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

import java.time.Duration;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.concurrent.Semaphore;

import java.util.function.Supplier;

import scala.runtime.BoxedUnit;

import com.google.common.base.Preconditions;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.apache.kafka.clients.consumer.KafkaConsumer;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.util.Clock;

import com.twitter.finatra.kafka.producers.BlockingFinagleKafkaProducer;

import com.twitter.gizmoduck.thriftjava.UserModification;

import com.twitter.search.common.indexing.thriftjava.AntisocialUserUpdate;

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

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

import com.twitter.search.common.util.io.kafka.CompactThriftSerializer;

import com.twitter.search.common.util.io.kafka.ThriftDeserializer;

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

import com.twitter.util.Future;

import com.twitter.util.Futures;

/\*\*

\* This class reads UserModification events from Kafka, transforms them into AntisocialUserUpdates,

\* and writes them to Kafka.

\*/

public final class UserUpdatesPipeline {

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

private static final Duration POLL\_TIMEOUT = Duration.ofSeconds(1);

private static final int MAX\_PENDING\_EVENTS = 100;

private static final String KAFKA\_CLIENT\_ID = "";

private static final int MAX\_POLL\_RECORDS = 1;

private static final String USER\_MODIFICATIONS\_KAFKA\_TOPIC = "";

private static final String USER\_UPDATES\_KAFKA\_TOPIC\_PREFIX = "";

private static final String KAFKA\_PRODUCER\_DEST = "";

private static final String KAFKA\_CONSUMER\_DEST = "";

// This semaphore stops us from having more than MAX\_PENDING\_EVENTS in the pipeline at any point

// in time.

private final Semaphore pendingEvents = new Semaphore(MAX\_PENDING\_EVENTS);

private final Supplier<Boolean> isRunning;

private final KafkaConsumer<Long, UserModification> userModificationConsumer;

private final UserUpdateIngester userUpdateIngester;

private final SearchRateCounter records;

private final SearchRateCounter success;

private final SearchRateCounter failure;

private final String userUpdatesKafkaTopic;

private final BlockingFinagleKafkaProducer<Long, AntisocialUserUpdate> userUpdatesProducer;

private final Clock clock;

/\*\*

\* Builds the pipeline.

\*/

public static UserUpdatesPipeline buildPipeline(

String environment,

WireModule wireModule,

String statsPrefix,

Supplier<Boolean> isRunning,

Clock clock) throws Exception {

// We only have Gizmoduck clients for staging and prod.

String gizmoduckClient;

if (environment.startsWith("staging")) {

gizmoduckClient = "";

} else {

Preconditions.checkState("prod".equals(environment));

gizmoduckClient = "";

}

LOG.info("Gizmoduck client: {}", gizmoduckClient);

String kafkaConsumerGroup = "" + environment;

KafkaConsumer<Long, UserModification> userModificationConsumer = wireModule.newKafkaConsumer(

KAFKA\_CONSUMER\_DEST,

new ThriftDeserializer<>(UserModification.class),

KAFKA\_CLIENT\_ID,

kafkaConsumerGroup,

MAX\_POLL\_RECORDS);

userModificationConsumer.subscribe(Collections.singleton(USER\_MODIFICATIONS\_KAFKA\_TOPIC));

LOG.info("User modifications topic: {}", USER\_MODIFICATIONS\_KAFKA\_TOPIC);

LOG.info("User updates Kafka topic prefix: {}", USER\_UPDATES\_KAFKA\_TOPIC\_PREFIX);

LOG.info("Kafka consumer group: {}", kafkaConsumerGroup);

LOG.info("Kafka client id: {}", KAFKA\_CLIENT\_ID);

UserUpdateIngester userUpdateIngester = new UserUpdateIngester(

statsPrefix,

wireModule.getGizmoduckClient(gizmoduckClient),

wireModule.getDecider());

