package com.twitter.search.ingester.pipeline.twitter;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

import java.util.Optional;

import java.util.Queue;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.TimeUnit;

import java.util.stream.Collectors;

import javax.naming.NamingException;

import scala.runtime.BoxedUnit;

import com.google.common.collect.Lists;

import com.google.common.collect.Queues;

import org.apache.commons.pipeline.StageException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.search.common.metrics.SearchCustomGauge;

import com.twitter.search.common.metrics.SearchRateCounter;

import com.twitter.search.common.metrics.SearchTimerStats;

import com.twitter.search.ingester.pipeline.util.BatchedElement;

import com.twitter.search.ingester.pipeline.util.PipelineStageException;

import com.twitter.util.Function;

import com.twitter.util.Future;

public abstract class TwitterBatchedBaseStage<T, R> extends

TwitterBaseStage<T, CompletableFuture<R>> {

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

protected final Queue<BatchedElement<T, R>> queue =

Queues.newLinkedBlockingQueue(MAX\_BATCHING\_QUEUE\_SIZE);

private int batchedStageBatchSize = 100;

private int forceProcessAfterMs = 500;

private long lastProcessingTime;

private SearchRateCounter timeBasedQueueFlush;

private SearchRateCounter sizeBasedQueueFlush;

private SearchRateCounter eventsFailed;

private SearchRateCounter numberOfCallsToNextBatchIfReady;

private SearchTimerStats batchExecutionTime;

private SearchTimerStats batchFailedExecutionTime;

private SearchRateCounter validElements;

private SearchRateCounter batchedElements;

private SearchRateCounter emittedElements;

private static final int MAX\_BATCHING\_QUEUE\_SIZE = 10000;

// force the implementing class to set type correctly to avoid catching issues at runtime

protected abstract Class<T> getQueueObjectType();

// up to the developer on how each batch is processed.

protected abstract Future<Collection<R>> innerProcessBatch(Collection<BatchedElement<T, R>>

batch);

// classes that need to update their batch e.g after a decider change

// can override this

protected void updateBatchSize() {

}

protected Collection<T> extractOnlyElementsFromBatch(Collection<BatchedElement<T, R>> batch) {

Collection<T> elementsOnly = new ArrayList<>();

for (BatchedElement<T, R> batchedElement : batch) {

elementsOnly.add(batchedElement.getItem());

}

return elementsOnly;

}

/\*\*

\* This function is used to filter the elements that we want to batch.

\* e.g. if a tweet has urls batch it to resolve the urls, if it doesn't contain urls

\* do not batch.

\*

\* @param element to be evaluated

\*/

protected abstract boolean needsToBeBatched(T element);

/\*\*

\* Tranform from type T to U element.

\* T and U might be different types so this function will help with the transformation

\* if the incoming T element is filtered out and is bypass directly to the next stage

\* that takes incoming objects of type U

\*

\* @param element incoming element

\*/

protected abstract R transform(T element);

protected void reEnqueueAndRetry(BatchedElement<T, R> batchedElement) {

queue.add(batchedElement);

}

@Override

protected void initStats() {

super.initStats();

commonInnerSetupStats();

}

private void commonInnerSetupStats() {

timeBasedQueueFlush = SearchRateCounter.export(getStageNamePrefix()

+ "\_time\_based\_queue\_flush");

sizeBasedQueueFlush = SearchRateCounter.export(getStageNamePrefix()

+ "\_size\_based\_queue\_flush");

batchExecutionTime = SearchTimerStats.export(getStageNamePrefix()

+ "\_batch\_execution\_time", TimeUnit.MILLISECONDS, false, true);

batchFailedExecutionTime = SearchTimerStats.export(getStageNamePrefix()

+ "\_batch\_failed\_execution\_time", TimeUnit.MILLISECONDS, false, true);

eventsFailed = SearchRateCounter.export(getStageNamePrefix() + "\_events\_dropped");

SearchCustomGauge.export(getStageNamePrefix() + "\_batched\_stage\_queue\_size", queue::size);

numberOfCallsToNextBatchIfReady = SearchRateCounter.export(getStageNamePrefix()

+ "\_calls\_to\_nextBatchIfReady");

validElements = SearchRateCounter.export(getStageNamePrefix() + "\_valid\_elements");

batchedElements = SearchRateCounter.export(getStageNamePrefix() + "\_batched\_elements");

emittedElements = SearchRateCounter.export(getStageNamePrefix() + "\_emitted\_elements");

}

@Override

protected void innerSetupStats() {

commonInnerSetupStats();

}

// return a possible batch of elements to process. If we have enough for one batch

protected Optional<Collection<BatchedElement<T, R>>> nextBatchIfReady() {

numberOfCallsToNextBatchIfReady.increment();

Optional<Collection<BatchedElement<T, R>>> batch = Optional.empty();

if (!queue.isEmpty()) {

long elapsed = clock.nowMillis() - lastProcessingTime;

if (elapsed > forceProcessAfterMs) {

batch = Optional.of(Lists.newArrayList(queue));

timeBasedQueueFlush.increment();

queue.clear();

} else if (queue.size() >= batchedStageBatchSize) {

batch = Optional.of(queue.stream()

.limit(batchedStageBatchSize)

.map(element -> queue.remove())

.collect(Collectors.toList()));

sizeBasedQueueFlush.increment();

}

}

return batch;

