package com.twitter.search.earlybird.search;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import java.util.Set;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Optional;

import com.google.common.base.Preconditions;

import com.google.common.collect.Maps;

import com.google.common.collect.Sets;

import org.apache.commons.collections.CollectionUtils;

import org.apache.lucene.index.LeafReader;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.search.DocIdSetIterator;

import org.apache.lucene.search.ScoreMode;

import com.twitter.common.util.Clock;

import com.twitter.search.common.constants.thriftjava.ThriftLanguage;

import com.twitter.search.common.partitioning.snowflakeparser.SnowflakeIdParser;

import com.twitter.search.common.relevance.features.EarlybirdDocumentFeatures;

import com.twitter.search.common.results.thriftjava.FieldHitAttribution;

import com.twitter.search.common.results.thriftjava.FieldHitList;

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

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

import com.twitter.search.common.schema.earlybird.EarlybirdCluster;

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants.EarlybirdFieldConstant;

import com.twitter.search.common.search.TwitterEarlyTerminationCollector;

import com.twitter.search.common.util.spatial.GeoUtil;

import com.twitter.search.core.earlybird.facets.AbstractFacetCountingArray;

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

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

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

import com.twitter.search.core.earlybird.index.inverted.QueryCostTracker;

import com.twitter.search.earlybird.common.config.EarlybirdConfig;

import com.twitter.search.earlybird.common.userupdates.UserTable;

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

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

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

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

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

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

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

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

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

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

import geo.google.datamodel.GeoCoordinate;

/\*\*

\* Abstract parent class for all results collectors in earlybird.

\* This collector should be able to handle both single-segment and

\* multi-segment collection.

\*/

public abstract class AbstractResultsCollector<R extends SearchRequestInfo,

S extends SearchResultsInfo>

extends TwitterEarlyTerminationCollector {

enum IdAndRangeUpdateType {

BEGIN\_SEGMENT,

END\_SEGMENT,

HIT

}

// Earlybird used to have a special early termination logic: at segment boundaries

// the collector estimates how much time it'll take to search the next segment.

// If this estimate \* 1.5 will cause the request to timeout, the search early terminates.

// That logic is removed in favor of more fine grained checks---now we check timeout

// within a segment, every 2,000,000 docs processed.

private static final int EXPENSIVE\_TERMINATION\_CHECK\_INTERVAL =

EarlybirdConfig.getInt("expensive\_termination\_check\_interval", 2000000);

private static final long NO\_TIME\_SLICE\_ID = -1;

protected final R searchRequestInfo;

// Sometimes maxHitsToProcess can also come from places other than collector params.

// E.g. from searchQuery.getRelevanceOptions(). This provides a way to allow

// subclasses to override the maxHitsToProcess on collector params.

private final long maxHitsToProcessOverride;

// min and max status id actually considered in the search (may not be a hit)

private long minSearchedStatusID = Long.MAX\_VALUE;

private long maxSearchedStatusID = Long.MIN\_VALUE;

private int minSearchedTime = Integer.MAX\_VALUE;

private int maxSearchedTime = Integer.MIN\_VALUE;

// per-segment start time. Will be re-started in setNextReader().

private long segmentStartTime;

// Current segment being searched.

protected EarlybirdIndexSegmentAtomicReader currTwitterReader;

protected TweetIDMapper tweetIdMapper;

protected TimeMapper timeMapper;

protected long currTimeSliceID = NO\_TIME\_SLICE\_ID;

private final long queryTime;

// Time periods, in milliseconds, for which hits are counted.

private final List<Long> hitCountsThresholdsMsec;

// hitCounts[i] is the number of hits that are more recent than hitCountsThresholdsMsec[i]

private final int[] hitCounts;

private final ImmutableSchemaInterface schema;

private final EarlybirdSearcherStats searcherStats;

// For collectors that fill in the results' geo locations, this will be used to retrieve the

// documents' lat/lon coordinates.

private GeoCoordinate resultGeoCoordinate;

protected final boolean fillInLatLonForHits;

protected EarlybirdDocumentFeatures documentFeatures;

protected boolean featuresRequested = false;

private final FacetLabelCollector facetCollector;

