package com.twitter.search.earlybird.search.queries;

import java.io.IOException;

import java.util.Arrays;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.concurrent.TimeUnit;

import java.util.stream.Collectors;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.Lists;

import com.google.common.collect.Maps;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.index.Term;

import org.apache.lucene.index.Terms;

import org.apache.lucene.index.TermsEnum;

import org.apache.lucene.search.BooleanClause;

import org.apache.lucene.search.BooleanQuery;

import org.apache.lucene.search.BulkScorer;

import org.apache.lucene.search.ConstantScoreQuery;

import org.apache.lucene.search.ConstantScoreWeight;

import org.apache.lucene.search.IndexSearcher;

import org.apache.lucene.search.Query;

import org.apache.lucene.search.Scorer;

import org.apache.lucene.search.ScoreMode;

import org.apache.lucene.search.Weight;

import org.apache.lucene.util.BytesRef;

import com.twitter.decider.Decider;

import com.twitter.search.common.decider.DeciderUtil;

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

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

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

import com.twitter.search.common.query.HitAttributeHelper;

import com.twitter.search.common.query.IDDisjunctionQuery;

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

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

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

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

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

import com.twitter.search.common.util.analysis.LongTermAttributeImpl;

import com.twitter.search.common.util.analysis.SortableLongTermAttributeImpl;

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

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

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

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

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

import com.twitter.search.earlybird.queryparser.EarlybirdQueryHelper;

import com.twitter.search.queryparser.query.QueryParserException;

/\*\*

\* A variant of a multi-term ID disjunction query (similar to {@link UserIdMultiSegmentQuery}),

\* that also uses a {@link MultiSegmentTermDictionary} where available, for more efficient

\* term lookups for queries that span multiple segments.

\*

\* By default, a IDDisjunctionQuery (or Lucene's MultiTermQuery), does a term dictionary lookup

\* for all of the terms in its disjunction, and it does it once for each segment (or AtomicReader)

\* that the query is searching.

\* This means that when the term dictionary is large, and the term lookups are expensive, and when

\* we are searching multiple segments, the query needs to make num\_terms \* num\_segments expensive

\* term dictionary lookups.

\*

\* With the help of a MultiSegmentTermDictionary, this multi-term disjunction query implementation

\* only does one lookup for all of the segments managed by the MultiSegmentTermDictionary.

\* If a segment is not supported by the MultiSegmentTermDictionary (e.g. if it's not optimized yet),

\* a regular lookup in that segment's term dictionary will be performed.

\*

\* Usually, we will make 'num\_terms' lookups in the current, un-optimized segment, and then if

\* more segments need to be searched, we will make another 'num\_terms' lookups, once for all of

\* the remaining segments.

\*

\* When performing lookups in the MultiSegmentTermDictionary, for each supported segment, we save

\* a list of termIds from that segment for all the searched terms that appear in that segment.

\*

\* For example, when querying for UserIdMultiSegmentQuery with user ids: {1L, 2L, 3L} and

\* segments: {1, 2}, where segment 1 has user ids {1L, 2L} indexed under termIds {100, 200},

\* and segment 2 has user ids {1L, 2L, 3L} indexed under termIds {200, 300, 400}, we will build

\* up the following map once:

\* segment1 -> [100, 200]

\* segment2 -> [200, 300, 400]

\*/

public class UserIdMultiSegmentQuery extends Query {

@VisibleForTesting

public static final SearchTimerStats TERM\_LOOKUP\_STATS =

SearchTimerStats.export("multi\_segment\_query\_term\_lookup", TimeUnit.NANOSECONDS, false);

public static final SearchTimerStats QUERY\_FROM\_PRECOMPUTED =

SearchTimerStats.export("multi\_segment\_query\_from\_precomputed", TimeUnit.NANOSECONDS, false);

public static final SearchTimerStats QUERY\_REGULAR =

SearchTimerStats.export("multi\_segment\_query\_regular", TimeUnit.NANOSECONDS, false);

