package com.twitter.search.earlybird.querycache;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

import java.util.List;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.ScheduledFuture;

import java.util.concurrent.TimeUnit;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.Lists;

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.concurrent.ScheduledExecutorServiceFactory;

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

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

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

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

import com.twitter.search.earlybird.factory.QueryCacheUpdaterScheduledExecutorService;

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

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

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

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

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

/\*\*

\* Class to manage the scheduler service and all the update tasks. Through this

\* class, update tasks are created and scheduled, canceled and removed.

\*

\* This class is not thread-safe.

\*/

@VisibleForTesting

final class QueryCacheUpdater extends ScheduledExecutorManager {

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

private final List<Task> tasks;

private final EarlybirdSearcherStats searcherStats;

private final Decider decider;

private final UserTable userTable;

private final Clock clock;

@VisibleForTesting

static final class Task {

@VisibleForTesting public final QueryCacheUpdateTask updateTask;

@VisibleForTesting public final ScheduledFuture future;

private Task(QueryCacheUpdateTask updateTask, ScheduledFuture future) {

this.updateTask = updateTask;

this.future = future;

}

}

public QueryCacheUpdater(Collection<QueryCacheFilter> cacheFilters,

ScheduledExecutorServiceFactory updaterScheduledExecutorServiceFactory,

UserTable userTable,

SearchStatsReceiver searchStatsReceiver,

EarlybirdSearcherStats searcherStats,

Decider decider,

CriticalExceptionHandler criticalExceptionHandler,

Clock clock) {

super(updaterScheduledExecutorServiceFactory.build("QueryCacheUpdateThread-%d", true),

ShutdownWaitTimeParams.immediately(), searchStatsReceiver,

criticalExceptionHandler, clock);

Preconditions.checkNotNull(cacheFilters);

Preconditions.checkArgument(getExecutor() instanceof QueryCacheUpdaterScheduledExecutorService,

getExecutor().getClass());

this.searcherStats = searcherStats;

this.decider = decider;

this.userTable = userTable;

this.clock = clock;

shouldLog = false;

// One update task per <query, segment>

tasks = Lists.newArrayListWithCapacity(cacheFilters.size() \* 20);

SearchCustomGauge.export(

"querycache\_num\_tasks",

tasks::size

);

}

/\*\*

\* Create an update task and add it to the executor

\*

\* @param filter The filter the task should execute

\* @param segmentInfo The segment that this task would be responsible for

\* @param updateInterval time in milliseconds between successive updates

\* @param initialDelay Introduce a delay when adding the task to the executor

\*/

void addTask(QueryCacheFilter filter, SegmentInfo segmentInfo,

Amount<Long, Time> updateInterval, Amount<Long, Time> initialDelay) {

String filterName = filter.getFilterName();

String query = filter.getQueryString();

// Create the task.

QueryCacheUpdateTask qcTask = new QueryCacheUpdateTask(

filter,

segmentInfo,

userTable,

updateInterval,

initialDelay,

getIterationCounter(),

searcherStats,

decider,

criticalExceptionHandler,

clock);

long initialDelayAsMS = initialDelay.as(Time.MILLISECONDS);

long updateIntervalAsMS = updateInterval.as(Time.MILLISECONDS);

Preconditions.checkArgument(

initialDelayAsMS >= initialDelay.getValue(), "initial delay unit granularity too small");

Preconditions.checkArgument(

updateIntervalAsMS >= updateInterval.getValue(),

"update interval unit granularity too small");

// Schedule the task.

ScheduledFuture future = scheduleNewTask(qcTask,

PeriodicActionParams.withIntialWaitAndFixedDelay(

initialDelayAsMS, updateIntervalAsMS, TimeUnit.MILLISECONDS

)

);

tasks.add(new Task(qcTask, future));

LOG.debug("Added a task for filter [" + filterName

+ "] for segment [" + segmentInfo.getTimeSliceID()

+ "] with query [" + query

+ "] update interval " + updateInterval + " "

+ (initialDelay.getValue() == 0 ? "without" : "with " + initialDelay)

+ " initial delay");

}

void removeAllTasksForSegment(SegmentInfo segmentInfo) {

int removedTasksCount = 0;

for (Iterator<Task> it = tasks.iterator(); it.hasNext();) {

Task task = it.next();

if (task.updateTask.getTimeSliceID() == segmentInfo.getTimeSliceID()) {

task.future.cancel(true);

it.remove();

removedTasksCount += 1;

}

}

LOG.info("Removed {} update tasks for segment {}.", removedTasksCount,

segmentInfo.getTimeSliceID());

}

public void clearTasks() {

int totalTasks = tasks.size();

LOG.info("Removing {} update tasks for all segments.", totalTasks);

for (Task task : tasks) {

task.future.cancel(true);

}

tasks.clear();

LOG.info("Canceled {} QueryCache update tasks", totalTasks);

}

// Have all tasks run at least once (even if they failed)?

public boolean allTasksRan() {

boolean allTasksRan = true;

for (Task task : tasks) {

if (!task.updateTask.ranOnce()) {

allTasksRan = false;

break;

}

}

return allTasksRan;

}

// Have all tasks for this run at least once (even if they failed)?

public boolean allTasksRanForSegment(SegmentInfo segmentInfo) {

boolean allTasksRanForSegment = true;

for (Task task : tasks) {

if ((task.updateTask.getTimeSliceID() == segmentInfo.getTimeSliceID())

&& !task.updateTask.ranOnce()) {

allTasksRanForSegment = false;

break;

}

}

return allTasksRanForSegment;

}

/\*\*

\* After startup, we want only one thread to update the query cache.

\*/

void setWorkerPoolSizeAfterStartup() {

QueryCacheUpdaterScheduledExecutorService executor =

(QueryCacheUpdaterScheduledExecutorService) getExecutor();

executor.setWorkerPoolSizeAfterStartup();

LOG.info("Done setting executor core pool size to one");

}

@Override

protected void shutdownComponent() {

clearTasks();

}

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

// for unit tests only

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

/\*\*

\* Returns the list of all query cache updater tasks. This method should be used only in tests.

\*/

@VisibleForTesting

List<Task> getTasksForTest() {

synchronized (tasks) {

return new ArrayList<>(tasks);

}

}

@VisibleForTesting

int getTasksSize() {

synchronized (tasks) {

return tasks.size();

}

}

@VisibleForTesting

boolean tasksContains(Task task) {

synchronized (tasks) {

return tasks.contains(task);

}

}

@VisibleForTesting

public ScheduledExecutorService getExecutorForTest() {

return getExecutor();

}

}