// debugMode set in request to determine debugging level.

private int requestDebugMode;

// debug info to be returned in earlybird response

protected List<String> debugInfo;

private int numHitsCollectedPerSegment;

public AbstractResultsCollector(

ImmutableSchemaInterface schema,

R searchRequestInfo,

Clock clock,

EarlybirdSearcherStats searcherStats,

int requestDebugMode) {

super(searchRequestInfo.getSearchQuery().getCollectorParams(),

searchRequestInfo.getTerminationTracker(),

QueryCostTracker.getTracker(),

EXPENSIVE\_TERMINATION\_CHECK\_INTERVAL,

clock);

this.schema = schema;

this.searchRequestInfo = searchRequestInfo;

ThriftSearchQuery thriftSearchQuery = searchRequestInfo.getSearchQuery();

this.maxHitsToProcessOverride = searchRequestInfo.getMaxHitsToProcess();

this.facetCollector = buildFacetCollector(searchRequestInfo, schema);

if (searchRequestInfo.getTimestamp() > 0) {

queryTime = searchRequestInfo.getTimestamp();

} else {

queryTime = System.currentTimeMillis();

}

hitCountsThresholdsMsec = thriftSearchQuery.getHitCountBuckets();

hitCounts = hitCountsThresholdsMsec == null || hitCountsThresholdsMsec.size() == 0

? null

: new int[hitCountsThresholdsMsec.size()];

this.searcherStats = searcherStats;

Schema.FieldInfo latLonCSFField =

schema.hasField(EarlybirdFieldConstant.LAT\_LON\_CSF\_FIELD.getFieldName())

? schema.getFieldInfo(EarlybirdFieldConstant.LAT\_LON\_CSF\_FIELD.getFieldName())

: null;

boolean loadLatLonMapperIntoRam = true;

if (latLonCSFField != null) {

// If the latlon\_csf field is explicitly defined, then take the config from the schema.

// If it's not defined, we assume that the latlon mapper is stored in memory.

loadLatLonMapperIntoRam = latLonCSFField.getFieldType().isCsfLoadIntoRam();

}

// Default to not fill in lat/lon if the lat/lon CSF field is not loaded into RAM

this.fillInLatLonForHits = EarlybirdConfig.getBool("fill\_in\_lat\_lon\_for\_hits",

loadLatLonMapperIntoRam);

this.requestDebugMode = requestDebugMode;

if (shouldCollectDetailedDebugInfo()) {

this.debugInfo = new ArrayList<>();

debugInfo.add("Starting Search");

}

}

private static FacetLabelCollector buildFacetCollector(

SearchRequestInfo request,

ImmutableSchemaInterface schema) {

if (CollectionUtils.isEmpty(request.getFacetFieldNames())) {

return null;

}

// Get all facet field ids requested.

Set<String> requiredFields = Sets.newHashSet();

for (String fieldName : request.getFacetFieldNames()) {

Schema.FieldInfo field = schema.getFacetFieldByFacetName(fieldName);

if (field != null) {

requiredFields.add(field.getFieldType().getFacetName());

}

}

if (requiredFields.size() > 0) {

return new FacetLabelCollector(requiredFields);

} else {

return null;

}

}

/\*\*

\* Subclasses should implement the following methods.

\*/

// Subclasses should process collected hits and construct a final

// AbstractSearchResults object.

protected abstract S doGetResults() throws IOException;

// Subclasses can override this method to add more collection logic.

protected abstract void doCollect(long tweetID) throws IOException;

public final ImmutableSchemaInterface getSchema() {

return schema;

}

// Updates the hit count array - each result only increments the first qualifying bucket.

protected final void updateHitCounts(long statusId) {

if (hitCounts == null) {

return;

}

long delta = queryTime - SnowflakeIdParser.getTimestampFromTweetId(statusId);

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

if (delta >= 0 && delta < hitCountsThresholdsMsec.get(i)) {

hitCounts[i]++;

// Increments to the rest of the count array are implied, and aggregated later, since the

// array is sorted.

