FLIP-XXX new Apicurio Avro format original.

Status

Discussion thread	https://lists.apache.org/thread/wtkl4yn847tdd0wrqm5xgv9wc0cb0kr8
Vote thread	
JIRA	
Release	1.20 or beyond

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Currently, if users have Avro schemas in an Apicurio Registry (an open source Apache 2 licensed schema registry), then the natural way to work with those Avro events is to use the schemas in the Apicurio Repository. This FLIP proposes a new Kafka oriented Avro Apicurio format, to allow Flink users to work with Avro schemas stored in the Apicurio Registry.

Messages in the Apicurio format have a schema ID (usually the global ID in a Kafka header), which identifies the schema in the Apicurio Registry. The new format will:

- For inbound messages, use the ID to find the Avro schema
- For outbound messages, register a schema in Apicurio and include the ID in the message

In this way Apache Flink can be used to consume and produce Apicurio events.

Public Interfaces

This FLIP adds a new Format called Apicurio Avro for Apache Kafka and Upsert Kafka.

Apicurio Avro Format

Format: Serialization Schema Format: Deserialization Schema

The Avro Schema Registry (`avro-apicurio`) format allows you to read records that were serialized by the `io.apicurio.registry.serde.avro.KafkaAvroSerializer` and to write records that can in turn be read by the `io.apicurio.registry.serde.avro.KafkaAvroDeserializer`.

When reading (deserializing) a record with this format the Avro writer schema is fetched from the configured Apicurio Registry based on the global ID or content ID encoded in the record while the reader schema is inferred from table schema.

When writing (serializing) a record with this format, the Avro schema is inferred from the table schema and used to register an Avro schema whose ID will be included in the outbound message.

The Apicurio Avro format can only be used in conjunction with the Apache Kafka SQL connector or the Upsert Kafka SQL Connector.

How to create tables with Apicurio Avro format

Example of a table using raw UTF-8 string as Kafka key and Avro records registered in the Apicurio Registry as Kafka values:

```
CREATE TABLE user_created (

  -- one column mapped to the Kafka raw UTF-8 key
  the_kafka_key STRING,

  -- a few columns mapped to the Avro fields of the Kafka value
  id STRING,
  name STRING,
  email STRING

) WITH (

  'connector' = 'kafka',
  'topic' = 'user_events_example1',
  'properties.bootstrap.servers' = 'localhost:9092',
  'properties.group.id' = 'mygroupid',
```

```
  -- UTF-8 string as Kafka keys, using the 'the_kafka_key' table column
  'key.format' = 'raw',
  'key.fields' = 'the_kafka_key',

  'value.format' = 'avro-apicurio',
  'value.avro-apicurio.url' = 'http://localhost:8080/apis/registry/v2',
  'value.fields-include' = 'EXCEPT_KEY'
);
```

We can write data into the Kafka table as follows:

```
INSERT INTO user_created
SELECT
  -- replicating the user id into a column mapped to the kafka key
  id as the_kafka_key,

  -- all values
  id, name, email
FROM some_table;
```

Example of a table with both the Kafka key and value registered as Avro records in the Apicurio Registry:

```
CREATE TABLE user_created (

  -- one column mapped to the 'id' Avro field of the Kafka key
  kafka_key_id STRING,

  -- a few columns mapped to the Avro fields of the Kafka value
  id STRING,
  name STRING,
  email STRING

) WITH (

  'connector' = 'kafka',
  'topic' = 'user_events_example1',
  'properties.bootstrap.servers' = 'localhost:9092',

  -- Watch out: schema evolution in the context of a Kafka key is almost never backward nor
  -- forward compatible due to hash partitioning.
  'key.format' = 'avro-apicurio',
  'key.avro-apicurio.url' = 'http://localhost:8080/apis/registry/v2',
  'key.fields' = 'kafka_key_id',

  -- In this example, we want the Avro types of both the Kafka key and value to contain the
field 'id'
  -- => adding a prefix to the table column associated to the Kafka key field avoids clashes
  'key.fields-prefix' = 'kafka_key_',

  'value.format' = 'avro-apicurio',
```

```
 'value.avro-apicurio.url' = 'http://localhost:8080/apis/registry/v2',
'value.fields-include' = 'EXCEPT_KEY'

);
```

