/\*\*

\* &copy; Copyright 2008, Summize, Inc. All rights reserved.

\*/

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

import java.util.Collections;

import java.util.NavigableSet;

import java.util.TreeSet;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicLong;

import org.apache.commons.pipeline.StageException;

import org.apache.commons.pipeline.validation.ConsumedTypes;

import org.apache.commons.pipeline.validation.ProducedTypes;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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

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

/\*\*

\* Collect incoming objects into batches of the configured size and then

\* emit the <code>Collection</code> of objects. Internally uses a <code>TreeSet</code>

\* to remove duplicates. Incoming objects MUST implement the <code>Comparable</code>

\* interface.

\*/

@ConsumedTypes(Comparable.class)

@ProducedTypes(NavigableSet.class)

public class CollectComparableObjectsStage extends TwitterBaseStage<Void, Void> {

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

// Batch size of the collections we are emitting.

private int batchSize = -1;

// Top tweets sorts the tweets in reverse order.

private Boolean reverseOrder = false;

// Batch being constructed.

private TreeSet<Object> currentCollection = null;

// Timestamp (ms) of last batch emission.

private final AtomicLong lastEmitTimeMillis = new AtomicLong(-1);

// If set, will emit a batch (only upon arrival of a new element), if time since last emit has

// exceeded this threshold.

private long emitAfterMillis = -1;

private SearchCounter sizeBasedEmitCount;

private SearchCounter timeBasedEmitCount;

private SearchCounter sizeAndTimeBasedEmitCount;

private SearchTimerStats batchEmitTimeStats;

@Override

protected void initStats() {

super.initStats();

SearchCustomGauge.export(getStageNamePrefix() + "\_last\_emit\_time",

() -> lastEmitTimeMillis.get());

sizeBasedEmitCount = SearchCounter.export(getStageNamePrefix() + "\_size\_based\_emit\_count");

timeBasedEmitCount = SearchCounter.export(getStageNamePrefix() + "\_time\_based\_emit\_count");

sizeAndTimeBasedEmitCount = SearchCounter.export(

getStageNamePrefix() + "\_size\_and\_time\_based\_emit\_count");

batchEmitTimeStats = SearchTimerStats.export(

getStageNamePrefix() + "\_batch\_emit\_time",

TimeUnit.MILLISECONDS,

false, // no cpu timers

true); // with percentiles

}

@Override

protected void doInnerPreprocess() throws StageException {

// We have to initialize this stat here, because initStats() is called before

// doInnerPreprocess(), so at that point the 'clock' is not set yet.

SearchCustomGauge.export(getStageNamePrefix() + "\_millis\_since\_last\_emit",

() -> clock.nowMillis() - lastEmitTimeMillis.get());

currentCollection = newBatchCollection();

if (batchSize <= 0) {

throw new StageException(this, "Must set the batchSize parameter to a value >0");

}

}

private TreeSet<Object> newBatchCollection() {

return new TreeSet<>(reverseOrder ? Collections.reverseOrder() : null);

}

@Override

public void innerProcess(Object obj) throws StageException {

if (!Comparable.class.isAssignableFrom(obj.getClass())) {

throw new StageException(

this, "Attempt to add a non-comparable object to a sorted collection");

}

currentCollection.add(obj);

if (shouldEmit()) {

// We want to trace here when we actually emit the batch, as tweets sit in this stage until

// a batch is full, and we want to see how long they actually stick around.

DebugEventUtil.addDebugEventToCollection(

currentCollection, "CollectComparableObjectsStage.outgoing", clock.nowMillis());

emitAndCount(currentCollection);

updateLastEmitTime();

currentCollection = newBatchCollection();

}

}

private boolean shouldEmit() {

if (lastEmitTimeMillis.get() < 0) {

// Initialize lastEmit at the first tweet seen by this stage.

lastEmitTimeMillis.set(clock.nowMillis());

}

final boolean sizeBasedEmit = currentCollection.size() >= batchSize;

final boolean timeBasedEmit =

emitAfterMillis > 0 && lastEmitTimeMillis.get() + emitAfterMillis <= clock.nowMillis();

if (sizeBasedEmit && timeBasedEmit) {

sizeAndTimeBasedEmitCount.increment();

return true;

} else if (sizeBasedEmit) {

sizeBasedEmitCount.increment();

return true;

} else if (timeBasedEmit) {

timeBasedEmitCount.increment();

return true;

} else {

return false;

}

}

@Override

public void innerPostprocess() throws StageException {

if (!currentCollection.isEmpty()) {

emitAndCount(currentCollection);

updateLastEmitTime();

currentCollection = newBatchCollection();

}

}

private void updateLastEmitTime() {

long currentEmitTime = clock.nowMillis();

long previousEmitTime = lastEmitTimeMillis.getAndSet(currentEmitTime);

// Also stat how long each emit takes.

batchEmitTimeStats.timerIncrement(currentEmitTime - previousEmitTime);

}

public void setBatchSize(Integer size) {

LOG.info("Updating all CollectComparableObjectsStage batchSize to {}.", size);

this.batchSize = size;

}

public Boolean getReverseOrder() {

return reverseOrder;

}

public void setReverseOrder(Boolean reverseOrder) {

this.reverseOrder = reverseOrder;

}

public void setEmitAfterMillis(long emitAfterMillis) {

LOG.info("Setting emitAfterMillis to {}.", emitAfterMillis);

this.emitAfterMillis = emitAfterMillis;

}

public long getSizeBasedEmitCount() {

return sizeBasedEmitCount.get();

}

public long getTimeBasedEmitCount() {

return timeBasedEmitCount.get();

}

}