break;

}

}

}

private boolean searchedStatusIDsAndTimesInitialized() {

return maxSearchedStatusID != Long.MIN\_VALUE;

}

// Updates the first searched status ID when starting to search a new segment.

private void updateFirstSearchedStatusID() {

// Only try to update the min/max searched ids, if this segment/reader actually has documents

// See SEARCH-4535

int minDocID = currTwitterReader.getSmallestDocID();

if (currTwitterReader.hasDocs() && minDocID >= 0 && !searchedStatusIDsAndTimesInitialized()) {

final long firstStatusID = tweetIdMapper.getTweetID(minDocID);

final int firstStatusTime = timeMapper.getTime(minDocID);

if (shouldCollectDetailedDebugInfo()) {

debugInfo.add(

"updateFirstSearchedStatusID. minDocId=" + minDocID + ", firstStatusID="

+ firstStatusID + ", firstStatusTime=" + firstStatusTime);

}

updateIDandTimeRanges(firstStatusID, firstStatusTime, IdAndRangeUpdateType.BEGIN\_SEGMENT);

}

}

public final R getSearchRequestInfo() {

return searchRequestInfo;

}

public final long getMinSearchedStatusID() {

return minSearchedStatusID;

}

public final long getMaxSearchedStatusID() {

return maxSearchedStatusID;

}

public final int getMinSearchedTime() {

return minSearchedTime;

}

public boolean isSetMinSearchedTime() {

return minSearchedTime != Integer.MAX\_VALUE;

}

public final int getMaxSearchedTime() {

return maxSearchedTime;

}

@Override

public final long getMaxHitsToProcess() {

return maxHitsToProcessOverride;

}

// Notifies classes that a new index segment is about to be searched.

@Override

public final void setNextReader(LeafReaderContext context) throws IOException {

super.setNextReader(context);

setNextReader(context.reader());

}

/\*\*

\* Notifies the collector that a new segment is about to be searched.

\*

\* It's easier to use this method from tests, because LeafReader is not a final class, so it can

\* be mocked (unlike LeafReaderContext).

\*/

@VisibleForTesting

public final void setNextReader(LeafReader reader) throws IOException {

if (!(reader instanceof EarlybirdIndexSegmentAtomicReader)) {

throw new RuntimeException("IndexReader type not supported: " + reader.getClass());

}

currTwitterReader = (EarlybirdIndexSegmentAtomicReader) reader;

documentFeatures = new EarlybirdDocumentFeatures(currTwitterReader);

tweetIdMapper = (TweetIDMapper) currTwitterReader.getSegmentData().getDocIDToTweetIDMapper();

timeMapper = currTwitterReader.getSegmentData().getTimeMapper();

currTimeSliceID = currTwitterReader.getSegmentData().getTimeSliceID();

updateFirstSearchedStatusID();

if (shouldCollectDetailedDebugInfo()) {

debugInfo.add("Starting search in segment with timeslice ID: " + currTimeSliceID);

}

segmentStartTime = getClock().nowMillis();

startSegment();

}

protected abstract void startSegment() throws IOException;

@Override

protected final void doCollect() throws IOException {

documentFeatures.advance(curDocId);

long tweetID = tweetIdMapper.getTweetID(curDocId);

updateIDandTimeRanges(tweetID, timeMapper.getTime(curDocId), IdAndRangeUpdateType.HIT);

doCollect(tweetID);

numHitsCollectedPerSegment++;

}

protected void collectFeatures(ThriftSearchResultMetadata metadata) throws IOException {

if (featuresRequested) {

ensureExtraMetadataIsSet(metadata);

metadata.getExtraMetadata().setDirectedAtUserId(

documentFeatures.getFeatureValue(EarlybirdFieldConstant.DIRECTED\_AT\_USER\_ID\_CSF));

metadata.getExtraMetadata().setQuotedTweetId(

documentFeatures.getFeatureValue(EarlybirdFieldConstant.QUOTED\_TWEET\_ID\_CSF));

metadata.getExtraMetadata().setQuotedUserId(

documentFeatures.getFeatureValue(EarlybirdFieldConstant.QUOTED\_USER\_ID\_CSF));

int cardLangValue =

(int) documentFeatures.getFeatureValue(EarlybirdFieldConstant.CARD\_LANG\_CSF);

