package com.twitter.search.earlybird.search.relevance;

import java.io.IOException;

import java.util.Objects;

import com.google.common.annotations.VisibleForTesting;

import org.apache.lucene.index.LeafReader;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.index.NumericDocValues;

import org.apache.lucene.search.BooleanClause;

import org.apache.lucene.search.BooleanQuery;

import org.apache.lucene.search.DocIdSetIterator;

import org.apache.lucene.search.IndexSearcher;

import org.apache.lucene.search.Query;

import org.apache.lucene.search.ScoreMode;

import org.apache.lucene.search.Weight;

import com.twitter.search.common.encoding.features.ByteNormalizer;

import com.twitter.search.common.encoding.features.ClampByteNormalizer;

import com.twitter.search.common.encoding.features.SingleBytePositiveFloatNormalizer;

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

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

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

import com.twitter.search.core.earlybird.index.util.RangeFilterDISI;

public final class MinFeatureValueFilter extends Query implements FilteredQuery.DocIdFilterFactory {

private final String featureName;

private final ByteNormalizer normalizer;

private final double minValue;

/\*\*

\* Creates a query that filters out all hits that have a value smaller than the given threshold

\* for the given feature.

\*

\* @param featureName The feature.

\* @param minValue The threshold for the feature values.

\* @return A query that filters out all hits that have a value smaller than the given threshold

\* for the given feature.

\*/

public static Query getMinFeatureValueFilter(String featureName, double minValue) {

return new BooleanQuery.Builder()

.add(new MinFeatureValueFilter(featureName, minValue), BooleanClause.Occur.FILTER)

.build();

}

public static FilteredQuery.DocIdFilterFactory getDocIdFilterFactory(

String featureName, double minValue) {

return new MinFeatureValueFilter(featureName, minValue);

}

/\*\*

\* Returns the normalizer that should be used to normalize the values for the given feature.

\*

\* @param featureName The feature.

\* @return The normalizer that should be used to normalize the values for the given feature.

\*/

@VisibleForTesting

public static ByteNormalizer getMinFeatureValueNormalizer(String featureName) {

if (featureName.equals(EarlybirdFieldConstant.USER\_REPUTATION.getFieldName())) {

return new ClampByteNormalizer(0, 100);

}

if (featureName.equals(EarlybirdFieldConstant.FAVORITE\_COUNT.getFieldName())

|| featureName.equals(EarlybirdFieldConstant.PARUS\_SCORE.getFieldName())

|| featureName.equals(EarlybirdFieldConstant.REPLY\_COUNT.getFieldName())

|| featureName.equals(EarlybirdFieldConstant.RETWEET\_COUNT.getFieldName())) {

return new SingleBytePositiveFloatNormalizer();

}

throw new IllegalArgumentException("Unknown normalization method for field " + featureName);

}

@Override

public int hashCode() {

// Probably doesn't make sense to include the schemaSnapshot and normalizer here.

return (int) ((featureName == null ? 0 : featureName.hashCode() \* 7) + minValue);

}

@Override

public boolean equals(Object obj) {

if (!(obj instanceof MinFeatureValueFilter)) {

return false;

}

// Probably doesn't make sense to include the schemaSnapshot and normalizer here.

MinFeatureValueFilter filter = MinFeatureValueFilter.class.cast(obj);

return Objects.equals(featureName, filter.featureName) && (minValue == filter.minValue);

}

@Override

public String toString(String field) {

return String.format("MinFeatureValueFilter(%s, %f)", featureName, minValue);

}

private MinFeatureValueFilter(String featureName, double minValue) {

this.featureName = featureName;

this.normalizer = getMinFeatureValueNormalizer(featureName);

this.minValue = normalizer.normalize(minValue);

}

@Override

public FilteredQuery.DocIdFilter getDocIdFilter(LeafReaderContext context) throws IOException {

final NumericDocValues featureDocValues = context.reader().getNumericDocValues(featureName);

return (docId) -> featureDocValues.advanceExact(docId)

&& ((byte) featureDocValues.longValue() >= minValue);

}

@Override

public Weight createWeight(IndexSearcher searcher, ScoreMode scoreMode, float boost) {

return new DefaultFilterWeight(this) {

@Override

protected DocIdSetIterator getDocIdSetIterator(LeafReaderContext context) throws IOException {

return new MinFeatureValueDocIdSetIterator(

context.reader(), featureName, minValue);

}

};

}

private static final class MinFeatureValueDocIdSetIterator extends RangeFilterDISI {

private final NumericDocValues featureDocValues;

private final double minValue;

MinFeatureValueDocIdSetIterator(LeafReader indexReader,

String featureName,

double minValue) throws IOException {

super(indexReader);

this.featureDocValues = indexReader.getNumericDocValues(featureName);

this.minValue = minValue;

}

@Override

public boolean shouldReturnDoc() throws IOException {

// We need this explicit casting to byte, because of how we encode and decode features in our

// encoded\_tweet\_features field. If a feature is an int (uses all 32 bits of the int), then

// encoding the feature and then decoding it preserves its original value. However, if the

// feature does not use the entire int (and especially if it uses bits somewhere in the middle

// of the int), then the feature value is assumed to be unsigned when it goes through this

// process of encoding and decoding. So a user rep of

// RelevanceSignalConstants.UNSET\_REPUTATION\_SENTINEL (-128) will be correctly encoded as the

// binary value 10000000, but will be treated as an unsigned value when decoded, and therefore

// the decoded value will be 128.

//

// In retrospect, this seems like a really poor design decision. It seems like it would be

// better if all feature values were considered to be signed, even if most features can never

// have negative values. Unfortunately, making this change is not easy, because some features

// store normalized values, so we would also need to change the range of allowed values

// produced by those normalizers, as well as all code that depends on those values.

//

// So for now, just cast this value to a byte, to get the proper negative value.

return featureDocValues.advanceExact(docID())

&& ((byte) featureDocValues.longValue() >= minValue);

}

}

public double getMinValue() {

return minValue;

}

public ByteNormalizer getNormalizer() {

return normalizer;

}

}