CONFLUENT

SerDes for Kafka Streams

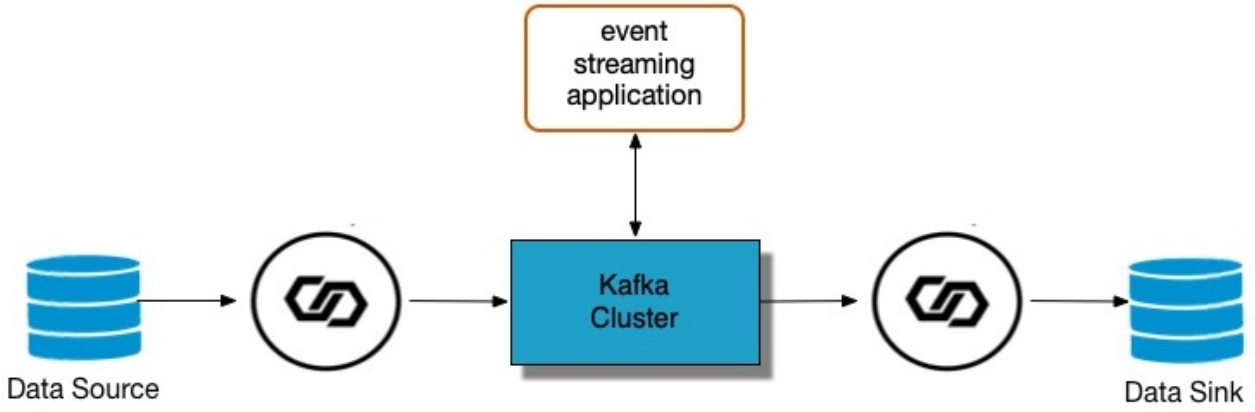
SpecificAvro, GenericAvro, and JSON Examples

Yeva Byzek, © 2019 Confluent, Inc.

Table of Contents

Introduction

Enterprises can build streaming ETL pipelines in Apache Kafka®, a distributed streaming platform that is the core of modern enterprise architectures. Connectors within the Kafka Connect framework extract data from different sources, the rich Kafka Streams API performs complex transformations and analysis from within your core applications, and other connectors load transformed data to other systems. You can deploy the Confluent Schema Registry to centrally manage schemas, validate compatibility, and provide warnings if data does not conform to the schema.



Developers writing event streaming applications using the Kafka Streams API need to consider data stream format and serialization:

1. Keys and values
2. Serialization/deserialization (SerDes)

For example, some Kafka Streams methods require record keys to be non-null, so either the connector or the application may need to add keys to the event stream. Or, another consideration is how the record keys or record values are serialized—you must use the appropriate SerDes, such as `SpecificAvro`, `GenericAvro`, `JsonPOJO`, any of the built-in SerDe implementations for Java primitives and basic types, or a custom one. Essentially, data may need to be appropriately transformed before applying business logic. There are many ways to do this in the pipeline—it can done by Kafka Connect or

© 2014-2020 Confluent, Inc.

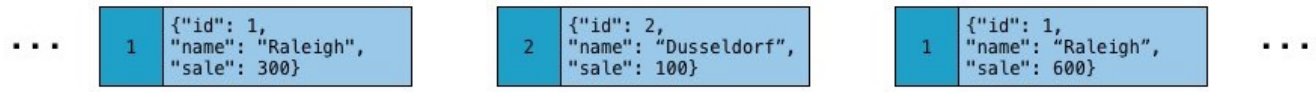
in the event streaming application itself. This white paper explores five examples:

Example	Produce to Kafka Topic	Topic Key	Topic Value
Example 1: Confluent CLI Producer with String	CLI	`String`	`String`
Example 2: JDBC source connector with JSON	JDBC with SMT to add key	`Long`	`Json`
Example 3: JDBC source connector with SpecificAvro	JDBC with SMT to set namespace	null	`SpecificAvro`
Example 4: JDBC source connector with GenericAvro	JDBC	null	`GenericAvro`
Example 5: Java producer with SpecificAvro	Producer	`Long`	`SpecificAvro`

© 2014-2020 Confluent, Inc.

Scenario

There is a global company that sells products to consumers, and every time a sale happens, it generates a Kafka event that captures the ID of the city where the sale happened, the name of the city, and the value of the sale. Because these sales events happen on an ongoing basis, there is a stream of these sales events.



There is a requirement to update a live sales dashboard, so you need to build a real-time event streaming application. Every time a sale occurs in any given city, the dashboard is refreshed with the latest **count** of sales per city and **sum** value of the sales per city. Kafka Streams has a rich Domain Specific Language (DSL), and most data processing operations can be expressed in just a few lines of code, including those that will calculate a count and sum. (See the Kafka Streams Code in the appendix for sample snippets of code that can calculate **sum** and **count**.)