}

@Override

public void innerProcess(Object obj) throws StageException {

T element;

if (getQueueObjectType().isInstance(obj)) {

element = getQueueObjectType().cast(obj);

} else {

throw new StageException(this, "Trying to add an object of the wrong type to a queue. "

+ getQueueObjectType().getSimpleName()

+ " is the expected type");

}

if (!tryToAddElementToBatch(element)) {

emitAndCount(transform(element));

}

tryToSendBatchedRequest();

}

@Override

protected CompletableFuture<R> innerRunStageV2(T element) {

CompletableFuture<R> completableFuture = new CompletableFuture<>();

if (!tryToAddElementToBatch(element, completableFuture)) {

completableFuture.complete(transform(element));

}

tryToSendBatchedRequestV2();

return completableFuture;

}

private boolean tryToAddElementToBatch(T element, CompletableFuture<R> cf) {

boolean needsToBeBatched = needsToBeBatched(element);

if (needsToBeBatched) {

queue.add(new BatchedElement<>(element, cf));

}

return needsToBeBatched;

}

private boolean tryToAddElementToBatch(T element) {

return tryToAddElementToBatch(element, CompletableFuture.completedFuture(null));

}

private void tryToSendBatchedRequest() {

Optional<Collection<BatchedElement<T, R>>> maybeToProcess = nextBatchIfReady();

if (maybeToProcess.isPresent()) {

Collection<BatchedElement<T, R>> batch = maybeToProcess.get();

lastProcessingTime = clock.nowMillis();

processBatch(batch, getOnSuccessFunction(lastProcessingTime),

getOnFailureFunction(batch, lastProcessingTime));

}

}

private void tryToSendBatchedRequestV2() {

Optional<Collection<BatchedElement<T, R>>> maybeToProcess = nextBatchIfReady();

if (maybeToProcess.isPresent()) {

Collection<BatchedElement<T, R>> batch = maybeToProcess.get();

lastProcessingTime = clock.nowMillis();

processBatch(batch, getOnSuccessFunctionV2(batch, lastProcessingTime),

getOnFailureFunctionV2(batch, lastProcessingTime));

}

}

private void processBatch(Collection<BatchedElement<T, R>> batch,

Function<Collection<R>, BoxedUnit> onSuccess,

Function<Throwable, BoxedUnit> onFailure) {

updateBatchSize();

Future<Collection<R>> futureComputation = innerProcessBatch(batch);

futureComputation.onSuccess(onSuccess);

futureComputation.onFailure(onFailure);

}

private Function<Collection<R>, BoxedUnit> getOnSuccessFunction(long started) {

return Function.cons((elements) -> {

elements.forEach(this::emitAndCount);

batchExecutionTime.timerIncrement(clock.nowMillis() - started);

});

}

private Function<Collection<R>, BoxedUnit> getOnSuccessFunctionV2(Collection<BatchedElement<T, R>>

batch, long started) {

return Function.cons((elements) -> {

Iterator<BatchedElement<T, R>> iterator = batch.iterator();

for (R element : elements) {

if (iterator.hasNext()) {

iterator.next().getCompletableFuture().complete(element);

} else {

LOG.error("Getting Response from Batched Request, but no CompleteableFuture object"

+ " to complete.");

}

}

batchExecutionTime.timerIncrement(clock.nowMillis() - started);

});

}

private Function<Throwable, BoxedUnit> getOnFailureFunction(Collection<BatchedElement<T, R>>

batch, long started) {

return Function.cons((throwable) -> {

batch.forEach(batchedElement -> {

eventsFailed.increment();

// pass the tweet event down better to index an incomplete event than nothing at all

emitAndCount(transform(batchedElement.getItem()));

});

batchFailedExecutionTime.timerIncrement(clock.nowMillis() - started);

LOG.error("Failed processing batch", throwable);

});

}

private Function<Throwable, BoxedUnit> getOnFailureFunctionV2(Collection<BatchedElement<T, R>>

batch, long started) {

return Function.cons((throwable) -> {

batch.forEach(batchedElement -> {

eventsFailed.increment();

R itemTransformed = transform(batchedElement.getItem());

// complete the future, its better to index an incomplete event than nothing at all

batchedElement.getCompletableFuture().complete(itemTransformed);

});

batchFailedExecutionTime.timerIncrement(clock.nowMillis() - started);

LOG.error("Failed processing batch", throwable);

});

}

@Override

protected void doInnerPreprocess() throws StageException, NamingException {

try {

commonInnerSetup();

} catch (PipelineStageException e) {

throw new StageException(this, e);

}

}

private void commonInnerSetup() throws PipelineStageException, NamingException {

updateBatchSize();

if (batchedStageBatchSize < 1) {

throw new PipelineStageException(this,

"Batch size must be set at least to 1 for batched stages but is set to"

+ batchedStageBatchSize);

}

if (forceProcessAfterMs < 1) {

throw new PipelineStageException(this, "forceProcessAfterMs needs to be at least 1 "

+ "ms but is set to " + forceProcessAfterMs);

}

}

@Override

protected void innerSetup() throws PipelineStageException, NamingException {

commonInnerSetup();

}

// Setters for configuration parameters

public void setBatchedStageBatchSize(int maxElementsToWaitFor) {

this.batchedStageBatchSize = maxElementsToWaitFor;

}

public void setForceProcessAfter(int forceProcessAfterMS) {

this.forceProcessAfterMs = forceProcessAfterMS;

}

}