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

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.query.DefaultFilterWeight;

import com.twitter.search.common.schema.thriftjava.ThriftCSFType;

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;

/\*\*

\* Filters tweets according to the specified CSF field value.

\* Note that min value is inclusive, and max value is exclusive.

\*/

public final class DocValRangeFilter extends Query {

private final String csfField;

private final ThriftCSFType csfFieldType;

private final Number minValInclusive;

private final Number maxValExclusive;

/\*\*

\* Returns a query that filters hits based on the value of a CSF.

\*

\* @param csfField The CSF name.

\* @param csfFieldType The CSF type.

\* @param minVal The minimum acceptable value (inclusive).

\* @param maxVal The maximum acceptable value (exclusive).

\* @return A query that filters hits based on the value of a CSF.

\*/

public static Query getDocValRangeQuery(String csfField, ThriftCSFType csfFieldType,

double minVal, double maxVal) {

return new BooleanQuery.Builder()

.add(new DocValRangeFilter(csfField, csfFieldType, minVal, maxVal),

BooleanClause.Occur.FILTER)

.build();

}

/\*\*

\* Returns a query that filters hits based on the value of a CSF.

\*

\* @param csfField The CSF name.

\* @param csfFieldType The CSF type.

\* @param minVal The minimum acceptable value (inclusive).

\* @param maxVal The maximum acceptable value (exclusive).

\* @return A query that filters hits based on the value of a CSF.

\*/

public static Query getDocValRangeQuery(String csfField, ThriftCSFType csfFieldType,

long minVal, long maxVal) {

return new BooleanQuery.Builder()

.add(new DocValRangeFilter(csfField, csfFieldType, minVal, maxVal),

BooleanClause.Occur.FILTER)

.build();

}

private DocValRangeFilter(String csfField, ThriftCSFType csfFieldType,

double minVal, double maxVal) {

this.csfField = csfField;

this.csfFieldType = csfFieldType;

this.minValInclusive = new Float(minVal);

this.maxValExclusive = new Float(maxVal);

}

private DocValRangeFilter(String csfField, ThriftCSFType csfFieldType,

long minVal, long maxVal) {

this.csfField = csfField;

this.csfFieldType = csfFieldType;

this.minValInclusive = new Long(minVal);

this.maxValExclusive = new Long(maxVal);

}

@Override

public int hashCode() {

return (csfField == null ? 0 : csfField.hashCode()) \* 29

+ (csfFieldType == null ? 0 : csfFieldType.hashCode()) \* 17

+ minValInclusive.hashCode() \* 7

+ maxValExclusive.hashCode();

}

@Override

public boolean equals(Object obj) {

if (!(obj instanceof DocValRangeFilter)) {

return false;

}

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

return Objects.equals(csfField, filter.csfField)

&& (csfFieldType == filter.csfFieldType)

&& minValInclusive.equals(filter.minValInclusive)

&& maxValExclusive.equals(filter.maxValExclusive);

}

@Override

public String toString(String field) {

return "DocValRangeFilter:" + csfField

+ ",type:" + csfFieldType.toString()

+ ",min:" + this.minValInclusive.toString()

+ ",max:" + this.maxValExclusive.toString();

}

@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 (csfFieldType == null) {

return new AllDocsIterator(reader);

}

int smallestDoc = (reader instanceof EarlybirdIndexSegmentAtomicReader)

? ((EarlybirdIndexSegmentAtomicReader) reader).getSmallestDocID() : 0;

int largestDoc = reader.maxDoc() - 1;

return new CSFRangeDocIdSetIterator(reader, csfField, csfFieldType,

smallestDoc, largestDoc,

minValInclusive, maxValExclusive);

}

};

}

private static final class CSFRangeDocIdSetIterator extends RangeFilterDISI {

private final NumericDocValues numericDocValues;

private final ThriftCSFType csfType;

private final Number minValInclusive;

private final Number maxValExclusive;

public CSFRangeDocIdSetIterator(LeafReader reader,

String csfField,

ThriftCSFType csfType,

int smallestDocID,

int largestDocID,

Number minValInclusive,

Number maxValExclusive) throws IOException {

super(reader, smallestDocID, largestDocID);

this.numericDocValues = reader.getNumericDocValues(csfField);

this.csfType = csfType;

this.minValInclusive = minValInclusive;

this.maxValExclusive = maxValExclusive;

}

@Override

protected boolean shouldReturnDoc() throws IOException {

if (!numericDocValues.advanceExact(docID())) {

return false;

}

long val = numericDocValues.longValue();

switch (csfType) {

case DOUBLE:

double doubleVal = Double.longBitsToDouble(val);

return doubleVal >= minValInclusive.doubleValue()

&& doubleVal < maxValExclusive.doubleValue();

case FLOAT:

float floatVal = Float.intBitsToFloat((int) val);

return floatVal >= minValInclusive.doubleValue()

&& floatVal < maxValExclusive.doubleValue();

case LONG:

return val >= minValInclusive.longValue() && val < maxValExclusive.longValue();

case INT:

return val >= minValInclusive.longValue() && (int) val < maxValExclusive.longValue();

case BYTE:

return (byte) val >= minValInclusive.longValue()

&& (byte) val < maxValExclusive.longValue();

default:

return false;

}

}

}

//////////////////////////

// for unit tests only

//////////////////////////

@VisibleForTesting

public Number getMinValForTest() {

return minValInclusive;

}

@VisibleForTesting

public Number getMaxValForTest() {

return maxValExclusive;

}

}