KIP-656: MirrorMaker2 Exactly-once Semantics

Status

Current state: *Draft*

Discussion thread: https://lists.apache.org/thread.html/r9d1c89b871792655cd14ff585980bb0ace639d85d9200e239cc0e1cd%40%3Cdev.kafka.apache.org%3E

Voting thread: https://lists.apache.org/thread.html/rbfe08bfb15e14db14c54d1ca5c86bfcd17dc952084ad0a4dec8255b6%40%3Cdev.kafka.apache.org%3E

JIRA: ~~KAFKA-10339~~ - **MirrorMaker2 Exactly-once Semantics** RESOLVED

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

Motivation

MirrorMaker2 is currently implemented on Kafka Connect Framework, more specifically the Source Connector / Task, which do not provide exactly-once semantics (EOS) out-of-the-box, as discussed in https://github.com/confluentinc/kafka-connect-jdbc/issues/461, https://github.com/apache/kafka/pull/5553, ~~KAFKA-6080~~ - Transactional EoS for source connectors RESOLVED and KAFKA-3821 - Allow Kafka Connect source tasks to produce offset without writing to topics OPEN . Therefore current MirrorMaker2 does not provide EOS as well.

This proposal is to provide an option to enable EOS for MirrorMaker 2 if no data loss or duplicate data is preferred. The key idea of this proposal is to extend SinkTask with a brand new **MirrorSinkTask** implementation, which has an option to manage the consumer offsets in transactional way (similar high-level idea as HDFS Sink Connector), such that the messages can be delivered across clusters:

(1) **Exactly-once**: by a transactional producer in MirrorSinkTask and consumer offsets are committed within a transaction by the transactional producer

OR

(2) **At-least-once**: by a non-transactional producer in MirrorSinkTask and consumer offsets are committed by the consumer from WorkerSinkTask separately

The reasons to implement a brand new **MirrorSinkTask** that extends SinkTask, rather than working on the existing MirrorSourceTask, are the following:

- as mentioned above, Kafka Connect Source Connector / Task do not provide EOS by nature, mostly because of async and periodic source task offset commit, which in MirorrMaker case, the "offset" is consumer offset. So this is one blocker to enable EOS without a lots of changes in Worker SourceTask.
- MirrorSourceTask explicitly avoided using subscribe() and instead handle rebalances and new topic-partitions explicitly. While **MirrorSinkTask** that extends SinkTask is initiated by WorkerSinkTask. WorkerSinkTask uses consumer.subscribe() in which the benefits of rebalances and auto-detection of new topics/partitions are out-of-the-box. When the consumer or its rebalance handling is upgraded in WorkerSinkTask as part of Kafka Connect, **MirrorSinkTask** will take the advantages transparently.
- Since **MirrorSinkTask** is a new implementation, on the other end we can fully control how a producer is created (transactional v.s. non-transactional) and handle various Exception cases (especially in transactional mode), purely in the new implementation, rather than changing the existing producer in WorkerSourceTask
- HDFS Sink Connector already achieved EOS, we can correctly implement MirrorSinkTask based on the methods of SinkTask by referring the best practices from HDFS Sink Connector.

Public Interfaces

New classes and interfaces include:

MirrorMakerConfig

Name	Type	Default	Doc
connector.type	string	source	if "source", the existing MirrorSourceConnector will be launched. If "sink", the new MirrorSinkConnector will be launched with further option to enable EOS

MirrorConnectorConfig

Name	Type	Default	Doc
transaction.producer	boolean	false	if True, EOS is enabled between consumer and producer

Proposed Changes

MirrorSinkTask

There are several key challenges to make EOS happen across clusters. Those challenges are discussed here one-by-one:

(1) in MirrorMaker case, there are source and target clusters. Normally the consumer pulls the data and stores its offsets in the source cluster, then the producer takes over the data from the consumer and sends them to target cluster. However Kafka transaction can not happen across clusters out-of-the-box. If we want EOS across clusters, what modifications need to be done?

A: The short answer is consumer group offsets are managed, committed by the transactional producer and are stored on the **target** cluster instead.

