

Flags:

```

-h, --help      help for schema-registry-cli
-e, --url string schema registry url (default "http://localhost:8081")
-v, --verbose    be verbose

```

Use `"schema-registry-cli [command] --help"` **for** more information about a command.

The documentation of the package can be found [here](#).

1.4 Fast Data UI's (Landoop)

Landoop has a number of UI's available to visually data in Kafka and schemas in the Schema Registry.

You can either build from their GitHub repo or install and run the docker images.

- [Kafka Topics Browser](#)
- [Schema Registry](#)
- [Kafka Connect](#)

1.4.1 Kafka Connect UI

The Kafka Connect UI lets you:

- Visualise your connect cluster sink & sources.
- Create new connectors with few clicks.
- Update & Delete connectors configuration.
- View workers tasks health & failures.

For the Connect UI please contact [Landoop](#).

Add new connectors.

1.4.2 Kafka Topic Browser

The Kafka Topic Browser allows you to look into topic without having to write code or use the command line console consumers.

Supported features are:

- Find topics & browse topic data (kafka messages)
- View topic metadata
- View topic configuration
- Download data

KAFKA CONNECT

7 Connectors

NEW

Search connectors

file-connector

2 x

file

→

influxDB

influx-basic

2 x

influxDB

logs

1 x

file

→

influxDB

redis-fx-avro

1 x

redis

sink-to-hbase

1 x

hbase

twitter-source

1 x

twitter

→

influxDB

yahoo-finance-source

2 x

yahoo

→

influxDB

Kafka Connect : <https://kafka-connect.demo.landoop.com>

Powered by LANDOOP

Dashboard

logs

kafka-connect-logs

sink-to-hbase

influx-basic2

influx-basic

file-connector

fstab

yahoo-finance-source

2 workers

redis-fx-avro

twitter-source

twitter

7 Connectors

NEW

Search connectors

file-connector

2 x

file

→

influxDB

influx-basic

2 x

influxDB

logs

1 x

file

→

influxDB

redis-fx-avro

1 x

redis

sink-to-hbase

1 x

hbase

twitter-source

1 x

twitter

→

influxDB

yahoo-finance-source

2 x

yahoo

→

influxDB

Kafka Connect : <https://kafka-connect.demo.landoop.com>

kafka-connect-ui.landoop.com/#/connector/redis-fx-avro

New Connector

Search

Sources

twitter

Subscribe to feeds using the Twitter API and stream data into a kafka topic

file

Read files and stream data into Kafka Topics

Blockchain

A Connector to hook into the live streaming providing a real time feed for new bitcoin blocks and transactions provided by www.blockchain.info

bloomberg

a source connector to subscribe to Bloomberg feeds via the Bloomberg labs open API and write to Kafka.

cassandra

The Cassandra source connector allows you to extract entries from Cassandra with the CQL driver and write them into a Kafka topic

yahoo

Subscribe to Yahoo Finance API and stream data into a kafka topic

jdbc

Sinks

cassandra

The Cassandra sink connector allows you to write data from a Kafka topic into a Cassandra table.

influxDB

The InfluxDB sink connector allows you to write data from a Kafka topic into a influxDB table.

elastic

A Connector and Sink to write events from Kafka to Elastic Search using Elastic4s client.

hbase

The HBase sink connector allows you to write data from a Kafka topic into HBase

redis

The Redis sink connector allows you to write data from a Kafka topic into Redis

kudu

The Kudu sink connector allows you to write data from a Kafka topic into Kudu. Kudu 0.9 or newer

jms

The JMS sink connector allows you to extract entries

7 Connectors

NEW

Search connectors

file-connector

2 x file →

influx-basic

2 x influxDB

logs

1 x file →

redis-fx-avro

1 x redis

sink-to-hbase

1 x hbase

twitter-source

1 x twitter →

yahoo-finance-source

2 x yahoo →

Kafka Connect : https://kafka-connect.demo.landoop.com

New Connector (Sink): kudu

kudu

The Kudu sink connector allows you to write data from a Kafka topic into Kudu. Kudu 0.9 or newer

class: com.datamountaineer.streamreactor.connect.kudu.sink.KuduSinkConnector

Show Hints

BASIC INFO

ADVANCED OPTIONS

FINISH

Kafka-Connect Query Language (KCQL)

