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

import java.util.Arrays;

import java.util.Collection;

import java.util.Collections;

import java.util.List;

import java.util.Optional;

import java.util.concurrent.ConcurrentMap;

import java.util.concurrent.TimeUnit;

import javax.naming.NamingException;

import com.google.common.base.Preconditions;

import com.google.common.collect.Maps;

import org.apache.commons.lang.StringUtils;

import org.apache.commons.pipeline.StageException;

import org.apache.commons.pipeline.stage.InstrumentedBaseStage;

import com.twitter.common.metrics.Metrics;

import com.twitter.common.util.Clock;

import com.twitter.decider.Decider;

import com.twitter.search.common.debug.DebugEventAccumulator;

import com.twitter.search.common.debug.DebugEventUtil;

import com.twitter.search.common.decider.DeciderUtil;

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

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

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

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

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

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

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

import com.twitter.search.common.schema.earlybird.EarlybirdCluster;

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

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

import com.twitter.search.ingester.pipeline.wire.WireModule;

/\*\*

\* Common functionality for all stages.

\*/

public class TwitterBaseStage<T, R> extends InstrumentedBaseStage {

// Currently, all stages run in separate threads, so we could use simple maps here.

// However, it seems safer to use concurrent maps, in case we ever change our stage set up.

// The performance impact should be negligible.

private final ConcurrentMap<Optional<String>, SearchRateCounter> branchEmitObjectsRateCounters =

Maps.newConcurrentMap();

private final ConcurrentMap<Optional<String>, SearchRateCounter>

branchEmitBatchObjectsRateCounters = Maps.newConcurrentMap();

private String stageNamePrefix = null;

protected WireModule wireModule;

protected Decider decider;

protected Clock clock;

protected EarlybirdCluster earlybirdCluster;

private String fullStageName = null;

private Percentile<Long> processPercentile = null;

private SearchTimerStats processTimerStats = null;

private SearchRateCounter droppedItems = null;

private SearchLongGauge stageExceptions = null;

private SearchRateCounter incomingBatchesRateCounter;

private SearchRateCounter incomingBatchObjectsRateCounter;

private List<String> passThroughToBranches = Collections.emptyList();

private List<String> additionalEmitToBranches = Collections.emptyList();

private boolean passThroughDownstream = false;

private boolean emitDownstream = true;

private String dropItemsDeciderKey;

// From XML config.

public void setPassThroughToBranches(String passThroughToBranchesString) {

// This is a comma-delimited string which is a list of branches to which we just

// pass through the incoming object without any processing/filtering.

this.passThroughToBranches = Arrays.asList(passThroughToBranchesString.split(","));

}

// From XML config.

public void setAdditionalEmitToBranches(String emitToBranchesString) {

// This is a comma-delimited string which is a list of branches to which we

// will emit when we call actuallyEmitAndCount(obj).

this.additionalEmitToBranches = Arrays.asList(emitToBranchesString.split(","));

}

// From XML config.

public void setPassThroughDownstream(boolean passThroughDownstream) {

// If true, we emit the raw object downstream

this.passThroughDownstream = passThroughDownstream;

}

// From XML config.

public void setEmitDownstream(boolean emitDownstream) {

// If true, we emit the processed object downstream.

this.emitDownstream = emitDownstream;

}

@Override

public final void innerPreprocess() throws StageException {

try {

setupEssentialObjects();

doInnerPreprocess();

} catch (NamingException e) {

throw new StageException(this, "Failed to initialize stage.", e);

}

}

/\*\*\*

\* Sets up all necessary objects for this stage of the Pipeline. Previously, this task was done

\* by the preprocess() method provided by the ACP library.

\* @throws PipelineStageException

\*/

public void setupStageV2() throws PipelineStageException {

try {

setupCommonStats();

innerSetupStats();

setupEssentialObjects();

innerSetup();

} catch (NamingException e) {

throw new PipelineStageException(this, "Failed to initialize stage", e);

}

}

protected void innerSetup() throws PipelineStageException, NamingException { }

/\*\*\*

\* Takes in an argument of type T, processes it and returns an argument of Type R. This is the

\* main method of a pipeline stage.

