package com.twitter.search.core.earlybird.index.inverted;

import java.io.IOException;

import java.util.Comparator;

import javax.annotation.Nullable;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.Terms;

import org.apache.lucene.index.TermsEnum;

import org.apache.lucene.util.BytesRef;

import org.apache.lucene.util.StringHelper;

import com.twitter.search.common.hashtable.HashTable;

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

import com.twitter.search.common.util.hash.KeysSource;

import com.twitter.search.common.util.io.flushable.DataDeserializer;

import com.twitter.search.common.util.io.flushable.DataSerializer;

import com.twitter.search.common.util.io.flushable.FlushInfo;

import com.twitter.search.common.util.io.flushable.Flushable;

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

public class InvertedRealtimeIndex extends InvertedIndex {

public static final int FIXED\_HASH\_SEED = 0;

public final class TermHashTable extends HashTable<BytesRef> {

private final TermPointerEncoding termPointerEncoding;

public TermHashTable(int size, TermPointerEncoding termPointerEncoding) {

super(size);

this.termPointerEncoding = termPointerEncoding;

}

public TermHashTable(int[] termsHash, TermPointerEncoding termPointerEncoding) {

super(termsHash);

this.termPointerEncoding = termPointerEncoding;

}

@Override

public boolean matchItem(BytesRef term, int candidateTermID) {

return ByteTermUtils.postingEquals(

getTermPool(),

termPointerEncoding.getTextStart(termsArray.termPointers[candidateTermID]), term);

}

@Override

public int hashCodeForItem(int itemID) {

return ByteTermUtils.hashCode(

getTermPool(), termPointerEncoding.getTextStart(termsArray.termPointers[itemID]));

}

/\*

\* Use a fixed hash seed to compute the hash code for the given item. This is necessary because

\* we want the TermHashTable to be consistent for lookups in indexes that have been flushed and

\* loaded across restarts and redeploys.

\*

\* Note: previously we used item.hashcode(), however that hash function relies on the seed value

\* StringHelper.GOOD\_FAST\_HASH\_SEED, which is initialized to System.currentTimeMillis() when the

\* JVM process starts up.

\*/

public long lookupItem(BytesRef item) {

int itemHashCode = StringHelper.murmurhash3\_x86\_32(item, FIXED\_HASH\_SEED);

return super.lookupItem(item, itemHashCode);

}

}

/\*\*

\* Skip list comparator used by {@link #termsSkipList}. The key would be the bytesRef of the term,

\* and the value would be the termID of a term.

\*

\* Notice this comparator is keeping states,

\* so different threads CANNOT share the same comparator.

\*/

public static final class TermsSkipListComparator implements SkipListComparator<BytesRef> {

private static final Comparator<BytesRef> BYTES\_REF\_COMPARATOR = Comparator.naturalOrder();

private static final int SENTINEL\_VALUE = HashTable.EMPTY\_SLOT;

// Initializing two BytesRef to use for later comparisons.

// Notice different threads cannot share the same comparator.

private final BytesRef bytesRef1 = new BytesRef();

private final BytesRef bytesRef2 = new BytesRef();

/\*\*

\* We have to pass each part of the index in since during load process, the comparator

\* needs to be build before the index.

\*/

private final InvertedRealtimeIndex invertedIndex;

public TermsSkipListComparator(InvertedRealtimeIndex invertedIndex) {

this.invertedIndex = invertedIndex;

}

@Override

public int compareKeyWithValue(BytesRef key, int targetValue, int targetPosition) {

// No key could represent SENTINEL\_VALUE and SENTINEL\_VALUE is greatest.

if (targetValue == SENTINEL\_VALUE) {

return -1;

} else {

getTerm(targetValue, bytesRef1);

return BYTES\_REF\_COMPARATOR.compare(key, bytesRef1);

}

}

@Override

public int compareValues(int v1, int v2) {

// SENTINEL\_VALUE is greatest.

if (v1 != SENTINEL\_VALUE && v2 != SENTINEL\_VALUE) {

getTerm(v1, bytesRef1);

getTerm(v2, bytesRef2);

return BYTES\_REF\_COMPARATOR.compare(bytesRef1, bytesRef2);

} else if (v1 == SENTINEL\_VALUE && v2 == SENTINEL\_VALUE) {

return 0;

} else if (v1 == SENTINEL\_VALUE) {

return 1;

} else {

return -1;

}

}

@Override

public int getSentinelValue() {

return SENTINEL\_VALUE;

}

/\*\*

\* Get the term specified by the termID.

