package com.twitter.search.earlybird.search;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.HashSet;

import java.util.LinkedHashMap;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.stream.Collectors;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.index.MultiReader;

import org.apache.lucene.index.Term;

import org.apache.lucene.search.Collector;

import org.apache.lucene.search.Explanation;

import org.apache.lucene.search.Query;

import org.apache.lucene.search.Scorer;

import org.apache.lucene.search.ScoreMode;

import org.apache.lucene.search.Weight;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.util.Clock;

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

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

import com.twitter.search.earlybird.EarlybirdSearcher;

import com.twitter.search.earlybird.index.EarlybirdSingleSegmentSearcher;

import com.twitter.search.earlybird.index.TweetIDMapper;

import com.twitter.search.earlybird.search.facets.AbstractFacetTermCollector;

import com.twitter.search.earlybird.search.facets.TermStatisticsCollector;

import com.twitter.search.earlybird.search.facets.TermStatisticsCollector.TermStatisticsSearchResults;

import com.twitter.search.earlybird.search.facets.TermStatisticsRequestInfo;

import com.twitter.search.earlybird.search.queries.SinceMaxIDFilter;

import com.twitter.search.earlybird.search.queries.SinceUntilFilter;

import com.twitter.search.earlybird.stats.EarlybirdSearcherStats;

import com.twitter.search.earlybird.thrift.ThriftFacetCount;

import com.twitter.search.earlybird.thrift.ThriftSearchResult;

import com.twitter.search.earlybird.thrift.ThriftSearchResults;

import com.twitter.search.earlybird.thrift.ThriftTermStatisticsResults;

import com.twitter.search.queryparser.util.IdTimeRanges;

public class EarlybirdMultiSegmentSearcher extends EarlybirdLuceneSearcher {

private static final Logger LOG = LoggerFactory.getLogger(EarlybirdMultiSegmentSearcher.class);

private final ImmutableSchemaInterface schema;

private final Map<Long, EarlybirdSingleSegmentSearcher> segmentSearchers;

protected final int numSegments;

private final Clock clock;

// This will prevent us from even considering segments that are out of range.

// It's an important optimization for a certain class of queries.

protected IdTimeRanges idTimeRanges = null;

private final EarlybirdSearcherStats searcherStats;

public EarlybirdMultiSegmentSearcher(

ImmutableSchemaInterface schema,

List<EarlybirdSingleSegmentSearcher> searchers,

EarlybirdSearcherStats searcherStats,

Clock clock) throws IOException {

// NOTE: We pass in an empty MultiReader to super and retain the list of searchers in this

// class since MultiReader does not allow an aggregate of more than Integer.MAX\_VALUE docs,

// which some of our larger archive indexes may have.

super(new MultiReader());

// segmentSearchers are mapped from time slice IDs to searchers so that we can quickly

// find the correct searcher for a given time slice ID (see fillPayload).

// make sure we maintain order of segments, hence a LinkedHashMap instead of just a HashMap

this.segmentSearchers = new LinkedHashMap<>();

this.schema = schema;

for (EarlybirdSingleSegmentSearcher searcher : searchers) {

if (searcher != null) {

long timeSliceID = searcher.getTimeSliceID();

this.segmentSearchers.put(timeSliceID, searcher);

}

}

// initializing this after populating the list. previously initialized before, and

// this may have lead to a race condition, although this doesn't seem possible given

// that segments should be an immutable cloned list.

this.numSegments = segmentSearchers.size();

this.searcherStats = searcherStats;

this.clock = clock;

}

public void setIdTimeRanges(IdTimeRanges idTimeRanges) {

this.idTimeRanges = idTimeRanges;

}

@Override

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

throws IOException {

Preconditions.checkState(coll instanceof AbstractResultsCollector);

AbstractResultsCollector<?, ?> collector = (AbstractResultsCollector<?, ?>) coll;

for (EarlybirdSingleSegmentSearcher segmentSearcher : segmentSearchers.values()) {

if (shouldSkipSegment(segmentSearcher)) {

collector.skipSegment(segmentSearcher);

} else {

segmentSearcher.search(weight.getQuery(), collector);

if (collector.isTerminated()) {

break;

}

}

}

}

@VisibleForTesting

protected boolean shouldSkipSegment(EarlybirdSingleSegmentSearcher segmentSearcher) {

EarlybirdIndexSegmentData segmentData =

segmentSearcher.getTwitterIndexReader().getSegmentData();

if (idTimeRanges != null) {

if (!SinceMaxIDFilter.sinceMaxIDsInRange(

(TweetIDMapper) segmentData.getDocIDToTweetIDMapper(),

idTimeRanges.getSinceIDExclusive().or(SinceMaxIDFilter.NO\_FILTER),

idTimeRanges.getMaxIDInclusive().or(SinceMaxIDFilter.NO\_FILTER))

|| !SinceUntilFilter.sinceUntilTimesInRange(

segmentData.getTimeMapper(),

idTimeRanges.getSinceTimeInclusive().or(SinceUntilFilter.NO\_FILTER),

idTimeRanges.getUntilTimeExclusive().or(SinceUntilFilter.NO\_FILTER))) {

return true;