\*/

public R runStageV2(T arg) {

long startingTime = startProcessing();

R processed = innerRunStageV2(arg);

endProcessing(startingTime);

return processed;

}

/\*\*\*

\* Takes in an argument of type T, processes it and pushes the processed element to some place.

\* This method does not return anything as any time this method is called on a stage, it means

\* there is no stage after this one. An example stage is any KafkaProducerStage.

\*/

public void runFinalStageOfBranchV2(T arg) {

long startingTime = startProcessing();

innerRunFinalStageOfBranchV2(arg);

endProcessing(startingTime);

}

protected R innerRunStageV2(T arg) {

return null;

}

protected void innerRunFinalStageOfBranchV2(T arg) { }

/\*\*\*

\* called at the end of a pipeline. Cleans up all resources of the stage.

\*/

public void cleanupStageV2() { }

private void setupEssentialObjects() throws NamingException {

wireModule = WireModule.getWireModule();

decider = wireModule.getDecider();

clock = wireModule.getClock();

earlybirdCluster = wireModule.getEarlybirdCluster();

dropItemsDeciderKey =

"drop\_items\_" + earlybirdCluster.getNameForStats() + "\_" + fullStageName;

}

protected void doInnerPreprocess() throws StageException, NamingException { }

@Override

protected void initStats() {

super.initStats();

setupCommonStats();

// Export stage timers

SearchCustomGauge.export(stageNamePrefix + "\_queue\_size",

() -> Optional.ofNullable(getQueueSizeAverage()).orElse(0.0));

SearchCustomGauge.export(stageNamePrefix + "\_queue\_percentage\_full",

() -> Optional.ofNullable(getQueuePercentFull()).orElse(0.0));

// This only called once on startup

// In some unit tests, getQueueCapacity can return null. Hence this guard is added.

// getQueueCapacity() does not return null here in prod.

SearchLongGauge.export(stageNamePrefix + "\_queue\_capacity")

.set(getQueueCapacity() == null ? 0 : getQueueCapacity());

}

private void setupCommonStats() {

// If the stage is instantiated only once, the class name is used for stats export

// If the stage is instantiated multiple times, the "stageName" specified in the

// pipeline definition xml file is also included.

if (StringUtils.isBlank(this.getStageName())) {

fullStageName = this.getClass().getSimpleName();

} else {

fullStageName = String.format(

"%s\_%s",

this.getClass().getSimpleName(),

this.getStageName());

}

stageNamePrefix = Metrics.normalizeName(fullStageName).toLowerCase();

droppedItems = SearchRateCounter.export(stageNamePrefix + "\_dropped\_messages");

stageExceptions = SearchLongGauge.export(stageNamePrefix + "\_stage\_exceptions");

processTimerStats = SearchTimerStats.export(stageNamePrefix, TimeUnit.NANOSECONDS,

true);

processPercentile = PercentileUtil.createPercentile(stageNamePrefix);

incomingBatchesRateCounter = SearchRateCounter.export(stageNamePrefix + "\_incoming\_batches");

incomingBatchObjectsRateCounter =

SearchRateCounter.export(stageNamePrefix + "\_incoming\_batch\_objects");

}

protected void innerSetupStats() {

}

protected SearchCounter makeStageCounter(String counterName) {

return SearchCounter.export(getStageNamePrefix() + "\_" + counterName);

}

private SearchRateCounter getEmitObjectsRateCounterFor(Optional<String> maybeBranch) {

return getRateCounterFor(maybeBranch, "emit\_objects", branchEmitObjectsRateCounters);

}

private SearchRateCounter getEmitBatchObjectsRateCounterFor(Optional<String> maybeBranch) {

return getRateCounterFor(maybeBranch, "emit\_batch\_objects", branchEmitBatchObjectsRateCounters);

}

private SearchRateCounter getRateCounterFor(

Optional<String> maybeBranch,

String statSuffix,

ConcurrentMap<Optional<String>, SearchRateCounter> rateCountersMap) {

SearchRateCounter rateCounter = rateCountersMap.get(maybeBranch);

if (rateCounter == null) {

String branchSuffix = maybeBranch.map(b -> "\_" + b.toLowerCase()).orElse("");

rateCounter = SearchRateCounter.export(stageNamePrefix + branchSuffix + "\_" + statSuffix);