ThriftLanguage thriftLanguage = ThriftLanguage.findByValue(cardLangValue);

metadata.getExtraMetadata().setCardLang(thriftLanguage);

long cardNumericUri =

(long) documentFeatures.getFeatureValue(EarlybirdFieldConstant.CARD\_URI\_CSF);

if (cardNumericUri > 0) {

metadata.getExtraMetadata().setCardUri(String.format("card://%s", cardNumericUri));

}

}

}

protected void collectIsProtected(

ThriftSearchResultMetadata metadata, EarlybirdCluster cluster, UserTable userTable)

throws IOException {

// 'isUserProtected' field is only set for archive cluster because only archive cluster user

// table has IS\_PROTECTED\_BIT populated.

// Since this bit is checked after UserFlagsExcludeFilter checked this bit, there is a slight

// chance that this bit is updated in-between. When that happens, it is possible that we will

// see a small number of protected Tweets in the response when we meant to exclude them.

if (cluster == EarlybirdCluster.FULL\_ARCHIVE) {

ensureExtraMetadataIsSet(metadata);

long userId = documentFeatures.getFeatureValue(EarlybirdFieldConstant.FROM\_USER\_ID\_CSF);

boolean isProtected = userTable.isSet(userId, UserTable.IS\_PROTECTED\_BIT);

metadata.getExtraMetadata().setIsUserProtected(isProtected);

}

}

protected void collectExclusiveConversationAuthorId(ThriftSearchResultMetadata metadata)

throws IOException {

if (searchRequestInfo.isCollectExclusiveConversationAuthorId()) {

long exclusiveConversationAuthorId = documentFeatures.getFeatureValue(

EarlybirdFieldConstant.EXCLUSIVE\_CONVERSATION\_AUTHOR\_ID\_CSF);

if (exclusiveConversationAuthorId != 0L) {

ensureExtraMetadataIsSet(metadata);

metadata.getExtraMetadata().setExclusiveConversationAuthorId(exclusiveConversationAuthorId);

}

}

}

// It only makes sense to collectFacets for search types that return individual results (recency,

// relevance and top\_tweets), which use the AbstractRelevanceCollector and SearchResultsCollector,

// so this method should only be called from these classes.

protected void collectFacets(ThriftSearchResultMetadata metadata) {

if (currTwitterReader == null) {

return;

}

AbstractFacetCountingArray facetCountingArray = currTwitterReader.getFacetCountingArray();

EarlybirdIndexSegmentData segmentData = currTwitterReader.getSegmentData();

if (facetCountingArray == null || facetCollector == null) {

return;

}

facetCollector.resetFacetLabelProviders(

segmentData.getFacetLabelProviders(),

segmentData.getFacetIDMap());

facetCountingArray.collectForDocId(curDocId, facetCollector);

List<ThriftFacetLabel> labels = facetCollector.getLabels();

if (labels.size() > 0) {

metadata.setFacetLabels(labels);

}

}

protected void ensureExtraMetadataIsSet(ThriftSearchResultMetadata metadata) {

if (!metadata.isSetExtraMetadata()) {

metadata.setExtraMetadata(new ThriftSearchResultExtraMetadata());

}

}

@Override

protected final void doFinishSegment(int lastSearchedDocID) {

if (shouldCollectDetailedDebugInfo()) {

long timeSpentSearchingSegmentInMillis = getClock().nowMillis() - segmentStartTime;

debugInfo.add("Finished segment at doc id: " + lastSearchedDocID);

debugInfo.add("Time spent searching " + currTimeSliceID

+ ": " + timeSpentSearchingSegmentInMillis + "ms");

debugInfo.add("Number of hits collected in segment " + currTimeSliceID + ": "

+ numHitsCollectedPerSegment);

}

if (!currTwitterReader.hasDocs()) {

// Due to race between the reader and the indexing thread, a seemingly empty segment that

// does not have document committed in the posting lists, might already have a document

// inserted into the id/time mappers, which we do not want to take into account.