For these operations to work, you first need to get the data in the right format. Each of the five examples compares different paths to serving an event stream with the following characteristics to the event streaming application:

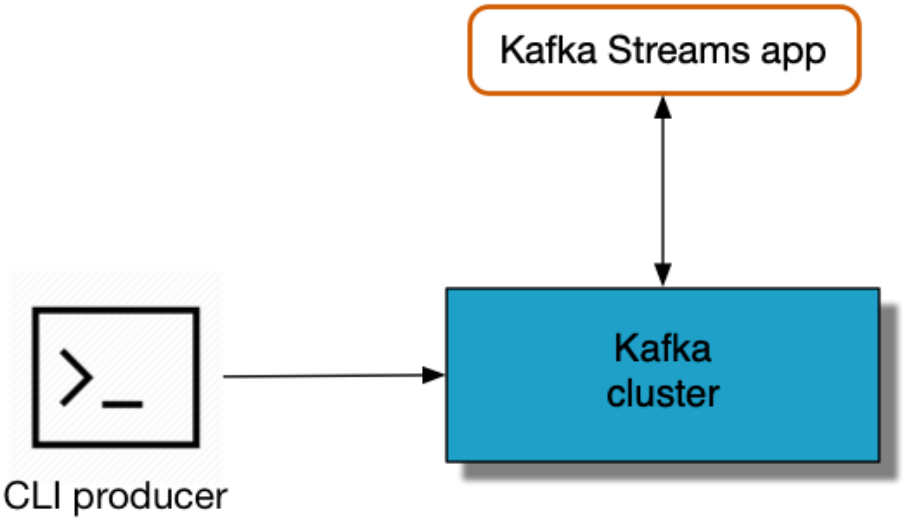
- A `KStream<Long, Long>` object that represents a stream of sales events
- Each record has a key of type `Long` that represents the ID of the city
- Each record has a value of type `Long` that represents the sale value of each event

© 2014-2020 Confluent, Inc.

Example 1: Confluent CLI Producer with String

Overview

This example is for users who are just starting out, have no real data source to connect to, and want a more basic introduction with the CLI before jumping into Kafka Connect.



Produce to Kafka Topic

If you have no data source to test with but you just want to get straight to the code development in Kafka Streams, your first steps might be to just use the Confluent CLI producer.

1. Start the CLI producer writing to a Kafka topic called `consoleproducer-locations` on a locally running cluster:

© 2014-2020 Confluent, Inc.

```
confluent local produce consoleproducer-locations -- \
--property parse.key=true \
--property key.separator='|'
```

2. Copy and paste the following messages one at a time. The key is the first value, e.g., 1, 2, 3, 4, or 5, which represents the ID of the city. Press `ctrl-c` when you are done:

```
1|Raleigh|300
2|Dusseldorf|100
1|Raleigh|600
3|Moscow|800
4|Sydney|200
2|Dusseldorf|400
5|Chennai|400
3|Moscow|100
3|Moscow|200
1|Raleigh|700
```

3. Consume messages from the topic `consoleproducer-locations`:

```
confluent local consume consoleproducer-locations -- \
--from-beginning \
--property print.key=true
```

Observe messages with `String` keys and `String` values:

© 2014-2020 Confluent, Inc.

```
1   Raleigh|300
2   Dusseldorf|100
1   Raleigh|600
3   Moscow|800
4   Sydney|200
2   Dusseldorf|400
5   Chennai|400
3   Moscow|100
3   Moscow|200
1   Raleigh|700
```

Create the KStream

See the source code.

It was super easy getting the data into the topic. But now it's going to require some dirty work in the Streams application to glean the fields and form meaningful structures.

1. Define the SerDes for the source topic:
   - key: `Serdes.String()`
   - value: `Serdes.String()`

```
final KStream<String, String> inputData = builder.stream("consoleproducer-
locations", Consumed.with(Serdes.String(), Serdes.String()));
```

2. The value of the messages is a `String` with a delimiter (`|`) separating the city name from the sale value, so it first needs to be split:

© 2014-2020 Confluent, Inc.

```
final KStream<Long, Location> locations = inputData.map((k, v) ->
    new KeyValue<>(Long.parseLong(k), new Location(Long.parseLong(k),
        v.split("\\|")[0], Long.parseLong( v.split( "\\|")[1])));
```

3. Create the target `KStream<Long, Long> sales` object. *Tada!*

```
final KStream<Long, Long> sales = locations.mapValues(v -> v.getSale());
```