Example of a table using the upsert-kafka connector with the Kafka value registered as an Avro record in the Schema Registry:

```
CREATE TABLE user_created (

  -- one column mapped to the Kafka raw UTF-8 key
  kafka_key_id STRING,

  -- a few columns mapped to the Avro fields of the Kafka value
  id STRING,
  name STRING,
  email STRING,

  -- upsert-kafka connector requires a primary key to define the upsert behavior
  PRIMARY KEY (kafka_key_id) NOT ENFORCED

) WITH (

  'connector' = 'upsert-kafka',
  'topic' = 'user_events_example1',
  'properties.bootstrap.servers' = 'localhost:9092',

  -- UTF-8 string as Kafka keys
  -- We don't specify 'key.fields' in this case since it's dictated by the primary key of
the table
  'key.format' = 'raw',

  -- In this example, we want the Avro types of both the Kafka key and value to contain the
field 'id'
  -- => adding a prefix to the table column associated to the kafka key field avoids clashes
  'key.fields-prefix' = 'kafka_key_',

  'value.format' = 'avro-apicurio',
  'value.avro-apicurio.url' = 'http://localhost:8080/apis/registry/v2'
'value.fields-include' = 'EXCEPT_KEY'
);
```

Kafka connector Options

Option	Required	Forwarded	Default	Type	Description

useAdditionalPropertiesWithSerDe	optional	no	false	Boolean	Specify true so additional properties can be used in serialization and deserialization.

Apicurio avro Format Options

Option	Required	Forwarded	Default	Type	Sink only	Description
Format	required	no	(none)	String		Specify what format to use, here should be `'avro-apicurio'`.
properties	optional	yes	(none)	Map		This is the apicurio-registry client configuration properties. Any Flink properties take precedence,
apicurio.registry.request.ssl.truststore.location	optional	yes	(none)	String		Location / File of SSL truststore
apicurio.registry.request.ssl.truststore.type	optional	yes	(none)	String		Type of SSL truststore
apicurio.registry.request.ssl.truststore.password	optional	yes	(none)	String		Password of SSL truststore
apicurio.registry.request.ssl.keystore.location	optional	yes	(none)	String		Location / File of SSL keystore
apicurio.registry.request.ssl.keystore.type	optional	yes	(none)	String		Type of SSL keystore
apicurio.registry.request.ssl.keystore.password	optional	yes	(none)	String		Password of SSL keystore

apicurio-auth-basic-userid	optional	yes	(none)	String	Basic auth userId
apicurio-auth-basic-password	optional	yes	(none)	String	Basic auth password
apicurio-auth-oidc-url	optional	yes	(none)l	String	The auth URL to use for OIDC
apicurio-auth-oidc-clientID	optional	yes	(none)	String	Client ID to use for OIDC

apicurio.auth.oidc.clientSecret	optional	yes	(none)	String	Client secret to use for OIDC
apicurio-auth-oidc-scope	optional	yes	(none)	String	Client scope to use for OIDC
apicurio-auth-oidc-client-token-expiration-reduction	optional	yes	1	String	The token expiration to use for OIDC. This is a Duration in seconds. This is the amount of time before the token expires that Apicurio requests a new token.
apicurio-avro.id-placement	optional	yes	HEADER	Enum - values HEADER, LEGACY CONFLUENT	The valid values are HEADER – the schema ID is processed using the header LEGACY– the schema ID is processed in the message payload as a long CONFLUENT - the schema ID is processed in the message payload as an int.

apicurio-avro.id-option	optional	yes	GLOBAL_ID	Enum - values GLOBAL_ID, CONTENT_ID		The valid values are GLOBAL_ID - global IDs will be used CONTENT_ID - content IDs will be used
avro-apicurio.artifactId	optional	yes	<topic-name>-value or <topic-name>-key	String	Y	Specifies the artifactId of the artifact to be registered. If not specified, then for a key this is the topic name suffixed with “-key” and for a value it is the topic name suffixed with “-value”. A key and value can be specified for the same topic.
avro-apicurio.artifactName	optional	yes	<topic-name>-value or <topic-name>-key	String	Y	This specifies the name of the artifact to be registered. If not specified, then for a key this is the topic name suffixed with “-key” and for a value it is the topic name suffixed with “-value”.A key and value can be specified for the same topic.
avro-apicurio.artifactDescription	optional	yes	`Schema registered by Apache Flink.`	String	Y	This specifies the description of the artifact to be registered.
avro-apicurio.artifactVersion	optional	yes	“1”	String	Y	This specifies the version of the artifact to be registered.
avro-apicurio.schema	optional	yes	(none)	String		The schema registered or to be registered in the Apicurio Registry. If no schema is provided Flink converts the table schema to avro schema. The schema provided must match the table schema.
avro-apicurio.groupId	optional	yes	(none)	String	Y	The group id to use when creating a schema.
avro-apicurio.register-schema	optional	yes	True	Boolean	Y	When true the schema is registered, otherwise the schema is not registered.