// If there are no documents in the segment, we don't update searched min/max ids to

// anything.

return;

} else if (lastSearchedDocID == DocIdSetIterator.NO\_MORE\_DOCS) {

// Segment exhausted.

if (shouldCollectDetailedDebugInfo()) {

debugInfo.add("Segment exhausted");

}

updateIDandTimeRanges(tweetIdMapper.getMinTweetID(), timeMapper.getFirstTime(),

IdAndRangeUpdateType.END\_SEGMENT);

} else if (lastSearchedDocID >= 0) {

long lastSearchedTweetID = tweetIdMapper.getTweetID(lastSearchedDocID);

int lastSearchTweetTime = timeMapper.getTime(lastSearchedDocID);

if (shouldCollectDetailedDebugInfo()) {

debugInfo.add("lastSearchedDocId=" + lastSearchedDocID);

}

updateIDandTimeRanges(lastSearchedTweetID, lastSearchTweetTime,

IdAndRangeUpdateType.END\_SEGMENT);

}

numHitsCollectedPerSegment = 0;

}

private void updateIDandTimeRanges(long tweetID, int time, IdAndRangeUpdateType updateType) {

// We need to update minSearchedStatusID/maxSearchedStatusID and

// minSearchedTime/maxSearchedTime independently: SEARCH-6139

minSearchedStatusID = Math.min(minSearchedStatusID, tweetID);

maxSearchedStatusID = Math.max(maxSearchedStatusID, tweetID);

if (time > 0) {

minSearchedTime = Math.min(minSearchedTime, time);

maxSearchedTime = Math.max(maxSearchedTime, time);

}

if (shouldCollectVerboseDebugInfo()) {

debugInfo.add(

String.format("call to updateIDandTimeRanges(%d, %d, %s)"

+ " set minSearchStatusID=%d, maxSearchedStatusID=%d,"

+ " minSearchedTime=%d, maxSearchedTime=%d)",

tweetID, time, updateType.toString(),

minSearchedStatusID, maxSearchedStatusID,

minSearchedTime, maxSearchedTime));

}

}

/\*\*

\* This is called when a segment is skipped but we would want to do accounting

\* for minSearchDocId as well as numDocsProcessed.

\*/

public void skipSegment(EarlybirdSingleSegmentSearcher searcher) throws IOException {

setNextReader(searcher.getTwitterIndexReader().getContext());

trackCompleteSegment(DocIdSetIterator.NO\_MORE\_DOCS);

if (shouldCollectDetailedDebugInfo()) {

debugInfo.add("Skipping segment: " + currTimeSliceID);

}

}

/\*\*

\* Returns the results collected by this collector.

\*/

public final S getResults() throws IOException {

// In order to make pagination work, if minSearchedStatusID is greater than the asked max\_id.

// We force the minSearchedStatusID to be max\_id + 1.

IdTimeRanges idTimeRanges = searchRequestInfo.getIdTimeRanges();

if (idTimeRanges != null) {

Optional<Long> maxIDInclusive = idTimeRanges.getMaxIDInclusive();

if (maxIDInclusive.isPresent() && minSearchedStatusID > maxIDInclusive.get()) {

searcherStats.numCollectorAdjustedMinSearchedStatusID.increment();

minSearchedStatusID = maxIDInclusive.get() + 1;

}

}

S results = doGetResults();

results.setNumHitsProcessed((int) getNumHitsProcessed());

results.setNumSearchedSegments(getNumSearchedSegments());

if (searchedStatusIDsAndTimesInitialized()) {

results.setMaxSearchedStatusID(maxSearchedStatusID);

results.setMinSearchedStatusID(minSearchedStatusID);

results.setMaxSearchedTime(maxSearchedTime);

results.setMinSearchedTime(minSearchedTime);

}

results.setEarlyTerminated(getEarlyTerminationState().isTerminated());

if (getEarlyTerminationState().isTerminated()) {

results.setEarlyTerminationReason(getEarlyTerminationState().getTerminationReason());

}

Map<Long, Integer> counts = getHitCountMap();

if (counts != null) {

results.hitCounts.putAll(counts);

}

return results;

}

/\*\*

\* Returns a map of timestamps (specified in the query) to the number of hits that are more recent

\* that the respective timestamps.

