package com.twitter.search.earlybird.search.facets;

import java.io.IOException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import org.apache.commons.lang.StringUtils;

import org.apache.lucene.index.PostingsEnum;

import org.apache.lucene.index.Term;

import org.apache.lucene.index.Terms;

import org.apache.lucene.index.TermsEnum;

import org.apache.lucene.search.DocIdSetIterator;

import com.twitter.common.util.Clock;

import com.twitter.search.common.metrics.SearchCounter;

import com.twitter.search.common.metrics.SearchResultsStats;

import com.twitter.search.common.schema.SchemaUtil;

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

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

import com.twitter.search.common.search.EarlyTerminationState;

import com.twitter.search.common.util.earlybird.TermStatisticsUtil;

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

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

import com.twitter.search.earlybird.search.AbstractResultsCollector;

import com.twitter.search.earlybird.search.SearchResultsInfo;

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

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

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

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

public class TermStatisticsCollector extends AbstractResultsCollector

<TermStatisticsRequestInfo, TermStatisticsCollector.TermStatisticsSearchResults> {

private static final EarlyTerminationState TERMINATED\_TERM\_STATS\_COUNTING\_DONE =

new EarlyTerminationState("terminated\_term\_stats\_counting\_done", true);

// Stats for tracking histogram results.

private static final SearchResultsStats TERM\_STATS\_HISTOGRAM\_REQUESTS\_WITH\_MOVED\_BACK\_BINS =

SearchResultsStats.export("term\_statistics\_collector\_queries\_with\_moved\_back\_bins");

private static final SearchCounter TERM\_STATS\_SKIPPED\_LARGER\_OUT\_OF\_BOUNDS\_HITS =

SearchCounter.export("term\_statistics\_collector\_skipped\_larger\_out\_of\_bounds\_hits");

@VisibleForTesting

static final class TermStatistics {

private final ThriftTermRequest termRequest;

private final Term term; // could be null, for count across all fields

private int termDF = 0;

private int termCount = 0;

private final int[] histogramBins;

// Per-segment information.

private PostingsEnum segmentDocsEnum; // could be null, for count across all fields

private boolean segmentDone;

@VisibleForTesting

TermStatistics(ThriftTermRequest termRequest, Term term, int numBins) {

this.termRequest = termRequest;

this.term = term;

this.histogramBins = new int[numBins];

}

/\*\*

\* Take the currently accumulated counts and "move them back" to make room for counts from more

\* recent binIds.

\*

\* For example, if the oldFirstBinID was set to 10, and the histogramBins were {3, 4, 5, 6, 7},

\* after this call with newFirstBinID set to 12, the histogramBins will be set

\* to {5, 6, 7, 0, 0}.

\*

\* @param oldFirstBinID the binId of the firstBin that's been used up to now.

\* @param newFirstBinID the new binId of the firstBin that will be used from now on.

\* The newFirstBinID is presumed to be larger than the oldFirstBinID, and is asserted.

\*/

@VisibleForTesting

void moveBackTermCounts(int oldFirstBinID, int newFirstBinID) {

Preconditions.checkState(oldFirstBinID < newFirstBinID);

// move counts back by this many bins

final int moveBackBy = newFirstBinID - oldFirstBinID;

this.termCount = 0;

for (int i = 0; i < histogramBins.length; i++) {

int oldCount = histogramBins[i];

histogramBins[i] = 0;

int newIndex = i - moveBackBy;

if (newIndex >= 0) {

histogramBins[newIndex] = oldCount;

this.termCount += oldCount;

}

}

}

@VisibleForTesting void countHit(int bin) {

termCount++;

histogramBins[bin]++;

}

@VisibleForTesting int getTermCount() {

return termCount;

}

@VisibleForTesting int[] getHistogramBins() {

return histogramBins;

}

}

private TermStatistics[] termStatistics;

// Histogram fields.

private int numBins;

private int binSize;

private int numTimesBinsWereMovedBack = 0;

private int numLargerOutOfBoundsBinsSkipped = 0;

private static final int SEEN\_OUT\_OF\_RANGE\_THRESHOLD = 10;

private int seenOutOfRange = 0;

// ID of the first bin - effectively time / binSize. This is calculated

// relative to the first collected in-order hit.

private int firstBinID = -1;

// List of per-segment debug information specifically useful for termstat request debugging.

private List<String> termStatisticsDebugInfo = new ArrayList<>();

/\*\*

\* Creates a new term stats collector.

