Connect Transforms - Proposed Design

Java API

```
public interface TransformableRecord<R extends TransformableRecord<R>> { // Implemented by SourceRecord and
SinkRecord

    String topic();

    Schema keySchema();

    Object key();

    Schema valueSchema();

    Object value();

    Long timestamp();

    R newRecord(String topic, Schema keySchema, Object key, Schema valueSchema, Object value, Long timestamp);

}

public interface Transformation<R extends TransformableRecord<R>> {

    void init(Map<String, String> config);

    R apply(R record);

    void close();

    ConfigDef config();

}
```

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
```

Application

For source connectors, transformations are applied on results from `SourceTask.poll()`.

For sink connectors, transformations are applied on the `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).

SourceTask.poll() → Transformation.apply()* → ? → Converter.fromConnectData() → Kafka

Kafka → Converter.toConnectData() → Transformation.apply()* → SinkTask.put()

Features

- Backwards compatible - no breaking change in the current APIs is required. Transformation is an additional layer at the edge of record exchange between the framework and connectors.
- Pluggable - initialized and configured somewhat similarly to `Converters`
- Stackable - can be chained in a defined order
- Fairly flexible - within the constraints of the `TransformableRecord` API and 1:{0,1} mapping
    - Any kind of filtering, renaming, masking operations on the data, adding fields, etc.
    - Filtering of records from the stream.
    - Routing for both source and sink - sink connectors can also just operate on the `TransformableRecord.topic` since the target 'bucket' (table, index, etc.) in always a function of that.
    - For any transformation that requires access to certain fields not exposed on the `TransformableRecord` i.e. `{SourceRecord, SinkRecord}.kafkaPartition`, `SinkRecord.kafkaOffset`, or `SinkRecord.timestampType` – it can set the R type parameter to specifically be `SourceRecord` and `SinkRecord` and use the relevant constructors instead of `newRecord()`. It can also just cast internally if an optional functionality requires access to such a field.

Example transformations

List of example transformations to demonstrate broad applicability - not in any particular order, and some more thought-through than others. We may want to include some of these with Connect itself to provide some useful out-of-the-box functionality and encourage standard ways to perform these transformations.

- `Mask`
    - Masks primitive fields: obscure sensitive info like credit card numbers.
    - Configure with list of fields to randomize or clobber.
- `Flatten`
    - Flatten nested `Structs` inside a top-level `Struct`, omitting all other non-primitive fields. Useful for connectors that can only deal with flat `Structs` like Confluent's JDBC Sink.
    - Configure with delimiter to use when flattening field names.
- `Replace`
    - Filter and rename fields. Useful for lightweight data munging.
    - Configure with whitelist and/or blacklist, map of fields to rename.
- `NumericCasts`
    - Casting of numeric field to some numeric type, useful in conjunction with source connectors that don't have enough information.
    - Configure with map of field to type (i.e. boolean, int8, int16, int32, int64, float32, float64).
- `TimestampRouter`
    - Useful for temporal data e.g. application log data being indexed to Elasticsearch with a sink connector can be routed to a daily index.
    - Configure with `SimpleDateFormat`-compatible timestamp format string, and a format string for the renamed `topic` that can have placeholders for original topic and the timestamp.
- `Insert`
    - Allow inserting into a top-level `Struct` record-level fields like the `topic`, `partition`, `offset`, `timestamp`. Can also allow a UUID field to be inserted.
    - Configure with names for desired fields.
- `RegexRouter`
    - Regex-based routing. There are too many inconsistent configs to route in different connectors.
    - Configure with matcher regex and replacement that can contain capture references.
- `TimestampConverter`
    - 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.
    - Configure with field name and desired type.
- `HoistToStruct`
    - Wrap data in a `Struct`.
    - Configure with schema name for the `Struct` schema and field name to insert the original data as.
- `ExtractFromStruct`
    - Extract a specific field from a `Struct`.
    - Configure with field name.
- `ValueToKey`
    - Useful when a source connector does not populate the `SourceRecord` key but only the value with a `Struct.`
    - Configure with list of field names to hoist into the record key as a primitive (single field) / `Struct` (multiple fields), and a flag to force wrapping in a `Struct` even when it is a single field.

Patterns for data transformations

- Data transformations could be applicable to the key or the value of the record. We could have *`Key` and *`Value` variants for these transformations that reuse the common functionality.
- Some common utilities for data transformations will probably 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 may want a consistent way to convey if it should be created as an optional field. E.g. a leading '?' character.
- Where field names are expected, we may want to allow for getting at nested fields by allowing a dotted syntax which is common in such usage (and accordingly, will need some reusable utilities around accessing a field that may be nested). Also implies actual dots in field names will need escaping.