package com.twitter.search.earlybird.common.userupdates;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.TimeUnit;

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

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

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

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

import com.twitter.tweetypie.thriftjava.UserScrubGeoEvent;

/\*\*

\* Map of users who have actioned to delete location data from their tweets. UserID's are mapped

\* to the maxTweetId that will eventually be scrubbed from the index (userId -> maxTweetId).

\*

\* ConcurrentHashMap is thread safe without synchronizing the whole map. Reads can happen very fast

\* while writes are done with a lock. This is ideal since many Earlybird Searcher threads could

\* be reading from the map at once, whereas we will only be adding to the map via kafka.

\*

\* This map is checked against to filter out tweets that should not be returned to geo queries.

\* See: go/realtime-geo-filtering

\*/

public class UserScrubGeoMap {

// The number of geo events that contain a user ID already present in the map. This count is used

// to verify the number of users in the map against the number of events consumed from kafka.

private static final SearchCounter USER\_SCRUB\_GEO\_EVENT\_EXISTING\_USER\_COUNT =

SearchCounter.export("user\_scrub\_geo\_event\_existing\_user\_count");

public static final SearchTimerStats USER\_SCRUB\_GEO\_EVENT\_LAG\_STAT =

SearchTimerStats.export("user\_scrub\_geo\_event\_lag",

TimeUnit.MILLISECONDS,

false,

true);

private ConcurrentHashMap<Long, Long> map;

public UserScrubGeoMap() {

map = new ConcurrentHashMap<>();

SearchCustomGauge.export("num\_users\_in\_geo\_map", this::getNumUsersInMap);

}

/\*\*

\* Ensure that the max\_tweet\_id in the userScrubGeoEvent is greater than the one already stored

\* in the map for the given user id (if any) before updating the entry for this user.

\* This will protect Earlybirds from potential issues where out of date UserScrubGeoEvents

\* appear in the incoming Kafka stream.

\*

\* @param userScrubGeoEvent

\*/

public void indexUserScrubGeoEvent(UserScrubGeoEvent userScrubGeoEvent) {

long userId = userScrubGeoEvent.getUser\_id();

long newMaxTweetId = userScrubGeoEvent.getMax\_tweet\_id();

long oldMaxTweetId = map.getOrDefault(userId, 0L);

if (map.containsKey(userId)) {

USER\_SCRUB\_GEO\_EVENT\_EXISTING\_USER\_COUNT.increment();

}

map.put(userId, Math.max(oldMaxTweetId, newMaxTweetId));

USER\_SCRUB\_GEO\_EVENT\_LAG\_STAT.timerIncrement(computeEventLag(newMaxTweetId));

}

/\*\*

\* A tweet is geo scrubbed if it is older than the max tweet id that is scrubbed for the tweet's

\* author.

\* If there is no entry for the tweet's author in the map, then the tweet is not geo scrubbed.

\*

\* @param tweetId

\* @param fromUserId

\* @return

\*/

public boolean isTweetGeoScrubbed(long tweetId, long fromUserId) {

return tweetId <= map.getOrDefault(fromUserId, 0L);

}

/\*\*

\* The lag (in milliseconds) from when a UserScrubGeoEvent is created, until it is applied to the

\* UserScrubGeoMap. Take the maxTweetId found in the current event and convert it to a timestamp.

\* The maxTweetId will give us a timestamp closest to when Tweetypie processes macaw-geo requests.

\*

\* @param maxTweetId

\* @return

\*/

private long computeEventLag(long maxTweetId) {

long eventCreatedAtTime = SnowflakeIdParser.getTimestampFromTweetId(maxTweetId);

return System.currentTimeMillis() - eventCreatedAtTime;

}

public long getNumUsersInMap() {

return map.size();

}

public ConcurrentHashMap<Long, Long> getMap() {

return map;

}

public boolean isEmpty() {

return map.isEmpty();

}

public boolean isSet(long userId) {

return map.containsKey(userId);

}

}