\*/

public TermStatisticsCollector(

ImmutableSchemaInterface schema,

TermStatisticsRequestInfo searchRequestInfo,

EarlybirdSearcherStats searcherStats,

Clock clock,

int requestDebugMode) {

super(schema, searchRequestInfo, clock, searcherStats, requestDebugMode);

// Set up the histogram bins.

if (searchRequestInfo.isReturnHistogram()) {

ThriftHistogramSettings histogramSettings = searchRequestInfo.getHistogramSettings();

this.numBins = histogramSettings.getNumBins();

binSize = TermStatisticsUtil.determineBinSize(histogramSettings);

} else {

this.numBins = 0;

this.binSize = 0;

}

// Set up the term statistics array.

List<ThriftTermRequest> termRequests = searchRequestInfo.getTermRequests();

if (termRequests == null) {

this.termStatistics = new TermStatistics[0];

return;

}

this.termStatistics = new TermStatistics[searchRequestInfo.getTermRequests().size()];

for (int i = 0; i < searchRequestInfo.getTermRequests().size(); i++) {

final ThriftTermRequest termRequest = searchRequestInfo.getTermRequests().get(i);

Term term = null;

String fieldName = termRequest.getFieldName();

if (!StringUtils.isBlank(fieldName)) {

// First check if it's a facet field.

Schema.FieldInfo facetField = schema.getFacetFieldByFacetName(termRequest.getFieldName());

if (facetField != null) {

term = new Term(facetField.getName(), termRequest.getTerm());

} else {

// EarlybirdSearcher.validateRequest() should've already checked that the field exists in

// the schema, and that the term can be converted to the type of this field. However, if

// that did not happen for some reason, an exception will be thrown here, which will be

// converted to a TRANSIENT\_ERROR response code.

Schema.FieldInfo fieldInfo = schema.getFieldInfo(fieldName);

Preconditions.checkNotNull(

fieldInfo,

"Found a ThriftTermRequest for a field that's not in the schema: " + fieldName

+ ". This should've been caught by EarlybirdSearcher.validateRequest()!");

term = new Term(fieldName, SchemaUtil.toBytesRef(fieldInfo, termRequest.getTerm()));

}

} else {

// NOTE: if the fieldName is empty, this is a catch-all term request for the count across

// all fields. We'll just use a null term in the TermStatistics object.

}

termStatistics[i] = new TermStatistics(termRequest, term, numBins);

}

}

@Override

public void startSegment() throws IOException {

termStatisticsDebugInfo.add(

"Starting segment in timestamp range: [" + timeMapper.getFirstTime()

+ ", " + timeMapper.getLastTime() + "]");

for (TermStatistics termStats : termStatistics) {

termStats.segmentDone = true; // until we know it's false later.

TermsEnum termsEnum = null;

if (termStats.term != null) {

Terms terms = currTwitterReader.terms(termStats.term.field());

if (terms != null) {

termsEnum = terms.iterator();

if (termsEnum != null && termsEnum.seekExact(termStats.term.bytes())) {

termStats.termDF += termsEnum.docFreq(); // Only meaningful for matchAll queries.

termStats.segmentDocsEnum =

termsEnum.postings(termStats.segmentDocsEnum, PostingsEnum.FREQS);

termStats.segmentDone = termStats.segmentDocsEnum == null

|| termStats.segmentDocsEnum.nextDoc() == DocIdSetIterator.NO\_MORE\_DOCS;

} else {

// this term doesn't exist in this segment.