\* This method should be the same as {@link InvertedRealtimeIndex#getTerm}

\*/

private void getTerm(int termID, BytesRef text) {

invertedIndex.getTerm(termID, text);

}

}

private static final int HASHMAP\_SIZE = 64 \* 1024;

private SkipListContainer<BytesRef> termsSkipList;

private final TermPointerEncoding termPointerEncoding;

private final ByteBlockPool termPool;

private final SkipListPostingList postingList;

private int numTerms;

private int numDocs;

private int sumTotalTermFreq;

private int sumTermDocFreq;

private int maxPosition;

private volatile TermHashTable hashTable;

private TermsArray termsArray;

/\*\*

\* Creates a new in-memory real-time inverted index for the given field.

\*/

public InvertedRealtimeIndex(EarlybirdFieldType fieldType,

TermPointerEncoding termPointerEncoding,

String fieldName) {

super(fieldType);

this.termPool = new ByteBlockPool();

this.termPointerEncoding = termPointerEncoding;

this.hashTable = new TermHashTable(HASHMAP\_SIZE, termPointerEncoding);

this.postingList = new SkipListPostingList(

fieldType.hasPositions()

? SkipListContainer.HasPositions.YES

: SkipListContainer.HasPositions.NO,

fieldType.isStorePerPositionPayloads()

? SkipListContainer.HasPayloads.YES

: SkipListContainer.HasPayloads.NO,

fieldName);

this.termsArray = new TermsArray(

HASHMAP\_SIZE, fieldType.isStoreFacetOffensiveCounters());

// Create termsSkipList to maintain order if field is support ordered terms.

if (fieldType.isSupportOrderedTerms()) {

// Terms skip list does not support position.

this.termsSkipList = new SkipListContainer<>(

new TermsSkipListComparator(this),

SkipListContainer.HasPositions.NO,

SkipListContainer.HasPayloads.NO,

"terms");

this.termsSkipList.newSkipList();

} else {

this.termsSkipList = null;

}

}

void setTermsSkipList(SkipListContainer<BytesRef> termsSkipList) {

this.termsSkipList = termsSkipList;

}

SkipListContainer<BytesRef> getTermsSkipList() {

return termsSkipList;

}

private InvertedRealtimeIndex(

EarlybirdFieldType fieldType,

int numTerms,

int numDocs,

int sumTermDocFreq,

int sumTotalTermFreq,

int maxPosition,

int[] termsHash,

TermsArray termsArray,

ByteBlockPool termPool,

TermPointerEncoding termPointerEncoding,

SkipListPostingList postingList) {

super(fieldType);

this.numTerms = numTerms;

this.numDocs = numDocs;

this.sumTermDocFreq = sumTermDocFreq;

this.sumTotalTermFreq = sumTotalTermFreq;

this.maxPosition = maxPosition;

this.termsArray = termsArray;

this.termPool = termPool;

this.termPointerEncoding = termPointerEncoding;

this.hashTable = new TermHashTable(termsHash, termPointerEncoding);

this.postingList = postingList;

}

void insertToTermsSkipList(BytesRef termBytesRef, int termID) {

if (termsSkipList != null) {

// Use the comparator passed in while building the skip list since we only have one writer.

termsSkipList.insert(termBytesRef, termID, SkipListContainer.FIRST\_LIST\_HEAD);

}

}

@Override

public int getNumTerms() {

return numTerms;

}

@Override

public int getNumDocs() {

return numDocs;

}

@Override

public int getSumTotalTermFreq() {

return sumTotalTermFreq;

}

@Override

public int getSumTermDocFreq() {

return sumTermDocFreq;

}

@Override

public Terms createTerms(int maxPublishedPointer) {

return new RealtimeIndexTerms(this, maxPublishedPointer);

}

@Override

public TermsEnum createTermsEnum(int maxPublishedPointer) {

// Use SkipListInMemoryTermsEnum if termsSkipList is not null, which indicates field required

// ordered term.

if (termsSkipList == null) {

return new RealtimeIndexTerms.InMemoryTermsEnum(this, maxPublishedPointer);

} else {

return new RealtimeIndexTerms.SkipListInMemoryTermsEnum(this, maxPublishedPointer);

}

}

int getPostingListPointer(int termID) {

return termsArray.getPostingsPointer(termID);

}

@Override

public int getLargestDocIDForTerm(int termID) {

if (termID == EarlybirdIndexSegmentAtomicReader.TERM\_NOT\_FOUND) {

return TermsArray.INVALID;

} else {

return postingList.getDocIDFromPosting(termsArray.largestPostings[termID]);