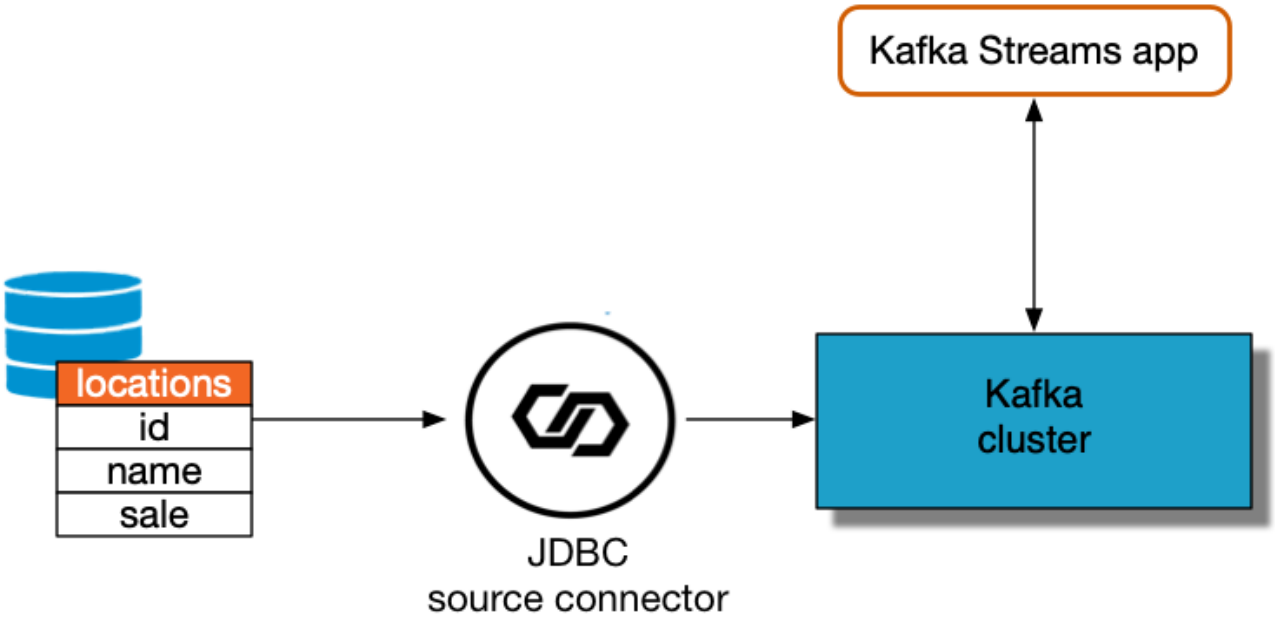
4. Now that you have `sales`, you can proceed with the application logic for count and sum.

© 2014-2020 Confluent, Inc.

Example 2: JDBC source connector with JSON

Overview

This is for users who want an example with Kafka Connect but are not ready for schemas and Avro. It writes the data in JSON to the Kafka topic. The connector uses a few Single Message Transforms (SMTs) to demonstrate how the connector can insert a key into the Kafka message.



© 2014-2020 Confluent, Inc.

Source Database Table Data

The next step is to source data from a database into Kafka. Let's assume there is a SQL database with a table called `locations` with some data:

id INTEGER KEY	name VARCHAR(255)	sale INTEGER
1	Raleigh	300
2	Dusseldorf	100
1	Raleigh	600
3	Moscow	800
4	Sydney	200
2	Dusseldorf	400
5	Chennai	400
3	Moscow	100
3	Moscow	200
1	Raleigh	700

© 2014-2020 Confluent, Inc.

Produce to Kafka Topic

You can use the JDBC source connector to read that data into the Kafka topic and write the message value in JSON. The Kafka Connect JDBC source connector can produce JSON values, but by default it does not insert a key. This is where the SMT is helpful: it will insert a key before writing the data into the Kafka topic.

1. In a file called `jdbcjson-connector.properties`, configure the primary connector properties. Notice the `key.converter` is `Long` (specifically `org.apache.kafka.connect.converters.LongConverter`, as provided by KAFKA-6913), and the `value.converter` is `Json` (specifically `org.apache.kafka.connect.json.JsonConverter`).

```
name=jdbcjson
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
connection.url=jdbc:sqlite:/usr/local/lib/retail.db
mode=incrementing
incrementing.column.name=id
topic.prefix=jdbcjson-
table.whitelist=locations
key.converter=org.apache.kafka.connect.converters.LongConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
tasks.max=1
```

2. In the same file, configure the SMTs to add a key to the records. The order of operations is: `org.apache.kafka.connect.transforms.ValueToKey` takes the ID field from the message value and replaces it as the key (which was previously null), `org.apache.kafka.connect.transforms.ExtractField$Key` extracts the value of ID itself, and `org.apache.kafka.connect.transforms.Cast$Key` casts the ID type to `Long`.

