KIP-66: Single Message Transforms for Kafka Connect

Status

Current state: *Accepted*

Discussion thread: here

JIRA: KAFKA-3209

Released: 0.10.2.0

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

The framework for Single Message Transforms was released on 0.10.2.0 but only some of the built-in transformations were included with that version. The table below indicates what version each transformation was or will be released with. A few don't have the exact name as listed in the KIP because they were found to be slightly inaccurate during code review.

Transformation	Version
InsertField	0.10.2.0
ReplaceField	0.10.2.0
MaskField	0.10.2.0
ValueToKey	0.10.2.0
HoistField	0.10.2.0
ExtractField	0.10.2.0
SetSchemaMetadata	0.10.2.0
TimestampRouter	0.10.2.0
RegexRouter	0.10.2.0
Flatten	0.11.0.0
Cast	0.11.0.0
TimestampConverter	0.11.0.0

The Kafka documentation also includes references for each transformation.

Motivation

This proposal is for adding a record transformation API to Kafka Connect as well as certain bundled transformations. At the same time, we should not extend Connect's area of focus beyond moving data between Kafka and other systems. We will only support simple 1:{0,1} transformations – i.e. map and filter operations.

The objective is to:

- Allow for lightweight updates to records.

    - Some transformations **must** be performed before the data hits Kafka (source) or another system (sink) e.g. filtering certain types of events or sensitive information.
    - It's also useful for very light modifications that are easier to perform inline with the data import/export. It may be inconvenient to add stream processing into the mix for simple data massaging or control over routing.

- Benefit the growing connector ecosystem since some common options that are widely applicable can now be implemented once and reused. For example,
    - It is common for source connectors to allow configuring what format the topic name should follow based on some aspect of the source data, or in the case of sink connectors what 'bucket' (table, index etc.) a record should end up in based on the topic. This is configured in many different ways currently.
    - Some sink connectors allow inserting record metadata like the Kafka topic/partition/offset into the record key or value, while others do not. This information can get lost in translation if the functionality is absent and makes a connector less useful.
    - See the 'bundled transformations' section below for more examples.

Public Interfaces and Proposed Changes

Java API

```
// Existing base class for SourceRecord and SinkRecord, new self type parameter.
public abstract class ConnectRecord<R extends ConnectRecord<R>> {

    // ...

    // New abstract method:

    /** Generate a new record of the same type as itself, with the specified parameter values. **/
    public abstract R newRecord(String topic, Schema keySchema, Object key, Schema valueSchema, Object value,
Long timestamp);

}

public interface Transformation<R extends ConnectRecord<R>> extends Configurable, Closeable {

    // via Configurable base interface:
    // void configure(Map<String, ?> configs);

    /**
     * Apply transformation to the {@code record} and return another record object (which may be {@code record}
itself) or {@code null},
     * corresponding to a map or filter operation respectively. The implementation must be thread-safe.
     */
    R apply(R record);

    /** Configuration specification for this transformation. **/
    ConfigDef config();

    /** Signal that this transformation instance will no longer will be used. **/
    @Override
    void close();

}
```

Configuration

A transformation chain will be configured at the connector-level. The order of transformations is defined by the `transforms` config which represents a list of aliases. An alias in `transforms` implies that some additional keys are configurable:
- `transforms.$alias.type` – fully qualified class name for the transformation
- `transforms.$alias.*` – all other keys as defined in `Transformation.config()` are embedded with this prefix

Example:

```
transforms=tsRouter,insertKafkaCoordinates

transforms.tsRouter.type=org.apache.kafka.connect.transforms.TimestampRouter
transforms.tsRouter.topic.format=${topic}-${timestamp}
transforms.tsRouter.timestamp.format=yyyyMMdd

transforms.insertKafkaCoordinates.type=org.apache.kafka.connect.transforms.InsertInValue
transforms.insertKafkaCoordinates.topic=kafka_topic
transforms.insertKafkaCoordinates.partition=kafka_partition
transforms.insertKafkaCoordinates.offset=kafka_offset
```

Runtime changes

For source connectors, transformations are applied on the collection of `SourceRecord` retrieved from `SourceTask.poll()`.

For sink connectors, transformations are applied on the collection of `SinkRecord` before being provided to `SinkTask.put()`.

If the result of any `Transformation.apply()` in a chain is `null`, that record is discarded (not written to Kafka in the case of a source connector, or not provided to sink connector).

Bundled transformations

Criteria: SMTs that are shipped with Kafka Connect should be general enough to apply to many data sources & serialization formats. They should also be simple enough to not cause any additional library dependency to be introduced.

Beyond those being initially included with this KIP, transformations can be adopted for inclusion in future with JIRA/ML discussion to weigh the tradeoffs.

