package com.twitter.search.earlybird.partition;

import java.util.ArrayDeque;

import java.util.Queue;

import java.util.Set;

import java.util.concurrent.ConcurrentSkipListSet;

import com.twitter.common.collections.Pair;

import com.twitter.common.util.Clock;

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

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

import com.twitter.util.Duration;

import com.twitter.util.Time;

public class AudioSpaceTable {

private static final String STATS\_PREFIX = "audio\_space\_";

private static final Duration AUDIO\_EVENT\_EXPIRATION\_DURATION =

Duration.fromHours(12);

private final Set<String> startedSpaces;

private final Set<String> finishedSpaces;

/\*\*

\* timestampedSpaceEvents contains both start and finish events.

\* This is to aid in the case in which we receive only on or the other for a spaceId -- start or finish

\* without doing this, we could potentially never purge from the sets.

\*/

private final Queue<Pair<Time, String>> timestampedSpaceEvents;

private final Clock clock;

private final SearchRateCounter audioSpaceStarts =

SearchRateCounter.export(STATS\_PREFIX + "stream\_starts");

private final SearchRateCounter audioSpaceFinishes =

SearchRateCounter.export(STATS\_PREFIX + "stream\_finishes");

private final SearchRateCounter isRunningCalls =

SearchRateCounter.export(STATS\_PREFIX + "is\_running\_calls");

private final SearchRateCounter audioSpaceDuplicateStarts =

SearchRateCounter.export(STATS\_PREFIX + "duplicate\_start\_events");

private final SearchRateCounter audioSpaceDuplicateFinishes =

SearchRateCounter.export(STATS\_PREFIX + "duplicate\_finish\_events");

private final SearchRateCounter startsProcessedAfterCorrespondingFinishes =

SearchRateCounter.export(STATS\_PREFIX + "starts\_processed\_after\_corresponding\_finishes");

private final SearchRateCounter finishesProcessedWithoutCorrespondingStarts =

SearchRateCounter.export(STATS\_PREFIX + "finishes\_processed\_without\_corresponding\_starts");

public AudioSpaceTable(Clock clock) {

// We read and write from different threads, so we need a thread-safe set implementation.

startedSpaces = new ConcurrentSkipListSet<>();

finishedSpaces = new ConcurrentSkipListSet<>();

timestampedSpaceEvents = new ArrayDeque<>();

this.clock = clock;

SearchCustomGauge.export(STATS\_PREFIX + "live", this::getNumberOfLiveAudioSpaces);

SearchCustomGauge.export(STATS\_PREFIX + "retained\_starts", startedSpaces::size);

SearchCustomGauge.export(STATS\_PREFIX + "retained\_finishes", finishedSpaces::size);

}

private int getNumberOfLiveAudioSpaces() {

// This call is a bit expensive, but I logged it and it's getting called once a minute, at

// the beginning of the minute, so it's fine.

int count = 0;

for (String startedSpace : startedSpaces) {

count += finishedSpaces.contains(startedSpace) ? 0 : 1;

}

return count;

}

/\*\*

\* We keep spaces that have started in the last 12 hours.

\* This is called on every start space event received, and cleans up

\* the retained spaces so memory usage does not become too high

\*/

private void purgeOldSpaces() {

Pair<Time, String> oldest = timestampedSpaceEvents.peek();

Time now = Time.fromMilliseconds(clock.nowMillis());

while (oldest != null) {

Duration durationSinceInsert = now.minus(oldest.getFirst());

if (durationSinceInsert.compareTo(AUDIO\_EVENT\_EXPIRATION\_DURATION) > 0) {

// This event has expired, so we purge it and move on to the next.

String oldSpaceId = oldest.getSecond();

startedSpaces.remove(oldSpaceId);

finishedSpaces.remove(oldSpaceId);

oldest = timestampedSpaceEvents.poll();

} else {

// Oldest event is not old enough so quit purging

break;

}

}

}

/\*\*

\* Record AudioSpace start event

\*/

public void audioSpaceStarts(String spaceId) {

audioSpaceStarts.increment();

boolean spaceSeenBefore = !startedSpaces.add(spaceId);

if (spaceSeenBefore) {

audioSpaceDuplicateStarts.increment();

}

if (finishedSpaces.contains(spaceId)) {

startsProcessedAfterCorrespondingFinishes.increment();

}

timestampedSpaceEvents.add(new Pair(Time.fromMilliseconds(clock.nowMillis()), spaceId));

purgeOldSpaces();

}

/\*\*

\* Record AudioSpace finish event

\*/

public void audioSpaceFinishes(String spaceId) {

audioSpaceFinishes.increment();

boolean spaceSeenBefore = !finishedSpaces.add(spaceId);

if (spaceSeenBefore) {

audioSpaceDuplicateFinishes.increment();

}

if (!startedSpaces.contains(spaceId)) {

finishesProcessedWithoutCorrespondingStarts.increment();

}

timestampedSpaceEvents.add(new Pair(Time.fromMilliseconds(clock.nowMillis()), spaceId));

purgeOldSpaces();

}

public boolean isRunning(String spaceId) {

isRunningCalls.increment();

return startedSpaces.contains(spaceId) && !finishedSpaces.contains(spaceId);

}

/\*\*

\* Print stats on this AudioSpaceTable

\* @return Stats string

\*/

public String toString() {

return "AudioSpaceTable: Starts: " + audioSpaceStarts.getCounter().get()

+ ", Finishes: " + audioSpaceFinishes.getCounter().get()

+ ", Retained starts: " + startedSpaces.size()

+ ", Retained finishes: " + finishedSpaces.size()

+ ", Currently live: " + getNumberOfLiveAudioSpaces();

}

public Set<String> getStartedSpaces() {

return startedSpaces;

}

public Set<String> getFinishedSpaces() {

return finishedSpaces;

}

}