SearchRateCounter existingRateCounter = rateCountersMap.putIfAbsent(maybeBranch, rateCounter);

if (existingRateCounter != null) {

Preconditions.checkState(

existingRateCounter == rateCounter,

"SearchRateCounter.export() should always return the same stat instance.");

}

}

return rateCounter;

}

public String getStageNamePrefix() {

return stageNamePrefix;

}

public String getFullStageName() {

return fullStageName;

}

@Override

public void process(Object obj) throws StageException {

long startTime = System.nanoTime();

try {

// this needs to be updated before calling super.process() so that innerProcess can actually

// use the updated incoming rates

updateIncomingBatchStats(obj);

// Track timing events for when tweets enter each stage.

captureStageDebugEvents(obj);

if (DeciderUtil.isAvailableForRandomRecipient(decider, dropItemsDeciderKey)) {

droppedItems.increment();

return;

}

super.process(obj);

// Now emit the object raw to wherever we need to

emitToPassThroughBranches(obj);

} finally {

long processTime = System.nanoTime() - startTime;

processTimerStats.timerIncrement(processTime);

processPercentile.record(processTime);

stageExceptions.set(stats.getExceptionCount());

}

}

protected long startProcessing() {

long startingTime = System.nanoTime();

checkIfObjectShouldBeEmittedOrThrowRuntimeException();

return startingTime;

}

protected void endProcessing(long startingTime) {

long processTime = System.nanoTime() - startingTime;

processTimerStats.timerIncrement(processTime);

processPercentile.record(processTime);

}

private void checkIfObjectShouldBeEmittedOrThrowRuntimeException() {

if (DeciderUtil.isAvailableForRandomRecipient(decider, dropItemsDeciderKey)) {

droppedItems.increment();

throw new PipelineStageRuntimeException("Object does not have to be processed and passed"

+ " to the next stage");

}

}

private void emitToPassThroughBranches(Object obj) {

for (String branch : passThroughToBranches) {

actuallyEmitAndCount(Optional.of(branch), obj);

}

if (passThroughDownstream) {

actuallyEmitAndCount(Optional.empty(), obj);

}

}

private void updateIncomingBatchStats(Object obj) {

incomingBatchesRateCounter.increment();

incomingBatchObjectsRateCounter.increment(getBatchSizeForStats(obj));

}

protected void captureStageDebugEvents(Object obj) {

if (obj instanceof DebugEventAccumulator) {

DebugEventUtil.addDebugEvent(

(DebugEventAccumulator) obj, getFullStageName(), clock.nowMillis());

} else if (obj instanceof Collection) {

DebugEventUtil.addDebugEventToCollection(

(Collection<?>) obj, getFullStageName(), clock.nowMillis());

} else {

SearchCounter debugEventsNotSupportedCounter = SearchCounter.export(

stageNamePrefix + "\_debug\_events\_not\_supported\_for\_" + obj.getClass());

debugEventsNotSupportedCounter.increment();

}

}

protected int getBatchSizeForStats(Object obj) {

return (obj instanceof Collection) ? ((Collection<?>) obj).size() : 1;

}

protected void emitAndCount(Object obj) {

for (String branch : additionalEmitToBranches) {

actuallyEmitAndCount(Optional.of(branch), obj);

}

if (emitDownstream) {

actuallyEmitAndCount(Optional.empty(), obj);

}

}

protected void emitToBranchAndCount(String branch, Object obj) {

actuallyEmitAndCount(Optional.of(branch), obj);

}

// If the branch is none, emit downstream

private void actuallyEmitAndCount(Optional<String> maybeBranch, Object obj) {

if (maybeBranch.isPresent()) {

emit(maybeBranch.get(), obj);

} else {

emit(obj);

}

getEmitObjectsRateCounterFor(maybeBranch).increment();

getEmitBatchObjectsRateCounterFor(maybeBranch).increment(getBatchSizeForStats(obj));

}

}