package com.twitter.search.earlybird.search;

import java.io.IOException;

import java.util.LinkedHashSet;

import java.util.Set;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.index.NumericDocValues;

import org.apache.lucene.search.Query;

import org.apache.lucene.spatial.prefix.tree.Cell;

import org.apache.lucene.spatial.prefix.tree.CellIterator;

import org.apache.lucene.util.BytesRef;

import org.locationtech.spatial4j.shape.Rectangle;

import com.twitter.search.common.query.MultiTermDisjunctionQuery;

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants.EarlybirdFieldConstant;

import com.twitter.search.common.search.GeoQuadTreeQueryBuilderUtil;

import com.twitter.search.common.search.TerminationTracker;

import com.twitter.search.common.util.spatial.BoundingBox;

import com.twitter.search.common.util.spatial.GeoUtil;

import com.twitter.search.common.util.spatial.GeohashChunkImpl;

import com.twitter.search.core.earlybird.index.EarlybirdIndexSegmentAtomicReader;

import com.twitter.search.earlybird.search.queries.GeoTwoPhaseQuery;

import com.twitter.search.earlybird.search.queries.GeoTwoPhaseQuery.SecondPhaseDocAccepter;

import com.twitter.search.queryparser.query.QueryParserException;

import com.twitter.search.queryparser.util.GeoCode;

import geo.google.datamodel.GeoCoordinate;

/\*\*

\* A class that builds queries to query the quadtree.

\*/

public final class GeoQuadTreeQueryBuilder {

private GeoQuadTreeQueryBuilder() {

}

/\*\*

\* Returns a GeoTwoPhaseQuery for the given geocode.

\*/

public static Query buildGeoQuadTreeQuery(final GeoCode geocode) {

return buildGeoQuadTreeQuery(geocode, null);

}

/\*\*

\* Returns a GeoTwoPhaseQuery for the given geocode.

\*

\* @param geocode The geocode.

\* @param terminationTracker The tracker that determines when the query needs to terminate.

\*/

public static Query buildGeoQuadTreeQuery(GeoCode geocode,

TerminationTracker terminationTracker) {

Query geoHashDisjuntiveQuery = GeoQuadTreeQueryBuilderUtil.buildGeoQuadTreeQuery(

geocode, EarlybirdFieldConstant.GEO\_HASH\_FIELD.getFieldName());

// 5. Create post filtering accepter

final SecondPhaseDocAccepter accepter = (geocode.distanceKm != GeoCode.DOUBLE\_DISTANCE\_NOT\_SET)

? new CenterRadiusAccepter(geocode.latitude, geocode.longitude, geocode.distanceKm)

: GeoTwoPhaseQuery.ALL\_DOCS\_ACCEPTER;

return new GeoTwoPhaseQuery(geoHashDisjuntiveQuery, accepter, terminationTracker);

}

/\*\*

\* Construct a query as below:

\* 1. Compute all quadtree cells that intersects the bounding box.

\* 2. Create a disjunction of the geohashes of all the intersecting cells.

\* 3. Add a filter to only keep points inside the giving bounding box.

\*/

public static Query buildGeoQuadTreeQuery(final Rectangle boundingBox,

final TerminationTracker terminationTracker)

throws QueryParserException {

// 1. Locate the main quadtree cell---the cell containing the bounding box's center point whose

// diagonal is just longer than the bounding box's diagonal.

final Cell centerCell = GeohashChunkImpl.getGeoNodeByBoundingBox(boundingBox);

// 2. Determine quadtree level to search.

int treeLevel = -1;

if (centerCell != null) {

treeLevel = centerCell.getLevel();

} else {

// This should not happen.

throw new QueryParserException(

"Unable to locate quadtree cell containing the given bounding box."

+ "Bounding box is: " + boundingBox);

}

// 3. get all quadtree cells at treeLevel that intersects the given bounding box.