@VisibleForTesting

public static final SearchCounter USED\_MULTI\_SEGMENT\_TERM\_DICTIONARY\_COUNT = SearchCounter.export(

"user\_id\_multi\_segment\_query\_used\_multi\_segment\_term\_dictionary\_count");

@VisibleForTesting

public static final SearchCounter USED\_ORIGINAL\_TERM\_DICTIONARY\_COUNT = SearchCounter.export(

"user\_id\_multi\_segment\_query\_used\_original\_term\_dictionary\_count");

private static final SearchCounter NEW\_QUERY\_COUNT =

SearchCounter.export("user\_id\_multi\_segment\_new\_query\_count");

private static final SearchCounter OLD\_QUERY\_COUNT =

SearchCounter.export("user\_id\_multi\_segment\_old\_query\_count");

private static final HashMap<String, SearchCounter> QUERY\_COUNT\_BY\_QUERY\_NAME = new HashMap<>();

private static final HashMap<String, SearchCounter> QUERY\_COUNT\_BY\_FIELD\_NAME = new HashMap<>();

private static final String DECIDER\_KEY\_PREFIX = "use\_multi\_segment\_id\_disjunction\_queries\_in\_";

/\*\*

\* Returns a new user ID disjunction query.

\*

\* @param ids The user IDs.

\* @param field The field storing the user IDs.

\* @param schemaSnapshot A snapshot of earlybird's schema.

\* @param multiSegmentTermDictionaryManager The manager for the term dictionaries that span

\* multiple segments.

\* @param decider The decider.

\* @param earlybirdCluster The earlybird cluster.

\* @param ranks The hit attribution ranks to be assigned to every user ID.

\* @param hitAttributeHelper The helper that tracks hit attributions.

\* @param queryTimeout The timeout to be enforced on this query.

\* @return A new user ID disjunction query.

\*/

public static Query createIdDisjunctionQuery(

String queryName,

List<Long> ids,

String field,

ImmutableSchemaInterface schemaSnapshot,

MultiSegmentTermDictionaryManager multiSegmentTermDictionaryManager,

Decider decider,

EarlybirdCluster earlybirdCluster,

List<Integer> ranks,

@Nullable HitAttributeHelper hitAttributeHelper,

@Nullable QueryTimeout queryTimeout) throws QueryParserException {

QUERY\_COUNT\_BY\_QUERY\_NAME.computeIfAbsent(queryName, name ->

SearchCounter.export("multi\_segment\_query\_name\_" + name)).increment();

QUERY\_COUNT\_BY\_FIELD\_NAME.computeIfAbsent(field, name ->

SearchCounter.export("multi\_segment\_query\_count\_for\_field\_" + name)).increment();

if (DeciderUtil.isAvailableForRandomRecipient(decider, getDeciderName(earlybirdCluster))) {

NEW\_QUERY\_COUNT.increment();

MultiSegmentTermDictionary multiSegmentTermDictionary =

multiSegmentTermDictionaryManager.getMultiSegmentTermDictionary(field);

return new UserIdMultiSegmentQuery(

ids,

field,

schemaSnapshot,

multiSegmentTermDictionary,

ranks,

hitAttributeHelper,

queryTimeout);

} else {

OLD\_QUERY\_COUNT.increment();

return new IDDisjunctionQuery(ids, field, schemaSnapshot);

}

}

@VisibleForTesting

public static String getDeciderName(EarlybirdCluster earlybirdCluster) {

return DECIDER\_KEY\_PREFIX + earlybirdCluster.name().toLowerCase();

}

private final boolean useOrderPreservingEncoding;

private final HitAttributeHelper hitAttributeHelper;

private final QueryTimeout queryTimeout;

private final MultiSegmentTermDictionary multiSegmentTermDictionary;

private final Schema.FieldInfo fieldInfo;

private final String field;

private final List<Long> ids;

private final List<Integer> ranks;

// For each segment where we have a multi-segment term dictionary, this map will contain the

// termIds of all the terms that actually appear in that segment's index.