© 2014-2020 Confluent, Inc.

```
transforms=InsertKey, ExtractId, CastLong
transforms.InsertKey.type=org.apache.kafka.connect.transforms.ValueToKey
transforms.InsertKey.fields=id
transforms.ExtractId.type=org.apache.kafka.connect.transforms.ExtractField$K
ey
transforms.ExtractId.field=id
transforms.CastLong.type=org.apache.kafka.connect.transforms.Cast$Key
transforms.CastLong.spec=int64
```

3. Run the connector. If you are testing this on a local host, use:

```
confluent local config jdbcjson -- -d ./jdbcjson-connector.properties
```

4. Consume messages from the topic `jdbcjson-locations`, and you can specify the `Long` deserializer since the key has been cast to a `Long`:

```
confluent local consume jdbcjson-locations -- \
--from-beginning \
--property print.key=true \
--key-deserializer org.apache.kafka.common.serialization.LongDeserializer
```

Observe messages with `Long` keys and `Json` values:

© 2014-2020 Confluent, Inc.

```
1   {"id":1,"name":"Raleigh","sale":300}
1   {"id":1,"name":"Raleigh","sale":600}
1   {"id":1,"name":"Raleigh","sale":700}
2   {"id":2,"name":"Dusseldorf","sale":100}
2   {"id":2,"name":"Dusseldorf","sale":400}
3   {"id":3,"name":"Moscow","sale":800}
3   {"id":3,"name":"Moscow","sale":100}
3   {"id":3,"name":"Moscow","sale":200}
4   {"id":4,"name":"Sydney","sale":200}
5   {"id":5,"name":"Chennai","sale":400}
```

Create the KStream

See the full source code.

1. Create a custom JSON data model for the messages in the Kafka topic (source):

© 2014-2020 Confluent, Inc.

```
...
public class LocationJSON {

  private Long id;
  private String name;
  private Long sale;

  public LocationJSON(final Long id, final String name, final Long sale) {
    this.id = id;
    this.name = name;
    this.sale = sale;
  }

  public Long getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public Long getSale() {
    return sale;
  }
}
...
```

2. Define the SerDes for the source topic:
   - Key: `Serdes.Long()`
   - Value: custom JSON `locationSerde`

© 2014-2020 Confluent, Inc.

```
final Serde<LocationJSON> locationSerde = Serdes.serdeFrom(new
JsonSerializer<>(),
                                        new JsonDeserializer<>(LocationJSON
.class));
final KStream<Long, LocationJSON>
    locationsJSON =
    builder.stream("jdbcjson-locations", Consumed.with(Serdes.Long(),
locationSerde));
```

3. Create the target `KStream<Long, Long> sales` object. *Tada!*

```
final KStream<Long, Long> sales = locationsJSON.mapValues(v -> v.getSale());
```

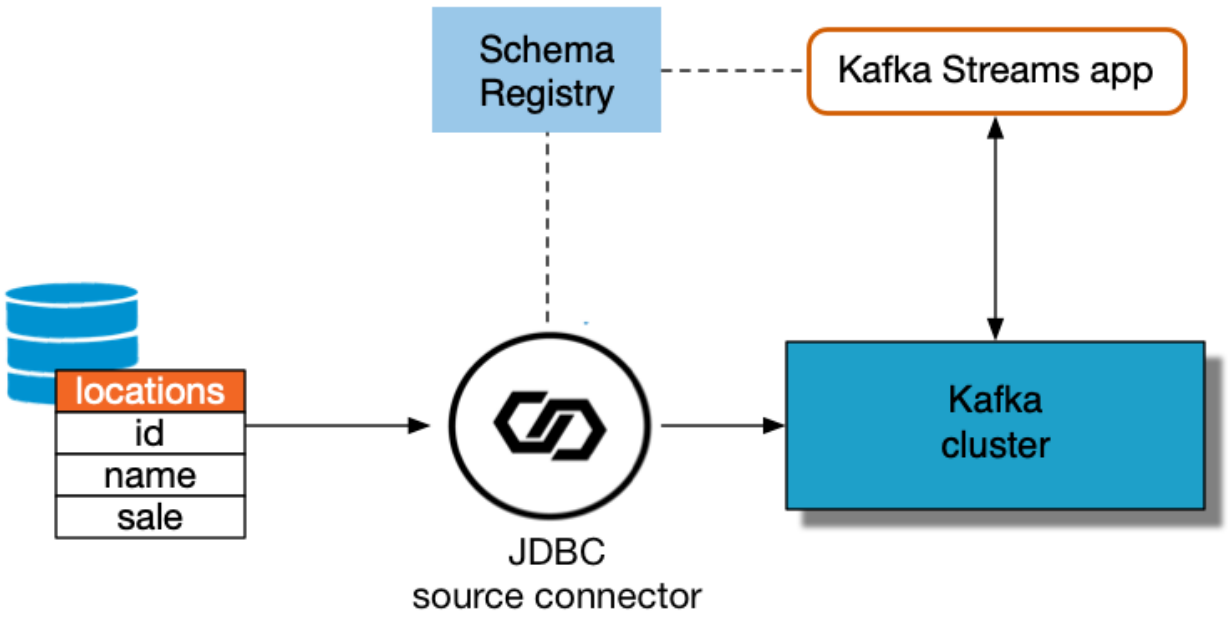
4. Now that you have `sales`, you can proceed with the application logic for count and sum.

© 2014-2020 Confluent, Inc.