String userUpdatesKafkaTopic = USER\_UPDATES\_KAFKA\_TOPIC\_PREFIX + environment;

BlockingFinagleKafkaProducer<Long, AntisocialUserUpdate> userUpdatesProducer =

wireModule.newFinagleKafkaProducer(

KAFKA\_PRODUCER\_DEST,

new CompactThriftSerializer<AntisocialUserUpdate>(),

KAFKA\_CLIENT\_ID,

null);

return new UserUpdatesPipeline(

isRunning,

userModificationConsumer,

userUpdateIngester,

userUpdatesProducer,

userUpdatesKafkaTopic,

clock);

}

private UserUpdatesPipeline(

Supplier<Boolean> isRunning,

KafkaConsumer<Long, UserModification> userModificationConsumer,

UserUpdateIngester userUpdateIngester,

BlockingFinagleKafkaProducer<Long, AntisocialUserUpdate> userUpdatesProducer,

String userUpdatesKafkaTopic,

Clock clock) {

this.isRunning = isRunning;

this.userModificationConsumer = userModificationConsumer;

this.userUpdateIngester = userUpdateIngester;

this.userUpdatesProducer = userUpdatesProducer;

this.userUpdatesKafkaTopic = userUpdatesKafkaTopic;

this.clock = clock;

String statPrefix = "user\_updates\_pipeline\_";

SearchCustomGauge.export(statPrefix + "semaphore\_permits", pendingEvents::availablePermits);

records = SearchRateCounter.export(statPrefix + "records\_processed\_total");

success = SearchRateCounter.export(statPrefix + "records\_processed\_success");

failure = SearchRateCounter.export(statPrefix + "records\_processed\_failure");

}

/\*\*

\* Start the user updates pipeline.

\*/

public void run() {

while (isRunning.get()) {

try {

pollFromKafka();

} catch (Throwable e) {

LOG.error("Exception processing event.", e);

}

}

close();

}

/\*\*

\* Polls records from Kafka and handles timeouts, back-pressure, and error handling.

\* All consumed messages are passed to the messageHandler.

\*/

private void pollFromKafka() throws Exception {

for (ConsumerRecord<Long, UserModification> record

: userModificationConsumer.poll(POLL\_TIMEOUT)) {

pendingEvents.acquire();

records.increment();

handleUserModification(record.value())

.onFailure(e -> {

failure.increment();

return null;

})

.onSuccess(u -> {

success.increment();

return null;

})

.ensure(() -> {

pendingEvents.release();

return null;

});

}

}

/\*\*

\* Handles the business logic for the user updates pipeline:

\* 1. Converts incoming event into possibly empty set of AntisocialUserUpdates

\* 2. Writes the result to Kafka so that Earlybird can consume it.

\*/

private Future<BoxedUnit> handleUserModification(UserModification event) {

return userUpdateIngester

.transform(event)

.flatMap(this::writeListToKafka);

}

private Future<BoxedUnit> writeListToKafka(List<AntisocialUserUpdate> updates) {

List<Future<BoxedUnit>> futures = new ArrayList<>();

for (AntisocialUserUpdate update : updates) {

futures.add(writeToKafka(update));

}

return Futures.join(futures).onFailure(e -> {

LOG.info("Exception while writing to kafka", e);

return null;

});

}

private Future<BoxedUnit> writeToKafka(AntisocialUserUpdate update) {

ProducerRecord<Long, AntisocialUserUpdate> record = new ProducerRecord<>(

userUpdatesKafkaTopic,

null,

clock.nowMillis(),

null,

update);

try {

return userUpdatesProducer.send(record).unit();

} catch (Exception e) {

return Future.exception(e);

}

}

private void close() {

userModificationConsumer.close();

try {

// Acquire all of the permits, so we know all pending events have been written.

pendingEvents.acquire(MAX\_PENDING\_EVENTS);

} catch (Exception e) {

LOG.error("Error shutting down stage", e);

}

}

}