@Nullable

private Map<InvertedIndex, List<TermRankPair>> termIdsPerSegment;

// A wrap class helps to associate termId with corresponding search operator rank if exist

private final class TermRankPair {

private final int termId;

private final int rank;

TermRankPair(int termId, int rank) {

this.termId = termId;

this.rank = rank;

}

public int getTermId() {

return termId;

}

public int getRank() {

return rank;

}

}

@VisibleForTesting

public UserIdMultiSegmentQuery(

List<Long> ids,

String field,

ImmutableSchemaInterface schemaSnapshot,

MultiSegmentTermDictionary termDictionary,

List<Integer> ranks,

@Nullable HitAttributeHelper hitAttributeHelper,

@Nullable QueryTimeout queryTimeout) {

this.field = field;

this.ids = ids;

this.multiSegmentTermDictionary = termDictionary;

this.ranks = ranks;

this.hitAttributeHelper = hitAttributeHelper;

this.queryTimeout = queryTimeout;

// check ids and ranks have same size

Preconditions.checkArgument(ranks.size() == 0 || ranks.size() == ids.size());

// hitAttributeHelper is not null iff ranks is not empty

if (ranks.size() > 0) {

Preconditions.checkNotNull(hitAttributeHelper);

} else {

Preconditions.checkArgument(hitAttributeHelper == null);

}

if (!schemaSnapshot.hasField(field)) {

throw new IllegalStateException("Tried to search a field which does not exist in schema");

}

this.fieldInfo = Preconditions.checkNotNull(schemaSnapshot.getFieldInfo(field));

IndexedNumericFieldSettings numericFieldSettings =

fieldInfo.getFieldType().getNumericFieldSettings();

if (numericFieldSettings == null) {

throw new IllegalStateException("Id field is not numerical");

}

this.useOrderPreservingEncoding = numericFieldSettings.isUseSortableEncoding();

}

/\*\*

\* If it hasn't been built yet, build up the map containing termIds of all the terms being

\* searched, for all of the segments that are managed by the multi-segment term dictionary.

\*

\* We only do this once, when we have to search the first segment that's supported by our

\* multi-segment term dictionary.

\*

\* Flow here is to:

\* 1. go through all the ids being queried.

\* 2. for each id, get the termIds for that term in all of the segments in the term dictionary

\* 3. for all of the segments that have that term, add the termId to that segment's list of

\* term ids (in the 'termIdsPerSegment' map).

\*/

private void createTermIdsPerSegment() {

if (termIdsPerSegment != null) {

// already created the map

return;

}

long start = System.nanoTime();

final BytesRef termRef = useOrderPreservingEncoding

? SortableLongTermAttributeImpl.newBytesRef()

: LongTermAttributeImpl.newBytesRef();

termIdsPerSegment = Maps.newHashMap();

List<? extends InvertedIndex> segmentIndexes = multiSegmentTermDictionary.getSegmentIndexes();

for (int idx = 0; idx < ids.size(); ++idx) {

long longTerm = ids.get(idx);

if (useOrderPreservingEncoding) {

SortableLongTermAttributeImpl.copyLongToBytesRef(termRef, longTerm);

} else {

LongTermAttributeImpl.copyLongToBytesRef(termRef, longTerm);

}

int[] termIds = multiSegmentTermDictionary.lookupTermIds(termRef);

Preconditions.checkState(segmentIndexes.size() == termIds.length,

"SegmentIndexes: %s, field: %s, termIds: %s",

segmentIndexes.size(), field, termIds.length);

for (int indexId = 0; indexId < termIds.length; indexId++) {

int termId = termIds[indexId];

if (termId != EarlybirdIndexSegmentAtomicReader.TERM\_NOT\_FOUND) {

InvertedIndex fieldIndex = segmentIndexes.get(indexId);

List<TermRankPair> termIdsList = termIdsPerSegment.get(fieldIndex);

if (termIdsList == null) {

termIdsList = Lists.newArrayList();

termIdsPerSegment.put(fieldIndex, termIdsList);

