package com.twitter.simclusters\_v2.summingbird.storm

import com.twitter.simclusters\_v2.common.ModelVersions.\_

import com.twitter.simclusters\_v2.summingbird.common.SimClustersProfile.SimClustersTweetProfile

import com.twitter.simclusters\_v2.summingbird.common.Configs

import com.twitter.simclusters\_v2.summingbird.common.Implicits

import com.twitter.simclusters\_v2.summingbird.common.SimClustersHashUtil

import com.twitter.simclusters\_v2.summingbird.common.SimClustersInterestedInUtil

import com.twitter.simclusters\_v2.summingbird.common.StatsUtil

import com.twitter.simclusters\_v2.thriftscala.\_

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.summingbird.\_

import com.twitter.summingbird.option.JobId

import com.twitter.timelineservice.thriftscala.Event

import com.twitter.conversions.DurationOps.\_

import com.twitter.timelineservice.thriftscala.EventAliases.FavoriteAlias

object TweetJob {

import Implicits.\_

import StatsUtil.\_

object NodeName {

final val TweetClusterScoreFlatMapNodeName: String = "TweetClusterScoreFlatMap"

final val TweetClusterUpdatedScoresFlatMapNodeName: String = "TweetClusterUpdatedScoreFlatMap"

final val TweetClusterScoreSummerNodeName: String = "TweetClusterScoreSummer"

final val TweetTopKNodeName: String = "TweetTopKSummer"

final val ClusterTopKTweetsNodeName: String = "ClusterTopKTweetsSummer"

final val ClusterTopKTweetsLightNodeName: String = "ClusterTopKTweetsLightSummer"

}

def generate[P <: Platform[P]](

profile: SimClustersTweetProfile,

timelineEventSource: Producer[P, Event],

userInterestedInService: P#Service[Long, ClustersUserIsInterestedIn],

tweetClusterScoreStore: P#Store[(SimClusterEntity, FullClusterIdBucket), ClustersWithScores],

tweetTopKClustersStore: P#Store[EntityWithVersion, TopKClustersWithScores],

clusterTopKTweetsStore: P#Store[FullClusterId, TopKTweetsWithScores],

clusterTopKTweetsLightStore: Option[P#Store[FullClusterId, TopKTweetsWithScores]]

)(

implicit jobId: JobId

): TailProducer[P, Any] = {

val userInterestNonEmptyCount = Counter(Group(jobId.get), Name("num\_user\_interests\_non\_empty"))

val userInterestEmptyCount = Counter(Group(jobId.get), Name("num\_user\_interests\_empty"))

val numClustersCount = Counter(Group(jobId.get), Name("num\_clusters"))

val entityClusterPairCount = Counter(Group(jobId.get), Name("num\_entity\_cluster\_pairs\_emitted"))

// Fav QPS is around 6K

val qualifiedFavEvents = timelineEventSource

.collect {

case Event.Favorite(favEvent)

if favEvent.userId != favEvent.tweetUserId && !isTweetTooOld(favEvent) =>

(favEvent.userId, favEvent)

}

.observe("num\_qualified\_favorite\_events")

val entityWithSimClustersProducer = qualifiedFavEvents

.leftJoin(userInterestedInService)

.map {

case (\_, (favEvent, userInterestOpt)) =>

(favEvent.tweetId, (favEvent, userInterestOpt))

}

.flatMap {

case (\_, (favEvent, Some(userInterests))) =>

userInterestNonEmptyCount.incr()

val timestamp = favEvent.eventTimeMs

val clustersWithScores = SimClustersInterestedInUtil.topClustersWithScores(userInterests)

// clusters.size is around 25 in average

numClustersCount.incrBy(clustersWithScores.size)

val simClusterScoresByHashBucket = clustersWithScores.groupBy {

case (clusterId, \_) => SimClustersHashUtil.clusterIdToBucket(clusterId)

}

for {

(hashBucket, scores) <- simClusterScoresByHashBucket

} yield {

entityClusterPairCount.incr()

val clusterBucket = FullClusterIdBucket(userInterests.knownForModelVersion, hashBucket)

val tweetId: SimClusterEntity = SimClusterEntity.TweetId(favEvent.tweetId)

(tweetId, clusterBucket) -> SimClustersInterestedInUtil

.buildClusterWithScores(

scores,

timestamp,

profile.favScoreThresholdForUserInterest

)

}

case \_ =>

userInterestEmptyCount.incr()

None

}

.observe("entity\_cluster\_delta\_scores")

.name(NodeName.TweetClusterScoreFlatMapNodeName)

.sumByKey(tweetClusterScoreStore)(clustersWithScoreMonoid)

.name(NodeName.TweetClusterScoreSummerNodeName)

