package com.twitter.search.earlybird;

import java.io.BufferedWriter;

import java.io.Closeable;

import java.io.File;

import java.io.IOException;

import java.nio.file.Files;

import java.util.ArrayList;

import java.util.List;

import java.util.Set;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.RejectedExecutionException;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicReference;

import javax.annotation.Nullable;

import javax.annotation.concurrent.GuardedBy;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Charsets;

import com.google.common.base.Stopwatch;

import com.google.common.cache.CacheBuilder;

import com.google.common.cache.CacheLoader;

import com.google.common.cache.LoadingCache;

import com.google.common.collect.ImmutableMap;

import com.google.common.collect.Lists;

import com.google.common.util.concurrent.AtomicLongMap;

import org.apache.commons.codec.binary.Base64;

import org.apache.lucene.search.IndexSearcher;

import org.apache.thrift.TBase;

import org.apache.thrift.TException;

import org.apache.thrift.TSerializer;

import org.apache.zookeeper.KeeperException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.collections.Pair;

import com.twitter.common.util.Clock;

import com.twitter.common.zookeeper.ServerSet.UpdateException;

import com.twitter.common.zookeeper.ZooKeeperClient;

import com.twitter.decider.Decider;

import com.twitter.finagle.Failure;

import com.twitter.search.common.database.DatabaseConfig;

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

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

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

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

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

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

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

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

import com.twitter.search.common.schema.DynamicSchema;

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

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

import com.twitter.search.common.search.termination.QueryTimeoutFactory;

import com.twitter.search.common.util.FinagleUtil;

import com.twitter.search.common.util.GCUtil;

import com.twitter.search.common.util.ml.tensorflow\_engine.TensorflowModelsManager;

import com.twitter.search.common.util.zookeeper.ZooKeeperProxy;

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

import com.twitter.search.earlybird.admin.LastSearchesSummary;

import com.twitter.search.earlybird.admin.QueriedFieldsAndSchemaStats;

import com.twitter.search.earlybird.common.ClientIdUtil;

import com.twitter.search.earlybird.common.EarlybirdRequestLogger;

import com.twitter.search.earlybird.common.EarlybirdRequestPostLogger;

import com.twitter.search.earlybird.common.EarlybirdRequestPreLogger;

import com.twitter.search.earlybird.common.EarlybirdRequestUtil;

import com.twitter.search.earlybird.common.RequestResponsePair;

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

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

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

import com.twitter.search.earlybird.ml.ScoringModelsManager;

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

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

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

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

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

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

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

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

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

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

import com.twitter.search.earlybird.querycache.QueryCacheManager;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

import com.twitter.snowflake.id.SnowflakeId;

import com.twitter.util.Duration;

import com.twitter.util.Function;

import com.twitter.util.Function0;

import com.twitter.util.Future;

public class EarlybirdServer implements EarlybirdService.ServiceIface, ServerSetMember {

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

private static final String EARLYBIRD\_STARTUP = "earlybird startup";

public static final String SERVICE\_NAME = "Earlybird";

private static final boolean REGISTER\_WITH\_ZK\_ON\_STARTUP =

EarlybirdConfig.getBool("register\_with\_zk\_on\_startup", true);

private static final Duration SERVER\_CLOSE\_WAIT\_TIME = Duration.apply(5L, TimeUnit.SECONDS);

private static final Failure QUEUE\_FULL\_FAILURE =

Failure.rejected("Rejected due to full executor queue");

private final int port = EarlybirdConfig.getThriftPort();

private final int warmUpPort = EarlybirdConfig.getWarmUpThriftPort();

private final int numSearcherThreads = EarlybirdConfig.getSearcherThreads();

private final SearchStatsReceiver earlybirdServerStatsReceiver;

private final EarlybirdRPCStats searchStats = new EarlybirdRPCStats("search");

private final EarlybirdSearcherStats tweetsSearcherStats;

private static final String REQUESTS\_RECEIVED\_BY\_FINAGLE\_ID\_COUNTER\_NAME\_PATTERN =

"requests\_for\_finagle\_id\_%s\_all";

private static final String REQUESTS\_RECEIVED\_BY\_FINAGLE\_ID\_AND\_CLIENT\_ID\_COUNTER\_NAME\_PATTERN =

"requests\_for\_finagle\_id\_%s\_and\_client\_id\_%s";

private static final String RESPONSES\_PER\_CLIENT\_ID\_STAT\_TEMPLATE =

"responses\_for\_client\_id\_%s\_with\_response\_code\_%s";

// Loading cache for per finagle-client-id stats. Storing them in a loading cache key-ed by

// finagle client id so we don't export the stat multiple times.

private final LoadingCache<String, SearchTimerStats> requestCountersByFinagleClientId =

CacheBuilder.newBuilder().build(

new CacheLoader<String, SearchTimerStats>() {

@Override

public SearchTimerStats load(String finagleClientId) {

return earlybirdServerStatsReceiver.getTimerStats(

String.format(

REQUESTS\_RECEIVED\_BY\_FINAGLE\_ID\_COUNTER\_NAME\_PATTERN,

finagleClientId), TimeUnit.MICROSECONDS, false, true, false);

}

});

// Counters per client and response code.

private final LoadingCache<String, SearchCounter> responseByClientIdAndResponseCode =

CacheBuilder.newBuilder().build(

new CacheLoader<String, SearchCounter>() {

@Override

public SearchCounter load(String key) {

return earlybirdServerStatsReceiver.getCounter(key);

}

});

private final LoadingCache<String, SearchCounter> resultsAgeCounter =

CacheBuilder.newBuilder().build(

new CacheLoader<String, SearchCounter>() {

@Override

public SearchCounter load(String key) {

return earlybirdServerStatsReceiver.getCounter(key);

}

}

);

// Loading cache for per finagle client id and client id stats. These are stored separate

// from the other stats because they are key-ed by the pair of finagle client id and client id

// in order to make sure the stats are only exported once.