}

termIdsList.add(new TermRankPair(

termId, ranks.size() > 0 ? ranks.get(idx) : -1));

}

}

}

long elapsed = System.nanoTime() - start;

TERM\_LOOKUP\_STATS.timerIncrement(elapsed);

}

@Override

public Weight createWeight(IndexSearcher searcher, ScoreMode scoreMode, float boost) {

return new UserIdMultiSegmentQueryWeight(searcher, scoreMode, boost);

}

@Override

public int hashCode() {

return Arrays.hashCode(

new Object[] {useOrderPreservingEncoding, queryTimeout, field, ids, ranks});

}

@Override

public boolean equals(Object obj) {

if (!(obj instanceof UserIdMultiSegmentQuery)) {

return false;

}

UserIdMultiSegmentQuery query = UserIdMultiSegmentQuery.class.cast(obj);

return Arrays.equals(

new Object[] {useOrderPreservingEncoding, queryTimeout, field, ids, ranks},

new Object[] {query.useOrderPreservingEncoding,

query.queryTimeout,

query.field,

query.ids,

query.ranks});

}

@Override

public String toString(String fieldName) {

StringBuilder builder = new StringBuilder();

builder.append(getClass().getSimpleName()).append("[").append(fieldName).append(":");

for (Long id : this.ids) {

builder.append(id);

builder.append(",");

}

builder.setLength(builder.length() - 1);

builder.append("]");

return builder.toString();

}

private final class UserIdMultiSegmentQueryWeight extends ConstantScoreWeight {

private final IndexSearcher searcher;

private final ScoreMode scoreMode;

private UserIdMultiSegmentQueryWeight(

IndexSearcher searcher,

ScoreMode scoreMode,

float boost) {

super(UserIdMultiSegmentQuery.this, boost);

this.searcher = searcher;

this.scoreMode = scoreMode;

}

@Override

public Scorer scorer(LeafReaderContext context) throws IOException {

Weight weight = rewrite(context);

if (weight != null) {

return weight.scorer(context);

} else {

return null;

}

}

@Override

public BulkScorer bulkScorer(LeafReaderContext context) throws IOException {

Weight weight = rewrite(context);

if (weight != null) {

return weight.bulkScorer(context);

} else {

return null;

}

}

@Override

public void extractTerms(Set<Term> terms) {

terms.addAll(ids

.stream()

.map(id -> new Term(field, LongTermAttributeImpl.copyIntoNewBytesRef(id)))

.collect(Collectors.toSet()));

}

@Override

public boolean isCacheable(LeafReaderContext ctx) {

return true;

}

private Weight rewrite(LeafReaderContext context) throws IOException {

final Terms terms = context.reader().terms(field);

if (terms == null) {

// field does not exist

return null;

}

final TermsEnum termsEnum = terms.iterator();

Preconditions.checkNotNull(termsEnum, "No termsEnum for field: %s", field);

BooleanQuery bq;

// See if the segment is supported by the multi-segment term dictionary. If so, build up

// the query using the termIds from the multi-segment term dictionary.

// If not (for the current segment), do the term lookups directly in the queried segment.

InvertedIndex fieldIndex = getFieldIndexFromMultiTermDictionary(context);

if (fieldIndex != null) {

createTermIdsPerSegment();

USED\_MULTI\_SEGMENT\_TERM\_DICTIONARY\_COUNT.increment();

SearchTimer timer = QUERY\_FROM\_PRECOMPUTED.startNewTimer();

bq = addPrecomputedTermQueries(fieldIndex, termsEnum);

QUERY\_FROM\_PRECOMPUTED.stopTimerAndIncrement(timer);

} else {

USED\_ORIGINAL\_TERM\_DICTIONARY\_COUNT.increment();

// This segment is not supported by the multi-segment term dictionary. Lookup terms

// directly.