CellIterator intersectingCells =

GeohashChunkImpl.getNodesIntersectingBoundingBox(boundingBox, treeLevel);

// 4. Construct disjunction query

final Set<BytesRef> geoHashSet = new LinkedHashSet<>();

// Add center node

geoHashSet.add(centerCell.getTokenBytesNoLeaf(new BytesRef()));

// If there are other nodes intersecting query circle, also add them in.

if (intersectingCells != null) {

while (intersectingCells.hasNext()) {

geoHashSet.add(intersectingCells.next().getTokenBytesNoLeaf(new BytesRef()));

}

}

MultiTermDisjunctionQuery geoHashDisjuntiveQuery = new MultiTermDisjunctionQuery(

EarlybirdFieldConstant.GEO\_HASH\_FIELD.getFieldName(), geoHashSet);

// 5. Create post filtering accepter

final GeoDocAccepter accepter = new BoundingBoxAccepter(boundingBox);

return new GeoTwoPhaseQuery(geoHashDisjuntiveQuery, accepter, terminationTracker);

}

private abstract static class GeoDocAccepter extends SecondPhaseDocAccepter {

private NumericDocValues latLonDocValues;

private final GeoCoordinate geoCoordReuse = new GeoCoordinate();

@Override

public void initialize(LeafReaderContext context) throws IOException {

final EarlybirdIndexSegmentAtomicReader reader =

(EarlybirdIndexSegmentAtomicReader) context.reader();

latLonDocValues =

reader.getNumericDocValues(EarlybirdFieldConstant.LAT\_LON\_CSF\_FIELD.getFieldName());

}

// Decides whether a point should be accepted.

protected abstract boolean acceptPoint(double lat, double lon);

// Decides whether a document should be accepted based on its geo coordinates.

@Override

public final boolean accept(int internalDocId) throws IOException {

// Cannot obtain valid geo coordinates for the document. Not acceptable.

if (latLonDocValues == null

|| !latLonDocValues.advanceExact(internalDocId)

|| !GeoUtil.decodeLatLonFromInt64(latLonDocValues.longValue(), geoCoordReuse)) {

return false;

}

return acceptPoint(geoCoordReuse.getLatitude(), geoCoordReuse.getLongitude());

}

}

// Accepts points within a circle defined by a center point and a radius.

private static final class CenterRadiusAccepter extends GeoDocAccepter {

private final double centerLat;

private final double centerLon;

private final double radiusKm;

public CenterRadiusAccepter(double centerLat, double centerLon, double radiusKm) {

this.centerLat = centerLat;

this.centerLon = centerLon;

this.radiusKm = radiusKm;

}

@Override

protected boolean acceptPoint(double lat, double lon) {

double actualDistance =

BoundingBox.approxDistanceC(centerLat, centerLon, lat, lon);

if (actualDistance < radiusKm) {

return true;

} else if (Double.isNaN(actualDistance)) {

// There seems to be a rare bug in GeoUtils that computes NaN

// for two identical lat/lon pairs on occasion. Check for that here.

if (lat == centerLat && lon == centerLon) {

return true;

}

}

return false;

}

@Override

public String toString() {

return String.format("CenterRadiusAccepter(Center: %.4f, %.4f Radius (km): %.4f)",

centerLat, centerLon, radiusKm);

}

}

// Accepts points within a BoundingBox

private static final class BoundingBoxAccepter extends GeoDocAccepter {

private final Rectangle boundingBox;

public BoundingBoxAccepter(Rectangle boundingBox) {

this.boundingBox = boundingBox;

}

@Override

protected boolean acceptPoint(double lat, double lon) {

return GeohashChunkImpl.isPointInBoundingBox(lat, lon, boundingBox);

}

@Override

public String toString() {

return String.format("PointInBoundingBoxAccepter((%.4f, %.4f), (%.4f, %.4f), "

+ "crossesDateLine=%b)",

boundingBox.getMinY(), boundingBox.getMinX(),

boundingBox.getMaxY(), boundingBox.getMaxX(),

boundingBox.getCrossesDateLine());

}

}

}