}

}

} else {

// Catch-all case

termStats.termDF += currTwitterReader.numDocs(); // Only meaningful for matchAll queries.

termStats.segmentDocsEnum = null;

termStats.segmentDone = false;

}

}

}

private int calculateBin(final int tweetTime) {

if (tweetTime == TimeMapper.ILLEGAL\_TIME) {

return -1;

}

final int binID = Math.abs(tweetTime) / binSize;

final int expectedFirstBinId = binID - numBins + 1;

if (firstBinID == -1) {

firstBinID = expectedFirstBinId;

} else if (expectedFirstBinId > firstBinID) {

numTimesBinsWereMovedBack++;

final int oldOutOfOrderFirstBinID = firstBinID;

firstBinID = expectedFirstBinId;

// We got a more recent out of order bin, move previous counts back.

for (TermStatistics ts : termStatistics) {

ts.moveBackTermCounts(oldOutOfOrderFirstBinID, firstBinID);

}

}

final int binIndex = binID - firstBinID;

if (binIndex >= numBins) {

// In-order times should be decreasing,

// and out of order times seen after an in-order tweet should also be smaller than the

// first in-order tweet's time. Will track these and export as a stat.

numLargerOutOfBoundsBinsSkipped++;

return -1;

} else if (binIndex < 0) {

// Early termination criteria.

seenOutOfRange++;

} else {

// Reset the counter, since we want to see consecutive tweets that are out of our bin range

// not single anomalies.

seenOutOfRange = 0;

}

return binIndex;

}

@Override

public void doCollect(long tweetID) throws IOException {

if (searchRequestInfo.isReturnHistogram()) {

final int tweetTime = timeMapper.getTime(curDocId);

final int binIndex = calculateBin(tweetTime);

if (binIndex >= 0) {

for (TermStatistics ts : termStatistics) {

if (!ts.segmentDone) {

countHist(ts, binIndex);

}

}

}

} else {

for (TermStatistics ts : termStatistics) {

if (!ts.segmentDone) {

countNoHist(ts);

}

}

}

}

@Override

public void skipSegment(EarlybirdSingleSegmentSearcher searcher) {

// Do nothing here.

// We don't do accounting that's done in AbstractResultsCollector for Term Stats

// requests because otherwise the bin ID calculation will be confused.

}

private boolean advance(TermStatistics ts) throws IOException {

PostingsEnum docsEnum = ts.segmentDocsEnum;

if (docsEnum.docID() < curDocId) {

if (docsEnum.advance(curDocId) == DocIdSetIterator.NO\_MORE\_DOCS) {

ts.segmentDone = true;

return false;

}

}

return docsEnum.docID() == curDocId;

}

private boolean countHist(TermStatistics ts, int bin) throws IOException {

if (ts.term != null && !advance(ts)) {

return false;

}

ts.countHit(bin);

return true;

}

private boolean countNoHist(TermStatistics ts) throws IOException {

if (ts.term != null && !advance(ts)) {

return false;

}

ts.termCount++;

return true;

}

@Override

public EarlyTerminationState innerShouldCollectMore() {

if (readyToTerminate()) {

return setEarlyTerminationState(TERMINATED\_TERM\_STATS\_COUNTING\_DONE);

}

return EarlyTerminationState.COLLECTING;

}

/\*\*

\* The termination logic is simple - we know what our earliest bin is and once we see a result

\* that's before our earliest bin, we terminate.

\*

\* Our results come with increasing internal doc ids, which should correspond to decreasing

\* timestamps. See SEARCH-27729, TWEETYPIE-7031.

\*

\* We early terminate after we have seen enough tweets that are outside of the bin

\* range that we want to return. This way we're not terminating too early because of single tweets

\* with wrong timestamps.

\*/

@VisibleForTesting

boolean readyToTerminate() {

return this.seenOutOfRange >= SEEN\_OUT\_OF\_RANGE\_THRESHOLD;

}

@Override

public TermStatisticsSearchResults doGetResults() {

return new TermStatisticsSearchResults();

}

public final class TermStatisticsSearchResults extends SearchResultsInfo {

public final List<Integer> binIds;

public final Map<ThriftTermRequest, ThriftTermResults> results;

public final int lastCompleteBinId;

public final List<String> termStatisticsDebugInfo;

private TermStatisticsSearchResults() {

// Initialize term stat debug info

termStatisticsDebugInfo = TermStatisticsCollector.this.termStatisticsDebugInfo;

if (termStatistics.length > 0) {

results = new HashMap<>();

if (searchRequestInfo.isReturnHistogram()) {

binIds = new ArrayList<>(numBins);

int minSearchedTime = TermStatisticsCollector.this.getMinSearchedTime();

if (shouldCollectDetailedDebugInfo()) {

termStatisticsDebugInfo.add("minSearchedTime: " + minSearchedTime);

int maxSearchedTime = TermStatisticsCollector.this.getMaxSearchedTime();

termStatisticsDebugInfo.add("maxSearchedTime: " + maxSearchedTime);

