KIP-610: Error Reporting in Sink Connectors

Status

Current state: *Adopted*

Discussion thread: Here

Vote thread: Here

JIRA: ⚠ Unable to render Jira issues macro, execution error.

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

Motivation

Currently, KIP-298 provides error handling in Kafka Connect that includes functionality such as retrying, logging, and sending errant records to a dead letter queue. However, the dead letter queue functionality from KIP-298 only supports error reporting within contexts of the transform operation, and key, value, and header converter operation. After records are sent to the connector for processing, there is no support for dead letter queue/error reporting functionality.

As stated in rejected alternatives, "**Write records that fail in the put() step of a sink connector to the dead letter queue**: since sink connectors can chose to batch records in a put() method, it is not clear what errors are caused by what records (they might be because of records that were immediately written to put(), or by some previous records that were processed later). Also, there might be connection issues that are not handled by the connector, and simply bubbled up as IOException (for example). Effectively, errors sent back to the framework from the put() method currently do not have sufficient context to determine the problematic records (if any). Addressing these issues would need a separate KIP. "

Thus, this proposal aims to extend KIP-298 and add error reporting functionality even after records are sent to connector tasks without adding redundancy in configuration.

Public Interfaces

This feature will directly make changes to the `SinkTaskContext` class with the addition of a new method and a new interface.

Method

This KIP will add a getter method to the `SinkTaskContext` class that will return a `error reporter` object, and by default this method will return null. Sink connectors that wish to use an error reporter can call this method within their tasks.

The method will look the following:

```
public interface SinkTaskContext {
...
        /**
    * Get the reporter to which the sink task can report problematic or failed {@link SinkRecord}
         * passed to the {@link SinkTask#put(Collection)} method. When reporting a failed record,
         * the sink task will receive a {@link Future} that the task can optionally use to wait until
         * the failed record and exception have been written to Kafka via Connect's DLQ. Note that
         * the result of this method may be null if this connector has not been configured with a DLQ.
         *
         * <p>This method was added in Apache Kafka 2.6. Sink tasks that use this method but want to
         * maintain backward compatibility so they can also be deployed to older Connect runtimes
         * should guard the call to this method with a try-catch block, since calling this method will result
in a
         * {@link NoSuchMethodException} or {@link NoClassDefFoundError} when the sink connector is deployed to
         * Connect runtimes older than Kafka 2.6. For example:
         * <pre>
         *     ErrantRecordReporter reporter;
         *     try {
         *         reporter = context.failedRecordReporter();
         *     } catch (NoSuchMethodError | NoClassDefFoundError e) {
         *         reporter = null;
         *     }
         * </pre>
         *
         * @return the reporter function; null if no error reporter has been configured for the connector
         * @since 2.6
         */
         default ErrantRecordReporter errantRecordReporter() {
        return null;
    }
}
```

Interface

This KIP will add an `ErrantRecordReporter` interface and will contain one method, `report(SinkRecord record, Throwable error)`.

The interface will look like the following:

```
/**
 * Component that the sink task can use as it {@link SinkTask#put(Collection<SinkRecord>)}
 * Reporter of problematic records and the corresponding problems.
 *
 * @since 2.6
 */
public interface ErrantRecordReporter {

    /**
     * Report a problematic record and the corresponding error to be written to the sink
     * connector's dead letter queue (DLQ).
     *
     * <p>This call is asynchronous and returns a {@link java.util.concurrent.Future Future}.
     * Invoking {@link java.util.concurrent.Future#get() get()} on this future will block until the
     * record has been written and then return the metadata for the record
     * or throw any exception that occurred while sending the record.
     * If you want to simulate a simple blocking call you can call the <code>get()</code> method
     * immediately.
     *
     * @param record the problematic record; may not be null
     * @param error  the error capturing the problem with the record; may not be null
     * @return a future that can be used to block until the record and error are written
     *         to the DLQ
     * @since 2.6
     */
    Future<Void> report(SinkRecord record, Throwable error);
}
```

Proposed Changes

Reporting

The error reporter interface will have one asynchronous reporting method. The method will accept the errant record and the exception thrown while processing the errant record, and return a `Future<RecordMetadata>`. In order to handle the case where the connector is deployed to an older version of AK, when calling the method failedRecordReporter() to get the error reporter, the developer must catch the resulting `NoClassDefError` or `NoSuchMethodError` and set the reporter to null. Thus, when processing errant records, the developer should check if the reporter is null; if it is, then the developer should wrap the original exception in a `ConnectException` and throw it. Records passed to the error reporter should be considered processed with respect to handling offsets, for example within the context of processing current offsets in `preCommit(...)`.

