package com.twitter.search.earlybird.querycache;

import java.io.IOException;

import java.util.concurrent.TimeUnit;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.cache.CacheBuilder;

import com.google.common.cache.CacheLoader;

import com.google.common.cache.LoadingCache;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.quantity.Amount;

import com.twitter.common.quantity.Time;

import com.twitter.common.util.Clock;

import com.twitter.decider.Decider;

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

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

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

import com.twitter.search.common.search.TerminationTracker;

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

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

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

import com.twitter.search.earlybird.exception.CriticalExceptionHandler;

import com.twitter.search.earlybird.exception.EarlybirdException;

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

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

import com.twitter.search.earlybird.partition.SegmentInfo;

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

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

import com.twitter.search.earlybird.util.ScheduledExecutorTask;

/\*\*

\* Each task is responsible for one filter on one segment. We should have a total

\* of num\_of\_filter \* num\_of\_segments tasks

\*/

@VisibleForTesting

class QueryCacheUpdateTask extends ScheduledExecutorTask {

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

// See OBSERVE-10347

private static final boolean EXPORT\_STATS =

EarlybirdConfig.getBool("export\_query\_cache\_update\_task\_stats", false);

private static final LoadingCache<String, TaskStats> TASK\_STATS =

CacheBuilder.newBuilder().build(new CacheLoader<String, TaskStats>() {

@Override

public TaskStats load(String statNamePrefix) {

return new TaskStats(statNamePrefix, EXPORT\_STATS);

}

});

private static final SearchCounter FINISHED\_TASKS = SearchCounter.export(

"querycache\_finished\_tasks");

private final QueryCacheFilter filter;

// Info/data of the segment this task is responsible for

private final SegmentInfo segmentInfo;

private final UserTable userTable;

private volatile boolean ranOnce;

private final TaskStats stats;

private Amount<Long, Time> lastRunFinishTime;

// See SEARCH-4346

private final String filterAndSegment;

private final Decider decider;

private static final class TaskStats {

private final SearchLongGauge numHitsStat;

private final SearchLongGauge updateLatencyStat;

private final SearchCounter updateSuccessCountStat;

private final SearchCounter updateFailureCountStat;

private TaskStats(String statNamePrefix, boolean exportStats) {

// See SEARCH-3698

numHitsStat = exportStats ? SearchLongGauge.export(statNamePrefix + "numhit")

: new SearchLongGauge(statNamePrefix + "numhit");

updateLatencyStat = exportStats

? SearchLongGauge.export(statNamePrefix + "update\_latency\_ms")

: new SearchLongGauge(statNamePrefix + "update\_latency\_ms");

updateSuccessCountStat = exportStats

? SearchCounter.export(statNamePrefix + "update\_success\_count")

: SearchCounter.create(statNamePrefix + "update\_success\_count");

updateFailureCountStat = exportStats

? SearchCounter.export(statNamePrefix + "update\_failure\_count")

: SearchCounter.create(statNamePrefix + "update\_failure\_count");

}

}

private final Amount<Long, Time> updateInterval;

private final Amount<Long, Time> initialDelay;

private final EarlybirdSearcherStats searcherStats;

private final CriticalExceptionHandler criticalExceptionHandler;

/\*\*

\* Constructor

\* @param filter Filter to be used to populate the cache

\* @param segmentInfo Segment this task is responsible for

\* @param updateInterval Time between successive updates

\* @param initialDelay Time before the first update

\* @param updateIterationCounter

\* @param decider

\*/

public QueryCacheUpdateTask(QueryCacheFilter filter,

SegmentInfo segmentInfo,

UserTable userTable,

Amount<Long, Time> updateInterval,

Amount<Long, Time> initialDelay,

SearchCounter updateIterationCounter,

EarlybirdSearcherStats searcherStats,

Decider decider,

CriticalExceptionHandler criticalExceptionHandler,

Clock clock) {

super(updateIterationCounter, clock);

this.filter = filter;

this.segmentInfo = segmentInfo;

this.userTable = userTable;

this.ranOnce = false;

this.updateInterval = updateInterval;

this.initialDelay = initialDelay;

this.stats = setupStats();

this.filterAndSegment = String.format(

"QueryCacheFilter: %s | Segment: %d",

filter.getFilterName(), segmentInfo.getTimeSliceID());

this.searcherStats = searcherStats;

this.criticalExceptionHandler = criticalExceptionHandler;

this.decider = decider;