.map {

case ((simClusterEntity, clusterBucket), (oldValueOpt, deltaValue)) =>

val updatedClusterIds = deltaValue.clustersToScore.map(\_.keySet).getOrElse(Set.empty[Int])

(simClusterEntity, clusterBucket) -> clustersWithScoreMonoid.plus(

oldValueOpt

.map { oldValue =>

oldValue.copy(

clustersToScore =

oldValue.clustersToScore.map(\_.filterKeys(updatedClusterIds.contains))

)

}.getOrElse(clustersWithScoreMonoid.zero),

deltaValue

)

}

.observe("entity\_cluster\_updated\_scores")

.name(NodeName.TweetClusterUpdatedScoresFlatMapNodeName)

val tweetTopK = entityWithSimClustersProducer

.flatMap {

case ((simClusterEntity, FullClusterIdBucket(modelVersion, \_)), clusterWithScores)

if simClusterEntity.isInstanceOf[SimClusterEntity.TweetId] =>

clusterWithScores.clustersToScore

.map { clustersToScores =>

val topClustersWithFavScores = clustersToScores.mapValues { scores: Scores =>

Scores(

favClusterNormalized8HrHalfLifeScore =

scores.favClusterNormalized8HrHalfLifeScore.filter(

\_.value >= Configs.scoreThresholdForTweetTopKClustersCache

)

)

}

(

EntityWithVersion(simClusterEntity, modelVersion),

TopKClustersWithScores(Some(topClustersWithFavScores), None)

)

}

case \_ =>

None

}

.observe("tweet\_topk\_updates")

.sumByKey(tweetTopKClustersStore)(topKClustersWithScoresMonoid)

.name(NodeName.TweetTopKNodeName)

val clusterTopKTweets = entityWithSimClustersProducer

.flatMap {

case ((simClusterEntity, FullClusterIdBucket(modelVersion, \_)), clusterWithScores) =>

simClusterEntity match {

case SimClusterEntity.TweetId(tweetId) =>

clusterWithScores.clustersToScore

.map { clustersToScores =>

clustersToScores.toSeq.map {

case (clusterId, scores) =>

val topTweetsByFavScore = Map(

tweetId -> Scores(favClusterNormalized8HrHalfLifeScore =

scores.favClusterNormalized8HrHalfLifeScore.filter(\_.value >=

Configs.scoreThresholdForClusterTopKTweetsCache)))

(

FullClusterId(modelVersion, clusterId),

TopKTweetsWithScores(Some(topTweetsByFavScore), None)

)

}

}.getOrElse(Nil)

case \_ =>

Nil

}

}

.observe("cluster\_topk\_tweets\_updates")

.sumByKey(clusterTopKTweetsStore)(topKTweetsWithScoresMonoid)

.name(NodeName.ClusterTopKTweetsNodeName)

val clusterTopKTweetsLight = clusterTopKTweetsLightStore.map { lightStore =>

entityWithSimClustersProducer

.flatMap {

case ((simClusterEntity, FullClusterIdBucket(modelVersion, \_)), clusterWithScores) =>

simClusterEntity match {

case SimClusterEntity.TweetId(tweetId) if isTweetTooOldForLight(tweetId) =>

clusterWithScores.clustersToScore

.map { clustersToScores =>

clustersToScores.toSeq.map {

case (clusterId, scores) =>

val topTweetsByFavScore = Map(

tweetId -> Scores(favClusterNormalized8HrHalfLifeScore =

scores.favClusterNormalized8HrHalfLifeScore.filter(\_.value >=

Configs.scoreThresholdForClusterTopKTweetsCache)))

(

FullClusterId(modelVersion, clusterId),

TopKTweetsWithScores(Some(topTweetsByFavScore), None)

)

}

}.getOrElse(Nil)

case \_ =>

Nil

}

}

.observe("cluster\_topk\_tweets\_updates")

.sumByKey(lightStore)(topKTweetsWithScoresLightMonoid)

.name(NodeName.ClusterTopKTweetsLightNodeName)

}

clusterTopKTweetsLight match {

case Some(lightNode) =>

tweetTopK.also(clusterTopKTweets).also(lightNode)

case None =>

tweetTopK.also(clusterTopKTweets)

}

}

// Boolean check to see if the tweet is too old

private def isTweetTooOld(favEvent: FavoriteAlias): Boolean = {

favEvent.tweet.forall { tweet =>

SnowflakeId.unixTimeMillisOptFromId(tweet.id).exists { millis =>

System.currentTimeMillis() - millis >= Configs.OldestTweetFavEventTimeInMillis

}

}

}

private def isTweetTooOldForLight(tweetId: Long): Boolean = {

SnowflakeId.unixTimeMillisOptFromId(tweetId).exists { millis =>

System.currentTimeMillis() - millis >= Configs.OldestTweetInLightIndexInMillis

}

}

}