package com.twitter.search.common.query;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.Objects;

import java.util.Set;

import java.util.stream.Collectors;

import org.apache.lucene.index.FilteredTermsEnum;

import org.apache.lucene.index.IndexReader;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.index.PostingsEnum;

import org.apache.lucene.index.Term;

import org.apache.lucene.index.TermState;

import org.apache.lucene.index.TermStates;

import org.apache.lucene.index.Terms;

import org.apache.lucene.index.TermsEnum;

import org.apache.lucene.search.BooleanClause.Occur;

import org.apache.lucene.search.BooleanQuery;

import org.apache.lucene.search.BulkScorer;

import org.apache.lucene.search.ConstantScoreQuery;

import org.apache.lucene.search.ConstantScoreScorer;

import org.apache.lucene.search.ConstantScoreWeight;

import org.apache.lucene.search.DocIdSet;

import org.apache.lucene.search.DocIdSetIterator;

import org.apache.lucene.search.IndexSearcher;

import org.apache.lucene.search.MultiTermQuery;

import org.apache.lucene.search.Query;

import org.apache.lucene.search.Scorer;

import org.apache.lucene.search.ScoreMode;

import org.apache.lucene.search.TermQuery;

import org.apache.lucene.search.Weight;

import org.apache.lucene.util.AttributeSource;

import org.apache.lucene.util.BytesRef;

import org.apache.lucene.util.DocIdSetBuilder;

import com.twitter.search.common.schema.base.ImmutableSchemaInterface;

import com.twitter.search.common.schema.base.IndexedNumericFieldSettings;

import com.twitter.search.common.util.analysis.LongTermAttributeImpl;

import com.twitter.search.common.util.analysis.SortableLongTermAttributeImpl;

import com.twitter.search.queryparser.query.QueryParserException;

/\*\*

\* An extension of Lucene's MultiTermQuery which creates a disjunction of

\* long ID terms. Lucene tries to rewrite the Query depending on the number

\* of clauses to perform as efficiently as possible.

\*/

public class IDDisjunctionQuery extends MultiTermQuery {

private final List<Long> ids;

private final boolean useOrderPreservingEncoding;

/\*\* Creates a new IDDisjunctionQuery instance. \*/

public IDDisjunctionQuery(List<Long> ids, String field, ImmutableSchemaInterface schemaSnapshot)

throws QueryParserException {

super(field);

this.ids = ids;

setRewriteMethod(new Rewrite());

if (!schemaSnapshot.hasField(field)) {

throw new QueryParserException(

"Tried to search a field which does not exist in schema: " + field);

}

IndexedNumericFieldSettings numericFieldSettings =

schemaSnapshot.getFieldInfo(field).getFieldType().getNumericFieldSettings();

if (numericFieldSettings == null) {

throw new QueryParserException("Requested id field is not numerical: " + field);

}

this.useOrderPreservingEncoding = numericFieldSettings.isUseSortableEncoding();

}

/\*\*

\* Work around for an issue where LongTerms are not valid utf8, so calling

\* toString on any TermQuery containing a LongTerm may cause exceptions.

\*/

private class Rewrite extends RewriteMethod {

@Override

public Query rewrite(IndexReader reader, MultiTermQuery query) throws IOException {

Query result = new MultiTermQueryConstantScoreWrapper(

(IDDisjunctionQuery) query, useOrderPreservingEncoding);

return result;

}

}

@Override

protected TermsEnum getTermsEnum(final Terms terms, AttributeSource atts) throws IOException {

final Iterator<Long> it = this.ids.iterator();

final TermsEnum termsEnum = terms.iterator();

return new FilteredTermsEnum(termsEnum) {

private final BytesRef term = useOrderPreservingEncoding

? SortableLongTermAttributeImpl.newBytesRef()

: LongTermAttributeImpl.newBytesRef();

@Override protected AcceptStatus accept(BytesRef term) throws IOException {

return AcceptStatus.YES;

}

@Override public BytesRef next() throws IOException {

while (it.hasNext()) {

Long longTerm = it.next();

if (useOrderPreservingEncoding) {

SortableLongTermAttributeImpl.copyLongToBytesRef(term, longTerm);