}

}

return false;

}

@Override

public void fillFacetResults(

AbstractFacetTermCollector collector, ThriftSearchResults searchResults) throws IOException {

for (EarlybirdSingleSegmentSearcher segmentSearcher : segmentSearchers.values()) {

segmentSearcher.fillFacetResults(collector, searchResults);

}

}

@Override

public TermStatisticsSearchResults collectTermStatistics(

TermStatisticsRequestInfo searchRequestInfo,

EarlybirdSearcher searcher,

int requestDebugMode) throws IOException {

TermStatisticsCollector collector = new TermStatisticsCollector(

schema, searchRequestInfo, searcherStats, clock, requestDebugMode);

search(collector.getSearchRequestInfo().getLuceneQuery(), collector);

searcher.maybeSetCollectorDebugInfo(collector);

return collector.getResults();

}

@Override

public void explainSearchResults(SearchRequestInfo searchRequestInfo,

SimpleSearchResults hits, ThriftSearchResults searchResults) throws IOException {

for (EarlybirdSingleSegmentSearcher segmentSearcher : segmentSearchers.values()) {

// the hits that are getting passed into this method are hits across

// all searched segments. We need to get the per segment hits and

// generate explanations one segment at a time.

List<Hit> hitsForCurrentSegment = new ArrayList<>();

Set<Long> tweetIdsForCurrentSegment = new HashSet<>();

List<ThriftSearchResult> hitResultsForCurrentSegment = new ArrayList<>();

for (Hit hit : hits.hits) {

if (hit.getTimeSliceID() == segmentSearcher.getTimeSliceID()) {

hitsForCurrentSegment.add(hit);

tweetIdsForCurrentSegment.add(hit.statusID);

}

}

for (ThriftSearchResult result : searchResults.getResults()) {

if (tweetIdsForCurrentSegment.contains(result.id)) {

hitResultsForCurrentSegment.add(result);

}

}

ThriftSearchResults resultsForSegment = new ThriftSearchResults()

.setResults(hitResultsForCurrentSegment);

SimpleSearchResults finalHits = new SimpleSearchResults(hitsForCurrentSegment);

segmentSearcher.explainSearchResults(searchRequestInfo, finalHits, resultsForSegment);

}

// We should not see hits that are not associated with an active segment

List<Hit> hitsWithUnknownSegment =

Arrays.stream(hits.hits()).filter(hit -> !hit.isHasExplanation())

.collect(Collectors.toList());

for (Hit hit : hitsWithUnknownSegment) {

LOG.error("Unable to find segment associated with hit: " + hit.toString());

}

}

@Override

public void fillFacetResultMetadata(Map<Term, ThriftFacetCount> facetResults,

ImmutableSchemaInterface documentSchema, byte debugMode)

throws IOException {

for (EarlybirdSingleSegmentSearcher segmentSearcher : segmentSearchers.values()) {

segmentSearcher.fillFacetResultMetadata(facetResults, documentSchema, debugMode);

}

}

@Override

public void fillTermStatsMetadata(ThriftTermStatisticsResults termStatsResults,

ImmutableSchemaInterface documentSchema, byte debugMode)

throws IOException {

for (EarlybirdSingleSegmentSearcher segmentSearcher : segmentSearchers.values()) {

segmentSearcher.fillTermStatsMetadata(termStatsResults, documentSchema, debugMode);

}

}

/\*\*

\* The searchers for individual segments will rewrite the query as they see fit, so the multi

\* segment searcher does not need to rewrite it. In fact, not rewriting the query here improves

\* the request latency by ~5%.

\*/

@Override

public Query rewrite(Query original) {

return original;

}

/\*\*

\* The searchers for individual segments will create their own weights. This method only creates

\* a dummy weight to pass the Lucene query to the search() method of these individual segment

\* searchers.

\*/

@Override

public Weight createWeight(Query query, ScoreMode scoreMode, float boost) {

return new DummyWeight(query);

}

/\*\*

\* Dummy weight used solely to pass Lucene Query around.

\*/

private static final class DummyWeight extends Weight {

private DummyWeight(Query luceneQuery) {

super(luceneQuery);

}

@Override

public Explanation explain(LeafReaderContext context, int doc) {

throw new UnsupportedOperationException();

}

@Override

public Scorer scorer(LeafReaderContext context) {

throw new UnsupportedOperationException();

}

@Override

public void extractTerms(Set<Term> terms) {

throw new UnsupportedOperationException();

}

@Override

public boolean isCacheable(LeafReaderContext context) {

return true;

}

}

}