package com.twitter.search.common.search;

import java.io.IOException;

import java.util.List;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.IndexReader;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.index.MultiDocValues;

import org.apache.lucene.index.NumericDocValues;

import org.apache.lucene.index.Term;

import org.apache.lucene.index.Terms;

import org.apache.lucene.search.CollectionStatistics;

import org.apache.lucene.search.Collector;

import org.apache.lucene.search.DocIdSetIterator;

import org.apache.lucene.search.IndexSearcher;

import org.apache.lucene.search.LeafCollector;

import org.apache.lucene.search.Scorer;

import org.apache.lucene.search.TermStatistics;

import org.apache.lucene.search.Weight;

/\*\*

\* An IndexSearch that works with TwitterEarlyTerminationCollector.

\* If a stock Lucene collector is passed into search(), this IndexSearch.search() behaves the

\* same as Lucene's stock IndexSearcher. However, if a TwitterEarlyTerminationCollector is passed

\* in, this IndexSearcher performs early termination without relying on

\* CollectionTerminatedException.

\*/

public class TwitterIndexSearcher extends IndexSearcher {

public TwitterIndexSearcher(IndexReader r) {

super(r);

}

/\*\*

\* search() main loop.

\* This behaves exactly like IndexSearcher.search() if a stock Lucene collector passed in.

\* However, if a TwitterCollector is passed in, this class performs Twitter style early

\* termination without relying on

\* {@link org.apache.lucene.search.CollectionTerminatedException}.

\*/

@Override

protected void search(List<LeafReaderContext> leaves, Weight weight, Collector coll)

throws IOException {

// If an TwitterCollector is passed in, we can do a few extra things in here, such

// as early termination. Otherwise we can just fall back to IndexSearcher.search().

if (coll instanceof TwitterCollector) {

TwitterCollector collector = (TwitterCollector) coll;

for (LeafReaderContext ctx : leaves) { // search each subreader

if (collector.isTerminated()) {

return;

}

// Notify the collector that we're starting this segment, and check for early

// termination criteria again. setNextReader() performs 'expensive' early

// termination checks in some implementations such as TwitterEarlyTerminationCollector.

LeafCollector leafCollector = collector.getLeafCollector(ctx);

if (collector.isTerminated()) {

return;

}

// Initialize the scorer - it should not be null. Note that constructing the scorer

// may actually do real work, such as advancing to the first hit.

Scorer scorer = weight.scorer(ctx);

if (scorer == null) {

collector.finishSegment(DocIdSetIterator.NO\_MORE\_DOCS);

continue;

}

leafCollector.setScorer(scorer);

// Start searching.

DocIdSetIterator docIdSetIterator = scorer.iterator();

int docID = docIdSetIterator.nextDoc();

if (docID != DocIdSetIterator.NO\_MORE\_DOCS) {

// Collect results. Note: check isTerminated() before calling nextDoc().

do {

leafCollector.collect(docID);

} while (!collector.isTerminated()

&& (docID = docIdSetIterator.nextDoc()) != DocIdSetIterator.NO\_MORE\_DOCS);

}

// Always finish the segment, providing the last docID advanced to.

collector.finishSegment(docID);

}

} else {

// The collector given is not a TwitterCollector, just use stock lucene search().

super.search(leaves, weight, coll);

}

}

/\*\* Returns {@link NumericDocValues} for this field, or

\* null if no {@link NumericDocValues} were indexed for

\* this field. The returned instance should only be

\* used by a single thread. \*/

public NumericDocValues getNumericDocValues(String field) throws IOException {

return MultiDocValues.getNumericValues(getIndexReader(), field);

}

@Override

public CollectionStatistics collectionStatistics(String field) throws IOException {

return collectionStatistics(field, getIndexReader());

}

@Override

public TermStatistics termStatistics(Term term, int docFreq, long totalTermFreq) {

return termStats(term, docFreq, totalTermFreq);

}

/\*\*

\* Lucene relies on the fact that maxDocID is typically equal to the number of documents in the

\* index, which is false when we have sparse doc IDs or when we start from 8 million docs and

\* decrement, so in this class we pass in numDocs instead of the maximum assigned document ID.