} else {

LongTermAttributeImpl.copyLongToBytesRef(term, longTerm);

}

if (termsEnum.seekExact(term)) {

return term;

}

}

return null;

}

};

}

@Override

public String toString(String field) {

StringBuilder builder = new StringBuilder();

builder.append("IDDisjunction[").append(this.field).append(":");

for (Long id : this.ids) {

builder.append(id);

builder.append(",");

}

builder.setLength(builder.length() - 1);

builder.append("]");

return builder.toString();

}

private static class TermQueryWithToString extends TermQuery {

private final boolean useOrderPreservingEncoding;

public TermQueryWithToString(Term t, TermStates states, boolean useOrderPreservingEncoding) {

super(t, states);

this.useOrderPreservingEncoding = useOrderPreservingEncoding;

}

@Override

public String toString(String field) {

StringBuilder buffer = new StringBuilder();

if (!getTerm().field().equals(field)) {

buffer.append(getTerm().field());

buffer.append(":");

}

long longTerm;

BytesRef termBytes = getTerm().bytes();

if (useOrderPreservingEncoding) {

longTerm = SortableLongTermAttributeImpl.copyBytesRefToLong(termBytes);

} else {

longTerm = LongTermAttributeImpl.copyBytesRefToLong(termBytes);

}

buffer.append(longTerm);

return buffer.toString();

}

}

/\*\*

\* This class provides the functionality behind {@link MultiTermQuery#CONSTANT\_SCORE\_REWRITE}.

\* It tries to rewrite per-segment as a boolean query that returns a constant score and otherwise

\* fills a DocIdSet with matches and builds a Scorer on top of this DocIdSet.

\*/

static final class MultiTermQueryConstantScoreWrapper extends Query {

// disable the rewrite option which will scan all posting lists sequentially and perform

// the intersection using a temporary DocIdSet. In earlybird this mode is slower than a "normal"

// disjunctive BooleanQuery, due to early termination and the fact that everything is in memory.

private static final int BOOLEAN\_REWRITE\_TERM\_COUNT\_THRESHOLD = 3000;

private static class TermAndState {

private final BytesRef term;

private final TermState state;

private final int docFreq;

private final long totalTermFreq;

TermAndState(BytesRef term, TermState state, int docFreq, long totalTermFreq) {

this.term = term;

this.state = state;

this.docFreq = docFreq;

this.totalTermFreq = totalTermFreq;

}

}

private static class WeightOrDocIdSet {

private final Weight weight;

private final DocIdSet docIdSet;

WeightOrDocIdSet(Weight weight) {

this.weight = Objects.requireNonNull(weight);

this.docIdSet = null;

}

WeightOrDocIdSet(DocIdSet docIdSet) {

this.docIdSet = docIdSet;

this.weight = null;

}

}

protected final IDDisjunctionQuery query;

private final boolean useOrderPreservingEncoding;

/\*\*

\* Wrap a {@link MultiTermQuery} as a Filter.

\*/

protected MultiTermQueryConstantScoreWrapper(

IDDisjunctionQuery query,

boolean useOrderPreservingEncoding) {

this.query = query;

this.useOrderPreservingEncoding = useOrderPreservingEncoding;

}

@Override

public String toString(String field) {

// query.toString should be ok for the filter, too, if the query boost is 1.0f

return query.toString(field);

}

@Override

public boolean equals(Object obj) {

if (!(obj instanceof MultiTermQueryConstantScoreWrapper)) {

return false;

}

return query.equals(MultiTermQueryConstantScoreWrapper.class.cast(obj).query);

}

@Override

public int hashCode() {

return query == null ? 0 : query.hashCode();

}

/\*\* Returns the field name for this query \*/

public String getField() {

return query.getField();

}

private List<Long> getIDs() {

return query.ids;

}

@Override

public Weight createWeight(

final IndexSearcher searcher,

final ScoreMode scoreMode,

final float boost) throws IOException {

return new ConstantScoreWeight(this, boost) {

/\*\* Try to collect terms from the given terms enum and return true iff all

\* terms could be collected. If {@code false} is returned, the enum is

\* left positioned on the next term. \*/

private boolean collectTerms(LeafReaderContext context,

TermsEnum termsEnum,

List<TermAndState> terms) throws IOException {

final int threshold = Math.min(BOOLEAN\_REWRITE\_TERM\_COUNT\_THRESHOLD,

BooleanQuery.getMaxClauseCount());

for (int i = 0; i < threshold; ++i) {

final BytesRef term = termsEnum.next();

if (term == null) {

return true;

