FLIP-171: Async Sink

Status

Document the state by adding a label to the FLIP page with one of "discussion", "accepted", "released", "rejected".

Discussion thread	https://mail-archives.apache.org/mod_mbox/flink-dev/202106.mbox/%3CC83F4222-4D07-412D-9BD5-DB92D59DDF03%40amazon.de%3E
Vote thread	https://mail-archives.apache.org/mod_mbox/flink-dev/202106.mbox/%3C860A1499-0166-4BCF-B24D-FBE9C823D46E%40amazon.de%3E
JIRA	~~FLINK-24041~~ - [FLIP-171] Generic AsyncSinkBase RESOLVED
Release	1.15

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

Motivation

Apache Flink has a rich connector ecosystem that can persist data in various destinations. Flink natively supports Apache Kafka, Amazon Kinesis Data Streams, Elasticsearch, HBase, and many more destinations. Additional connectors are maintained in Apache Bahir or directly on GitHub. The basic functionality of these sinks is quite similar. They batch events according to user defined buffering hints, sign requests and send them to the respective endpoint, retry unsuccessful or throttled requests, and participate in checkpointing. They primarily just differ in the way they interface with the destination. Yet, all the above-mentioned sinks are developed and maintained independently.

We hence propose to create a sink that abstracts away this common functionality into a generic sink. Adding support for a new destination then just means creating a lightweight shim that only implements the specific interfaces of the destination using a client that supports async requests. Having a common abstraction will reduce the effort required to maintain all these individual sinks. It will also make it much easier and faster to create integrations with additional destinations. Moreover, improvements or bug fixes to the core of the sink will benefit all implementations that are based on it.

The design of the sink focusses on extensibility and a broad support of destinations. The core of the sink is kept generic and free of any connector specific dependencies. The sink is designed to participate in checkpointing to provide at-least once semantics, but it is limited to destinations that provide a client that supports async requests.

Public Interfaces

There are two user-facing aspects of the generic sink. First, an abstract class that is used to implement a new sink for a concrete destination. Second, the interface that is used by end-users, who want to leverage an existing sink to persist events in a destination. Appendix A contains a simplified sample implementation for a Kinesis Data Stream sink.

The async sink is based on FLIP-143 and FLIP-177. It is based on the following generic types to be extensible and remain agnostic to the destination.

`InputT` – elements of a DataStream that should be persisted in the destination

`RequestEntryT` – the payload of the element and additional metadata that is required to submit a single element to the destination

To use an existing sink, end users just need to specify the basic sink configuration (API endpoint URI, buffering hints, etc) and an `ElementConverter`, which provides a mapping from `InputT` to `RequestEntryT`.

ElementConverter Interface

```
/**
 * This interface specifies the mapping between elements of a stream to request
 * entries that can be sent to the destination. The mapping is provided by the
 * end-user of a sink, not the sink creator.
 * <p>
 * The request entries contain all relevant information required to create and
 * sent the actual request. Eg, for Kinesis Data Streams, the request entry
 * includes the payload and the partition key.
 */
public interface ElementConverter<InputT, RequestEntryT> extends Serializable {
    RequestEntryT apply(InputT element, SinkWriter.Context context);
}
```

To add support for a new destination, sink creators need to specify how to make an async request against the destination for a set of given RequestEntryTs . The logic also needs to identify from the response of the call which `RequestEntryT`s were not persisted successfully and resubmit them to the internal queue for later retry.

AsyncSinkWriter

```
public abstract class AsyncSinkWriter<InputT, RequestEntryT extends Serializable> implements SinkWriter<InputT,
Void, Collection<RequestEntryT>> {

    /**
     * This method specifies how to persist buffered request entries into the
     * destination. It is implemented when support for a new destination is
     * added.
     * <p>
     * The method is invoked with a set of request entries according to the
     * buffering hints (and the valid limits of the destination). The logic then
     * needs to create and execute the request against the destination (ideally
     * by batching together multiple request entries to increase efficiency).
     * The logic also needs to identify individual request entries that were not
     * persisted successfully and resubmit them using the {@code
     * requestResult}.
     * <p>
     * During checkpointing, the sink needs to ensure that there are no
     * outstanding in-flight requests.
     *
     * @param requestEntries a set of request entries that should be sent to the
     *                       destination
     * @param requestResult  a ResultFuture that needs to be completed once all
     *                       request entries that have been passed to the method
     *                       on invocation have either been successfully
     *                       persisted in the destination or have been
     *                       re-queued
     */
    protected abstract void submitRequestEntries(List<RequestEntryT> requestEntries,
ResultFuture<RequestEntryT> requestResult);

    ...
}
```

Internally, the `AsyncSinkWriter` buffers `RequestEntryT`s and invokes the `submitRequestEntries` method with a set of `RequestEntryT`s according to user specified buffering hints. The `AsyncSinkWriter` also tracks in-flight requests, ie, calls to the API that have been sent but not completed. During a commit, the sink enforces that all in-flight requests have completed and currently buffered `RequestEntryT`s are persisted in the application state snapshot.

