package com.twitter.search.earlybird.search.queries;

import java.io.IOException;

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.query.DefaultFilterWeight;

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

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

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

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

public final class BadUserRepFilter extends Query {

/\*\*

\* Creates a query that filters out results coming from users with bad reputation.

\*

\* @param minTweepCred The lowest acceptable user reputation.

\* @return A query that filters out results from bad reputation users.

\*/

public static Query getBadUserRepFilter(int minTweepCred) {

if (minTweepCred <= 0) {

return null;

}

return new BooleanQuery.Builder()

.add(new BadUserRepFilter(minTweepCred), BooleanClause.Occur.FILTER)

.build();

}

private final int minTweepCred;

private BadUserRepFilter(int minTweepCred) {

this.minTweepCred = minTweepCred;

}

@Override

public int hashCode() {

return minTweepCred;

}

@Override

public boolean equals(Object obj) {

if (!(obj instanceof BadUserRepFilter)) {

return false;

}

return minTweepCred == BadUserRepFilter.class.cast(obj).minTweepCred;

}

@Override

public String toString(String field) {

return "BadUserRepFilter:" + minTweepCred;

}

@Override

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

return new DefaultFilterWeight(this) {

@Override

protected DocIdSetIterator getDocIdSetIterator(LeafReaderContext context) throws IOException {

LeafReader reader = context.reader();

if (!(reader instanceof EarlybirdIndexSegmentAtomicReader)) {

return new AllDocsIterator(reader);

}

return new BadUserExcludeDocIdSetIterator(

(EarlybirdIndexSegmentAtomicReader) context.reader(), minTweepCred);

}

};

}

private static final class BadUserExcludeDocIdSetIterator extends RangeFilterDISI {

private final NumericDocValues userReputationDocValues;

private final int minTweepCred;

BadUserExcludeDocIdSetIterator(EarlybirdIndexSegmentAtomicReader indexReader,

int minTweepCred) throws IOException {

super(indexReader);

this.userReputationDocValues =

indexReader.getNumericDocValues(EarlybirdFieldConstant.USER\_REPUTATION.getFieldName());

this.minTweepCred = minTweepCred;

}

@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 userReputationDocValues.advanceExact(docID())

&& ((byte) userReputationDocValues.longValue() >= minTweepCred);

}

}

}