package com.twitter.interaction\_graph.scio.agg\_all

import com.google.cloud.bigquery.BigQueryOptions

import com.google.cloud.bigquery.QueryJobConfiguration

import com.spotify.scio.ScioContext

import com.spotify.scio.ScioMetrics

import com.spotify.scio.values.SCollection

import com.twitter.beam.io.dal.DAL

import com.twitter.beam.io.dal.DAL.DiskFormat

import com.twitter.beam.io.dal.DAL.PathLayout

import com.twitter.beam.io.dal.DAL.WriteOptions

import com.twitter.beam.io.exception.DataNotFoundException

import com.twitter.beam.job.ServiceIdentifierOptions

import com.twitter.interaction\_graph.scio.agg\_all.InteractionGraphAggregationTransform.\_

import com.twitter.interaction\_graph.scio.common.DateUtil

import com.twitter.interaction\_graph.scio.common.FeatureGeneratorUtil

import com.twitter.interaction\_graph.scio.common.UserUtil

import com.twitter.interaction\_graph.thriftscala.Edge

import com.twitter.interaction\_graph.thriftscala.Vertex

import com.twitter.scalding\_internal.multiformat.format.keyval.KeyVal

import com.twitter.scio\_internal.job.ScioBeamJob

import com.twitter.statebird.v2.thriftscala.Environment

import com.twitter.user\_session\_store.thriftscala.UserSession

import com.twitter.util.Duration

import com.twitter.wtf.candidate.thriftscala.ScoredEdge

import java.time.Instant

import org.apache.avro.generic.GenericRecord

import org.apache.beam.sdk.io.gcp.bigquery.BigQueryIO

import org.apache.beam.sdk.io.gcp.bigquery.BigQueryIO.TypedRead

import org.apache.beam.sdk.io.gcp.bigquery.SchemaAndRecord

import org.apache.beam.sdk.transforms.SerializableFunction

import org.joda.time.Interval

import scala.collection.JavaConverters.\_

object InteractionGraphAggregationJob extends ScioBeamJob[InteractionGraphAggregationOption] {

// to parse latest date from the BQ table we're reading from

val parseDateRow = new SerializableFunction[SchemaAndRecord, String] {

override def apply(input: SchemaAndRecord): String = {

val genericRecord: GenericRecord = input.getRecord()

genericRecord.get("ds").toString

}

}

// note that we're using the prob\_explicit for real\_graph\_features (for Home)

val parseRow = new SerializableFunction[SchemaAndRecord, ScoredEdge] {

override def apply(record: SchemaAndRecord): ScoredEdge = {

val genericRecord: GenericRecord = record.getRecord()

ScoredEdge(

genericRecord.get("source\_id").asInstanceOf[Long],

genericRecord.get("destination\_id").asInstanceOf[Long],

genericRecord.get("prob\_explicit").asInstanceOf[Double],

genericRecord.get("followed").asInstanceOf[Boolean],

)

}

}

override def runPipeline(

sc: ScioContext,

opts: InteractionGraphAggregationOption

): Unit = {

val dateStr: String = opts.getDate().value.getStart.toString("yyyyMMdd")

logger.info(s"dateStr $dateStr")

val project: String = "twttr-recos-ml-prod"

val datasetName: String = "realgraph"

val bqTableName: String = "scores"

val fullBqTableName: String = s"$project:$datasetName.$bqTableName"

if (opts.getDALWriteEnvironment.toLowerCase == "prod") {

val bqClient =

BigQueryOptions.newBuilder.setProjectId(project).build.getService

val query =

s"""

|SELECT total\_rows

|FROM `$project.$datasetName.INFORMATION\_SCHEMA.PARTITIONS`

|WHERE partition\_id ="$dateStr" AND

|table\_name="$bqTableName" AND total\_rows > 0

|""".stripMargin

val queryConfig = QueryJobConfiguration.of(query)

val results = bqClient.query(queryConfig).getValues.asScala.toSeq

if (results.isEmpty || results.head.get(0).getLongValue == 0) {

throw new DataNotFoundException(s"$dateStr not present in $fullBqTableName.")