AsyncSinkWriter Internals

```
    /**
     * The ElementConverter provides a mapping between for the elements of a
     * stream to request entries that can be sent to the destination.
     * <p>
     * The resulting request entry is buffered by the AsyncSinkWriter and sent
     * to the destination when the {@code submitRequestEntries} method is
     * invoked.
     */
    private final ElementConverter<InputT, RequestEntryT> elementConverter;


    /**
     * Buffer to hold request entries that should be persisted into the
     * destination.
     * <p>
     * A request entry contain all relevant details to make a call to the
     * destination. Eg, for Kinesis Data Streams a request entry contains the
     * payload and partition key.
     * <p>
     * It seems more natural to buffer InputT, ie, the events that should be
     * persisted, rather than RequestEntryT. However, in practice, the response
     * of a failed request call can make it very hard, if not impossible, to
     * reconstruct the original event. It is much easier, to just construct a
     * new (retry) request entry from the response and add that back to the
     * queue for later retry.
```

```
 */
private final Deque<RequestEntryT> bufferedRequestEntries = new ArrayDeque<>();


/**
 * Tracks all pending async calls that have been executed since the last
 * checkpoint. Calls that completed (successfully or unsuccessfully) are
 * automatically decrementing the counter. Any request entry that was not
 * successfully persisted needs to be handled and retried by the logic in
 * {@code submitRequestsToApi}.
 * <p>
 * There is a limit on the number of concurrent (async) requests that can be
 * handled by the client library. This limit is enforced by checking the
 * size of this queue before issuing new requests.
 * <p>
 * To complete a checkpoint, we need to make sure that no requests are in
 * flight, as they may fail, which could then lead to data loss.
 */
private int inFlightRequestsCount;


@Override
public void write(InputT element, Context context) throws IOException, InterruptedException {
    // blocks if too many elements have been buffered
    while (bufferedRequestEntries.size() >= MAX_BUFFERED_REQUESTS_ENTRIES) {
        mailboxExecutor.yield();
    }

    bufferedRequestEntries.add(elementConverter.apply(element, context));

    // blocks if too many async requests are in flight
    flush();
}


/**
 * Persists buffered RequestsEntries into the destination by invoking {@code
 * submitRequestEntries} with batches according to the user specified
 * buffering hints.
 *
 * The method blocks if too many async requests are in flight.
 */
private void flush() throws InterruptedException {
    while (bufferedRequestEntries.size() >= MAX_BATCH_SIZE) {

        // create a batch of request entries that should be persisted in the destination
        ArrayList<RequestEntryT> batch = new ArrayList<>(MAX_BATCH_SIZE);

        while (batch.size() <= MAX_BATCH_SIZE && !bufferedRequestEntries.isEmpty()) {
            try {
                batch.add(bufferedRequestEntries.remove());
            } catch (NoSuchElementException e) {
                // if there are not enough elements, just create a smaller batch
                break;
            }
        }

        ResultFuture<RequestEntryT> requestResult =
                failedRequestEntries -> mailboxExecutor.execute(
                        () -> completeRequest(failedRequestEntries),
                        "Mark in-flight request as completed and requeue %d request entries",
                        failedRequestEntries.size());

        while (inFlightRequestsCount >= MAX_IN_FLIGHT_REQUESTS) {
            mailboxExecutor.yield();
        }

        inFlightRequestsCount++;
        submitRequestEntries(batch, requestResult);
    }
}
```

```
/**
 * Marks an in-flight request as completed and prepends failed requestEntries back to the
 * internal requestEntry buffer for later retry.
 *
 * @param failedRequestEntries requestEntries that need to be retried
 */
private void completeRequest(Collection<RequestEntryT> failedRequestEntries) {
    inFlightRequestsCount--;

    // By just iterating through failedRequestEntries, it reverses the order of the
    // failedRequestEntries. It doesn't make a difference for kinesis:putRecords, as the api
    // does not make any order guarantees, but may cause avoidable reorderings for other
    // destinations.
    failedRequestEntries.forEach(bufferedRequestEntries::addFirst);
}


/**
 * In flight requests will be retried if the sink is still healthy. But if in-flight requests
 * fail after a checkpoint has been triggered and Flink needs to recover from the checkpoint,
 * the (failed) in-flight requests are gone and cannot be retried. Hence, there cannot be any
 * outstanding in-flight requests when a commit is initialized.
 *
 * <p>To this end, all in-flight requests need to completed before proceeding with the commit.
 */
@Override
public List<Void> prepareCommit(boolean flush) throws IOException, InterruptedException {
    if (flush) {
        flush();
    }

    // wait until all in-flight requests completed
    while (inFlightRequestsCount > 0) {
        mailboxExecutor.yield();
    }

    return Collections.emptyList();
}


/**
 * All in-flight requests have been completed, but there may still be
 * request entries in the internal buffer that are yet to be sent to the
 * endpoint. These request entries are stored in the snapshot state so that
 * they don't get lost in case of a failure/restart of the application.
 */
@Override
public List<Collection<RequestEntryT>> snapshotState() throws IOException {
    return Collections.singletonList(bufferedRequestEntries);
}
```