// In the key-pair the first element is the finagle client id while the second element is the

// client id.

private final LoadingCache<Pair<String, String>, SearchRateCounter>

requestCountersByFinagleIdAndClientId = CacheBuilder.newBuilder().build(

new CacheLoader<Pair<String, String>, SearchRateCounter>() {

@Override

public SearchRateCounter load(Pair<String, String> clientKey) {

return earlybirdServerStatsReceiver.getRateCounter(

String.format(

REQUESTS\_RECEIVED\_BY\_FINAGLE\_ID\_AND\_CLIENT\_ID\_COUNTER\_NAME\_PATTERN,

clientKey.getFirst(),

clientKey.getSecond()));

}

});

// Loading cache for per-client-id latency stats. Stored in a loading cache here mainly because

// the tests assert the mock stats receiver that each stat is only exported once.

private final LoadingCache<String, SearchTimerStats> clientIdSearchStats =

CacheBuilder.newBuilder().build(

new CacheLoader<String, SearchTimerStats>() {

@Override

public SearchTimerStats load(String clientId) {

String formattedClientId = ClientIdUtil.formatClientId(clientId);

return earlybirdServerStatsReceiver.getTimerStats(formattedClientId,

TimeUnit.MICROSECONDS, false, true, true);

}

});

private final LoadingCache<String, SearchTimerStats> clientIdScoringPerQueryStats =

CacheBuilder.newBuilder().build(

new CacheLoader<String, SearchTimerStats>() {

@Override

public SearchTimerStats load(String clientId) {

String statName =

String.format("scoring\_time\_per\_query\_for\_client\_id\_%s", clientId);

return earlybirdServerStatsReceiver.getTimerStats(statName,

TimeUnit.NANOSECONDS, false, true, false);

}

});

private final LoadingCache<String, SearchTimerStats> clientIdScoringPerHitStats =

CacheBuilder.newBuilder().build(

new CacheLoader<String, SearchTimerStats>() {

@Override

public SearchTimerStats load(String clientId) {

String statName =

String.format("scoring\_time\_per\_hit\_for\_client\_id\_%s", clientId);

return earlybirdServerStatsReceiver.getTimerStats(statName,

TimeUnit.NANOSECONDS, false, true, false);

}

});

private final LoadingCache<String, Percentile<Integer>> clientIdScoringNumHitsProcessedStats =

CacheBuilder.newBuilder().build(

new CacheLoader<String, Percentile<Integer>>() {

@Override

public Percentile<Integer> load(String clientId) {

String statName =

String.format("scoring\_num\_hits\_processed\_for\_client\_id\_%s", clientId);

return PercentileUtil.createPercentile(statName);

}

});

private final LoadingCache<String, AtomicReference<RequestResponsePair>> lastRequestPerClientId =

CacheBuilder.newBuilder().build(

new CacheLoader<String, AtomicReference<RequestResponsePair>>() {

@Override

public AtomicReference<RequestResponsePair> load(String key) throws Exception {

return new AtomicReference<>(null);

}

});

private final SearchTimerStats overallScoringTimePerQueryStats;

private final SearchTimerStats overallScoringTimePerHitStats;

private final Percentile<Integer> overallScoringNumHitsProcessedStats;

private final EarlybirdIndexConfig earlybirdIndexConfig;

private final DynamicPartitionConfig dynamicPartitionConfig;

private final SegmentManager segmentManager;

private final UpdateableEarlybirdStateManager stateManager;

private final AudioSpaceTable audioSpaceTable;

private final SearchLongGauge startupTimeGauge;

// Time spent in an internal thread pool queue, between the time we get the search request

// from finagle until it actually starts being executed.

private final SearchTimerStats internalQueueWaitTimeStats;

// Tracking request that have exceeded their allocated timeout prior to us actually being able

// to start executing the search.

private final SearchCounter requestTimeoutExceededBeforeSearchCounter;

// Current number of running searcher threads.

private final SearchLongGauge numSearcherThreadsGauge;

private final QueryTimeoutFactory queryTimeoutFactory;

private PartitionManager partitionManager;

private QueryCacheManager queryCacheManager;

private final ScoringModelsManager scoringModelsManager;

private final TensorflowModelsManager tensorflowModelsManager;

private final EarlybirdRequestPreLogger requestPreLogger;

private final EarlybirdRequestPostLogger requestLogger;

private final TweetCountMonitor tweetCountMonitor;

private final TermCountMonitor termCountMonitor;

private final EarlybirdServerSetManager serverSetManager;

private final EarlybirdWarmUpManager warmUpManager;

private final MultiSegmentTermDictionaryManager multiSegmentTermDictionaryManager;

private final Object shutdownLock = new Object();

@GuardedBy("shutdownLock")

private final EarlybirdFuturePoolManager futurePoolManager;

@GuardedBy("shutdownLock")

private final EarlybirdFinagleServerManager finagleServerManager;

// If a search request comes in with a client-side start time, and we see that based on that

// the timeout has expired, whether we should drop that query immediately.

private final boolean skipTimedOutRequests =

EarlybirdConfig.getBool("skip\_timedout\_requests", false);

// client of szookeeper.local.twitter.com.

