The Outbox Pattern: A practical example with Kafka

# Prelude

I have a Spring Boot based [simple message aggregator](http://palashray.com/kafka-message-aggregation-using-camel-and-spring-boot/) based on Kafka. It aggregates a batch of messages from a Kafka Topic and publishes the aggregated result on a new Topic.

# Problem Statement

I have deployed 2 instances of this application, and both of them are active at any given point. When I start publishing messages on Kafka, only one of these would latch on to the message batch (as all messages of the same batch have the same Kafka.Key) and then start aggregating. Now if that instance crashes, what happens?

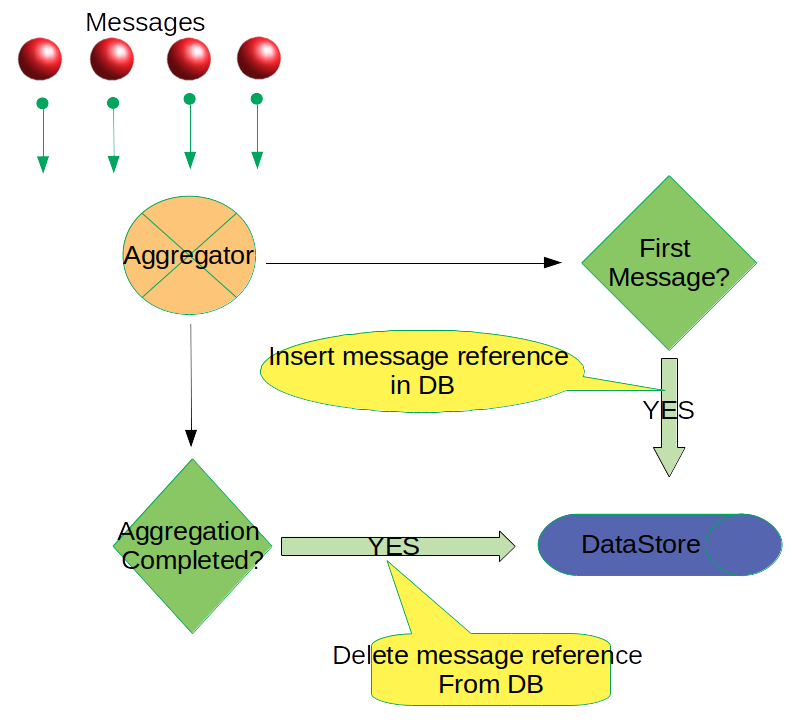
1. Kafka rebalances its consumers and the 2nd instance picks up the remaining messages of the same batch
2. The 2nd instance does not know that it has missed many messages, so after getting the end signal to complete the aggregation, it publishes the aggregated result on to the Kafka Topic
3. However, this aggregated result is incorrect, as it has missed a number of messages from the batch which were consumed by the 1st instance

# Solution

One of the ways of solving this is through the Outbox Pattern.