}

int lastCompleteBin = -1;

computeFirstBinId(TermStatisticsCollector.this.isSetMinSearchedTime(), minSearchedTime);

trackHistogramResultStats();

// Example:

// minSearchTime = 53s

// binSize = 10

// firstBinId = 5

// numBins = 4

// binId = 5, 6, 7, 8

// binTimeStamp = 50s, 60s, 70s, 80s

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

int binId = firstBinID + i;

int binTimeStamp = binId \* binSize;

binIds.add(binId);

if (lastCompleteBin == -1 && binTimeStamp > minSearchedTime) {

lastCompleteBin = binId;

}

}

if (!getEarlyTerminationState().isTerminated()) {

// only if we didn't early terminate we can be sure to use the firstBinID as

// lastCompleteBinId

lastCompleteBinId = firstBinID;

if (shouldCollectDetailedDebugInfo()) {

termStatisticsDebugInfo.add("no early termination");

}

} else {

lastCompleteBinId = lastCompleteBin;

if (shouldCollectDetailedDebugInfo()) {

termStatisticsDebugInfo.add(

"early terminated for reason: " + getEarlyTerminationReason());

}

}

if (shouldCollectDetailedDebugInfo()) {

termStatisticsDebugInfo.add("lastCompleteBinId: " + lastCompleteBinId);

}

} else {

binIds = null;

lastCompleteBinId = -1;

}

for (TermStatistics ts : termStatistics) {

ThriftTermResults termResults = new ThriftTermResults().setTotalCount(ts.termCount);

if (searchRequestInfo.isReturnHistogram()) {

List<Integer> list = new ArrayList<>();

for (int count : ts.histogramBins) {

list.add(count);

}

termResults.setHistogramBins(list);

}

results.put(ts.termRequest, termResults);

}

} else {

binIds = null;

results = null;

lastCompleteBinId = -1;

}

}

@Override

public String toString() {

StringBuilder res = new StringBuilder();

res.append("TermStatisticsSearchResults(\n");

if (binIds != null) {

res.append(" binIds=").append(binIds).append("\n");

}

res.append(" lastCompleteBinId=").append(lastCompleteBinId).append("\n");

if (results != null) {

res.append(" results=").append(results).append("\n");

}

res.append(")");

return res.toString();

}

public List<String> getTermStatisticsDebugInfo() {

return termStatisticsDebugInfo;

}

}

/\*\*

\* Figure out what the actual firstBinId is for this query.

\*/

private void computeFirstBinId(boolean isSetMinSearchedTime, int minSearchedTime) {

if (firstBinID == -1) {

if (!isSetMinSearchedTime) {

// This would only happen if we don't search any segments, which for now we have

// only seen happening if since\_time or until\_time don't intersect at all with

// the range of the served segments.

firstBinID = 0;

} else {

// Example:

// minSearchedTime = 54

// binSize = 10

// firstBinId = 5

firstBinID = minSearchedTime / binSize;

}

if (shouldCollectDetailedDebugInfo()) {

termStatisticsDebugInfo.add("firstBinId: " + firstBinID);

}

}

}

@VisibleForTesting

int getSeenOutOfRange() {

return seenOutOfRange;

}

private void trackHistogramResultStats() {

if (numLargerOutOfBoundsBinsSkipped > 0) {

TERM\_STATS\_SKIPPED\_LARGER\_OUT\_OF\_BOUNDS\_HITS.increment();

}

if (numTimesBinsWereMovedBack > 0) {

TERM\_STATS\_HISTOGRAM\_REQUESTS\_WITH\_MOVED\_BACK\_BINS.recordResults(numTimesBinsWereMovedBack);

}

}

}