// This is used to perform distributed locking and layout reading etc.

private final ZooKeeperProxy sZooKeeperClient;

private final Decider decider;

private final Clock clock;

private final List<Closeable> toClose = new ArrayList<>();

private final SearchIndexingMetricSet searchIndexingMetricSet;

private final EarlybirdDarkProxy earlybirdDarkProxy;

private final ImmutableMap<EarlybirdResponseCode, SearchCounter> responseCodeCounters;

private final SegmentSyncConfig segmentSyncConfig;

private final EarlybirdStartup earlybirdStartup;

private final QualityFactor qualityFactor;

private boolean isShutdown = false;

private boolean isShuttingDown = false;

private final AtomicLongMap<String> queriedFieldsCounts = AtomicLongMap.create();

public EarlybirdServer(QueryCacheManager queryCacheManager,

ZooKeeperProxy sZkClient,

Decider decider,

EarlybirdIndexConfig earlybirdIndexConfig,

DynamicPartitionConfig dynamicPartitionConfig,

PartitionManager partitionManager,

SegmentManager segmentManager,

AudioSpaceTable audioSpaceTable,

TermCountMonitor termCountMonitor,

TweetCountMonitor tweetCountMonitor,

UpdateableEarlybirdStateManager earlybirdStateManager,

EarlybirdFuturePoolManager futurePoolManager,

EarlybirdFinagleServerManager finagleServerManager,

EarlybirdServerSetManager serverSetManager,

EarlybirdWarmUpManager warmUpManager,

SearchStatsReceiver earlybirdServerStatsReceiver,

EarlybirdSearcherStats tweetsSearcherStats,

ScoringModelsManager scoringModelsManager,

TensorflowModelsManager tensorflowModelsManager,

Clock clock,

MultiSegmentTermDictionaryManager multiSegmentTermDictionaryManager,

EarlybirdDarkProxy earlybirdDarkProxy,

SegmentSyncConfig segmentSyncConfig,

QueryTimeoutFactory queryTimeoutFactory,

EarlybirdStartup earlybirdStartup,

QualityFactor qualityFactor,

SearchIndexingMetricSet searchIndexingMetricSet) {

LOG.info("Creating EarlybirdServer");

this.decider = decider;

this.clock = clock;

this.sZooKeeperClient = sZkClient;

this.earlybirdIndexConfig = earlybirdIndexConfig;

this.dynamicPartitionConfig = dynamicPartitionConfig;

this.segmentManager = segmentManager;

this.queryCacheManager = queryCacheManager;

this.termCountMonitor = termCountMonitor;

this.tweetCountMonitor = tweetCountMonitor;

this.stateManager = earlybirdStateManager;

this.partitionManager = partitionManager;

this.futurePoolManager = futurePoolManager;

this.finagleServerManager = finagleServerManager;

this.serverSetManager = serverSetManager;

this.warmUpManager = warmUpManager;

this.earlybirdServerStatsReceiver = earlybirdServerStatsReceiver;

this.tweetsSearcherStats = tweetsSearcherStats;

this.scoringModelsManager = scoringModelsManager;

this.tensorflowModelsManager = tensorflowModelsManager;

this.multiSegmentTermDictionaryManager = multiSegmentTermDictionaryManager;

this.searchIndexingMetricSet = searchIndexingMetricSet;

this.earlybirdDarkProxy = earlybirdDarkProxy;

this.segmentSyncConfig = segmentSyncConfig;

this.queryTimeoutFactory = queryTimeoutFactory;

this.earlybirdStartup = earlybirdStartup;

this.qualityFactor = qualityFactor;

this.audioSpaceTable = audioSpaceTable;

EarlybirdStatus.setStartTime(System.currentTimeMillis());

// Our initial status code is STARTING.

EarlybirdStatus.setStatus(EarlybirdStatusCode.STARTING);

EarlybirdStatus.THRIFT\_SERVICE\_STARTED.set(false);

PartitionConfig partitionConfig = dynamicPartitionConfig.getCurrentPartitionConfig();

earlybirdServerStatsReceiver.getLongGauge(

"search\_cluster\_" + partitionConfig.getClusterName()).set(1);

earlybirdServerStatsReceiver.getLongGauge(

"tier\_name\_" + partitionConfig.getTierName()).set(1);

earlybirdServerStatsReceiver.getLongGauge("partition").set(

partitionConfig.getIndexingHashPartitionID());

earlybirdServerStatsReceiver.getLongGauge("replica").set(

partitionConfig.getHostPositionWithinHashPartition());

earlybirdServerStatsReceiver.getLongGauge("penguin\_version").set(

EarlybirdConfig.getPenguinVersionByte());

earlybirdServerStatsReceiver.getLongGauge("flush\_version").set(

FlushVersion.CURRENT\_FLUSH\_VERSION.ordinal());

String buildGen = EarlybirdConfig.getString("offline\_segment\_build\_gen", "unknown");

earlybirdServerStatsReceiver.getLongGauge("build\_gen\_" + buildGen).set(1);

this.startupTimeGauge = earlybirdServerStatsReceiver.getLongGauge("startup\_time\_millis");

this.internalQueueWaitTimeStats = earlybirdServerStatsReceiver.getTimerStats(

"internal\_queue\_wait\_time", TimeUnit.MILLISECONDS, false, true, false);

this.requestTimeoutExceededBeforeSearchCounter = earlybirdServerStatsReceiver.getCounter(