}

}

@Override

public int getDF(int termID) {

if (termID == HashTable.EMPTY\_SLOT) {

return 0;

} else {

return this.postingList.getDF(termID, termsArray);

}

}

@Override

public int getMaxPublishedPointer() {

return this.postingList.getMaxPublishedPointer();

}

@Override

public int lookupTerm(BytesRef term) {

return HashTable.decodeItemId(hashTable.lookupItem(term));

}

@Override

public FacetLabelAccessor getLabelAccessor() {

final TermsArray termsArrayCopy = this.termsArray;

return new FacetLabelAccessor() {

@Override protected boolean seek(long termID) {

if (termID == HashTable.EMPTY\_SLOT) {

return false;

}

int termPointer = termsArrayCopy.termPointers[(int) termID];

hasTermPayload = termPointerEncoding.hasPayload(termPointer);

int textStart = termPointerEncoding.getTextStart(termPointer);

int termPayloadStart = ByteTermUtils.setBytesRef(termPool, termRef, textStart);

if (hasTermPayload) {

ByteTermUtils.setBytesRef(termPool, termPayload, termPayloadStart);

}

offensiveCount = termsArrayCopy.offensiveCounters != null

? termsArrayCopy.offensiveCounters[(int) termID] : 0;

return true;

}

};

}

@Override

public boolean hasMaxPublishedPointer() {

return true;

}

@Override

public void getTerm(int termID, BytesRef text) {

getTerm(termID, text, termsArray, termPointerEncoding, termPool);

}

/\*\*

\* Extract to helper method so the logic can be shared with

\* {@link TermsSkipListComparator#getTerm}

\*/

private static void getTerm(int termID, BytesRef text,

TermsArray termsArray,

TermPointerEncoding termPointerEncoding,

ByteBlockPool termPool) {

int textStart = termPointerEncoding.getTextStart(termsArray.termPointers[termID]);

ByteTermUtils.setBytesRef(termPool, text, textStart);

}

/\*\*

\* Called when postings hash is too small (> 50% occupied).

\*/

void rehashPostings(int newSize) {

TermHashTable newTable = new TermHashTable(newSize, termPointerEncoding);

hashTable.rehash(newTable);

hashTable = newTable;

}

/\*\*

\* Returns per-term array containing the number of documents indexed with that term that were

\* considered to be offensive.

\*/

@Nullable

int[] getOffensiveCounters() {

return this.termsArray.offensiveCounters;

}

/\*\*

\* Returns access to all the terms in this index as a {@link KeysSource}.

\*/

public KeysSource getKeysSource() {

final int localNumTerms = this.numTerms;

final TermsArray termsArrayCopy = this.termsArray;

return new KeysSource() {

private int termID = 0;

private BytesRef text = new BytesRef();

@Override

public int getNumberOfKeys() {

return localNumTerms;

}

/\*\* Must not be called more often than getNumberOfKeys() before rewind() is called \*/

@Override

public BytesRef nextKey() {

Preconditions.checkState(termID < localNumTerms);

int textStart = termPointerEncoding.getTextStart(termsArrayCopy.termPointers[termID]);

ByteTermUtils.setBytesRef(termPool, text, textStart);

termID++;

return text;

}

@Override

public void rewind() {

termID = 0;

}

};

}

/\*\*

\* Returns byte pool containing term text for all terms in this index.

\*/

public ByteBlockPool getTermPool() {

return this.termPool;

}

/\*\*

\* Returns per-term array containing pointers to where the text of each term is stored in the

\* byte pool returned by {@link #getTermPool()}.

\*/

public int[] getTermPointers() {

return this.termsArray.termPointers;

}

/\*\*

\* Returns the hash table used to look up terms in this index.

\*/

InvertedRealtimeIndex.TermHashTable getHashTable() {

return hashTable;

}

TermsArray getTermsArray() {

return termsArray;

}

TermsArray growTermsArray() {

termsArray = termsArray.grow();

return termsArray;

}

@SuppressWarnings("unchecked")

@Override

public FlushHandler getFlushHandler() {

return new FlushHandler(this);

}

TermPointerEncoding getTermPointerEncoding() {

return termPointerEncoding;

}

SkipListPostingList getPostingList() {

return postingList;

}

void incrementNumTerms() {

numTerms++;

}

void incrementSumTotalTermFreq() {

sumTotalTermFreq++;

}

public void incrementSumTermDocFreq() {

sumTermDocFreq++;

}

public void incrementNumDocs() {

numDocs++;

}

void setNumDocs(int numDocs) {

this.numDocs = numDocs;