Example 3: JDBC source connector with SpecificAvro

Overview

This is for users who want an example with Kafka Connect along with schemas, Avro, and Confluent Schema Registry. The Kafka topic does not have messages with keys, so the example demonstrates how Kafka Streams, instead of Kafka Connect, can insert a key into the event stream. The connector uses the SMT `SetSchemaMetadata`, fixed by KAFKA-5164, because the JDBC source connector by default doesn't set a namespace when it generates a schema name for the data it is producing to Kafka. This SMT manually sets the namespace to `io.confluent.examples.connectandstreams.avro.Location`, which in turn, enables the schema lookup in Confluent Schema Registry. If you do not have the fix for KAFKA-5164, see Example 4: JDBC source connector with GenericAvro, which uses `GenericAvro` instead of `SpecificAvro`.



© 2014-2020 Confluent, Inc.

Data Source

The data source is exactly the same as the Source Database Table Data in example 2.

Produce to Kafka Topic

As in Example 2: JDBC source connector with JSON, this example uses the Kafka Connect JDBC source connector. Instead of JSON, it produces Avro data.

1. In a file called `jdbcspecificavro-connector.properties`, configure the primary connector properties. Notice that the `value.converter` is `Avro` (specifically `io.confluent.connect.avro.AvroConverter`).

```
name=jdbcspecificavro
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
connection.url=jdbc:sqlite:/usr/local/lib/retail.db
mode=incrementing
incrementing.column.name=id
topic.prefix=jdbcspecificavro-
table.whitelist=locations
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081
value.converter.schemas.enable=true
tasks.max=1
```

2. In the same file, add the SMT `SetSchemaMetadata` to manually set the namespace to `io.confluent.examples.connectandstreams.avro.Location`:

© 2014-2020 Confluent, Inc.

```
transforms=SetValueSchema
transforms.SetValueSchema.type=org.apache.kafka.connect.transforms.SetSchema
Metadata$Value
transforms.SetValueSchema.schema.name=io.confluent.examples.connectandstream
s.avro.Location
```

Unlike Example 2: JDBC source connector with JSON,this example intentionally does not use the SMTs to add the key, demonstrating how to instead set the key from within the Kafka Streams application.

3. Run the connector. If you are testing this on a local host, use:

```
confluent local config jdbcspecificavro -- -d ./jdbcspecificavro-
connector.properties
```

4. Consume messages from the topic `jdbcspecificavro-locations`:

```
confluent local consume jdbcspecificavro-locations -- \
--value-format avro \
--from-beginning \
--property print.key=true
```

Observe messages with no keys and deserialized `Avro` values:

© 2014-2020 Confluent, Inc.

```
null    {"id":{"long":1},"name":{"string":"Raleigh"},"sale":{"int":300}}
null    {"id":{"long":1},"name":{"string":"Raleigh"},"sale":{"int":600}}
null    {"id":{"long":1},"name":{"string":"Raleigh"},"sale":{"int":700}}
null    {"id":{"long":2},"name":{"string":"Dusseldorf"},"sale":{"int":100}}
null    {"id":{"long":2},"name":{"string":"Dusseldorf"},"sale":{"int":400}}
null    {"id":{"long":3},"name":{"string":"Moscow"},"sale":{"int":800}}
null    {"id":{"long":3},"name":{"string":"Moscow"},"sale":{"int":100}}
null    {"id":{"long":3},"name":{"string":"Moscow"},"sale":{"int":200}}
null    {"id":{"long":4},"name":{"string":"Sydney"},"sale":{"int":200}}
null    {"id":{"long":5},"name":{"string":"Chennai"},"sale":{"int":400}}
```

5. View the schema registered in Confluent Schema Registry for the `jdbcspecificavro-locations-value`, which is the subject for the topic's message value. Notice that the namespace is set to `io.confluent.examples.connectandstreams.avro` because the connector used the SMT `SetSchemaMetadata`.

```
$ curl -s http://localhost:8081/subjects/jdbcspecificavro-locations-
value/versions/1 | jq -r '.schema'
{"type":"record","name":"Location","namespace":"io.confluent.examples.connec
tandstreams.avro","fields":[{"name":"id","type":["null","long"],"default":nu
ll},{"name":"name","type":["null","string"],"default":null},{"name":"sale","
type":["null","int"],"default":null}],"connect.name":"io.confluent.examples.
connectandstreams.avro.Location"}
```

Create the KStream

See the source code.

1. Define the SerDes for the source topic:
   - Key: `Serdes.String()`
   - Value: custom SpecificAvro `locationSerde`, with a pointer to the Schema

© 2014-2020 Confluent, Inc.