There are a number of sink only format configuration options that affect the schema being registered and how the ID of the schema is included in the message being sent.

When registering a schema, if an existing schema is found with a matching artifactId, then it is updated. If no schema is found, then a new one is created.

Data Type Mapping

Currently, Apache Flink always uses the table schema to derive the Avro reader schema during deserialization and Avro writer schema during serialization. Explicitly defining an Avro schema is not supported yet. See the Apache Avro Format for the mapping between Avro and Flink DataTypes.

In addition to the types listed there, Flink supports reading/writing nullable types. Flink maps nullable types to Avro `union(null, something)`, where `something` is the Avro type converted from Flink type.

You can refer to Avro Specification for more information about Avro types.

Custom types and References

When reading from a Kafka source, the Avro writer schema is obtained from the Apicurio Registry and may contain references. These references will be expanded so the writer schema can be mapped to the reader schema.

Recursive references are references that refer to its own type or a parent type. These are not supported by the Apicurio Avro format. This is because the writer schema is expanded, which could not complete if there are circularities.

When using Kafka sinks, the schema is registered to the Apicurio Registry; the registered schema will not contain any references. The artifactId configuration property ensures that only one schema is created for a Kafka sink, subsequent serializations attempting to register the schema with the same artifactId do not result in more schemas.

Compatibility, Deprecation, and Migration Plan

No plans for Deprecation and Migration.

In terms of compatibility for the Kafka source this change is across 2 repositories, with 4 main changes are:

In core Flink add 2 new default methods to the DeserializeSchema interface

```
/**
 * Deserializes the byte message with additional input Properties.
 *
 * @param message The message, as a byte array.
 * @param additionalInputProperties map of additional input Properties that
can be used
 *     for deserialization. Override this method to make use of the
additionalInputProperties,
 * @return The deserialized message as an object (null if the message cannot
be deserialized).
 */
@PublicEvolving
default T deserializeWithAdditionalProperties(
        byte[] message, Map<String, Object> additionalInputProperties) throws
IOException {
    return deserialize(message);
}
@PublicEvolving
default void deserializeWithAdditionalProperties(
        byte[] message, Map<String, Object> additionalInputProperties,
Collector<T> out)
        throws IOException {
    T deserialize = deserializeWithAdditionalProperties(message,
additionalInputProperties);
    if (deserialize != null) {
        out.collect(deserialize);
    }
}
```

- In core Flink add 2 new default methods to the SerializeSchema interface

```
/**
 * Serializes the incoming element to a specified type and populates an output.
 * The additional input properties can be used by the serialization and the output
 * additionalProperties can be populated by the serialization.
 *
 * @param element The incoming element to be serialized
 * @param additionalInputProperties additional input properties map supplied to serialization
 * @param additionalOutputProperties additional output properties that can be populated by the
 *    serialization
 * @return The serialized element.
 */
@PublicEvolving
default byte[] serialize(
      T element,
      Map<String, Object> additionalInputProperties,
```

```
        Map<String, Object> additionalOutputProperties) {
    throw new RuntimeException(
            "Method serialize(T element, Map<String, Object> additionalInputProperties,\n"
                    + "Map<String, Object> additionalOutputProperties) should be Overridden.");
}
```

- Change the Kafka connector to call the new method in the DeserializeSchema interface called `deserializeWithAdditionalInputProperties(byte[] message, Map<String, Object> additionalProperties, Collector<T> out.` Passing a map<String, Object> Map of the Kafka headers in the additionalProperties parameter.

