package com.twitter.search.common.search;

import java.io.IOException;

import java.util.List;

import javax.annotation.Nonnull;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.LeafReader;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.search.LeafCollector;

import org.apache.lucene.search.Scorable;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.util.Clock;

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

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

import com.twitter.search.common.query.thriftjava.CollectorParams;

import com.twitter.search.common.query.thriftjava.CollectorTerminationParams;

/\*\*

\* A TwitterCollector containing the most common early termination logic based on

\* timeout, cost, and max hits. This class does not do any actual hit collection---this class

\* is abstract and cannot be instantiated.

\*

\* If a Collector and all its subclasses need early termination, it should extend this class.

\*

\* However, if one just wants to add EarlyTermination to any single collector, he can just

\* use {@link DelegatingEarlyTerminationCollector}

\* as a wrapper.

\*/

public abstract class TwitterEarlyTerminationCollector

extends TwitterCollector implements LeafCollector {

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

private static final SearchCounter NEGATIVE\_TIME\_PER\_SEGMENT =

SearchCounter.export("TwitterEarlyTerminationCollector\_negative\_time\_per\_segment");

private static final SearchRateCounter QUERY\_TIMEOUT\_ENFORCED =

SearchRateCounter.export("TwitterEarlyTerminationCollector\_query\_timeout\_enforced");

protected int curDocId = -1;

protected Scorable scorer = null;

private LeafReader curReader = null;

private final long maxHitsToProcess;

private long numHitsProcessed = 0;

private int lastEarlyTerminationCheckDocId = -1;

private final Clock clock;

@Nullable

private final QueryCostProvider queryCostProvider;

private final TerminationTracker terminationTracker;

// This determines how often the expensive early termination check is performed.

// If set to be negative, expensive early termination check only performed at segment boundaries.

// If set to a positive number X, this check is performed every X docs processed.

private int numDocsBetweenTimeoutChecks;

// Number of segments searched so far.

// This is used to predicatively early terminate.

// Expensive early termination checks may not happen often enough. Sometimes the request

// times out in between the termination checks.

// After finishing searching a segment, we estimate how much time is needed to search one

// segment on average. If searching the next segment would cause a timeout, we early terminate.

private int numSearchedSegments = 0;

/\*\*

\* Creates a new TwitterEarlyTerminationCollector instance.

\*

\* @param collectorParams the parameters needed to guide early termination.

\* @param terminationTracker If null is passed in, a new TerminationTrack is created. Otherwise,

\* the one passed in is used.

\* @param numDocsBetweenTimeoutChecks TerminationTracker based check are performed upon a hit

\* every numDocsBetweenTimeoutChecks docs. If a non-positive number is passed

\* in, TerminationTracker based checks are disabled.

\* If collectorParams specifies a value as well, that value is used.

\*/

public TwitterEarlyTerminationCollector(

CollectorParams collectorParams,

TerminationTracker terminationTracker,

@Nullable QueryCostProvider queryCostProvider,

int numDocsBetweenTimeoutChecks,

Clock clock) {

CollectorTerminationParams terminationParams = collectorParams.getTerminationParams();

if (terminationParams == null) {

terminationParams = new CollectorTerminationParams()

.setMaxHitsToProcess(Integer.MAX\_VALUE)

.setMaxQueryCost(Double.MAX\_VALUE)

.setTimeoutMs(Integer.MAX\_VALUE);

}

if (!terminationParams.isSetMaxHitsToProcess() || terminationParams.getMaxHitsToProcess() < 0) {

maxHitsToProcess = Integer.MAX\_VALUE;

} else {

maxHitsToProcess = terminationParams.getMaxHitsToProcess();

}

if (terminationParams.isSetNumDocsBetweenTimeoutChecks()) {

this.numDocsBetweenTimeoutChecks = terminationParams.getNumDocsBetweenTimeoutChecks();

} else {

this.numDocsBetweenTimeoutChecks = numDocsBetweenTimeoutChecks;

}

this.terminationTracker = Preconditions.checkNotNull(terminationTracker);

this.queryCostProvider = queryCostProvider;

this.clock = clock;

}

public final LeafCollector getLeafCollector(LeafReaderContext context) throws IOException {

this.setNextReader(context);

return this;

}

/\*\*

\* Sub-classes may override this to add more collection logic.

\*/

protected abstract void doCollect() throws IOException;

/\*\*

\* Sub-classes may override this to add more segment completion logic.

\* @param lastSearchedDocID is the last docid searched before termination,

\* or NO\_MORE\_DOCS if there was no early termination. This doc may not be a hit!

\*/

protected abstract void doFinishSegment(int lastSearchedDocID) throws IOException;

/\*\*

\* sub classes can override this to perform more early termination checks.

\*/

public EarlyTerminationState innerShouldCollectMore() throws IOException {

return EarlyTerminationState.COLLECTING;

}

/\*\*

\* After early termination, this method can be used to retrieve early termination reason.

\*/

@Nonnull

public final EarlyTerminationState getEarlyTerminationState() {

return terminationTracker.getEarlyTerminationState();

}

protected final EarlyTerminationState setEarlyTerminationState(

EarlyTerminationState newEarlyTerminationState) {

terminationTracker.setEarlyTerminationState(newEarlyTerminationState);

return newEarlyTerminationState;

}

@Override

public final boolean isTerminated() throws IOException {

EarlyTerminationState earlyTerminationState = getEarlyTerminationState();

if (earlyTerminationState.isTerminated()) {

return true;

}

if (getNumHitsProcessed() >= getMaxHitsToProcess()) {

collectedEnoughResults();

if (shouldTerminate()) {

return setEarlyTerminationState(EarlyTerminationState.TERMINATED\_MAX\_HITS\_EXCEEDED)

.isTerminated();

} else {

return false;

}

}

return innerShouldCollectMore().isTerminated();

}

/\*\*

\* Note: subclasses overriding this method are expected to call "super.setNextReader"

\* in their setNextReader().