}

TermState state = termsEnum.termState();

terms.add(new TermAndState(BytesRef.deepCopyOf(term),

state,

termsEnum.docFreq(),

termsEnum.totalTermFreq()));

}

return termsEnum.next() == null;

}

/\*\*

\* On the given leaf context, try to either rewrite to a disjunction if

\* there are few terms, or build a DocIdSet containing matching docs.

\*/

private WeightOrDocIdSet rewrite(LeafReaderContext context)

throws IOException {

final Terms terms = context.reader().terms(query.getField());

if (terms == null) {

// field does not exist

return new WeightOrDocIdSet((DocIdSet) null);

}

final TermsEnum termsEnum = query.getTermsEnum(terms);

assert termsEnum != null;

PostingsEnum docs = null;

final List<TermAndState> collectedTerms = new ArrayList<>();

if (collectTerms(context, termsEnum, collectedTerms)) {

// build a boolean query

BooleanQuery.Builder bqBuilder = new BooleanQuery.Builder();

for (TermAndState t : collectedTerms) {

final TermStates termStates = new TermStates(searcher.getTopReaderContext());

termStates.register(t.state, context.ord, t.docFreq, t.totalTermFreq);

final Term term = new Term(query.getField(), t.term);

bqBuilder.add(

new TermQueryWithToString(term, termStates, useOrderPreservingEncoding),

Occur.SHOULD);

}

Query q = BoostUtils.maybeWrapInBoostQuery(

new ConstantScoreQuery(bqBuilder.build()), score());

return new WeightOrDocIdSet(

searcher.rewrite(q).createWeight(searcher, scoreMode, boost));

}

// Too many terms: go back to the terms we already collected and start building

// the DocIdSet

DocIdSetBuilder builder = new DocIdSetBuilder(context.reader().maxDoc());

if (!collectedTerms.isEmpty()) {

TermsEnum termsEnum2 = terms.iterator();

for (TermAndState t : collectedTerms) {

termsEnum2.seekExact(t.term, t.state);

docs = termsEnum2.postings(docs, PostingsEnum.NONE);

builder.add(docs);

}

}

// Then keep filling the DocIdSet with remaining terms

do {

docs = termsEnum.postings(docs, PostingsEnum.NONE);

builder.add(docs);

} while (termsEnum.next() != null);

return new WeightOrDocIdSet(builder.build());

}

private Scorer scorer(DocIdSet set) throws IOException {

if (set == null) {

return null;

}

final DocIdSetIterator disi = set.iterator();

if (disi == null) {

return null;

}

return new ConstantScoreScorer(this, score(), ScoreMode.COMPLETE\_NO\_SCORES, disi);

}

@Override

public BulkScorer bulkScorer(LeafReaderContext context) throws IOException {

final WeightOrDocIdSet weightOrDocIdSet = rewrite(context);

if (weightOrDocIdSet.weight != null) {

return weightOrDocIdSet.weight.bulkScorer(context);

} else {

final Scorer scorer = scorer(weightOrDocIdSet.docIdSet);

if (scorer == null) {

return null;

}

return new DefaultBulkScorer(scorer);

}

}

@Override

public Scorer scorer(LeafReaderContext context) throws IOException {

final WeightOrDocIdSet weightOrDocIdSet = rewrite(context);

if (weightOrDocIdSet.weight != null) {

return weightOrDocIdSet.weight.scorer(context);

} else {

return scorer(weightOrDocIdSet.docIdSet);

}

}

@Override

public void extractTerms(Set<Term> terms) {

terms.addAll(getIDs()

.stream()

.map(id -> new Term(getField(), LongTermAttributeImpl.copyIntoNewBytesRef(id)))

.collect(Collectors.toSet()));

}

@Override

public boolean isCacheable(LeafReaderContext ctx) {

return false;

}

};

}

}

}