Registry

```
final Serde<Location> locationSerde = new SpecificAvroSerde<>();
final boolean isKeySerde = false;
locationSerde.configure(
    Collections.singletonMap(AbstractKafkaAvroSerDeConfig
.SCHEMA_REGISTRY_URL_CONFIG, SCHEMA_REGISTRY_URL),
    isKeySerde);

final KStream<String, Location>
    locationsNoKey =
    builder.stream("jdbcspecificavro-locations", Consumed.with(Serdes.
String(), locationSerde));
```

2. Create the new event stream `locations` from the extracted `id` value of the message payload, adding a key

```
final KStream<Long, Location>
    locations =
    locationsNoKey.map((k, v) -> new KeyValue<>(v.getId(), v));
```

3. Create the target `KStream<Long, Long> sales` object. *Tada!*

```
final KStream<Long, Long> sales = locations.mapValues(v -> v.getSale());
```

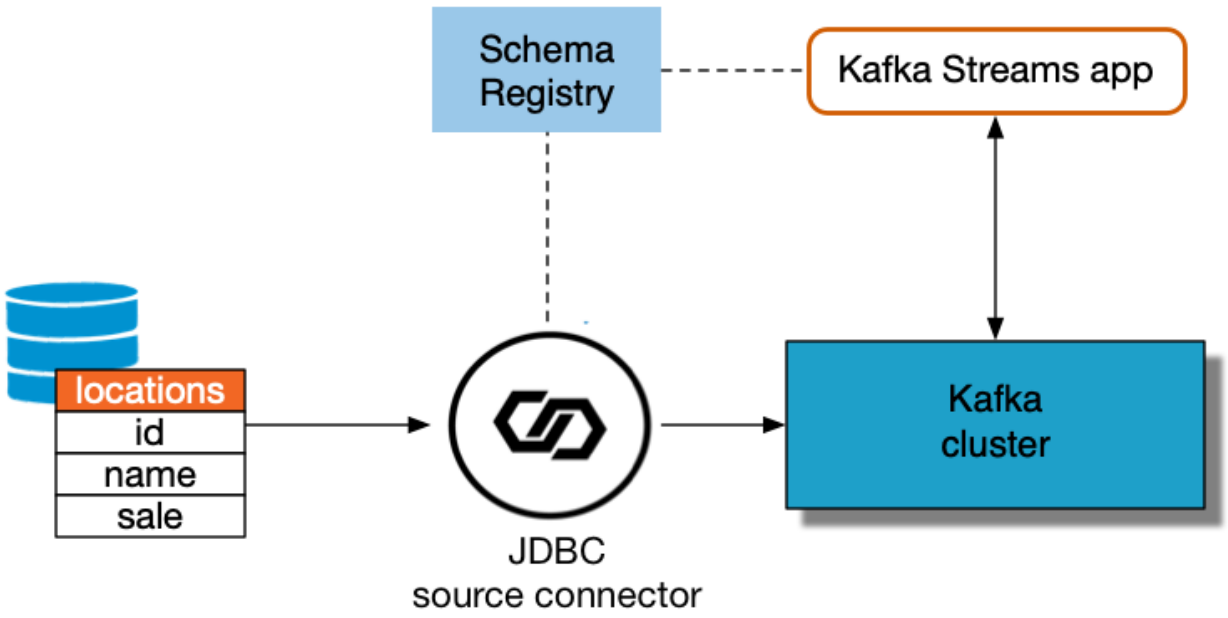
4. Now that you have `sales`, you can proceed with the application logic for count and sum.

© 2014-2020 Confluent, Inc.

Example 4: JDBC source connector with GenericAvro

Overview

This example is for users who are not running code that has the fix for KAFKA-5164. It's just like Example 3: JDBC source connector with SpecificAvro, except instead of `SpecificAvro`, the Kafka Streams application uses `GenericAvro`. If you're a reader of the Confluent blog, this example may remind you of the previous blog post Building a Real-Time Streaming ETL Pipeline in 20 Minutes.



Data Source

The data source is exactly the same as the Source Database Table Data in example 2.

© 2014-2020 Confluent, Inc.

Produce to Kafka Topic

Like Example 3: JDBC source connector with SpecificAvro, this example uses the Kafka Connect JDBC source connector. But instead of using the SMT `SetSchemaMetadata`, this example does not use any SMTs, so the topic will have null keys.

1. In a file called `jdbcgenericavro-connector.properties`, configure the primary connector properties. Notice that the `value.converter` is `Avro` (specifically `io.confluent.connect.avro.AvroConverter`).

```
name=jdbcgenericavro
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
connection.url=jdbc:sqlite:/usr/local/lib/retail.db
mode=incrementing
incrementing.column.name=id
topic.prefix=jdbcgenericavro-
table.whitelist=locations
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081
value.converter.schemas.enable=true
tasks.max=1
```

2. Run the connector. If you are testing this on a local host, use:

```
confluent local config jdbcgenericavro -- -d ./jdbcgenericavro-
connector.properties
```

3. Consume messages from the topic `jdbcgenericavro-locations`:

© 2014-2020 Confluent, Inc.

```
confluent local consume jdbcgenericavro-locations -- \
--value-format avro \
--from-beginning \
--property print.key=true
```