The error reporter will use the same configurations as the dead letter queue in KIP-298 to avoid redundant configuration. There will be no additional configurations for the Producer and `AdminClient` under the hood aside from the existing `producer.` and `admin.` configurations present in the worker configurations and `producer.override.` and `admin.override.` configurations present in the connector configurations. Serialization for the errant records will be done in the same manner as KIP-298.

Example Usage

The following is an example of how a sink task can use the error reporter and support connectors being deployed in earlier versions of the Connect runtime:

```
private ErrantRecordReporter reporter;

@Override
public void start(Map<String, String> props) {
  ...
  try {
    reporter = context.errantRecordReporter(); // may be null if DLQ not enabled
  } catch (NoSuchMethodException | NoClassDefFoundError e) {
    // Will occur in Connect runtimes earlier than 2.6
    reporter = null;
  }
}

@Override
public void put(Collection<SinkRecord> records) {
  for (SinkRecord record: records) {
    try {
      // attempt to send record to data sink
      process(record);
    } catch(Exception e) {
      if (reporter != null) {
        // Send errant record to error reporter
        Future<Void> future = reporter.report(record, e);
        // Optionally wait till the failure's been recorded in Kafka
        future.get();
      } else {
        // There's no error reporter, so fail
        throw new ConnectException("Failed on record", e);
      }
    }
  }
}
```

Synchrony

The error reporting functionality is asynchronous, although tasks can use the resulting future to wait for the record and exception to be written to Kafka.

Guarantees

The Connect framework also guarantees that by the time `preCommit(...)` is called on the task, the error reporter will have successfully and fully recorded all reported records with offsets at or before those passed to the `preCommit` method. Sink task implementations that need more strict guarantees can use the futures returned by `report(...)` to wait for confirmation that reported records have been successfully recorded.

Metrics

No new metrics will be added for the error reporter. Instead, the metrics detailed in KIP-298 for the dead letter queue will be used for the error reporter.

Errors Tolerance

The Errant Record Reporter will adhere to the existing DLQ error tolerance functionality. For example, if `errors.tolerance` is set to `ALL`, all errors will be tolerated; if the property is set to `NONE`, then the Errant Record Reporter will throw an exception detailing that the tolerance has been exceeded. Even if the developer chooses to catch and swallow any of these exceptions thrown during `report(...)`, the task is guaranteed to be killed following the completion of `put(...)`.

FileSinkTask Example

This KIP will update the `FileSinkTask` example to use the new API, to demonstrate proper error handling with the ability to use it on older versions of Kafka Connect.

Compatibility, Deprecation, and Migration Plan

Backwards Compatibility

This proposal is backward compatible such that existing sink connector implementations will continue to work as before. Developers can optionally modify sink connector implementations to use the new error reporting feature, yet still easily support installing and running those connectors in older Connect runtimes where the feature does not exist.

Moreover, to ensure that new connectors using this new method and interface can still be deployed on older versions of Kafka Connect, the developer should use a try catch block to catch the `NoSuchMethodError` or `NoClassDefFoundError` thrown by worker with an older version of AK.

Rejected Alternatives

1. **New library or interface:** creating a separate library like `connect-reporter` will cause redundancy in configuration, and creating any new interface or API will limit backwards compatibility. Deploying a connector on an old version of connect will not be a seamless integration with the introduction of a new interface
2. **Exposing `ErrorReporter` and `ProcessingContext` as public APIs:** the main issue with this design is that the the API exposes packages with `runtime` in them and package names cannot be changed.
3. **Labeling as a dead letter queue:** this is too specific of a label; using error reporter creates a more general contract of use case
4. **Batch error reporting:** this would introduce the need for keeping track of order, and the increase in throughput doesn't seem to outweigh the added complication
5. **Creating a setter method in `SinkTask` that accepts an error reporter object:** while this allows for backwards compatibility, it doesn't follow previous patterns of adding these types of methods to `SinkTaskContext` and is not a very aesthetic addition to the interface
6. **Creating an overloaded `put(...)` method that accepts an error reporter object and deprecating the original `put(...)` method:** deprecat ing the original method can cause confusion on which method to implement and if the wrong method is implemented, it can cause backwards compatibility issues
7. **Synchronous-only functionality:** this limits developer control on functionality and traditionally a lot of functionality within Kafka Connect has had asynchronous functionality
8. **Using `Callback` rather than `Future`:** because a sink task's callback is called from the producer thread, it can risk a poorly written sink task callback killing the reporter's producer without necessarily failing the task