package com.twitter.search.common.search;

import java.util.HashSet;

import java.util.Set;

import com.google.common.base.Preconditions;

import com.twitter.common.util.Clock;

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

/\*\*

\* Used for tracking termination criteria for earlybird queries.

\*

\* Currently this tracks the query time out and query cost, if they are set on the

\* {@link com.twitter.search.common.query.thriftjava.CollectorTerminationParams}.

\*/

public class TerminationTracker {

/\*\* Query start time provided by client. \*/

private final long clientStartTimeMillis;

/\*\* Timeout end times, calculated from {@link #clientStartTimeMillis}. \*/

private final long timeoutEndTimeMillis;

/\*\* Query start time recorded at earlybird server. \*/

private final long localStartTimeMillis;

/\*\* Tracking query cost \*/

private final double maxQueryCost;

// Sometimes, we want to early terminate before timeoutEndTimeMillis, to reserve time for

// work that needs to be done after early termination (E.g. merging results).

private final int postTerminationOverheadMillis;

// We don't check for early termination often enough. Some times requests timeout in between

// early termination checks. This buffer time is also substracted from deadline.

// To illustrate how this is used, let's use a simple example:

// If we spent 750ms searching 5 segments, a rough estimate is that we need 150ms to search

// one segment. If the timeout is set to 800ms, we should not starting searching the next segment.

// In this case, on can set preTerminationSafeBufferTimeMillis to 150ms, so that when early

// termination check computes the deadline, this buffer is also subtracted. See SEARCH-29723.

private int preTerminationSafeBufferTimeMillis = 0;

private EarlyTerminationState earlyTerminationState = EarlyTerminationState.COLLECTING;

// This flag determines whether the last searched doc ID trackers should be consulted when a

// timeout occurs.

private final boolean useLastSearchedDocIdOnTimeout;

private final Set<DocIdTracker> lastSearchedDocIdTrackers = new HashSet<>();

/\*\*

\* Creates a new termination tracker that will not specify a timeout or max query cost.

\* Can be used for queries that explicitly do not want to use a timeout. Meant to be used for

\* tests, and background queries running for the query cache.

\*/

public TerminationTracker(Clock clock) {

this.clientStartTimeMillis = clock.nowMillis();

this.localStartTimeMillis = clientStartTimeMillis;

this.timeoutEndTimeMillis = Long.MAX\_VALUE;

this.maxQueryCost = Double.MAX\_VALUE;

this.postTerminationOverheadMillis = 0;

this.useLastSearchedDocIdOnTimeout = false;

}

/\*\*

\* Convenient method overloading for

\* {@link #TerminationTracker(CollectorTerminationParams, long, Clock, int)}.

\*/

public TerminationTracker(

CollectorTerminationParams terminationParams, Clock clock,

int postTerminationOverheadMillis) {

this(terminationParams, clock.nowMillis(), clock, postTerminationOverheadMillis);

}

/\*\*

\* Convenient method overloading for

\* {@link #TerminationTracker(CollectorTerminationParams, long, Clock, int)}.

\*/

public TerminationTracker(

CollectorTerminationParams terminationParams, int postTerminationOverheadMillis) {

this(

terminationParams,

System.currentTimeMillis(),

Clock.SYSTEM\_CLOCK,

postTerminationOverheadMillis);

}

/\*\*

\* Creates a new TerminationTracker instance.

\*

\* @param terminationParams CollectorParams.CollectorTerminationParams carrying parameters

\* about early termination.

\* @param clientStartTimeMillis The query start time (in millis) specified by client. This is used

\* to calculate timeout end time, like {@link #timeoutEndTimeMillis}.

\* @param clock used to sample {@link #localStartTimeMillis}.

\* @param postTerminationOverheadMillis How much time should be reserved. E.g. if request time

\* out is 800ms, and this is set to 200ms, early termination

\* will kick in at 600ms mark.

\*/

public TerminationTracker(

CollectorTerminationParams terminationParams,

long clientStartTimeMillis,

Clock clock,

int postTerminationOverheadMillis) {

Preconditions.checkNotNull(terminationParams);

Preconditions.checkArgument(postTerminationOverheadMillis >= 0);

this.clientStartTimeMillis = clientStartTimeMillis;

this.localStartTimeMillis = clock.nowMillis();

if (terminationParams.isSetTimeoutMs()

&& terminationParams.getTimeoutMs() > 0) {

Preconditions.checkState(terminationParams.getTimeoutMs() >= postTerminationOverheadMillis);

this.timeoutEndTimeMillis = this.clientStartTimeMillis + terminationParams.getTimeoutMs();

} else {

// Effectively no timeout.

this.timeoutEndTimeMillis = Long.MAX\_VALUE;

}

// Tracking query cost

if (terminationParams.isSetMaxQueryCost()

&& terminationParams.getMaxQueryCost() > 0) {

maxQueryCost = terminationParams.getMaxQueryCost();

} else {

maxQueryCost = Double.MAX\_VALUE;

}

this.useLastSearchedDocIdOnTimeout = terminationParams.isEnforceQueryTimeout();

this.postTerminationOverheadMillis = postTerminationOverheadMillis;

}

/\*\*

\* Returns the reserve time to perform post termination work. Return the deadline timestamp

\* with postTerminationWorkEstimate subtracted.

\*/

public long getTimeoutEndTimeWithReservation() {

// Return huge value if time out is disabled.

if (timeoutEndTimeMillis == Long.MAX\_VALUE) {

return timeoutEndTimeMillis;

} else {

return timeoutEndTimeMillis

- postTerminationOverheadMillis

- preTerminationSafeBufferTimeMillis;

}

}

public void setPreTerminationSafeBufferTimeMillis(int preTerminationSafeBufferTimeMillis) {

Preconditions.checkArgument(preTerminationSafeBufferTimeMillis >= 0);

this.preTerminationSafeBufferTimeMillis = preTerminationSafeBufferTimeMillis;

}

public long getLocalStartTimeMillis() {

return localStartTimeMillis;

}

public long getClientStartTimeMillis() {

return clientStartTimeMillis;

}

public double getMaxQueryCost() {

return maxQueryCost;

}

public boolean isEarlyTerminated() {

return earlyTerminationState.isTerminated();

}

public EarlyTerminationState getEarlyTerminationState() {

return earlyTerminationState;

}

public void setEarlyTerminationState(EarlyTerminationState earlyTerminationState) {

this.earlyTerminationState = earlyTerminationState;

}

/\*\*

\* Return the minimum searched doc ID amongst all registered trackers, or -1 if there aren't any

\* trackers. Doc IDs are stored in ascending order, and trackers update their doc IDs as they

\* search, so the minimum doc ID reflects the most recent fully searched doc ID.

\*/

int getLastSearchedDocId() {

return lastSearchedDocIdTrackers.stream()

.mapToInt(DocIdTracker::getCurrentDocId).min().orElse(-1);

}

void resetDocIdTrackers() {

lastSearchedDocIdTrackers.clear();

}

/\*\*

\* Add a DocIdTracker, to keep track of the last fully-searched doc ID when early termination

\* occurs.

\*/

public void addDocIdTracker(DocIdTracker docIdTracker) {

lastSearchedDocIdTrackers.add(docIdTracker);

}

public boolean useLastSearchedDocIdOnTimeout() {

return useLastSearchedDocIdOnTimeout;

}

}