Limitations

- The sink is designed for destinations that provide an async client. Destinations that cannot ingest events in an async fashion cannot be supported by the sink.
- The sink usually persist InputTs in the order they are added to the sink, but reorderings may occur, eg, when RequestEntryTs need to be retried.
- We are not focusing on support for exactly-once semantics beyond simple upsert capable and idempotent destinations at this point.

Appendix A – Simplified example implementation of a sink for Kinesis Data Streams

Example ElementConverter for Kinesis Data Streams

This is the functionality that needs to be implemented by end users of the sink. It specifies how an element of a DataStream is mapped to a `PutRecordsRequestEntry` that can be submitted to the Kinesis Data Streams API.

ElementConverter for Kinesis Data Streams

```
new ElementConverter<InputT, PutRecordsRequestEntry>() {
    @Override
    public PutRecordsRequestEntry apply(InputT element, SinkWriter.Context context) {
        return PutRecordsRequestEntry
                .builder()
                .data(SdkBytes.fromUtf8String(element.toString()))
                .partitionKey(String.valueOf(element.hashCode()))
                .build();
    }
}
```

Simplified AsyncSinkWriter for Kinesis Data Streams

This is a simplified sample implementation of the `AsyncSinkWriter` for Kinesis Data Streams. Given a set of buffered `PutRecordRequestEntries`, it creates and submits a batch request against the Kinesis Data Stream API using the `KinesisAsyncClient`. The response of the API call is then checked for events that were not persisted successfully (eg, because of throttling or network failures) and those events are added back to the internal buffer of the A `syncSinkWriter`.

AmazonKinesisDataStreamWriter

```
private class AmazonKinesisDataStreamWriter extends AsyncSinkWriter<InputT, PutRecordsRequestEntry> {

    @Override
    protected void submitRequestEntries(List<PutRecordsRequestEntry> requestEntries,
ResultFuture<PutRecordsRequestEntry> requestResult) {

        // create a batch request
        PutRecordsRequest batchRequest = PutRecordsRequest
                .builder()
                .records(requestEntries)
                .streamName(streamName)
                .build();

        // call api with batch request
        CompletableFuture<PutRecordsResponse> future = client.putRecords(batchRequest);

        // re-queue elements of failed requests
        future.whenComplete((response, err) -> {
                if (response.failedRecordCount() > 0) {
                    ArrayList<PutRecordsRequestEntry> failedRequestEntries = new ArrayList<>(response.
failedRecordCount());
                    List<PutRecordsResultEntry> records = response.records();

                    for (int i = 0; i < records.size(); i++) {
                        if (records.get(i).errorCode() != null) {
                            failedRequestEntries.add(requestEntries.get(i));
                        }
                    }

                    requestResult.complete(failedRequestEntries);
                } else {
                    requestResult.complete(Collections.emptyList());
                }

                //TODO: handle errors of the entire request...
            });
    }

    ...
}
```

Rejected Alternatives

The sink needs a way to build back pressure, eg, if the throughput limit of the destination is exceeded. Initially we were planning to adopt the isAvailable pattern from the source interface. But the benefits are too vague at this point and it would require substantial changes to the sink API. We'll hence start with a blocking implementation of the `write` function and see how far we get.

isAvailable pattern

```
/**
 * Signals if enough RequestEntryTs have been buffered according to the user
 * specified buffering hints to make a request against the destination. This
 * functionality will be added to the sink interface by means of an
 * additional FLIP.
 *
 * @return a future that will be completed once a request against the
 * destination can be made
 */
public CompletableFuture<Void> isAvailable() {
    ...
}
```