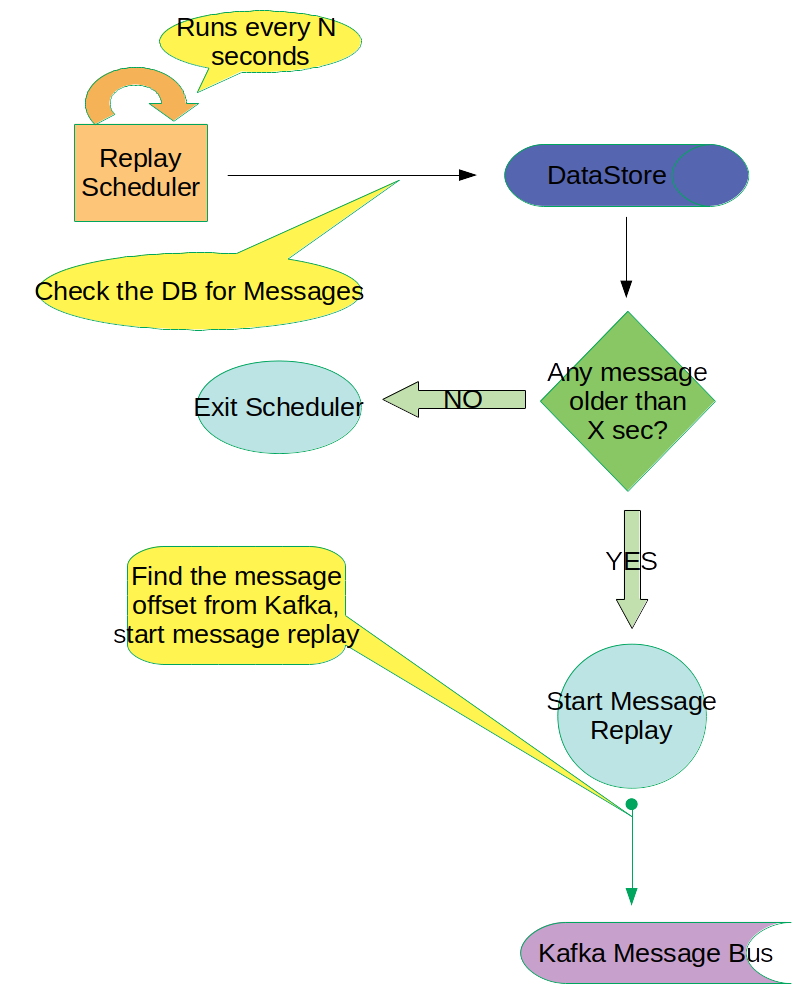
## High Level Design

### The Outbox Persister



1. Messages are consumed by the Aggregtor from a given Kafka Topic
2. The Aggregator saves a reference of the 1st Message (the CorrelationId for the batch, the Kafka Topic, Partition ID, Message Offset and the Timestamp) of a given batch into a Persistent Data Store, called the Outbox
3. If the aggregation operation is successful, this message reference is deleted and the aggregated result is published on a Kafka Topic
4. If the Aggregator instance crashes before the aggregation is completed, the reference of the 1st message of the batch remains in the Data Store

### The Message Replayer



1. The Message Replayer is a scheduled job that runs every N seconds
2. It polls the Persistent Data Store to see if any records exists
3. If it finds any records, it looks to see if the Timestamp of the message is older by X seconds from the current time
4. If YES, it latches on to the specific Topic and Partition in Kafka and starts replaying all the messages from the given offset of the 1st Message
5. The Replayer also sets a special header in the message to indicate that a Message Replay is in progress
6. It stops when it finds a different CorrelationId in a message
7. Thus, the Aggregator receives all the messages from the same batch
8. The Aggregator also can differentiate a normal operation from a *Replay Operation*

## Low Level Design

For a quick start, we have taken the entire source code of the [kafka-camel-aggregation-simple](https://github.com/paawak/spring-boot-demo/tree/master/kafka-spring/kafka-camel-aggregation-simple). We rename this to [kafka-camel-aggregation-with-replay](https://github.com/paawak/spring-boot-demo/tree/master/kafka-spring/kafka-camel-aggregation-with-replay). Since the 2 code bases are identical, we will refer a lot to [this blog entry](http://palashray.com/kafka-message-aggregation-using-camel-and-spring-boot/), which has the details of the basic structure. Here, I will only highlight the changes that we have to bring in to cater to the Outbox Pattern.

### The Persistent Data Store

First of all, we would need a Data Store to persist the message details. We would be using MySql. The schema looks like this:

CREATE TABLE message\_outbox (

correlation\_id varchar(50) PRIMARY KEY NOT NULL,

processor\_id varchar(50) NOT NULL,

topic\_name varchar(200) NOT NULL,

partition\_id smallint NOT NULL,

offset bigint NOT NULL,

insert\_time timestamp NOT NULL DEFAULT NOW()

);

### Replay Scheduler

Every fixed time interval, a scheduler polls the Data Store for new message reference to process.

@Service

public class BackfillScheduler {

private final KafkaMessageReplayer kafkaMessageReplayer;

@Scheduled(initialDelay = 5000, fixedDelay = 60000)

public void checkForBackill() {

LOG.info("checking for backfill...");

String sql =

"SELECT \* FROM message\_outbox WHERE TIMESTAMPDIFF(MINUTE, insert\_time, NOW()) > :intervalInMinutes";

Map<String, Object> params = new HashMap<>();

params.put("intervalInMinutes", MIN\_INTERVAL\_MINUTES);

List<MessageOutBox> outbox =

jdbcTemplate.query(sql, params, (ResultSet resultSet, int rowNum) -> {

String correlationId = resultSet.getString("correlation\_id");

String topicName = resultSet.getString("topic\_name");

int partitionId = resultSet.getInt("partition\_id");

long offset = resultSet.getLong("offset");

LOG.trace("topicName: {}, partitionId: {}, offset: {}, correlationId: {}",

topicName, partitionId, offset, correlationId);

return new MessageOutBox(correlationId, topicName, partitionId, offset);

});

if (outbox.isEmpty()) {

LOG.info("No backfill found");

return;

} else {

LOG.info("{} records found for backfill", outbox.size());

}

outbox.forEach(messageOutBox -> kafkaMessageReplayer.replayMessages(messageOutBox.topicName,

messageOutBox.partitionId, messageOutBox.offset, messageOutBox.correlationId));

}

}

### Kafka Message Replayer

This component is responsible for reading messages from a particular message-offset in a given Kafka Topic and Partition ID, and having the given CorrelationID. The Replayer keeps reading the messages till it hits a message which has a different CorrelationID. Thus, it replays all the messages that belongs to the same batch, indicated by the same CorrelationID.

