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

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.TimeUnit;

import javax.annotation.Nonnull;

import com.google.common.annotations.VisibleForTesting;

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.metrics.SearchCounter;

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

import com.twitter.search.common.partitioning.snowflakeparser.SnowflakeIdParser;

import com.twitter.search.ingester.model.IngesterTweetEvent;

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

import com.twitter.tweetypie.thriftjava.Tweet;

import com.twitter.tweetypie.thriftjava.TweetCreateEvent;

import com.twitter.tweetypie.thriftjava.TweetEvent;

import com.twitter.tweetypie.thriftjava.TweetEventData;

import com.twitter.tweetypie.thriftjava.TweetEventFlags;

/\*\*

\* Only lets through the create events that match the specified safety type.

\* Also lets through all delete events.

\*/

@ConsumedTypes(IngesterTweetEvent.class)

@ProducedTypes(IngesterTweetEvent.class)

public class FilterEventsBySafetyTypeStage extends TwitterBaseStage

<IngesterTweetEvent, IngesterTweetEvent> {

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

private SearchCounter totalEventsCount;

private SearchCounter createEventsCount;

private SearchCounter createPublicEventsCount;

private SearchCounter createProtectedEventsCount;

private SearchCounter createRestrictedEventsCount;

private SearchCounter createInvalidSafetyTypeCount;

private SearchCounter deleteEventsCount;

private SearchCounter deletePublicEventsCount;

private SearchCounter deleteProtectedEventsCount;

private SearchCounter deleteRestrictedEventsCount;

private SearchCounter deleteInvalidSafetyTypeCount;

private SearchCounter otherEventsCount;

private SearchDelayStats tweetCreateDelayStats;

private long tweetCreateLatencyLogThresholdMillis = -1;

private SafetyType safetyType = null;

private Map<String, Map<String, SearchCounter>> invalidSafetyTypeByEventTypeStatMap =

new ConcurrentHashMap<>();

public FilterEventsBySafetyTypeStage() { }

public FilterEventsBySafetyTypeStage(String safetyType, long tweetCreateLatencyThresholdMillis) {

setSafetyType(safetyType);

this.tweetCreateLatencyLogThresholdMillis = tweetCreateLatencyThresholdMillis;

}

/\*\*

\* To be called by XML config. Can be made private after we delete ACP code.

\*/

public void setSafetyType(@Nonnull String safetyTypeString) {

this.safetyType = SafetyType.valueOf(safetyTypeString);

if (this.safetyType == SafetyType.INVALID) {

throw new UnsupportedOperationException(

"Can't create a stage that permits 'INVALID' safetytypes");

}

}

@Override

protected void initStats() {

super.initStats();

innerSetupStats();

}

@Override

protected void innerSetupStats() {

totalEventsCount = SearchCounter.export(getStageNamePrefix() + "\_total\_events\_count");

createEventsCount = SearchCounter.export(getStageNamePrefix() + "\_create\_events\_count");

createPublicEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_create\_public\_events\_count");

createProtectedEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_create\_protected\_events\_count");

createRestrictedEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_create\_restricted\_events\_count");

createInvalidSafetyTypeCount =

SearchCounter.export(getStageNamePrefix() + "\_create\_missing\_or\_unknown\_safetytype");

deleteEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_delete\_events\_count");

deletePublicEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_delete\_public\_events\_count");

deleteProtectedEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_delete\_protected\_events\_count");

deleteRestrictedEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_delete\_restricted\_events\_count");

deleteInvalidSafetyTypeCount =

SearchCounter.export(getStageNamePrefix() + "\_delete\_missing\_or\_unknown\_safetytype");

otherEventsCount =

SearchCounter.export(getStageNamePrefix() + "\_other\_events\_count");

tweetCreateDelayStats = SearchDelayStats.export(

"create\_histogram\_" + getStageNamePrefix(), 90,

TimeUnit.SECONDS, TimeUnit.MILLISECONDS);

}

@Override

public void innerProcess(Object obj) throws StageException {

if (obj instanceof IngesterTweetEvent) {

IngesterTweetEvent tweetEvent = (IngesterTweetEvent) obj;

if (tryToRecordCreateLatency(tweetEvent)) {

emitAndCount(tweetEvent);

}

} else {

throw new StageException(this, "Object is not a IngesterTweetEvent: " + obj);

}

}

@Override

protected IngesterTweetEvent innerRunStageV2(IngesterTweetEvent tweetEvent) {

if (!tryToRecordCreateLatency(tweetEvent)) {

throw new PipelineStageRuntimeException("Event does not have to pass to the next stage.");

}

return tweetEvent;

}

private boolean tryToRecordCreateLatency(IngesterTweetEvent tweetEvent) {

incrementCounters(tweetEvent);

boolean shouldEmit = shouldEmit(tweetEvent);

if (shouldEmit) {

if (isCreateEvent(tweetEvent.getData())) {

recordCreateLatency(tweetEvent.getData().getTweet\_create\_event());

}

}

return shouldEmit;

}

private void incrementCounters(@Nonnull TweetEvent tweetEvent) {

totalEventsCount.increment();