"request\_timeout\_exceeded\_before\_search");

this.numSearcherThreadsGauge =

earlybirdServerStatsReceiver.getLongGauge("num\_searcher\_threads");

this.overallScoringTimePerQueryStats = earlybirdServerStatsReceiver.getTimerStats(

"overall\_scoring\_time\_per\_query", TimeUnit.NANOSECONDS, false, true, false);

// For most of our scoring functions the scoring\_time\_per\_hit records the actual time to score a

// single hit. However, the tensorflow based scoring function uses batch scoring, so we do not

// know the actual time it takes to score a single hit. We are now including batch scoring time

// in all scoring time stats (SEARCH-26014), which means that the scoring\_time\_per\_hit stat may

// be a bit misleading for tensorflow based queries. For these queries the scoring\_time\_per\_hit

// represents the ratio between total\_scoring\_time and the number\_of\_hits, instead of the actual

// time to score a single hit.

this.overallScoringTimePerHitStats = earlybirdServerStatsReceiver.getTimerStats(

"overall\_scoring\_time\_per\_hit", TimeUnit.NANOSECONDS, false, true, false);

this.overallScoringNumHitsProcessedStats = PercentileUtil.createPercentile(

"overall\_scoring\_num\_hits\_processed");

ImmutableMap.Builder<EarlybirdResponseCode, SearchCounter> responseCodeCountersBuilder =

new ImmutableMap.Builder<>();

for (EarlybirdResponseCode responseCode : EarlybirdResponseCode.values()) {

responseCodeCountersBuilder.put(

responseCode,

earlybirdServerStatsReceiver.getCounter(

"responses\_with\_response\_code\_" + responseCode.name().toLowerCase()));

}

responseCodeCounters = responseCodeCountersBuilder.build();

disableLuceneQueryCache();

initManagers();

requestPreLogger = EarlybirdRequestPreLogger.buildForShard(

EarlybirdConfig.getInt("latency\_warn\_threshold", 100), decider);

requestLogger = EarlybirdRequestPostLogger.buildForShard(

EarlybirdConfig.getInt("latency\_warn\_threshold", 100), decider);

this.qualityFactor.startUpdates();

LOG.info("Created EarlybirdServer");

}

public boolean isShutdown() {

return this.isShutdown;

}

private void initManagers() {

LOG.info("Created EarlybirdIndexConfig: " + earlybirdIndexConfig.getClass().getSimpleName());

segmentManager.addUpdateListener(queryCacheManager);

}

public PartitionManager getPartitionManager() {

return partitionManager;

}

public QueryCacheManager getQueryCacheManager() {

return queryCacheManager;

}

public SegmentManager getSegmentManager() {

return segmentManager;

}

public MultiSegmentTermDictionaryManager getMultiSegmentTermDictionaryManager() {

return this.multiSegmentTermDictionaryManager;

}

@VisibleForTesting

public int getPort() {

return port;

}

private void disableLuceneQueryCache() {

// SEARCH-30046: Look into possibly re-enabling the query -> weight cache.

// We can't use this cache until we upgrade to Lucene 6.0.0, because we have queries with a

// boost of 0.0, and they don't play nicely with Lucene's LRUQueryCache.get() method.

//

// Lucene 6.0.0 changes how boosts are handled: "real" boosts should be wrapped into BoostQuery

// instances, and queries with a boost of 0.0 should be rewritten as "filters"

// (BooleanQuery.add(query, BooleanClause.Occur.FILTER)). So when we upgrade to Lucene 6.0.0 we

// will be forced to refactor how we handle our current queries with a boost of 0.0, which might

// allow us to re-enable this cache.

//

// Note that disabling this cache is not a regression: it should give us the behavior that we

// had with Lucene 5.2.1 (and it's unclear if this cache is useful at all).

//

// WARNING: The default 'DefaultQueryCache' maintains a static reference to the weight forever,

// causing a memory leak. Our weights hold references to an entire segment so the memory leak is

// significant.

IndexSearcher.setDefaultQueryCache(null);

}

/\*\*

\* Starts the earlybird server.

\*/

public void start() throws EarlybirdStartupException {

// Make sure this is at the top of the function before other parts of the system start running

new EarlybirdBlacklistHandler(Clock.SYSTEM\_CLOCK, sZooKeeperClient)

.blockThenExitIfBlacklisted();

Stopwatch startupWatch = Stopwatch.createStarted();

EarlybirdStatus.beginEvent(EARLYBIRD\_STARTUP, searchIndexingMetricSet.startupInProgress);

LOG.info("java.library.path is: " + System.getProperty("java.library.path"));

PartitionConfig partitionConfig = dynamicPartitionConfig.getCurrentPartitionConfig();

SegmentVulture.removeUnusedSegments(partitionManager, partitionConfig,

earlybirdIndexConfig.getSchema().getMajorVersionNumber(), segmentSyncConfig);

// Start the schema manager

schedule(stateManager);

Closeable closeable = earlybirdStartup.start();

toClose.add(closeable);

if (EarlybirdStatus.getStatusCode() == EarlybirdStatusCode.STOPPING) {

LOG.info("Server is shutdown. Exiting...");

return;

}

startupTimeGauge.set(startupWatch.elapsed(TimeUnit.MILLISECONDS));

EarlybirdStatus.endEvent(EARLYBIRD\_STARTUP, searchIndexingMetricSet.startupInProgress);