Name	Functionality	Rationale	Configuration
`Mask{Key, Value}`	Mask or replace the specified primitive fields, assuming there is a top-level `Struct`.	Obscure sensitive info like credit card numbers.	• `randomize.fields` – fields to replace with random data • `clobber.fields` – map of fields to replacement string/number
`InsertIn{Key, Value}`	Insert specified fields with given name, assuming there is a top-level `Struct`.	Widely applicable to insert certain record metadata.	• `topic` – the target field name for record topic • `partition` – the target field name for record partition • `offset` – the target field name for record offset • `timestamp` – the target field name for record timestamp
`TimestampRouter`	Timestamp-based routing.	Useful for temporal data e.g. application log data being indexed to a search system with a sink connector can be routed to a daily index.	• `topic.format` – format string which can contain `${topic}` and `${timestamp}` as placeholders for the original topic and the timestamp, respectively • `timestamp.format` – a format string compatible with `SimpleDateFormat`
`RegexRouter`	Regex-based routing.	There are too many inconsistent configs to route in different connectors.	• `regex` • `replacement` See http://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#replaceFirst(java.lang.String)
`ValueToKey`	Create or replace record key with data from record value.	Useful when a source connector does not populate the record key but only the value with a `Struct`.	• `fields` – list of field names to hoist into the record key as a primitive (single field) / `Struct` (multiple fields) • `force.struct` – force wrapping in a `Struct` even when it is a single field
`Flatten`	Flatten nested `Structs` inside a top-level `Struct`, omitting all other non-primitive fields.	Useful for sink connectors that can only deal with flat `Structs`.	• `delimiter` – the delimiter to use when flattening field names

`Replace`	Filter and rename fields.	Useful for lightweight data munging.	• `whitelist` – fields to include • `blacklist` – fields to exclude • `rename` – map of old field names to new field names
`NumericCasts`	Casting of numeric field to some specified numeric type.	Useful in conjunction with source connectors that don't have enough information and utilize an unnecessarily wide data type.	• `spec` – map of field name to type (i.e. boolean, int8, int16, int32, int64, float32, float64)
`TimestampConverter`	Convert datatype of a timestamp field.	Timestamps are represented in a ton of different ways, provide a transformation from going between strings, epoch times as longs, and Connect date/time types.	• `field` – the field name (optional, can be left out in case of primitive data) • `target.type` – desired type (i.e. string, long, Date, Time, Timestamp) • `format` – in case converting to or from a string, a `SimpleDateFormat`-compatible format string
`Hoist{Key,Value}ToStruct`	Wrap data in a `Struct`.	Useful when a transformation or sink connector expects `Struct` but the data is a primitive type.	• `schema.name` – name for the new `Struct` schema • `field` – field name for the original data within this `Struct`
`Extract{Key,Value}FromStruct`	Extract a specific field from a `Struct`.	The inverse of `Hoist{Key,Value}ToStruct`	• `field` – field name to extract
`Set{Key,Value}SchemaMetadata`	Set/clobber `Schema` name or version.	Allow setting or overriding the schema name and/or version where necessary.	• `name` – the schema name, allowing for `${topic}` as placeholder. • `version` – the schema version

Patterns for implementing data transformations

- Data transformations could be applicable to the key or the value of the record. We will have `*Key` and `*Value` variants for these transformations that reuse the common functionality from a shared base class.
- Some common utilities for data transformations will shape up:
    - Cache the changes they make to `Schema` objects, possibly only preserving last-seen one as the likelihood of source data `Schema` changing is low.
    - Copying of `Schema` objects with the possible exclusion of some fields, which they are modifying. Likewise, copying of `Struct` object to another `Struct` having a different `Schema` with the exception of some fields, which they are modifying.
    - Where fields are being added and a field name specified in configuration, we will want a consistent way to convey if it should be created as a required or optional field. We can use a leading '!' or '?' character for this purpose if the user wants to make a different choice than the default determined by the transformation.
    - `ConfigDef` does not provide a `Type.MAP`, but for the time being we can piggyback on top of `Type.LIST` and represent maps as a list of key-value pairs separated by `:`.
    - Where field names are expected, in some cases we should allow for getting at nested fields by allowing a dotted syntax which is common in such usage (and accordingly, will need some utilities around accessing a field that may be nested).
    - There are escaping considerations to several such configs, so we will need utilities that that assume a consistent escaping style (e.g. backslashes).

Compatibility, Deprecation, and Migration Plan

There are no backwards compatibility concerns. Transformation is an additional layer at the edge of record exchange between the framework and connectors.

Test Plan

Unit tests for runtime changes and each bundled transformation, as well as system test exercising a few different transformation chains.

Rejected Alternatives

Transformation chains as top-level construct

The current proposal is to have transformation chains be configured in the connector config under the prefix `transforms`. An alternative would be to reference a transformation chain by a name in the connector configuration, with the transformation chain specification managed separately by Connect.

However, the surface area for such a change is much larger - we would need additional REST APIs for creating, updating and validating transformation chain configs. The current proposal does not prevent taking this direction down the line.

Not including any transformations with Connect

In the interest of providing a better out-of-the-box experience and avoiding duplication of effort in the ecosystem, we will be bundling certain transformations with Connect.

One concern here is that we should have a well-defined criteria for what belongs in Connect vs external dependencies, which was addressed.