- Change the Apicurio Avro deserialization to use the header content to look up the ID, then call the registry client to get the associated Avro Schema.

In terms of compatibility for the Kafka sink:

With the new Flink Kafka connector change and the new Core Flink changes, the new capability is there. At some stage the lowest Flink level supported by the Kafka connector will contain the additionalProperties methods in code Flink. In the meantime so as not to impact existing users, there will be a new Kafka connector configuration parameter “passHeadersToSerDe” (default false) when this is true the code will reflectively call the appropriate serialization and deserialization methods passing the headers; if false then the headers will not be passed.

With the new Kafka connector and an old Flink – this will fail with an Exception indicating that the Flink level is too low.

With the original Kafka connector and a new Flink then the old deserialize / serialize method will be called and an error will be issued indicating that the Kafka level is too low.

Deserialisation

There existing Kafka deserialization for the writer schema passes down the message body to be deserialized. This works for the Confluent Schema Registry as it has the schema id in the payload after the magic byte.

For normal operation in Apicurio, the schema identifier (the global ID or content ID) is in a header. So, the Kafka connector needs to be changed to send the header content and the payload in `KafkaValueOnlyDeserializationSchemaWrapper deserialize.` Like this :

```
@Override
public void deserialize(ConsumerRecord<byte[], byte[]> message, Collector<T>
out)
      throws IOException {
      Map<String, Object> additionalPropertiesMap =  new HashMap<>();
      for (Header header : message.additionalProperties()) {
      additionalPropertiesMap.put(header.key(), header.value());
      }
      deserializationSchema.deserialize(message.value(),
additionalPropertiesMap, out);
}
```

New default methods will be added to the interfaces to tolerate the new additionalProperties content, but also be backwards compatible as they supply default implementation. The additionalProperties are sent as a map so as not to send down any Kafka specific class.

A new readSchema method in the Apicurio SchemaCoder will be added

```
public Schema readSchema(InputStream in, Map<String, Object>
additionalInputProperties) throws IOException {
```

The input stream and additionalProperties will be sent, so the Apicurio SchemaCoder which will try getting the ID from the headers, then 4 bytes from the payload then 8 bytes from the payload.

A new writeSchema method in the Apicurio SchemaCoder will be added

```
/**
* The Avro schema will be written depending on the configuration.
*
* @param schema Avro Schema to write into the output stream,
* @param out the output stream where the schema is serialized.
* @param inputProperties properties containing the topic name
* @param headers headers to populate
* @throws IOException Exception has occurred
*/
@Override
```

```
public void writeSchema(
    Schema schema,
    OutputStream out,
    Map<String, Object> inputProperties,
    Map<String, Object> outputProperties)
    throws IOException {
```

Serialisation

For normal operation in Apicurio, the schema identifier (the global ID or content ID) is in a header. So, the Kafka connector needs to be changed to send the header content. This is done by sending an empty output map that the serialization can populate with the header content. The Kafka connector then can add these headers onto the message. An inputProperties map is also sent, detailing the topic name and whether the serialization is for a key (this is required so that the Apicurio format knows which header to insert (a key or value orientated header).

The Avro Apicurio format will also register the schema in the Apicurio Registry.

Test Plan

1. Run Flink tests multiple times
2. Junits (end to end tests might be tricky as the code lives in 2 git repositories
3. Manually test as a source and sink
    1. With global ID in header
    2. With global ID in the payload as a long
    3. With global ID in the payload as a long
    4. With content ID.
    5. With keys and values
    6. With security

Other considerations

The implementation does not use the Apicurio SerDe libraries, as the Kafka application lives in the Kafka connector and the serialization and deserialization is done in core Flink. To use the SerDe libraries would take a larger refactor. For example being able to specify the artifact and schema resolver strategy is not present in this proposal.

We could do something more sophisticated around resolving artifacts using something more akin to the artifact resolver strategy. In this proposal, the artifact id is used to find the artifact during the registering of the schema and defaults to topic name, which is one of the artifact resolver strategies.

We have not included Debezium at this stage.

Rejected Alternatives

- Running Apicurio in Confluent mode, so that the Confluent Avro format could be used. This would be great for some users , but this FLIP is providing new function to facilitate working naturally with Apicurio schemas where the global ID is in the headers.