GCUtil.runGC(); // Attempt to force a full GC before joining the serverset

try {

startThriftService(null, true);

} catch (InterruptedException e) {

LOG.info("Interrupted while starting thrift server, quitting earlybird");

throw new EarlybirdStartupException("Interrupted while starting thrift server");

}

EarlybirdStatus.THRIFT\_SERVICE\_STARTED.set(true);

// only once we're current, kick off daily tweet count monitors only for archive cluster

if (EarlybirdConfig.getInt(TweetCountMonitor.RUN\_INTERVAL\_MINUTES\_CONFIG\_NAME, -1) > 0) {

schedule(tweetCountMonitor);

}

// only once we're current, kick off per-field term count monitors

if (EarlybirdConfig.getInt(TermCountMonitor.RUN\_INTERVAL\_MINUTES\_CONFIG\_NAME, -1) > 0) {

schedule(termCountMonitor);

}

startupTimeGauge.set(startupWatch.elapsed(TimeUnit.MILLISECONDS));

LOG.info("EarlybirdServer start up time: {}", startupWatch);

}

/\*\*

\* Starts the thrift server if the server is not running.

\* If searcherThreads is null, it uses the value specified by EarlybirdConfig.

\*/

public void startThriftService(@Nullable Integer searcherThreads, boolean isStartingUp)

throws InterruptedException {

synchronized (shutdownLock) {

if (!finagleServerManager.isWarmUpServerRunning()

&& !finagleServerManager.isProductionServerRunning()) {

int threadCount = searcherThreads != null

? searcherThreads : this.numSearcherThreads;

LOG.info("Starting searcher pool with " + threadCount + " threads");

futurePoolManager.createUnderlyingFuturePool(threadCount);

numSearcherThreadsGauge.set(threadCount);

// If the server is not shutting down, go through the warm up stage. If the server is

// instructed to shut down during warm up, warmUpManager.warmUp() should return within a

// second, and should leave the warm up server set. We should still shut down the warm up

// Finagle server.

if (isStartingUp && (EarlybirdStatus.getStatusCode() != EarlybirdStatusCode.STOPPING)) {

LOG.info("Opening warmup thrift port...");

finagleServerManager.startWarmUpFinagleServer(this, SERVICE\_NAME, warmUpPort);

EarlybirdStatus.WARMUP\_THRIFT\_PORT\_OPEN.set(true);

try {

warmUpManager.warmUp();

} catch (UpdateException e) {

LOG.warn("Could not join or leave the warm up server set.", e);

} finally {

finagleServerManager.stopWarmUpFinagleServer(SERVER\_CLOSE\_WAIT\_TIME);

EarlybirdStatus.WARMUP\_THRIFT\_PORT\_OPEN.set(false);

}

}

// If the server is not shutting down, we can start the production Finagle server and join

// the production server set.

if (EarlybirdStatus.getStatusCode() != EarlybirdStatusCode.STOPPING) {

LOG.info("Opening production thrift port...");

finagleServerManager.startProductionFinagleServer(

earlybirdDarkProxy.getDarkProxy(), this, SERVICE\_NAME, port);

EarlybirdStatus.THRIFT\_PORT\_OPEN.set(true);

if (REGISTER\_WITH\_ZK\_ON\_STARTUP) {

// After the earlybird starts up, register with ZooKeeper.

try {

joinServerSet("internal start-up");

// Join separate server set for ServiceProxy on Archive Earlybirds

if (!EarlybirdConfig.isAurora()) {

joinServerSetForServiceProxy();

}

} catch (UpdateException e) {

throw new RuntimeException("Unable to join ServerSet during startup.", e);

}

}

}

}

}

}

/\*\*

\* Stops the thrift server if the server is already running.

\*/

public void stopThriftService(boolean shouldShutDown) {

synchronized (shutdownLock) {

try {

leaveServerSet(shouldShutDown ? "internal shutdown" : "admin stopThriftService");

} catch (UpdateException e) {

LOG.warn("Leaving production ServerSet failed.", e);

}

if (finagleServerManager.isProductionServerRunning()) {

try {

finagleServerManager.stopProductionFinagleServer(SERVER\_CLOSE\_WAIT\_TIME);

futurePoolManager.stopUnderlyingFuturePool(

SERVER\_CLOSE\_WAIT\_TIME.inSeconds(), TimeUnit.SECONDS);

numSearcherThreadsGauge.set(0);

} catch (InterruptedException e) {

LOG.error("Interrupted while stopping thrift service", e);

Thread.currentThread().interrupt();

}

EarlybirdStatus.THRIFT\_PORT\_OPEN.set(false);

}

}

}

/\*\*

\* Gets a string with information about the last request we've seen from each client.

\*/

public Future<String> getLastSearchesByClient(boolean includeResults) {

LastSearchesSummary summary = new LastSearchesSummary(

lastRequestPerClientId, clientIdSearchStats, includeResults);

return Future.value(summary.getSummary());

}

/\*\*

\* The following are all the Thrift RPC methods inherited from EarlybirdService.Iface

\*/

// Thrift getName RPC.

@Override

public Future<String> getName() {

return Future.value(SERVICE\_NAME);

}

// Thrift getStatus RPC.