}

}

sc.run()

}

override protected def configurePipeline(

scioContext: ScioContext,

pipelineOptions: InteractionGraphAggregationOption

): Unit = {

@transient

implicit lazy val sc: ScioContext = scioContext

implicit lazy val dateInterval: Interval = pipelineOptions.interval

val yesterday = DateUtil.subtract(dateInterval, Duration.fromDays(1))

val dalEnvironment: String = pipelineOptions

.as(classOf[ServiceIdentifierOptions])

.getEnvironment()

val dalWriteEnvironment = if (pipelineOptions.getDALWriteEnvironment != null) {

pipelineOptions.getDALWriteEnvironment

} else {

dalEnvironment

}

val dateStr: String = pipelineOptions.getDate().value.getStart.toString("yyyy-MM-dd")

logger.info(s"dateStr $dateStr")

val project: String = "twttr-recos-ml-prod"

val datasetName: String = "realgraph"

val bqTableName: String = "scores"

val fullBqTableName: String = s"$project:$datasetName.$bqTableName"

val scoreExport: SCollection[ScoredEdge] =

sc.customInput(

s"Read from BQ table $fullBqTableName",

BigQueryIO

.read(parseRow)

.fromQuery(s"""SELECT source\_id, destination\_id, prob\_explicit, followed

|FROM `$project.$datasetName.$bqTableName`

|WHERE ds = '$dateStr'""".stripMargin)

.usingStandardSql()

.withMethod(TypedRead.Method.DEFAULT)

)

val source = InteractionGraphAggregationSource(pipelineOptions)

val (addressEdgeFeatures, addressVertexFeatures) = source.readAddressBookFeatures()

val (clientEventLogsEdgeFeatures, clientEventLogsVertexFeatures) =

source.readClientEventLogsFeatures(dateInterval)

val (flockEdgeFeatures, flockVertexFeatures) = source.readFlockFeatures()

val (directInteractionsEdgeFeatures, directInteractionsVertexFeatures) =

source.readDirectInteractionsFeatures(dateInterval)

val invalidUsers = UserUtil.getInvalidUsers(source.readFlatUsers())

val (prevAggEdge, prevAggVertex) = source.readAggregatedFeatures(yesterday)

val prevAggregatedVertex: SCollection[Vertex] =

UserUtil

.filterUsersByIdMapping[Vertex](

prevAggVertex,

invalidUsers,

v => v.userId

)

/\*\* Remove status-based features (flock/ab) from current graph, because we only need the latest

\* This is to allow us to filter and roll-up a smaller dataset, to which we will still add

\* back the status-based features for the complete scoredAggregates (that other teams will read).

\*/

val prevAggEdgeFiltered = prevAggEdge

.filter { e =>

e.sourceId != e.destinationId

}

.withName("filtering status-based edges")

.flatMap(FeatureGeneratorUtil.removeStatusFeatures)

val prevAggEdgeValid: SCollection[Edge] =

UserUtil

.filterUsersByMultipleIdMappings[Edge](

prevAggEdgeFiltered,

invalidUsers,

Seq(e => e.sourceId, e => e.destinationId)

)

val aggregatedActivityVertexDaily = UserUtil

.filterUsersByIdMapping[Vertex](

FeatureGeneratorUtil

.combineVertexFeatures(

clientEventLogsVertexFeatures ++

directInteractionsVertexFeatures ++

addressVertexFeatures ++

flockVertexFeatures

),

invalidUsers,

v => v.userId

)

// we split up the roll-up of decayed counts between status vs activity/count-based features

val aggregatedActivityEdgeDaily = FeatureGeneratorUtil

.combineEdgeFeatures(clientEventLogsEdgeFeatures ++ directInteractionsEdgeFeatures)

// Vertex level, Add the decay sum for history and daily

val aggregatedActivityVertex = FeatureGeneratorUtil

.combineVertexFeaturesWithDecay(

prevAggregatedVertex,

aggregatedActivityVertexDaily,

InteractionGraphScoringConfig.ONE\_MINUS\_ALPHA,

InteractionGraphScoringConfig.ALPHA

)

// Edge level, Add the decay sum for history and daily

val aggregatedActivityEdge = FeatureGeneratorUtil

.combineEdgeFeaturesWithDecay(

prevAggEdgeValid,

aggregatedActivityEdgeDaily,

InteractionGraphScoringConfig.ONE\_MINUS\_ALPHA,

InteractionGraphScoringConfig.ALPHA

)

.filter(FeatureGeneratorUtil.edgeWithFeatureOtherThanDwellTime)

.withName("removing edges that only have dwell time features")

val edgeKeyedScores = scoreExport.keyBy { e => (e.sourceId, e.destinationId) }

val scoredAggregatedActivityEdge = aggregatedActivityEdge

.keyBy { e => (e.sourceId, e.destinationId) }

.withName("join with scores")

.leftOuterJoin(edgeKeyedScores)