Observe messages with no keys and deserialized `Avro` values, just as shown in Example 3: JDBC source connector with SpecificAvro:

```
null    {"id":{"long":1},"name":{"string":"Raleigh"},"sale":{"int":300}}
null    {"id":{"long":1},"name":{"string":"Raleigh"},"sale":{"int":600}}
null    {"id":{"long":1},"name":{"string":"Raleigh"},"sale":{"int":700}}
null    {"id":{"long":2},"name":{"string":"Dusseldorf"},"sale":{"int":100}}
null    {"id":{"long":2},"name":{"string":"Dusseldorf"},"sale":{"int":400}}
null    {"id":{"long":3},"name":{"string":"Moscow"},"sale":{"int":800}}
null    {"id":{"long":3},"name":{"string":"Moscow"},"sale":{"int":100}}
null    {"id":{"long":3},"name":{"string":"Moscow"},"sale":{"int":200}}
null    {"id":{"long":4},"name":{"string":"Sydney"},"sale":{"int":200}}
null    {"id":{"long":5},"name":{"string":"Chennai"},"sale":{"int":400}}
```

4. View the schema registered in Confluent Schema Registry for the `jdbcgenericavro-locations-value`, which is the subject for the topic's message value. Notice that there is no namespace field.

```
$ curl -s http://localhost:8081/subjects/jdbcgenericavro-locations-
value/versions/1 | jq -r '.schema'
{"type":"record","name":"locations","fields":[{"name":"id","type":["null","l
ong"],"default":null},{"name":"name","type":["null","string"],"default":null
},{"name":"sale","type":["null","int"],"default":null}],"connect.name":"loca
tions"}
```

© 2014-2020 Confluent, Inc.

Create the KStream

See the source code.

1. Define the SerDes for the source topic:
   - Value: `GenericAvroSerde`

```
streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
GenericAvroSerde.class);
...
final KStream<String, GenericRecord> locationsGeneric = builder.stream(
"jdbcgenericavro-locations");
```

2. Create the new event stream `locations` from the extracted `id` value of the message payload and custom class values. It uses a map function to convert the stream of messages into having `Long` keys.

```
final KStream<Long, Location>
    locations =
    locationsGeneric.map((k, v) -> new KeyValue<>((Long) v.get("id"),
                                    new Location((Long) v.get("id"),
                                                 v.get("name").toString(),
                                                 Long.valueOf(
                                                     (Integer) v.get("
sale")))));
```

3. Create the target `KStream<Long, Long> sales` object. *Tada!*

```
final KStream<Long, Long> sales = locations.mapValues(v -> v.getSale());
```

© 2014-2020 Confluent, Inc.

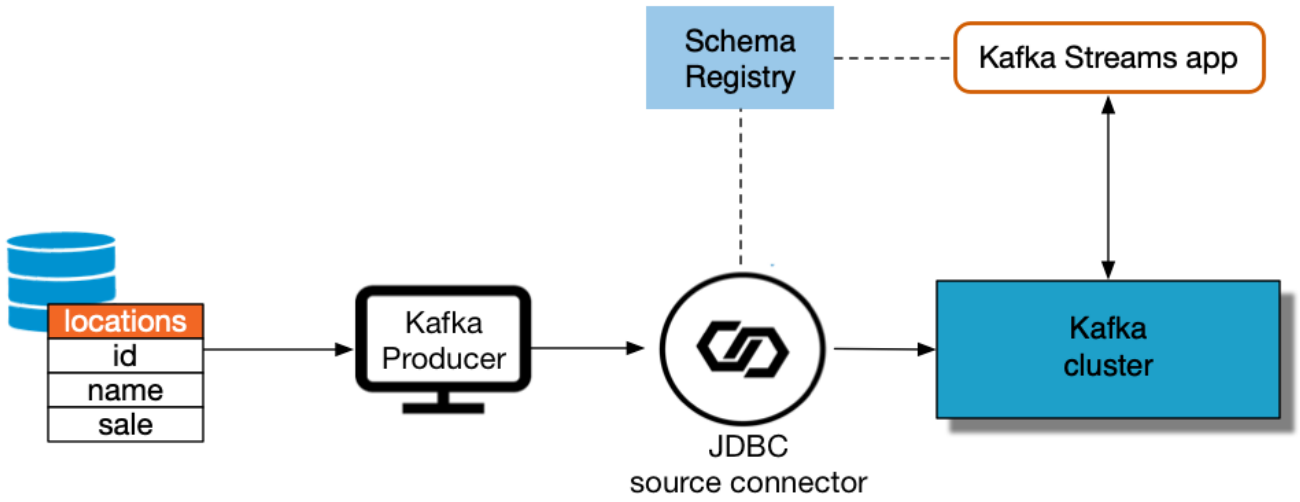
4. Now that you have `sales`, you can proceed with the application logic for count and sum.

© 2014-2020 Confluent, Inc.