@Override

public Future<EarlybirdStatusResponse> getStatus() {

EarlybirdStatusResponse response = new EarlybirdStatusResponse();

response.setCode(EarlybirdStatus.getStatusCode());

response.setAliveSince(EarlybirdStatus.getStartTime());

response.setMessage(EarlybirdStatus.getStatusMessage());

return Future.value(response);

}

public Future<List<String>> getSegmentMetadata() {

return Future.value(segmentManager.getSegmentMetadata());

}

public Future<String> getQueryCachesData() {

return Future.value(segmentManager.getQueryCachesData());

}

/\*\*

\* Get a text summary for which fields did we use in a schema.

\*/

public Future<String> getQueriedFieldsAndSchemaStats() {

ImmutableSchemaInterface schema = this.earlybirdIndexConfig.getSchema().getSchemaSnapshot();

QueriedFieldsAndSchemaStats summary = new QueriedFieldsAndSchemaStats(schema,

queriedFieldsCounts);

return Future.value(summary.getSummary());

}

/\*\*

\* Shuts down the earlybird server.

\*/

public void shutdown() {

LOG.info("shutdown(): status set to STOPPING");

EarlybirdStatus.setStatus(EarlybirdStatusCode.STOPPING);

try {

LOG.info("Stopping Finagle server.");

stopThriftService(true);

EarlybirdStatus.THRIFT\_SERVICE\_STARTED.set(false);

if (queryCacheManager != null) {

queryCacheManager.shutdown();

} else {

LOG.info("No queryCacheManager to shut down");

}

earlybirdIndexConfig.getResourceCloser().shutdownExecutor();

isShuttingDown = true;

LOG.info("Closing {} closeables.", toClose.size());

for (Closeable closeable : toClose) {

closeable.close();

}

} catch (InterruptedException | IOException e) {

EarlybirdStatus.setStatus(EarlybirdStatusCode.UNHEALTHY, e.getMessage());

LOG.error("Interrupted during shutdown, status set to UNHEALTHY");

}

LOG.info("Earlybird server stopped!");

isShutdown = true;

}

@Override

public Future<EarlybirdResponse> search(final EarlybirdRequest request) {

final long requestReceivedTimeMillis = System.currentTimeMillis();

// Record clock diff as early as possible.

EarlybirdRequestUtil.recordClientClockDiff(request);

if (!futurePoolManager.isPoolReady()) {

return Future.exception(new TransientException("Earlybird not yet able to handle requests."));

}

return futurePoolManager.apply(new Function0<EarlybirdResponse>() {

@Override

public EarlybirdResponse apply() {

return doSearch(request, requestReceivedTimeMillis);

}

}).rescue(Function.func(

// respond with Nack when the queue is full

t -> Future.exception((t instanceof RejectedExecutionException) ? QUEUE\_FULL\_FAILURE : t)));

}

private EarlybirdResponse doSearch(EarlybirdRequest request, long requestReceivedTimeMillis) {

final long queueWaitTime = System.currentTimeMillis() - requestReceivedTimeMillis;

internalQueueWaitTimeStats.timerIncrement(queueWaitTime);

// request restart time, not to be confused with startTime which is server restart time

Timer timer = new Timer(TimeUnit.MICROSECONDS);

requestPreLogger.logRequest(request);

String clientId = ClientIdUtil.getClientIdFromRequest(request);

String finagleClientId = FinagleUtil.getFinagleClientName();

requestCountersByFinagleIdAndClientId.getUnchecked(new Pair<>(finagleClientId, clientId))

.increment();

EarlybirdRequestUtil.checkAndSetCollectorParams(request);

// If the thrift logger is busy logging, queue the thrift request for logging.

if (EarlybirdThriftRequestLoggingUtil.thriftLoggerBusy) {

EarlybirdThriftRequestLoggingUtil.REQUEST\_BUFFER.offer(request);

}

EarlybirdRequestUtil.logAndFixExcessiveValues(request);

final EarlybirdSearcher searcher = new EarlybirdSearcher(

request,

segmentManager,

audioSpaceTable,

queryCacheManager,

earlybirdIndexConfig.getSchema().getSchemaSnapshot(),

earlybirdIndexConfig.getCluster(),

dynamicPartitionConfig.getCurrentPartitionConfig(),

decider,

tweetsSearcherStats,

scoringModelsManager,

tensorflowModelsManager,

clock,

multiSegmentTermDictionaryManager,

queryTimeoutFactory,

qualityFactor);

QueryCostTracker queryCostTracker = QueryCostTracker.getTracker();

EarlybirdResponse response = null;

try {

if (skipTimedOutRequests

&& searcher.getTerminationTracker().getTimeoutEndTimeWithReservation()

<= clock.nowMillis()) {

requestTimeoutExceededBeforeSearchCounter.increment();

response = new EarlybirdResponse();

response.setResponseCode(EarlybirdResponseCode.SERVER\_TIMEOUT\_ERROR);

} else {

queryCostTracker.reset();

response = searcher.search();

}

} finally {

if (response == null) {

// This can only happen if we failed to catch an exception in the searcher.