\* Note that the comment on {@link CollectionStatistics#maxDoc()} says that it returns the number

\* of documents in the segment, not the maximum ID, and that it is only used this way. This is

\* necessary for all lucene scoring methods, e.g.

\* {@link org.apache.lucene.search.similarities.TFIDFSimilarity#idfExplain}. This method body is

\* largely copied from {@link IndexSearcher#collectionStatistics(String)}.

\*/

public static CollectionStatistics collectionStatistics(String field, IndexReader indexReader)

throws IOException {

Preconditions.checkNotNull(field);

int docsWithField = 0;

long sumTotalTermFreq = 0;

long sumDocFreq = 0;

for (LeafReaderContext leaf : indexReader.leaves()) {

Terms terms = leaf.reader().terms(field);

if (terms == null) {

continue;

}

docsWithField += terms.getDocCount();

sumTotalTermFreq += terms.getSumTotalTermFreq();

sumDocFreq += terms.getSumDocFreq();

}

if (docsWithField == 0) {

// The CollectionStatistics API in Lucene is designed poorly. On one hand, starting with

// Lucene 8.0.0, searchers are expected to always produce valid CollectionStatistics instances

// and all int fields in these instances are expected to be strictly greater than 0. On the

// other hand, Lucene itself produces null CollectionStatistics instances in a few places.

// Also, there's no good placeholder value to indicate that a field is empty, which is a very

// reasonable thing to happen (for example, the first few tweets in a new segment might not

// have any links, so then the resolved\_links\_text would be empty). So to get around this

// issue, we do here what Lucene does: we return a CollectionStatistics instance with all

// fields set to 1.

return new CollectionStatistics(field, 1, 1, 1, 1);

}

// The writer could have added more docs to the index since this searcher started processing

// this request, or could be in the middle of adding a doc, which could mean that only some of

// the docsWithField, sumTotalTermFreq and sumDocFreq stats have been updated. I don't think

// this is a big deal, as these stats are only used for computing a hit's score, and minor

// inaccuracies should have very little effect on a hit's final score. But CollectionStatistic's

// constructor has some strict asserts for the relationship between these stats. So we need to

// make sure we cap the values of these stats appropriately.

//

// Adjust numDocs based on docsWithField (instead of doing the opposite), because:

// 1. If new documents were added to this segment after the reader was created, it seems

// reasonable to take the more recent information into account.

// 2. The termStats() method below will return the most recent docFreq (not the value that

// docFreq was set to when this reader was created). If this value is higher than numDocs,

// then Lucene might end up producing negative scores, which must never happen.

int numDocs = Math.max(indexReader.numDocs(), docsWithField);

sumDocFreq = Math.max(sumDocFreq, docsWithField);

sumTotalTermFreq = Math.max(sumTotalTermFreq, sumDocFreq);

return new CollectionStatistics(field, numDocs, docsWithField, sumTotalTermFreq, sumDocFreq);

}

/\*\*

\* This method body is largely copied from {@link IndexSearcher#termStatistics(Term, int, long)}.

\* The only difference is that we make sure all parameters we pass to the TermStatistics instance

\* we create are set to at least 1 (because Lucene 8.0.0 expects them to be).

\*/

public static TermStatistics termStats(Term term, int docFreq, long totalTermFreq) {

// Lucene expects the doc frequency and total term frequency to be at least 1. This assumption

// doesn't always make sense (the segment can be empty -- see comment above), but to make Lucene

// happy, make sure to always set these parameters to at least 1.

int adjustedDocFreq = Math.max(docFreq, 1);

return new TermStatistics(

term.bytes(),

adjustedDocFreq,

Math.max(totalTermFreq, adjustedDocFreq));

}

}