Example 5: Java producer with SpecificAvro

Overview

This example is for users who have a Kafka producer client application instead of Kafka Connect. It uses Avro as well.



Data Source

The data source is exactly the same as the Source Database Table Data in example 2.

Produce to Kafka Topic

Instead of using a JDBC source connector, this example shows the Java producer writing Avro data to the Kafka topic.

1. Create an Avro schema definition for `Location`. The plugin `avro-maven-plugin`

© 2014-2020 Confluent, Inc.

generates Java class files from this source schema.

```
{"namespace": "io.confluent.examples.connectandstreams.avro",
 "type": "record",
 "name": "Location",
 "fields": [
     {"name": "id", "type": "long", "doc" : "id"},
     {"name": "name", "type": "string", "doc" : "name"},
     {"name": "sale", "type": "long", "doc" : "sale"}
 ]
}
```

2. Develop a Java producer application that writes Avro data to the Kafka topic `javaproducer-locations`:

```
...
final Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer
.class);
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
schemaRegistryUrl);
final KafkaProducer<Long, Location>
    locationProducer = new KafkaProducer<Long, Location>(props);

// Assume List<Location> locationsList contains a list of `Location` objects
// That were read from the database
locationsList.forEach(t -> {
  final ProducerRecord<Long, Location> record = new ProducerRecord<Long,
Location>("javaproducer-locations",

t.getId(), t);
  locationProducer.send(record);
});

locationProducer.close();
...
```

© 2014-2020 Confluent, Inc.

3. Consume messages from the topic `javaproducer-locations`:

```
confluent local consume javaproducer-locations -- \
--value-format avro \
--key-deserializer org.apache.kafka.common.serialization.LongDeserializer \
--from-beginning \
--property print.key=true
```

Observe messages with `Long` keys and deserialized `Avro` values:

```
1   {"id":1,"name":"Raleigh","sale":300}
2   {"id":2,"name":"Dusseldorf","sale":100}
1   {"id":1,"name":"Raleigh","sale":600}
3   {"id":3,"name":"Moscow","sale":800}
4   {"id":4,"name":"Sydney","sale":200}
2   {"id":2,"name":"Dusseldorf","sale":400}
5   {"id":5,"name":"Chennai","sale":400}
3   {"id":3,"name":"Moscow","sale":100}
3   {"id":3,"name":"Moscow","sale":200}
1   {"id":1,"name":"Raleigh","sale":700}
```

4. View the schema registered in Confluent Schema Registry for the `jdbcgenericavro-locations-value`, which is the subject for the topic's message value. Notice that the namespace is set to `io.confluent.examples.connectandstreams.avro` because this was the namespace set in the Avro schema definition.

```
$ curl -s http://localhost:8081/subjects/javaproducer-locations-
value/versions/1 | jq -r '.schema'
{"type":"record","name":"Location","namespace":"io.confluent.examples.connec
tandstreams.avro","fields":[{"name":"id","type":"long","doc":"id"},{"name":"
name","type":"string","doc":"name"},{"name":"sale","type":"long","doc":"sale
"}]}
```

© 2014-2020 Confluent, Inc.

Create the KStream

See the source code.

1. Define the SerDes for the source topic. It can be immediately set to the event stream `locations`. No intermediate `KStream` is required like in the previous examples because it's already properly serialized.
   - Key: `Serdes.Long()`
   - Value: `SpecificAvroSerde`

```
final Serde<Location> locationSerde = new SpecificAvroSerde<>();
final boolean isKeySerde = false;
locationSerde.configure(
    Collections.singletonMap(AbstractKafkaAvroSerDeConfig
.SCHEMA_REGISTRY_URL_CONFIG, SCHEMA_REGISTRY_URL),
    isKeySerde);

final KStream<Long, Location> locations = builder.stream("javaproducer-
locations", Consumed.with(Serdes.Long(), locationSerde));
```

2. Create the target `KStream<Long, Long> sales` object. *Tada!*

```
final KStream<Long, Long> sales = locations.mapValues(v -> v.getSale());
```

3. Now that you have `sales`, you can proceed with the application logic for count and sum.

© 2014-2020 Confluent, Inc.

Resources

View the complete source code for all these examples in the confluentinc/examples GitHub repository.

Here are additional resources to further your education on developing Kafka Streams or KSQL event streaming applications:

- Kafka Tutorials
- Kafka Streams documentation
- KSQL documentation

Appendix

Kafka Streams Code

Here is an example of the Kafka Streams DSL code that achieves the **count**:

```
final KStream<Long, Long> countKeys = sales.groupByKey(Grouped.with(Serdes
.Long(), Serdes.Long()))
    .count()
    .toStream();
```

Here is an example of the Kafka Streams DSL code that achieves the **sum**:

```
final KStream<Long, Long> salesAgg = sales.groupByKey(Grouped.with(Serdes
.Long(), Serdes.Long()))
    .reduce(Long::sum)
    .toStream();
```

© 2014-2020 Confluent, Inc.