LOG.error("Response was null: " + request.toString());

response = new EarlybirdResponse();

response.setResponseCode(EarlybirdResponseCode.TRANSIENT\_ERROR);

}

if (response.getSearchResults() == null) {

List<ThriftSearchResult> emptyResultSet = Lists.newArrayList();

response.setSearchResults(new ThriftSearchResults(emptyResultSet));

}

long reqLatency = timer.stop();

response.setResponseTime(reqLatency / 1000);

response.setResponseTimeMicros(reqLatency);

response.getSearchResults().setQueryCost(queryCostTracker.getTotalCost());

requestLogger.logRequest(request, response, timer);

int numResults = EarlybirdRequestLogger.numResultsForLog(response);

boolean success = response.getResponseCode() == EarlybirdResponseCode.SUCCESS;

boolean clientError = response.getResponseCode() == EarlybirdResponseCode.CLIENT\_ERROR;

boolean earlyTerminated = (response.getSearchResults().isSetNumPartitionsEarlyTerminated()

&& response.getSearchResults().getNumPartitionsEarlyTerminated() > 0)

|| searcher.getTerminationTracker().isEarlyTerminated();

// Update termination stats.

searcher.getTerminationTracker().getEarlyTerminationState().incrementCount();

searchStats.requestComplete(reqLatency, numResults, success, earlyTerminated, clientError);

if (searcher.getRequestStats() != null) {

searcher.getRequestStats().requestComplete(reqLatency, numResults, success,

earlyTerminated, clientError);

}

getResponseCodeCounter(response.getResponseCode()).increment();

// Adding this counter to make it easier to debug cases where we see a spike in

// bad client request errors but don't know where they're coming from. (The

// alternative is to ssh to a machine in the cluster and sample

// /var/log/earlybird/earlybird.failed\_requests).

getClientIdResponseCodeCounter(clientId, response.getResponseCode()).increment();

// Export request latency as a stat.

clientIdSearchStats.getUnchecked(clientId).timerIncrement(reqLatency);

requestCountersByFinagleClientId.getUnchecked(finagleClientId).timerIncrement(reqLatency);

addEarlybirdServerStats(response, queueWaitTime);

// Export scoring stats for the request.

exportScoringTimeStats(response, clientId);

}

Set<String> queriedFields = searcher.getQueriedFields();

if (queriedFields != null) {

for (String queriedField : queriedFields) {

queriedFieldsCounts.incrementAndGet(queriedField);

}

}

// Increment counters for age of the returned results.

if (response.getSearchResults() != null && response.getSearchResults().getResults() != null) {

long currentTime = System.currentTimeMillis();

for (ThriftSearchResult result : response.getSearchResults().getResults()) {

long tweetId = result.getId();

if (SnowflakeId.isSnowflakeId(tweetId)) {

long ageMillis = Math.max(0L,

currentTime - SnowflakeId.unixTimeMillisFromId(tweetId));

int ageDays = Duration.fromMilliseconds(ageMillis).inDays();

if (EarlybirdConfig.isRealtimeOrProtected()) {

String key = "result\_age\_in\_days\_" + ageDays;

resultsAgeCounter.getUnchecked(key).increment();

} else {

int ageYears = ageDays / 365;

String key = "result\_age\_in\_years\_" + ageYears;

resultsAgeCounter.getUnchecked(key).increment();

}

}

}

}

try {

lastRequestPerClientId.get(clientId).set(

new RequestResponsePair(request, searcher.getParsedQuery(),

searcher.getLuceneQuery(), response));

} catch (ExecutionException ex) {

// Not a big problem, we'll just notice that the admin page doesn't work, and it

// probably won't happen.

}

return response;

}

private void exportScoringTimeStats(EarlybirdResponse response, String clientId) {

if (response.isSetSearchResults()

&& response.getSearchResults().isSetScoringTimeNanos()

&& response.getSearchResults().isSetNumHitsProcessed()) {

int numHitsProcessed = response.getSearchResults().getNumHitsProcessed();

long scoringTimeNanos = response.getSearchResults().getScoringTimeNanos();

if (numHitsProcessed > 0) {

// Only compute and report scoring time per hit when we have hits. (i.e. we don't just want

// to report 0's for cases where there were no hits, and only want to report legit per-hit

// times.

long scoringTimePerHit = scoringTimeNanos / numHitsProcessed;

this.clientIdScoringPerHitStats.getUnchecked(clientId).timerIncrement(scoringTimePerHit);

this.overallScoringTimePerHitStats.timerIncrement(scoringTimePerHit);

}

this.clientIdScoringPerQueryStats.getUnchecked(clientId).timerIncrement(scoringTimeNanos);

this.overallScoringTimePerQueryStats.timerIncrement(scoringTimeNanos);

// The num hits processed stats here are scoped only to queries that were actually scored.

// This would exclude queries like term stats (that would otherwise have huge num hits

// processed).

this.clientIdScoringNumHitsProcessedStats.getUnchecked(clientId).record(numHitsProcessed);

this.overallScoringNumHitsProcessedStats.record(numHitsProcessed);

}

}

private void addEarlybirdServerStats(EarlybirdResponse response, long queueWaitTime) {

PartitionConfig curPartitionConfig = dynamicPartitionConfig.getCurrentPartitionConfig();

EarlybirdServerStats earlybirdServerStats = new EarlybirdServerStats();

response.setEarlybirdServerStats(earlybirdServerStats);

earlybirdServerStats.setHostname(DatabaseConfig.getLocalHostname());

earlybirdServerStats.setPartition(curPartitionConfig.getIndexingHashPartitionID());

earlybirdServerStats.setTierName(curPartitionConfig.getTierName());

earlybirdServerStats.setCurrentQps(searchStats.getRequestRate());

earlybirdServerStats.setQueueTimeMillis(queueWaitTime);

earlybirdServerStats.setAverageQueueTimeMillis(

(long) (double) internalQueueWaitTimeStats.read());

earlybirdServerStats.setAverageLatencyMicros(searchStats.getAverageLatency());

