package com.twitter.tweetypie.config

import com.twitter.servo.cache.{Cache, Cached, CachedValue, CachedValueStatus}

import com.twitter.servo.util.Scribe

import com.twitter.tweetypie.TweetId

import com.twitter.tweetypie.repository.TweetKey

import com.twitter.tweetypie.serverutil.logcachewrites.WriteLoggingCache

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.tweetypie.thriftscala.{CachedTweet, ComposerSource, TweetCacheWrite}

import com.twitter.util.Time

class ScribeTweetCacheWrites(

val underlyingCache: Cache[TweetKey, Cached[CachedTweet]],

logYoungTweetCacheWrites: TweetId => Boolean,

logTweetCacheWrites: TweetId => Boolean)

extends WriteLoggingCache[TweetKey, Cached[CachedTweet]] {

private[this] lazy val scribe = Scribe(TweetCacheWrite, "tweetypie\_tweet\_cache\_writes")

private[this] def mkTweetCacheWrite(

id: Long,

action: String,

cachedValue: CachedValue,

cachedTweet: Option[CachedTweet] = None

): TweetCacheWrite = {

/\*

\* If the Tweet id is a Snowflake id, calculate the offset since Tweet creation.

\* If it is not a Snowflake id, then the offset should be 0. See [[TweetCacheWrite]]'s Thrift

\* documentation for more details.

\*/

val timestampOffset =

if (SnowflakeId.isSnowflakeId(id)) {

SnowflakeId(id).unixTimeMillis.asLong

} else {

0

}

TweetCacheWrite(

tweetId = id,

timestamp = Time.now.inMilliseconds - timestampOffset,

action = action,

cachedValue = cachedValue,

cachedTweet = cachedTweet

)

}

/\*\*

\* Scribe a TweetCacheWrite record to tweetypie\_tweet\_cache\_writes. We scribe the

\* messages instead of writing them to the regular log file because the

\* primary use of this logging is to get a record over time of the cache

\* actions that affected a tweet, so we need a durable log that we can

\* aggregate.

\*/

override def log(action: String, k: TweetKey, v: Option[Cached[CachedTweet]]): Unit =

v match {

case Some(cachedTweet) => {

val cachedValue = CachedValue(

status = cachedTweet.status,

cachedAtMsec = cachedTweet.cachedAt.inMilliseconds,

readThroughAtMsec = cachedTweet.readThroughAt.map(\_.inMilliseconds),

writtenThroughAtMsec = cachedTweet.writtenThroughAt.map(\_.inMilliseconds),

doNotCacheUntilMsec = cachedTweet.doNotCacheUntil.map(\_.inMilliseconds),

)

scribe(mkTweetCacheWrite(k.id, action, cachedValue, cachedTweet.value))

}

// `v` is only None if the action is a "delete" so set CachedValue with a status `Deleted`

case None => {

val cachedValue =

CachedValue(status = CachedValueStatus.Deleted, cachedAtMsec = Time.now.inMilliseconds)

scribe(mkTweetCacheWrite(k.id, action, cachedValue))

}

}

private[this] val YoungTweetThresholdMs = 3600 \* 1000

private[this] def isYoungTweet(tweetId: TweetId): Boolean =

(SnowflakeId.isSnowflakeId(tweetId) &&

((Time.now.inMilliseconds - SnowflakeId(tweetId).unixTimeMillis.asLong) <=

YoungTweetThresholdMs))

/\*\*

\* Select all tweets for which the log\_tweet\_cache\_writes decider returns

\* true and "young" tweets for which the log\_young\_tweet\_cache\_writes decider

\* returns true.

\*/

override def selectKey(k: TweetKey): Boolean =

// When the tweet is young, we log it if it passes either decider. This is

// because the deciders will (by design) select a different subset of

// tweets. We do this so that we have a full record for all tweets for which

// log\_tweet\_cache\_writes is on, but also cast a wider net for tweets that

// are more likely to be affected by replication lag, race conditions

// between different writes, or other consistency issues

logTweetCacheWrites(k.id) || (isYoungTweet(k.id) && logYoungTweetCacheWrites(k.id))

/\*\*

\* Log newscamera tweets as well as any tweets for which selectKey returns

\* true. Note that for newscamera tweets, we will possibly miss "delete"

\* actions since those do not have access to the value, and so do not call

\* this method.

\*/

override def select(k: TweetKey, v: Cached[CachedTweet]): Boolean =

v.value.exists(\_.tweet.composerSource.contains(ComposerSource.Camera)) || selectKey(k)

}