.map {

case (\_, (e, scoredEdgeOpt)) =>

val scoreOpt = scoredEdgeOpt.map(\_.score)

e.copy(weight = if (scoreOpt.nonEmpty) {

ScioMetrics.counter("after joining edge with scores", "has score").inc()

scoreOpt

} else {

ScioMetrics.counter("after joining edge with scores", "no score").inc()

None

})

}

val combinedFeatures = FeatureGeneratorUtil

.combineEdgeFeatures(aggregatedActivityEdge ++ addressEdgeFeatures ++ flockEdgeFeatures)

.keyBy { e => (e.sourceId, e.destinationId) }

val aggregatedActivityScoredEdge =

edgeKeyedScores

.withName("join with combined edge features")

.leftOuterJoin(combinedFeatures)

.map {

case (\_, (scoredEdge, combinedFeaturesOpt)) =>

if (combinedFeaturesOpt.exists(\_.features.nonEmpty)) {

ScioMetrics.counter("after joining scored edge with features", "has features").inc()

Edge(

sourceId = scoredEdge.sourceId,

destinationId = scoredEdge.destinationId,

weight = Some(scoredEdge.score),

features = combinedFeaturesOpt.map(\_.features).getOrElse(Nil)

)

} else {

ScioMetrics.counter("after joining scored edge with features", "no features").inc()

Edge(

sourceId = scoredEdge.sourceId,

destinationId = scoredEdge.destinationId,

weight = Some(scoredEdge.score),

features = Nil

)

}

}

val realGraphFeatures =

getTopKTimelineFeatures(aggregatedActivityScoredEdge, pipelineOptions.getMaxDestinationIds)

aggregatedActivityVertex.saveAsCustomOutput(

"Write History Aggregated Vertex Records",

DAL.writeSnapshot[Vertex](

dataset = InteractionGraphHistoryAggregatedVertexSnapshotScalaDataset,

pathLayout = PathLayout.DailyPath(pipelineOptions.getOutputPath + "/aggregated\_vertex"),

endDate = Instant.ofEpochMilli(dateInterval.getEndMillis),

diskFormat = DiskFormat.Parquet,

environmentOverride = Environment.valueOf(dalWriteEnvironment),

writeOption = WriteOptions(numOfShards = Some(pipelineOptions.getNumberOfShards / 10))

)

)

scoredAggregatedActivityEdge.saveAsCustomOutput(

"Write History Aggregated Edge Records",

DAL.writeSnapshot[Edge](

dataset = InteractionGraphHistoryAggregatedEdgeSnapshotScalaDataset,

pathLayout = PathLayout.DailyPath(pipelineOptions.getOutputPath + "/aggregated\_raw\_edge"),

endDate = Instant.ofEpochMilli(dateInterval.getEndMillis),

diskFormat = DiskFormat.Parquet,

environmentOverride = Environment.valueOf(dalWriteEnvironment),

writeOption = WriteOptions(numOfShards = Some(pipelineOptions.getNumberOfShards))

)

)

aggregatedActivityVertexDaily.saveAsCustomOutput(

"Write Daily Aggregated Vertex Records",

DAL.write[Vertex](

dataset = InteractionGraphAggregatedVertexDailyScalaDataset,

pathLayout =

PathLayout.DailyPath(pipelineOptions.getOutputPath + "/aggregated\_vertex\_daily"),

interval = dateInterval,

diskFormat = DiskFormat.Parquet,

environmentOverride = Environment.valueOf(dalWriteEnvironment),

writeOption = WriteOptions(numOfShards = Some(pipelineOptions.getNumberOfShards / 10))

)

)

aggregatedActivityEdgeDaily.saveAsCustomOutput(

"Write Daily Aggregated Edge Records",

DAL.write[Edge](

dataset = InteractionGraphAggregatedEdgeDailyScalaDataset,

pathLayout = PathLayout.DailyPath(pipelineOptions.getOutputPath + "/aggregated\_edge\_daily"),

interval = dateInterval,

diskFormat = DiskFormat.Parquet,

environmentOverride = Environment.valueOf(dalWriteEnvironment),

writeOption = WriteOptions(numOfShards = Some(pipelineOptions.getNumberOfShards))

)

)

realGraphFeatures.saveAsCustomOutput(

"Write Timeline Real Graph Features",

DAL.writeVersionedKeyVal[KeyVal[Long, UserSession]](

dataset = RealGraphFeaturesScalaDataset,

pathLayout =

PathLayout.VersionedPath(pipelineOptions.getOutputPath + "/real\_graph\_features"),

environmentOverride = Environment.valueOf(dalWriteEnvironment),

writeOption = WriteOptions(numOfShards = Some(pipelineOptions.getNumberOfShards))

)

)

}

}