}

@Override

public void joinServerSet(String username) throws UpdateException {

serverSetManager.joinServerSet(username);

}

@Override

public int getNumberOfServerSetMembers() throws InterruptedException,

ZooKeeperClient.ZooKeeperConnectionException, KeeperException {

return serverSetManager.getNumberOfServerSetMembers();

}

@Override

public void leaveServerSet(String username) throws UpdateException {

serverSetManager.leaveServerSet(username);

}

@Override

public void joinServerSetForServiceProxy() {

serverSetManager.joinServerSetForServiceProxy();

}

@VisibleForTesting

protected static class EarlybirdThriftRequestLoggingUtil {

private static final int DEFAULT\_MAX\_ENTRIES\_TO\_LOG = 50000;

private static final int DEFAULT\_BUFFER\_SIZE = 10000;

private static final int DEFAULT\_LOGGING\_SLEEP\_MS = 100;

@VisibleForTesting

protected static volatile boolean thriftLoggerBusy = false;

private static final ExecutorService LOGGING\_EXECUTOR = Executors.newCachedThreadPool();

// Synchronized circular buffer used for buffering requests.

// If buffer is full, the oldest requests are replaced. This should not be a problem for

// logging purpose.

@VisibleForTesting

protected static final ArrayBlockingQueue<EarlybirdRequest> REQUEST\_BUFFER =

new ArrayBlockingQueue<>(DEFAULT\_BUFFER\_SIZE);

/\*\*

\* Create a separate thread to log thrift request to the given file. If a thread is already

\* logging thrift requests, this does nothing and throws an IOException indicating that the

\* logging thread is busy.

\*

\* @param logFile File to log to.

\* @param maxEntriesToLog Number of entries to log.

\* @param postLoggingHook Code to run after logging finishes. Only used for testing as of now.

\*/

@VisibleForTesting

protected static synchronized void startThriftLogging(final File logFile,

final int maxEntriesToLog,

final Runnable postLoggingHook)

throws IOException {

if (thriftLoggerBusy) {

throw new IOException("Already busy logging thrift request. No action taken.");

}

if (!logFile.canWrite()) {

throw new IOException("Unable to open log file for writing: " + logFile);

}

final BufferedWriter thriftLogWriter =

Files.newBufferedWriter(logFile.toPath(), Charsets.UTF\_8);

// TSerializer used by the writer thread.

final TSerializer serializer = new TSerializer();

REQUEST\_BUFFER.clear();

thriftLoggerBusy = true;

LOG.info("Started to log thrift requests into file " + logFile.getAbsolutePath());

LOGGING\_EXECUTOR.submit(() -> {

try {

int count = 0;

while (count < maxEntriesToLog) {

if (REQUEST\_BUFFER.isEmpty()) {

Thread.sleep(DEFAULT\_LOGGING\_SLEEP\_MS);

continue;

}

try {

EarlybirdRequest ebRequest = REQUEST\_BUFFER.poll();

String logLine = serializeThriftObject(ebRequest, serializer);

thriftLogWriter.write(logLine);

count++;

} catch (TException e) {

LOG.warn("Unable to serialize EarlybirdRequest for logging.", e);

}

}

return count;

} finally {

thriftLogWriter.close();

thriftLoggerBusy = false;

LOG.info("Finished logging thrift requests into file " + logFile.getAbsolutePath());

REQUEST\_BUFFER.clear();

if (postLoggingHook != null) {

postLoggingHook.run();

}

}

});

}

/\*\*

\* Serialize a thrift object to a base 64 encoded string.

\*/

private static String serializeThriftObject(TBase<?, ?> tObject, TSerializer serializer)

throws TException {

return new Base64().encodeToString(serializer.serialize(tObject)) + "\n";

}

}

/\*\*

\* Start to log thrift EarlybirdRequests.

\*

\* @param logFile Log file to write to.

\* @param numRequestsToLog Number of requests to collect. Default value of 50000 used if

\* 0 or negative numbers are pass in.

\*/

public void startThriftLogging(File logFile, int numRequestsToLog) throws IOException {

int requestToLog = numRequestsToLog <= 0

? EarlybirdThriftRequestLoggingUtil.DEFAULT\_MAX\_ENTRIES\_TO\_LOG : numRequestsToLog;

EarlybirdThriftRequestLoggingUtil.startThriftLogging(logFile, requestToLog, null);

}

@VisibleForTesting

@Override

public boolean isInServerSet() {

return serverSetManager.isInServerSet();

}

@VisibleForTesting

SearchCounter getResponseCodeCounter(EarlybirdResponseCode responseCode) {

return responseCodeCounters.get(responseCode);

}

@VisibleForTesting

SearchCounter getClientIdResponseCodeCounter(

String clientId, EarlybirdResponseCode responseCode) {

String key = String.format(RESPONSES\_PER\_CLIENT\_ID\_STAT\_TEMPLATE,

clientId, responseCode.name().toLowerCase());

return responseByClientIdAndResponseCode.getUnchecked(key);

}

public void setNoShutdownWhenNotInLayout(boolean noShutdown) {

stateManager.setNoShutdownWhenNotInLayout(noShutdown);

}

private void schedule(OneTaskScheduledExecutorManager manager) {

if (!isShuttingDown) {

manager.schedule();

toClose.add(manager);

}

}

public DynamicSchema getSchema() {

return earlybirdIndexConfig.getSchema();

}

public AudioSpaceTable getAudioSpaceTable() {

return audioSpaceTable;

}

}