@Service

public class KafkaMessageReplayer {

private static final Logger LOG = LoggerFactory.getLogger(KafkaMessageReplayer.class);

private final ProducerTemplate producerTemplate;

private final String kafkaBrokers;

public KafkaMessageReplayer(ProducerTemplate producerTemplate,

@Value("${spring.kafka.bootstrap-servers}") String kafkaBrokers) {

this.producerTemplate = producerTemplate;

this.kafkaBrokers = kafkaBrokers;

}

public void replayMessages(String topicName, int partitionId, long offset,

String correlationId) {

LOG.info(

"Kafka message replay in progress: topicName: {}, partitionId: {}, offset: {}, correlationId: {}",

topicName, partitionId, offset, correlationId);

KafkaConsumer<String, String> kafkaConsumer = getKafkaConsumer();

TopicPartition topicPartition = new TopicPartition(topicName, partitionId);

kafkaConsumer.assign(Arrays.asList(topicPartition));

kafkaConsumer.seek(topicPartition, offset);

while (true) {

ConsumerRecords<String, String> records = kafkaConsumer.poll(Duration.ofMillis(500));

for (ConsumerRecord<String, String> record : records) {

String key = record.key();

if (!key.equals(correlationId)) {

return;

}

String value = record.value();

LOG.trace("correlationId: {}, value: {}", correlationId, value);

Map<String, Object> headers = new HashMap<>();

headers.put(RouteConstants.BACKFILL\_IN\_PROGRESS, Boolean.TRUE);

headers.put(KafkaConstants.KEY, correlationId);

headers.putAll(StreamSupport.stream(record.headers().spliterator(), false)

.collect(Collectors.toMap(header -> header.key(),

header -> new String(header.value()))));

producerTemplate.sendBodyAndHeaders(RouteConstants.AGGREGATION\_CHANNEL, value,

headers);

}

}

}

private KafkaConsumer<String, String> getKafkaConsumer() {

Properties props = new Properties();

props.put("bootstrap.servers", kafkaBrokers);

props.put("group.id", "bank-detail-backfill");

props.put("key.deserializer", StringDeserializer.class);

props.put("value.deserializer", StringDeserializer.class);

return new KafkaConsumer<>(props);

}

}

### Changes in the BankDetailAggregationStrategy

The BankDetailAggregationStrategy is the component which persists the first message reference to the Data Store. This happens in the *aggregate()* method when the *oldExchange* is *null*, it knows that this is the first message.

However, it has to also check whether the first message is *dirty*, i.e., partial messages due to an Aggregator crash. This is done by checking the Data Store for a message with the same CorrelationId. If one is found, it means that this operation is *dirty*, the result of which should be ignored.

For all the subsequent messages, the *oldExchange* and the *newExchange* are added up to return a partially aggregated result. Care should be taken to merge the headers as well in the partial result that is returned.

@Override

public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {

if (oldExchange == null) {

if (Boolean.TRUE.equals(newExchange.getIn()

.getHeader(RouteConstants.BACKFILL\_IN\_PROGRESS, Boolean.class))) {

LOG.debug("++++ Message replay in progress");

return newExchange;

}

// insert only the first message in the group

if (checkforDirtyAggregation(newExchange)) {

newExchange.getIn().setHeader(PRECOMPLETE\_DIRTY\_AGGREGATION, Boolean.TRUE);

return newExchange;

}

persistMessageOffsetDetails(newExchange);

return newExchange;

}

JobCount partialResults = oldExchange.getIn().getBody(JobCount.class);

JobCount newMessage = newExchange.getIn().getBody(JobCount.class);

oldExchange.getIn().setBody(doAggregation(newMessage, partialResults), JobCount.class);

// any header that is used by the predicate should be set in the message

// returned from this method

Map<String, Object> headers = new HashMap<>();

headers.putAll(oldExchange.getIn().getHeaders());

headers.putAll(newExchange.getIn().getHeaders());

oldExchange.getIn().setHeaders(headers);

return oldExchange;

