package com.twitter.search.common.query;

import java.io.IOException;

import java.util.Set;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.IndexReader;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.index.Term;

import org.apache.lucene.search.DocIdSetIterator;

import org.apache.lucene.search.Explanation;

import org.apache.lucene.search.IndexSearcher;

import org.apache.lucene.search.Query;

import org.apache.lucene.search.Scorer;

import org.apache.lucene.search.ScoreMode;

import org.apache.lucene.search.Weight;

/\*\*

\* A pairing of a query and a filter. The hits traversal is driven by the query's DocIdSetIterator,

\* and the filter is used only to do post-filtering. In other words, the filter is never used to

\* find the next doc ID: it's only used to filter out the doc IDs returned by the query's

\* DocIdSetIterator. This is useful when we need to have a conjunction between a query that can

\* quickly iterate through doc IDs (eg. a posting list), and an expensive filter (eg. a filter based

\* on the values stored in a CSF).

\*

\* For example, let say we want to build a query that returns all docs that have at least 100 faves.

\* 1. One option is to go with the [min\_faves 100] query. This would be very expensive though,

\* because this query would have to walk through every doc in the segment and for each one of

\* them it would have to extract the number of faves from the forward index.

\* 2. Another option is to go with a conjunction between this query and the HAS\_ENGAGEMENT filter:

\* (+[min\_faves 100] +[cached\_filter has\_engagements]). The HAS\_ENGAGEMENT filter could

\* traverse the doc ID space faster (if it's backed by a posting list). But this approach would

\* still be slow, because as soon as the HAS\_ENGAGEMENT filter finds a doc ID, the conjunction

\* scorer would trigger an advance(docID) call on the min\_faves part of the query, which has

\* the same problem as the first option.

\* 3. Finally, a better option for this particular case would be to drive by the HAS\_ENGAGEMENT

\* filter (because it can quickly jump over all docs that do not have any engagement), and use

\* the min\_faves filter as a post-processing step, on a much smaller set of docs.

\*/

public class FilteredQuery extends Query {

/\*\*

\* A doc ID predicate that determines if the given doc ID should be accepted.

\*/

@FunctionalInterface

public static interface DocIdFilter {

/\*\*

\* Determines if the given doc ID should be accepted.

\*/

boolean accept(int docId) throws IOException;

}

/\*\*

\* A factory for creating DocIdFilter instances based on a given LeafReaderContext instance.

\*/

@FunctionalInterface

public static interface DocIdFilterFactory {

/\*\*

\* Returns a DocIdFilter instance for the given LeafReaderContext instance.

\*/

DocIdFilter getDocIdFilter(LeafReaderContext context) throws IOException;

}

private static class FilteredQueryDocIdSetIterator extends DocIdSetIterator {

private final DocIdSetIterator queryScorerIterator;

private final DocIdFilter docIdFilter;

public FilteredQueryDocIdSetIterator(

DocIdSetIterator queryScorerIterator, DocIdFilter docIdFilter) {

this.queryScorerIterator = Preconditions.checkNotNull(queryScorerIterator);

this.docIdFilter = Preconditions.checkNotNull(docIdFilter);

}

@Override

public int docID() {

return queryScorerIterator.docID();

}

@Override

public int nextDoc() throws IOException {

int docId;

do {

docId = queryScorerIterator.nextDoc();

} while (docId != NO\_MORE\_DOCS && !docIdFilter.accept(docId));

return docId;

}

@Override

public int advance(int target) throws IOException {

int docId = queryScorerIterator.advance(target);

if (docId == NO\_MORE\_DOCS || docIdFilter.accept(docId)) {

return docId;

}

return nextDoc();

}

@Override

public long cost() {

return queryScorerIterator.cost();

}

}

private static class FilteredQueryScorer extends Scorer {

private final Scorer queryScorer;

private final DocIdFilter docIdFilter;

public FilteredQueryScorer(Weight weight, Scorer queryScorer, DocIdFilter docIdFilter) {

super(weight);

this.queryScorer = Preconditions.checkNotNull(queryScorer);

this.docIdFilter = Preconditions.checkNotNull(docIdFilter);

}

@Override

public int docID() {

return queryScorer.docID();

}

@Override

public float score() throws IOException {

return queryScorer.score();

}

@Override

public DocIdSetIterator iterator() {

return new FilteredQueryDocIdSetIterator(queryScorer.iterator(), docIdFilter);

}

@Override

public float getMaxScore(int upTo) throws IOException {

return queryScorer.getMaxScore(upTo);

}

}

private static class FilteredQueryWeight extends Weight {

private final Weight queryWeight;

private final DocIdFilterFactory docIdFilterFactory;

public FilteredQueryWeight(

FilteredQuery query, Weight queryWeight, DocIdFilterFactory docIdFilterFactory) {

super(query);

this.queryWeight = Preconditions.checkNotNull(queryWeight);

this.docIdFilterFactory = Preconditions.checkNotNull(docIdFilterFactory);

}

@Override

public void extractTerms(Set<Term> terms) {

queryWeight.extractTerms(terms);

}

@Override

public Explanation explain(LeafReaderContext context, int doc) throws IOException {

return queryWeight.explain(context, doc);

}

@Override

public Scorer scorer(LeafReaderContext context) throws IOException {

Scorer queryScorer = queryWeight.scorer(context);

if (queryScorer == null) {

return null;

}

return new FilteredQueryScorer(this, queryScorer, docIdFilterFactory.getDocIdFilter(context));

}

@Override

public boolean isCacheable(LeafReaderContext ctx) {

return queryWeight.isCacheable(ctx);

}

}

private final Query query;

private final DocIdFilterFactory docIdFilterFactory;

public FilteredQuery(Query query, DocIdFilterFactory docIdFilterFactory) {

this.query = Preconditions.checkNotNull(query);

this.docIdFilterFactory = Preconditions.checkNotNull(docIdFilterFactory);

}

public Query getQuery() {

return query;

}

@Override

public Query rewrite(IndexReader reader) throws IOException {

Query rewrittenQuery = query.rewrite(reader);

if (rewrittenQuery != query) {

return new FilteredQuery(rewrittenQuery, docIdFilterFactory);

}

return this;

}

@Override

public int hashCode() {

return query.hashCode() \* 13 + docIdFilterFactory.hashCode();

}

@Override

public boolean equals(Object obj) {

if (!(obj instanceof FilteredQuery)) {

return false;

}

FilteredQuery filteredQuery = FilteredQuery.class.cast(obj);

return query.equals(filteredQuery.query)

&& docIdFilterFactory.equals(filteredQuery.docIdFilterFactory);

}

@Override

public String toString(String field) {

StringBuilder sb = new StringBuilder();

sb.append("FilteredQuery(")

.append(query)

.append(" -> ")

.append(docIdFilterFactory)

.append(")");

return sb.toString();

}

@Override

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

throws IOException {

Weight queryWeight = Preconditions.checkNotNull(query.createWeight(searcher, scoreMode, boost));

return new FilteredQueryWeight(this, queryWeight, docIdFilterFactory);

}

}