}

@Override

protected void runOneIteration() {

try {

if (LOG.isDebugEnabled()) {

LOG.debug(

"[{}] Updating with query [{}] for the {} th time.",

filterAndSegment,

filter.getQueryString(),

stats.updateSuccessCountStat.get() + stats.updateFailureCountStat.get() + 1

);

if (lastRunFinishTime != null) {

LOG.debug(

"[{}] Last run, {} th time, finished {} secs ago. Should run every {} secs",

filterAndSegment,

stats.updateSuccessCountStat.get() + stats.updateFailureCountStat.get(),

TimeUnit.NANOSECONDS.toSeconds(

System.nanoTime() - lastRunFinishTime.as(Time.NANOSECONDS)),

updateInterval.as(Time.SECONDS)

);

}

}

Timer timer = new Timer(TimeUnit.MILLISECONDS);

SearchResultsInfo result = null;

try {

result = update();

} catch (Exception e) {

String msg = "Failed to update query cache entry [" + filter.getFilterName()

+ "] on segment [" + segmentInfo.getTimeSliceID() + "]";

LOG.warn(msg, e);

}

long endTime = timer.stop();

updateStats(result, endTime);

if (LOG.isDebugEnabled()) {

LOG.debug("[{}] Updated in {} ms, hit {} docs.",

filterAndSegment, endTime, stats.numHitsStat.read());

}

// Need to catch throwable here instead of exception so we handle errors like OutOfMemory

// See RB=528695 and SEARCH-4402

} catch (Throwable t) {

String message = String.format("Got unexpected throwable in %s", getClass().getName());

LOG.error(message, t);

// Wrap the Throwable in a FatalEarlybirdException to categorize it and ensure it's

// handled as a fatal exception

criticalExceptionHandler.handle(this,

new EarlybirdException(message, t));

} finally {

// Earlybird won't become CURRENT until all tasks are run at least once. We don't want

// failed "run" (update) to prevent Earlybird from becoming CURRENT. As long as all tasks

// got a chance to run at least once, we are good to go.

ranOnce = true;

lastRunFinishTime = Amount.of(System.nanoTime(), Time.NANOSECONDS);

}

}

public boolean ranOnce() {

return ranOnce;

}

private TaskStats setupStats() {

return TASK\_STATS.getUnchecked(statNamePrefix());

}

private SearchResultsInfo update() throws IOException {

// There's a chance that the EarlybirdSegment of a SegmentInfo to change at any

// time. Therefore, it's not safe to operate segments on the SegmentInfo level.

// On the archive clusters we create a new EarlybirdSegment and then swap it in when there's

// new data instead of appending to an existing EarlybirdSegment.

EarlybirdSegment earlybirdSegment = segmentInfo.getIndexSegment();

EarlybirdSingleSegmentSearcher searcher = earlybirdSegment.getSearcher(userTable);

if (searcher == null) {

LOG.warn("Unable to get searcher from TwitterIndexManager for segment ["

+ segmentInfo.getTimeSliceID() + "]. Has it been dropped?");

return null;

}

QueryCacheResultCollector collector = new QueryCacheResultCollector(

searcher.getSchemaSnapshot(), filter, searcherStats, decider, clock, 0);

searcher.search(filter.getLuceneQuery(), collector);

QueryCacheResultForSegment cacheResult = collector.getCachedResult();

searcher.getTwitterIndexReader().getSegmentData().updateQueryCacheResult(

filter.getFilterName(), cacheResult);

FINISHED\_TASKS.increment();

if (LOG.isDebugEnabled()) {

TerminationTracker tracker = collector.getSearchRequestInfo().getTerminationTracker();

LOG.debug(

"[{}] Updating query finished, start time ms is {}, termination reason is {}",

filterAndSegment,

tracker.getLocalStartTimeMillis(),

tracker.getEarlyTerminationState().getTerminationReason());

}

return collector.getResults();

}

private void updateStats(SearchResultsInfo result, long endTime) {

if (result != null) {

stats.numHitsStat.set(result.getNumHitsProcessed());

stats.updateSuccessCountStat.increment();

} else {

stats.updateFailureCountStat.increment();

}

stats.updateLatencyStat.set(endTime);

}

@VisibleForTesting

String statNamePrefix() {

// If we use this and try to display in monviz "ts(partition, single\_instance, querycache\*)",

// the UI shows "Really expensive query" message. We can keep this around for times when we

// want to start things manually and debug.

return "querycache\_" + filter.getFilterName() + "\_" + segmentInfo.getTimeSliceID() + "\_";

}

public long getTimeSliceID() {

return segmentInfo.getTimeSliceID();

}

//////////////////////////

// for unit tests only

//////////////////////////

@VisibleForTesting

String getFilterNameForTest() {

return filter.getFilterName();

}

@VisibleForTesting

Amount<Long, Time> getUpdateIntervalForTest() {

return updateInterval;

}

@VisibleForTesting

Amount<Long, Time> getInitialDelayForTest() {

return initialDelay;

}

@VisibleForTesting

TaskStats getTaskStatsForTest() {

return stats;

}

}