\* @deprecated Remove this methods in favor of {@link #getLeafCollector(LeafReaderContext)}

\*/

@Deprecated

public void setNextReader(LeafReaderContext context) throws IOException {

if (!terminationTracker.useLastSearchedDocIdOnTimeout()) {

expensiveEarlyTerminationCheck();

}

// Reset curDocId for next segment

curDocId = -1;

lastEarlyTerminationCheckDocId = -1;

curReader = context.reader();

}

/\*\*

\* Sub-classes overriding this method are expected to call super.setScorer()

\*/

@Override

public void setScorer(Scorable scorer) throws IOException {

this.scorer = scorer;

}

@Override

public final void collect(int doc) throws IOException {

curDocId = doc;

doCollect();

numHitsProcessed++;

if (numDocsBetweenTimeoutChecks > 0

&& (curDocId - lastEarlyTerminationCheckDocId) >= numDocsBetweenTimeoutChecks) {

lastEarlyTerminationCheckDocId = curDocId;

if (!terminationTracker.useLastSearchedDocIdOnTimeout()) {

expensiveEarlyTerminationCheck();

}

}

}

/\*\*

\* Accounting for a segment searched.

\* @param lastSearchedDocID is the last docid searched before termination,

\* or NO\_MORE\_DOCS if there was no early termination. This doc may not be a hit!

\*/

protected final void trackCompleteSegment(int lastSearchedDocID) throws IOException {

doFinishSegment(lastSearchedDocID);

}

@Override

public final void finishSegment(int lastSearchedDocID) throws IOException {

// finished searching a segment. Computer average time needed to search a segment.

Preconditions.checkState(curReader != null, "Did subclass call super.setNextReader()?");

numSearchedSegments++;

long totalTime = clock.nowMillis() - terminationTracker.getLocalStartTimeMillis();

if (totalTime >= Integer.MAX\_VALUE) {

String msg = String.format(

"%s: A query runs for %d that is longer than Integer.MAX\_VALUE ms. lastSearchedDocID: %d",

getClass().getSimpleName(), totalTime, lastSearchedDocID

);

LOG.error(msg);

throw new IllegalStateException(msg);

}

int timePerSegment = ((int) totalTime) / numSearchedSegments;

if (timePerSegment < 0) {

NEGATIVE\_TIME\_PER\_SEGMENT.increment();

timePerSegment = 0;

}

// If we're enforcing timeout via the last searched doc ID, we don't need to add this buffer,

// since we'll detect the timeout right away.

if (!terminationTracker.useLastSearchedDocIdOnTimeout()) {

terminationTracker.setPreTerminationSafeBufferTimeMillis(timePerSegment);

}

// Check whether we timed out and are checking for timeout at the leaves. If so, we should use

// the captured lastSearchedDocId from the tracker instead, which is the most up-to-date amongst

// the query nodes.

if (terminationTracker.useLastSearchedDocIdOnTimeout()

&& EarlyTerminationState.TERMINATED\_TIME\_OUT\_EXCEEDED.equals(

terminationTracker.getEarlyTerminationState())) {

QUERY\_TIMEOUT\_ENFORCED.increment();

trackCompleteSegment(terminationTracker.getLastSearchedDocId());

} else {

trackCompleteSegment(lastSearchedDocID);

}

// We finished a segment, so clear out the DocIdTrackers. The next segment will register its

// own trackers, and we don't need to keep the trackers from the current segment.

terminationTracker.resetDocIdTrackers();

curDocId = -1;

curReader = null;

scorer = null;

}

/\*\*

\* More expensive Early Termination checks, which are not called every hit.

\* This sets EarlyTerminationState if it decides that early termination should kick in.

\* See: SEARCH-29723.

\*/

private void expensiveEarlyTerminationCheck() {

if (queryCostProvider != null) {

double totalQueryCost = queryCostProvider.getTotalCost();

double maxQueryCost = terminationTracker.getMaxQueryCost();

if (totalQueryCost >= maxQueryCost) {

setEarlyTerminationState(EarlyTerminationState.TERMINATED\_MAX\_QUERY\_COST\_EXCEEDED);

}

}

final long nowMillis = clock.nowMillis();

if (nowMillis >= terminationTracker.getTimeoutEndTimeWithReservation()) {

setEarlyTerminationState(EarlyTerminationState.TERMINATED\_TIME\_OUT\_EXCEEDED);

}

}

public long getMaxHitsToProcess() {

return maxHitsToProcess;

}

public final void setNumHitsProcessed(long numHitsProcessed) {

this.numHitsProcessed = numHitsProcessed;

}

protected final long getNumHitsProcessed() {

return numHitsProcessed;

}

protected final int getNumSearchedSegments() {

return numSearchedSegments;

}

protected final Clock getClock() {

return clock;

}

@VisibleForTesting

protected final TerminationTracker getTerminationTracker() {

return this.terminationTracker;

}

protected void collectedEnoughResults() throws IOException {

}

protected boolean shouldTerminate() {

return true;

}

/\*\*

\* Debug info collected during execution.

\*/

public abstract List<String> getDebugInfo();

}