KCQL

INSERT INTO TABLE2 SELECT field1, field2, field3 as renamedField FROM TOPIC2

Kudu configuration

Master Server

quickstart

The schema registry url

https://schema-registry.demo.landoop.com

Error policy

noop

« BACK

NEXT »

KAFKA TOPICS

6 Topics

System Topics

Search topics

_schemas

1 Replication x 1 Partition

json

connect-status

1 Replication x 1 Partition

binary

kafka-connect-logs

1 Replication x 1 Partition

..

twitter

1 Replication x 1 Partition

avro

yahoo-fx

1 Replication x 1 Partition

avro

yahoo-stocks

1 Replication x 1 Partition

avro

* Default configuration has been amended

Kafka Rest : https://kafka-rest-proxy.demo.landoop.com

Kafka Brokers : 1

yahoo-fx

Filter

TOPIC	TABLE	RAW DATA
Offset	Partition	Key Value
0	0	▶ Value: { symbol: [object Object], price: [obje
1	0	▼ Value: ▶ symbol: { string: EURGBP=X } ▼ price: double: 0.8581
2	0	▶ Value: { symbol: [object Object], price: [obje
3	0	▶ Value: { symbol: [object Object], price: [obje
4	0	▶ Value: { symbol: [object Object], price: [obje
5	0	▶ Value: { symbol: [object Object], price: [obje
6	0	▶ Value: { symbol: [object Object], price: [obje
7	0	▶ Value: { symbol: [object Object], price: [obje
8	0	▶ Value: { symbol: [object Object], price: [obje
9	0	▶ Value: { symbol: [object Object], price: [obje

<

1

2

3

4

5

6

7

...

81

>

Powered by Landoop

KAFKA TOPICS

6 Topics System Topics

Search topics

* _schemas
1 Replication x 1 Partition
json

connect-status
1 Replication x 1 Partition
binary

kafka-connect-logs
1 Replication x 1 Partition
..

twitter
1 Replication x 1 Partition
avro

* yahoo-fx
1 Replication x 1 Partition
avro

yahoo-stocks
1 Replication x 1 Partition
avro

★ Default configuration has been amended

Kafka Rest : <https://kafka-rest-proxy.demo.landoop.com>

Kafka Brokers : 1

yahoo-fx

Filter

TOPIC

TABLE

RAW DATA

offset	partition	key	value	
			symbol.string	price.double
0	0		USDGBP=X	0.7671
1	0		EURGBP=X	0.8581
2	0		USDGBP=X	0.767
3	0		EURGBP=X	0.8583
4	0		USDGBP=X	0.767
5	0		EURGBP=X	0.8583
6	0		USDGBP=X	0.767
7	0		EURGBP=X	0.8583
8	0		USDGBP=X	0.767
9	0		EURGBP=X	0.8583

<

1

2

3

4

5

6

7

...

81

>

Powered by Landoop

1.4.3 Schema Registry Browser

The Schema Registry is an integral part of the Kafka Streaming Platform.

Schema Registry provides a serving layer for your metadata. It provides a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas, provides multiple compatibility settings and allows evolution of schemas according to the configured compatibility setting. It provides serializers that plug into Kafka clients that handle schema storage and retrieval for Kafka messages that are sent in the Avro format.

Our web tool for schema registry allows:

- Visibility of schemas and their versions in the topics
- Schema validation
- Ability to add new schemas
- Ability to track changes with a graphical diff.

1.4.4 Install

For docker, pull the images:

```
docker pull landoop/kafka-topics-ui
docker pull landoop/schema-registry-ui
```

To run

```
docker run --rm -it -p 8000:8000 \
  -e "SCHEMAREGISTRY_URL=http://confluent-schema-registry-host:port" \
  landoop/schema-registry-ui

docker run --rm -it -p 8000:8000 \
```

```
-e "KAFKA_REST_PROXY_URL=http://kafka-rest-proxy-host:port" \
  landoop/kafka-topics-ui

#Start both in one docker
#docker run --rm -it -p 8000:8000 \
#   -e "SCHEMAREGISTRY_UI_URL=http://confluent-schema-registry-host:port" \
#   -e "KAFKA_REST_PROXY_URL=http://kafka-rest-proxy-host:port" \
#   landoop/kafka-topics-ui
```

Your schema-registry service will need to allow CORS (!!)

To do that, and in `/opt/confluent-3.0.0/etc/schema-registry/schema-registry.properties`

```
access.control.allow.methods=GET,POST,OPTIONS
access.control.allow.origin=*
```