SafetyType eventSafetyType = getEventSafetyType(tweetEvent);

if (isCreateEvent(tweetEvent.getData())) {

createEventsCount.increment();

switch (eventSafetyType) {

case PUBLIC:

createPublicEventsCount.increment();

break;

case PROTECTED:

createProtectedEventsCount.increment();

break;

case RESTRICTED:

createRestrictedEventsCount.increment();

break;

default:

createInvalidSafetyTypeCount.increment();

incrementInvalidSafetyTypeStatMap(tweetEvent, "create");

}

} else if (isDeleteEvent(tweetEvent.getData())) {

deleteEventsCount.increment();

switch (eventSafetyType) {

case PUBLIC:

deletePublicEventsCount.increment();

break;

case PROTECTED:

deleteProtectedEventsCount.increment();

break;

case RESTRICTED:

deleteRestrictedEventsCount.increment();

break;

default:

deleteInvalidSafetyTypeCount.increment();

incrementInvalidSafetyTypeStatMap(tweetEvent, "delete");

}

} else {

otherEventsCount.increment();

}

}

private void incrementInvalidSafetyTypeStatMap(TweetEvent tweetEvent, String eventType) {

com.twitter.tweetypie.thriftjava.SafetyType thriftSafetyType =

tweetEvent.getFlags().getSafety\_type();

String safetyTypeString =

thriftSafetyType == null ? "null" : thriftSafetyType.toString().toLowerCase();

invalidSafetyTypeByEventTypeStatMap.putIfAbsent(eventType, new ConcurrentHashMap<>());

SearchCounter stat = invalidSafetyTypeByEventTypeStatMap.get(eventType).computeIfAbsent(

safetyTypeString,

safetyTypeStr -> SearchCounter.export(

getStageNamePrefix()

+ String.format("\_%s\_missing\_or\_unknown\_safetytype\_%s",

eventType, safetyTypeStr)));

stat.increment();

}

@VisibleForTesting

boolean shouldEmit(@Nonnull TweetEvent tweetEvent) {

// Do not emit any undelete events.

if (isUndeleteEvent(tweetEvent.getData())) {

return false;

}

SafetyType eventSafetyType = getEventSafetyType(tweetEvent);

// Custom logic for REALTIME\_CG cluster

if (safetyType == SafetyType.PUBLIC\_OR\_PROTECTED) {

return eventSafetyType == SafetyType.PUBLIC || eventSafetyType == SafetyType.PROTECTED;

} else {

return eventSafetyType == safetyType;

}

}

private SafetyType getEventSafetyType(@Nonnull TweetEvent tweetEvent) {

TweetEventFlags tweetEventFlags = tweetEvent.getFlags();

return SafetyType.fromThriftSafetyType(tweetEventFlags.getSafety\_type());

}

private boolean isCreateEvent(@Nonnull TweetEventData tweetEventData) {

return tweetEventData.isSet(TweetEventData.\_Fields.TWEET\_CREATE\_EVENT);

}

private boolean isDeleteEvent(@Nonnull TweetEventData tweetEventData) {

return tweetEventData.isSet(TweetEventData.\_Fields.TWEET\_DELETE\_EVENT);

}

private boolean isUndeleteEvent(@Nonnull TweetEventData tweetEventData) {

return tweetEventData.isSet(TweetEventData.\_Fields.TWEET\_UNDELETE\_EVENT);

}

private void recordCreateLatency(TweetCreateEvent tweetCreateEvent) {

Tweet tweet = tweetCreateEvent.getTweet();

if (tweet != null) {

long tweetCreateLatency =

clock.nowMillis() - SnowflakeIdParser.getTimestampFromTweetId(tweet.getId());

tweetCreateDelayStats.recordLatency(tweetCreateLatency, TimeUnit.MILLISECONDS);

if (tweetCreateLatency < 0) {

LOG.warn("Received a tweet created in the future: {}", tweet);

} else if (tweetCreateLatencyLogThresholdMillis > 0

&& tweetCreateLatency > tweetCreateLatencyLogThresholdMillis) {

LOG.debug("Found late incoming tweet: {}. Create latency: {}ms. Tweet: {}",

tweet.getId(), tweetCreateLatency, tweet);

}

}

}

public void setTweetCreateLatencyLogThresholdMillis(long tweetCreateLatencyLogThresholdMillis) {

LOG.info("Setting tweetCreateLatencyLogThresholdMillis to {}.",

tweetCreateLatencyLogThresholdMillis);

this.tweetCreateLatencyLogThresholdMillis = tweetCreateLatencyLogThresholdMillis;

}

public enum SafetyType {

PUBLIC,

PROTECTED,

RESTRICTED,

PUBLIC\_OR\_PROTECTED,

INVALID;

/\*\* Converts a tweetypie SafetyType instance to an instance of this enum. \*/

@Nonnull

public static SafetyType fromThriftSafetyType(

com.twitter.tweetypie.thriftjava.SafetyType safetyType) {

if (safetyType == null) {

return INVALID;

}

switch(safetyType) {

case PRIVATE:

return PROTECTED;

case PUBLIC:

return PUBLIC;

case RESTRICTED:

return RESTRICTED;

default:

return INVALID;

}

}

}

}