However the consumer still has to live on the source cluster in order to pull the data, but “source-of-truth” offsets are no longer stored in the source cluster. We propose to use the following idea to rewind the consumer correctly when data mirroring task restarts or rebalances, while the “source-of-truth” of consumer offsets are stored in the target cluster: (the pseudocode are shown in below)

- Consumer offsets are stored on the target cluster using a “fake” consumer group, that can be created programmatically as long as we know the name of consumer group. The “fake” means there would be no actual records being consumed by the group, just offsets being stored in *__consumer_offsets* topic. However, the *__consumer_offsets* topic on the target cluster (managed by the “fake” consumer group) is the “source of truth” offsets.
- With the “fake” consumer group on target cluster, MirrorSinkTask don't rely on Connect's internal offsets tracking or *__consumer_offsets* on the source cluster.
- the consumer offsets are only written by the producer evolved in the transaction to the target cluster.
- all records are written in a transaction, as if in the single cluster
- when MirrorSinkTask starts or rebalances, it loads initial offsets from __consumer_offsets on the target cluster.

The outcome of the above idea:

- if the transaction succeeds, the *__consumer_offsets* topic on the target cluster is updated by following the current protocol of Exactly-Once framework
- if the transaction aborts, all data records are dropped, and the __consumer_offsets topic on the target cluster is not updated.
- when MirrorSinkTask starts/restarts, it resumes at the last committed offsets, as stored in the target cluster.

Some items to pay attention in order to make above idea work correctly:

- If consumer group already exists on source cluster, while the "fake" consumer group (with same Group Id) on the target cluster does not exist or its offsets lower than the high watermark. To avoid duplicate data, it may need to do a one-time offline job to sync the offsets from source cluster to target cluster.

(2) Since the offsets of the consumer group on the source cluster is NOT "source of truth" and not involved in the transaction. Are they still being updated? Do we still need them in some cases?

A: the offsets of the consumer group on the source cluster are still being updated periodically and independently by the logics in WorkerSInkTask. However they may be lagging behind a little, since (1) they are not involved in transaction, (2) they are periodically committed.

However, they may be still useful in some cases: (1) measure the replication lag between the upstream produce on the source cluster and MirrorMaker's consumption. (2) restore the lost "fake" consumer group with small # of duplicate data.

The following is the pseudocode illustrates the high-level key implementation:

MirrorSinkTask

```
    private boolean isTransactional = config.getTransactionalProducer();
    private boolean transactionInProgress = false;
        protected Map<TopicPartition, OffsetAndMetadata> offsetsMap = new HashMap<>();
    private Set<TopicPartition> taskTopicPartitions;
        private KafkaProducer<byte[], byte[]> producer;
    private String connectConsumerGroup;

    @Override
        public void start(Map<String, String> props) {
                config = new MirrorTaskConfig(props);
            taskTopicPartitions = config.taskTopicPartitions();
        isTransactional = config.transactionalProducer();
            producer = initProducer(isTransactional);
        connectConsumerGroup = getSourceConsumerGroupId();
                if (isTransactional) {
                loadContextOffsets();
        }
    }

    @Override
    public void open(Collection<TopicPartition> partitions) {
                if (isTransactional) {
                loadContextOffsets();
                }
    }

    private void loadContextOffsets() {
                Map<TopicPartition, OffsetAndMetadata> initOffsetsOnTarget = listTargetConsumerGroupOffsets
(connectConsumerGroup);

        Set<TopicPartition> assignments = context.assignment();

        // only keep the offsets of the partitions assigned to this task
        Map<TopicPartition, Long> contextOffsets = assignments.stream()
                                                                      .filter(x -> currentOffsets.containsKey
(x))
                                                              .collect(Collectors.toMap(
                                                                      x -> x, x -> currentOffsets.get(x)));

        context.offset(contextOffsets);
    }

        protected KafkaProducer<byte[], byte[]> initProducer(boolean isTransactional) {
        Map<String, Object> producerConfig = config.targetProducerConfig();
        if (isTransactional) {
                String transactionId = getTransactionId();
                log.info("use transactional producer with Id: {} ", transactionId);
            producerConfig.put(ProducerConfig.ACKS_CONFIG, "all");
            producerConfig.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
                producerConfig.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionId);
                producerConfig.put(ProducerConfig.CLIENT_ID_CONFIG, transactionId);
        }
        return MirrorUtils.newProducer(producerConfig);
        }

    /**
     * Per some articles, to avoid ProducerFencedException, transaction id is suggested to set application name
+ hostname
     * Each MirrorSinkTask is also assigned with different set of <topic, partition>. To get unique transaction
id,
     * one way is to append connector name, hostname and string of each <topic, partition> pair
     */
    protected String getTransactionId() {
            return getHostName() + "-" + getUniquePredictableStr();
    }

    @Override
        public void put(Collection<SinkRecord> records) {
        log.info("receive {} messages from consumer", records.size());
        if (records.size() == 0) {
                return;
        }
        try {
            sendBatch(records, producer);
        } catch (RebalanceException e) {
            producer.close();
            producer = initProducer(isTransactional);
        } catch (ResendRecordsException e) {
            abortTransaction(producer);
            //TODO: add limited retry
            sendBatch(e.getRemainingRecords(), producer);
        } catch (Throwable e) {
            log.error(getHostName() + " terminating on exception: {}", e);
            return;
        }
    }

    private void sendBatch(Collection<SinkRecord> records, KafkaProducer<byte[], byte[]> producer) {
        try {
            Map<TopicPartition, List<SinkRecord> remainingRecordsMap = new HashMap<>();
            offsetsMap.clear();
            beginTransaction(producer);
            SinkRecord record;
            for ((record = records.peek()) != null) {
                ProducerRecord<byte[], byte[]> producerRecord = convertToProducerRecord(record);
                offsetsMap.compute(new TopicPartition(record.topic(), record.kafkaPartition()),
                       (tp, curOffsetMetadata) ->
                               (curOffsetMetadata == null || record.kafkaOffset() > curOffsetMetadata.offset())
                                       ?
                                       new OffsetAndMetadata(record.kafkaOffset())
                                       : curOffsetMetadata);
                Future<RecordMetadata> future = producer.send(producerRecord, (recordMetadata, e) -> {
                    if (e != null) {
                        log.error("{} failed to send record to {}: ", MirrorSinkTask.this, producerRecord.
topic(), e);
                        log.debug("{} Failed record: {}", MirrorSinkTask.this, producerRecord);
                        throw new KafkaException(e);
                    } else {
                        log.info("{} Wrote record successfully: topic {} partition {} offset {}", //log.trace
                                        MirrorSinkTask.this,
                                recordMetadata.topic(), recordMetadata.partition(),
                                recordMetadata.offset());
                        commitRecord(record, recordMetadata);
                    }
                });
                futures.add(future);
                records.poll();
            }

        } catch (KafkaException e) {
            // Any unsent messages are added to the remaining remainingRecordsMap for re-send
            for (SinkRecord record = records.poll(); record != null; record = records.poll()) {
                    addConsumerRecordToTopicPartitionRecordsMap(record, remainingRecordsMap);
        } finally {  //TODO: may add more exception handling case
            for (Future<RecordMetadata> future : futures) {
                try {
                    future.get();
                } catch (Exception e) {
                    SinkRecord record = futureMap.get(future);
                    // Any record failed to send, add to the remainingRecordsMap
                    addConsumerRecordToTopicPartitionRecordsMap(record, remainingRecordsMap);
                }
            }
         }

         if (isTransactional && remainingRecordsMap.size() == 0) {
             producer.sendOffsetsToTransaction(offsetsMap, consumerGroupId);
             commitTransaction(producer);
         }

        if (remainingRecordsMap.size() > 0) {
            // For transaction case, all records should be put into remainingRecords, as the whole transaction
should be redone
            Collection<SinkRecord> recordsToReSend;
            if (isTransactional) {
                // transactional: retry all records, as the transaction will cancel all successful and failed
records.
                recordsToReSend = records;
            } else {
                // non-transactional: only retry failed records, others were finished and sent.
                recordsToReSend = remainingRecordsMap;
            }
            throw new ResendRecordsException(recordsToReSend);
        }
    }

    // This commitRecord() follows the same logics as commitRecord() in MirrorSourceTask, to
    public void commitRecord(SinkRecord record, RecordMetadata metadata) {
        try {
            if (stopping) {
                return;
            }
            if (!metadata.hasOffset()) {
                log.error("RecordMetadata has no offset -- can't sync offsets for {}.", record.topic());
                return;
            }
            TopicPartition topicPartition = new TopicPartition(record.topic(), record.kafkaPartition());
            long latency = System.currentTimeMillis() - record.timestamp();
            metrics.countRecord(topicPartition);
            metrics.replicationLatency(topicPartition, latency);
            TopicPartition sourceTopicPartition = MirrorUtils.unwrapPartition(record.sourcePartition());
            long upstreamOffset = MirrorUtils.unwrapOffset(record.sourceOffset());
            long downstreamOffset = metadata.offset();
            maybeSyncOffsets(sourceTopicPartition, upstreamOffset, downstreamOffset);
        } catch (Throwable e) {
            log.warn("Failure committing record.", e);
        }
    }

    private void beginTransaction(KafkaProducer<byte[], byte[]> producer) {
        if (isTransactional) {
            producer.beginTransaction();
            transactionInProgress = true;
        }
    }

    private void initTransactions(KafkaProducer<byte[], byte[]> producer) {
        if (isTransactional) {
            producer.initTransactions();
        }
    }

    private void commitTransaction(KafkaProducer<byte[], byte[]> producer) {
        if (isTransactional) {
            producer.commitTransaction();
            transactionInProgress = false;
        }
    }

    private void abortTransaction(KafkaProducer<byte[], byte[]> producer) {
        if (isTransactional && transactionInProgress) {
            producer.abortTransaction();
            transactionInProgress = false;
        }
    }

    public static class ResendRecordsException extends Exception {
        private Collection<SinkRecord> remainingRecords;

        public ResendRecordsException(Collection<SinkRecord> remainingRecords) {
            super(cause);
            this.remainingRecords = remainingRecords;
        }

        public Collection<SinkRecord> getRemainingRecords() {
            return remainingRecords;
        }
    }
```