}

private JobCount doAggregation(JobCount newMessage, JobCount partialResults) {

JobCount aggregate = new JobCount();

aggregate.setAdminCount(newMessage.getAdminCount() + partialResults.getAdminCount());

aggregate.setBlueCollarCount(

newMessage.getBlueCollarCount() + partialResults.getBlueCollarCount());

aggregate.setEntrepreneurCount(

newMessage.getEntrepreneurCount() + partialResults.getEntrepreneurCount());

aggregate.setHouseMaidCount(

newMessage.getHouseMaidCount() + partialResults.getHouseMaidCount());

aggregate.setManagementCount(

newMessage.getManagementCount() + partialResults.getManagementCount());

aggregate.setRetiredCount(newMessage.getRetiredCount() + partialResults.getRetiredCount());

aggregate.setSelfEmployedCount(

newMessage.getSelfEmployedCount() + partialResults.getSelfEmployedCount());

aggregate.setServicesCount(

newMessage.getServicesCount() + partialResults.getServicesCount());

aggregate.setStudentCount(newMessage.getStudentCount() + partialResults.getStudentCount());

aggregate.setTechnicianCount(

newMessage.getTechnicianCount() + partialResults.getTechnicianCount());

aggregate.setUnemployedCount(

newMessage.getUnemployedCount() + partialResults.getUnemployedCount());

aggregate.setUnknownCount(newMessage.getUnknownCount() + partialResults.getUnknownCount());

return aggregate;

}

private void persistMessageOffsetDetails(Exchange message) {

if (Boolean.TRUE.equals(

message.getIn().getHeader(RouteConstants.BACKFILL\_IN\_PROGRESS, Boolean.class))) {

return;

}

String sql =

"INSERT INTO message\_outbox (correlation\_id, processor\_id, topic\_name, partition\_id, offset) VALUES (:correlationId, :processorId, :topicName, :partitionId, :offset)";

Map<String, Object> params = new HashMap<>();

params.put("correlationId", getCorrelationId(message));

String topicName = message.getIn().getHeader(KafkaConstants.TOPIC, String.class);

params.put("topicName", topicName);

params.put("partitionId",

message.getIn().getHeader(KafkaConstants.PARTITION, Integer.class));

params.put("offset", message.getIn().getHeader(KafkaConstants.OFFSET, Long.class));

params.put("processorId", System.getenv("processorId"));

jdbcTemplate.update(sql, params);

}

The *matches()* method should now return false if the operation is *dirty*, so that it *times-out* without removing the message reference from the Data Store.

@Override

public boolean matches(Exchange oldExchange) {

Boolean completionCommand = oldExchange.getIn()

.getHeader(RouteConstants.COMPLETE\_JOB\_AGGREGATION\_COMMAND, Boolean.class);

if (completionCommand == null) {

return false;

}

Boolean isDirty =

oldExchange.getIn().getHeader(PRECOMPLETE\_DIRTY\_AGGREGATION, Boolean.class);

if (isDirty == null) {

return completionCommand;

}

boolean shouldComplete = !isDirty && completionCommand;

LOG.debug("\*\*\*\*\*\*shouldComplete: {}", shouldComplete);

return shouldComplete;

}

Lastly, the *onCompletion()* method, which is called when the aggregation operation is successful, removes the message reference from the Data Store.

@Override

public void onCompletion(Exchange exchange) {

String correlationId = getCorrelationId(exchange);

LOG.info("########### Aggregation is complete for {}", correlationId);

String sql = "DELETE FROM message\_outbox WHERE correlation\_id = :correlationId";

Map<String, Object> params = new HashMap<>();

params.put("correlationId", correlationId);

jdbcTemplate.update(sql, params);

LOG.info("Record from \*message\_outbox\* table deleted for {}", correlationId);

}

## Drawbacks Of The Current Design

The main drawback is the time needed for the message replay. When an instance of the aggregator dies, we need to wait till all the messages of the same batch are published by the producer before we can replay.

There is also a chance that the replay and the consumption of messages from Kafka overlaps with each other, and happens at the same time. This is a real possibility in cases where the no. of messages in a batch are very high.

# Source Code

The fully working source code can be found here:

<https://github.com/paawak/spring-boot-demo/tree/master/kafka-spring/kafka-camel-aggregation-with-replay>

# Running The Example

## As Simple Java App

This is a Spring Boot application, so running it locally as a Java Application is just about running the main class KafkaCamelDemoApplication.

However, the below environment variable should be set to uniquely identify this node.

-DprocessorId=camel\_8080

You can run another instance by changing the port

-Dserver.port=8080

## As Docker Image

docker run -it -p 8080:8080 -e server.port=8080 -e processorId=camel\_8080 -e spring.profiles.active=default paawak/kafka-camel-aggregation-with-replay:latest

### Allowing docker host to access MySql

When running this as a Docker image, in order for MySql to give access to it, we would have to run the below commands on the MySql command prompt:

CREATE USER 'root'@'172.17.0.9' IDENTIFIED BY 'my\_pswd';

GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'172.17.0.9' WITH GRANT OPTION;

FLUSH PRIVILEGES;

# Further Reading

<https://dzone.com/articles/implementing-the-outbox-pattern>

<https://microservices.io/patterns/data/transactional-outbox.html>