SearchTimer timer = QUERY\_REGULAR.startNewTimer();

bq = addTermQueries(termsEnum);

QUERY\_REGULAR.stopTimerAndIncrement(timer);

}

return searcher.rewrite(new ConstantScoreQuery(bq)).createWeight(

searcher, scoreMode, score());

}

/\*\*

\* If the multi-segment term dictionary supports this segment/LeafReader, then return the

\* InvertedIndex representing this segment.

\*

\* If the segment being queried right now is not in the multi-segment term dictionary (e.g.

\* if it's not optimized yet), return null.

\*/

@Nullable

private InvertedIndex getFieldIndexFromMultiTermDictionary(LeafReaderContext context)

throws IOException {

if (multiSegmentTermDictionary == null) {

return null;

}

if (context.reader() instanceof EarlybirdIndexSegmentAtomicReader) {

EarlybirdIndexSegmentAtomicReader reader =

(EarlybirdIndexSegmentAtomicReader) context.reader();

EarlybirdIndexSegmentData segmentData = reader.getSegmentData();

InvertedIndex fieldIndex = segmentData.getFieldIndex(field);

if (multiSegmentTermDictionary.supportSegmentIndex(fieldIndex)) {

return fieldIndex;

}

}

return null;

}

private BooleanQuery addPrecomputedTermQueries(

InvertedIndex fieldIndex,

TermsEnum termsEnum) throws IOException {

BooleanQuery.Builder bqBuilder = new BooleanQuery.Builder();

int numClauses = 0;

List<TermRankPair> termRankPairs = termIdsPerSegment.get(fieldIndex);

if (termRankPairs != null) {

for (TermRankPair pair : termRankPairs) {

int termId = pair.getTermId();

if (numClauses >= BooleanQuery.getMaxClauseCount()) {

BooleanQuery saved = bqBuilder.build();

bqBuilder = new BooleanQuery.Builder();

bqBuilder.add(saved, BooleanClause.Occur.SHOULD);

numClauses = 1;

}

Query query;

if (pair.getRank() != -1) {

query = EarlybirdQueryHelper.maybeWrapWithHitAttributionCollector(

new SimpleTermQuery(termsEnum, termId),

pair.getRank(),

fieldInfo,

hitAttributeHelper);

} else {

query = new SimpleTermQuery(termsEnum, termId);

}

bqBuilder.add(EarlybirdQueryHelper.maybeWrapWithTimeout(query, queryTimeout),

BooleanClause.Occur.SHOULD);

++numClauses;

}

}

return bqBuilder.build();

}

private BooleanQuery addTermQueries(TermsEnum termsEnum) throws IOException {

final BytesRef termRef = useOrderPreservingEncoding

? SortableLongTermAttributeImpl.newBytesRef()

: LongTermAttributeImpl.newBytesRef();

BooleanQuery.Builder bqBuilder = new BooleanQuery.Builder();

int numClauses = 0;

for (int idx = 0; idx < ids.size(); ++idx) {

long longTerm = ids.get(idx);

if (useOrderPreservingEncoding) {

SortableLongTermAttributeImpl.copyLongToBytesRef(termRef, longTerm);

} else {

LongTermAttributeImpl.copyLongToBytesRef(termRef, longTerm);

}

if (termsEnum.seekExact(termRef)) {

if (numClauses >= BooleanQuery.getMaxClauseCount()) {

BooleanQuery saved = bqBuilder.build();

bqBuilder = new BooleanQuery.Builder();

bqBuilder.add(saved, BooleanClause.Occur.SHOULD);

numClauses = 1;

}

if (ranks.size() > 0) {

bqBuilder.add(EarlybirdQueryHelper.maybeWrapWithHitAttributionCollector(

new SimpleTermQuery(termsEnum, termsEnum.ord()),

ranks.get(idx),

fieldInfo,

hitAttributeHelper),

BooleanClause.Occur.SHOULD);

} else {

bqBuilder.add(new SimpleTermQuery(termsEnum, termsEnum.ord()),

BooleanClause.Occur.SHOULD);

}

++numClauses;

}

}

return bqBuilder.build();

}

}

}