MirrorSinkConnector

As SinkTask can only be created by SinkConnector, MirrorSinkConnector will be implemented and follow the most same logics as current MirrorSourceConnector. To minimize the duplicate code, a new class, e.g. "MirrorCommonConnector", may be proposed to host the common code as a separate code change merged before this KIP.

Migration from MirrorSourceConnector to MirrorSinkConnector /w EOS

This is a simply high-level guidance without real-world practices and is subject to change. Also each migration case may be handled differently with different requirements.

By default, "connector.type" is set to "source", when the latest MirrorMaker2 is deployed, the current mirroring behavior should not be changed.

Next, if there are multiple instances of MirrorMaker2, consider to change "connector.type" to "sink" on one instance and deploy it. Once the config change looks stable, repeat for other instances. The message delivery semantics is still at-least-once, but all instances of MirrorMaker2 are now using MirrorSinkConnector.

Since "Transactional mode" or EOS will inevitably consume more resources and deliver lower throughput, it is always recommended to benchmark the impact and provision the enough capacity before switching to EOS.

If a short downtime is allowed, stopping all MirrorMaker2 instances, setting "transaction.producer" to "true", then starting them again. From now, MirrorMaker2 should mirror the data with EOS.

if expect "no downtime", the migration should be conducted more carefully and the operational experiences could refer to "how to migrate from non-transactional to transactional Kafka producer", which is out of scope of this KIP.

Deprecation

A config "connector.type" is proposed to choose which type of Connector (source or sink) to use in MirrorMaker2. So both MirrorSourceConnector and MIrrorSinkConnector will co-exist in the codebase in the near future.

In the long term, if MirrorSinkConnector covers all use cases of MirrorSourceConnector and the migration is proven seamless, then in the future release, deprecation of MirrorSource Connector could be considered.

Rejected Alternatives

https://github.com/apache/kafka/pull/5553, **~~KAFKA-6080~~** - Transactional EoS for source connectors **RESOLVED** and **KAFKA-3821** - Allow Kafka Connect source tasks to produce offset without writing to topics **OPEN** are relevant efforts in a bigger scope, but it seems none of them proceeded successfully for a quite amount of time.