}

void adjustMaxPosition(int position) {

if (position > maxPosition) {

maxPosition = position;

}

}

int getMaxPosition() {

return maxPosition;

}

public static class FlushHandler extends Flushable.Handler<InvertedRealtimeIndex> {

private static final String NUM\_DOCS\_PROP\_NAME = "numDocs";

private static final String SUM\_TOTAL\_TERM\_FREQ\_PROP\_NAME = "sumTotalTermFreq";

private static final String SUM\_TERM\_DOC\_FREQ\_PROP\_NAME = "sumTermDocFreq";

private static final String NUM\_TERMS\_PROP\_NAME = "numTerms";

private static final String POSTING\_LIST\_PROP\_NAME = "postingList";

private static final String TERMS\_SKIP\_LIST\_PROP\_NAME = "termsSkipList";

private static final String MAX\_POSITION = "maxPosition";

protected final EarlybirdFieldType fieldType;

protected final TermPointerEncoding termPointerEncoding;

public FlushHandler(EarlybirdFieldType fieldType,

TermPointerEncoding termPointerEncoding) {

this.fieldType = fieldType;

this.termPointerEncoding = termPointerEncoding;

}

public FlushHandler(InvertedRealtimeIndex objectToFlush) {

super(objectToFlush);

this.fieldType = objectToFlush.fieldType;

this.termPointerEncoding = objectToFlush.getTermPointerEncoding();

}

@Override

protected void doFlush(FlushInfo flushInfo, DataSerializer out)

throws IOException {

InvertedRealtimeIndex objectToFlush = getObjectToFlush();

flushInfo.addIntProperty(NUM\_TERMS\_PROP\_NAME, objectToFlush.getNumTerms());

flushInfo.addIntProperty(NUM\_DOCS\_PROP\_NAME, objectToFlush.numDocs);

flushInfo.addIntProperty(SUM\_TERM\_DOC\_FREQ\_PROP\_NAME, objectToFlush.sumTermDocFreq);

flushInfo.addIntProperty(SUM\_TOTAL\_TERM\_FREQ\_PROP\_NAME, objectToFlush.sumTotalTermFreq);

flushInfo.addIntProperty(MAX\_POSITION, objectToFlush.maxPosition);

out.writeIntArray(objectToFlush.hashTable.slots());

objectToFlush.termsArray.getFlushHandler()

.flush(flushInfo.newSubProperties("termsArray"), out);

objectToFlush.getTermPool().getFlushHandler()

.flush(flushInfo.newSubProperties("termPool"), out);

objectToFlush.getPostingList().getFlushHandler()

.flush(flushInfo.newSubProperties(POSTING\_LIST\_PROP\_NAME), out);

if (fieldType.isSupportOrderedTerms()) {

Preconditions.checkNotNull(objectToFlush.termsSkipList);

objectToFlush.termsSkipList.getFlushHandler()

.flush(flushInfo.newSubProperties(TERMS\_SKIP\_LIST\_PROP\_NAME), out);

}

}

@Override

protected InvertedRealtimeIndex doLoad(FlushInfo flushInfo, DataDeserializer in)

throws IOException {

int[] termsHash = in.readIntArray();

TermsArray termsArray = (new TermsArray.FlushHandler())

.load(flushInfo.getSubProperties("termsArray"), in);

ByteBlockPool termPool = (new ByteBlockPool.FlushHandler())

.load(flushInfo.getSubProperties("termPool"), in);

SkipListPostingList postingList = (new SkipListPostingList.FlushHandler())

.load(flushInfo.getSubProperties(POSTING\_LIST\_PROP\_NAME), in);

InvertedRealtimeIndex index = new InvertedRealtimeIndex(

fieldType,

flushInfo.getIntProperty(NUM\_TERMS\_PROP\_NAME),

flushInfo.getIntProperty(NUM\_DOCS\_PROP\_NAME),

flushInfo.getIntProperty(SUM\_TERM\_DOC\_FREQ\_PROP\_NAME),

flushInfo.getIntProperty(SUM\_TOTAL\_TERM\_FREQ\_PROP\_NAME),

flushInfo.getIntProperty(MAX\_POSITION),

termsHash,

termsArray,

termPool,

termPointerEncoding,

postingList);

if (fieldType.isSupportOrderedTerms()) {

SkipListComparator<BytesRef> comparator = new TermsSkipListComparator(index);

index.setTermsSkipList((new SkipListContainer.FlushHandler<>(comparator))

.load(flushInfo.getSubProperties(TERMS\_SKIP\_LIST\_PROP\_NAME), in));

}

return index;

}

}

}