\*/

public final Map<Long, Integer> getHitCountMap() {

int total = 0;

if (hitCounts == null) {

return null;

}

Map<Long, Integer> map = Maps.newHashMap();

// since the array is incremental, need to aggregate here.

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

map.put(hitCountsThresholdsMsec.get(i), total += hitCounts[i]);

}

return map;

}

/\*\*

\* Common helper for collecting per-field hit attribution data (if it's available).

\*

\* @param metadata the metadata to fill for this hit.

\*/

protected final void fillHitAttributionMetadata(ThriftSearchResultMetadata metadata) {

if (searchRequestInfo.getHitAttributeHelper() == null) {

return;

}

Map<Integer, List<String>> hitAttributeMapping =

searchRequestInfo.getHitAttributeHelper().getHitAttribution(curDocId);

Preconditions.checkNotNull(hitAttributeMapping);

FieldHitAttribution fieldHitAttribution = new FieldHitAttribution();

for (Map.Entry<Integer, List<String>> entry : hitAttributeMapping.entrySet()) {

FieldHitList fieldHitList = new FieldHitList();

fieldHitList.setHitFields(entry.getValue());

fieldHitAttribution.putToHitMap(entry.getKey(), fieldHitList);

}

metadata.setFieldHitAttribution(fieldHitAttribution);

}

/\*\*

\* Fill the geo location of the given document in metadata, if we have the lat/lon for it.

\* For queries that specify a geolocation, this will also have the distance from

\* the location specified in the query, and the location of this document.

\*/

protected final void fillResultGeoLocation(ThriftSearchResultMetadata metadata)

throws IOException {

Preconditions.checkNotNull(metadata);

if (currTwitterReader != null && fillInLatLonForHits) {

// See if we can have a lat/lon for this doc.

if (resultGeoCoordinate == null) {

resultGeoCoordinate = new GeoCoordinate();

}

// Only fill if necessary

if (searchRequestInfo.isCollectResultLocation()

&& GeoUtil.decodeLatLonFromInt64(

documentFeatures.getFeatureValue(EarlybirdFieldConstant.LAT\_LON\_CSF\_FIELD),

resultGeoCoordinate)) {

ThriftSearchResultGeoLocation resultLocation = new ThriftSearchResultGeoLocation();

resultLocation.setLatitude(resultGeoCoordinate.getLatitude());

resultLocation.setLongitude(resultGeoCoordinate.getLongitude());

metadata.setResultLocation(resultLocation);

}

}

}

@Override

public ScoreMode scoreMode() {

return ScoreMode.COMPLETE;

}

private int terminationDocID = -1;

@Override

protected void collectedEnoughResults() throws IOException {

// We find 'terminationDocID' once we collect enough results, so that we know the point at which

// we can stop searching. We must do this because with the unordered doc ID mapper, tweets

// are not ordered within a millisecond, so we must search the entire millisecond bucket before

// terminating the search, otherwise we could skip over tweets and have an incorrect

// minSearchedStatusID.

if (curDocId != -1 && terminationDocID == -1) {

long tweetId = tweetIdMapper.getTweetID(curDocId);

// We want to find the highest possible doc ID for this tweetId, so pass true.

boolean findMaxDocID = true;

terminationDocID = tweetIdMapper.findDocIdBound(tweetId,

findMaxDocID,

curDocId,

curDocId);

}

}

@Override

protected boolean shouldTerminate() {

return curDocId >= terminationDocID;

}

@Override

public List<String> getDebugInfo() {

return debugInfo;

}

protected boolean shouldCollectDetailedDebugInfo() {

return requestDebugMode >= 5;

}

// Use this for per-result debug info. Useful for queries with no results

// or a very small number of results.

protected boolean shouldCollectVerboseDebugInfo() {

return requestDebugMode >= 6;

}

}