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

import java.util.Iterator;

import java.util.TreeSet;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.BaseTermsEnum;

import org.apache.lucene.index.ImpactsEnum;

import org.apache.lucene.index.PostingsEnum;

import org.apache.lucene.index.SlowImpactsEnum;

import org.apache.lucene.index.Terms;

import org.apache.lucene.index.TermsEnum;

import org.apache.lucene.util.BytesRef;

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

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

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

public class RealtimeIndexTerms extends Terms {

// Calling InMemoryTermsEnum.next() creates a full copy of the entire term dictionary, and can

// be quite expensive. We don't expect these calls to happen, and they shpould not happen on the

// regular read path. We stat them here just in case to see if there is any unexpected usage.

private static final SearchCounter TERMS\_ENUM\_NEXT\_CALLS =

SearchCounter.export("in\_memory\_terms\_enum\_next\_calls");

private static final SearchCounter TERMS\_ENUM\_CREATE\_TERM\_SET =

SearchCounter.export("in\_memory\_terms\_enum\_next\_create\_term\_set");

private static final SearchCounter TERMS\_ENUM\_CREATE\_TERM\_SET\_SIZE =

SearchCounter.export("in\_memory\_terms\_enum\_next\_create\_term\_set\_size");

private final InvertedRealtimeIndex index;

private final int maxPublishedPointer;

public RealtimeIndexTerms(InvertedRealtimeIndex index, int maxPublishedPointer) {

this.index = index;

this.maxPublishedPointer = maxPublishedPointer;

}

@Override

public long size() {

return index.getNumTerms();

}

@Override

public TermsEnum iterator() {

return index.createTermsEnum(maxPublishedPointer);

}

/\*\*

\* This TermsEnum use a tree set to support {@link TermsEnum#next()} method. However, this is not

\* efficient enough to support realtime operation. {@link TermsEnum#seekCeil} is not fully

\* supported in this termEnum.

\*/

public static class InMemoryTermsEnum extends BaseTermsEnum {

private final InvertedRealtimeIndex index;

private final int maxPublishedPointer;

private int termID = -1;

private BytesRef bytesRef = new BytesRef();

private Iterator<BytesRef> termIter;

private TreeSet<BytesRef> termSet;

public InMemoryTermsEnum(InvertedRealtimeIndex index, int maxPublishedPointer) {

this.index = index;

this.maxPublishedPointer = maxPublishedPointer;

termIter = null;

}

@Override

public int docFreq() {

return index.getDF(termID);

}

@Override

public PostingsEnum postings(PostingsEnum reuse, int flags) {

int postingsPointer = index.getPostingListPointer(termID);

return index.getPostingList().postings(postingsPointer, docFreq(), maxPublishedPointer);

}

@Override

public ImpactsEnum impacts(int flags) {

return new SlowImpactsEnum(postings(null, flags));

}

@Override

public SeekStatus seekCeil(BytesRef text) {

// Nullify termIter.

termIter = null;

termID = index.lookupTerm(text);

if (termID == -1) {

return SeekStatus.END;

} else {

index.getTerm(termID, bytesRef);

return SeekStatus.FOUND;

}

}

@Override

public BytesRef next() {

TERMS\_ENUM\_NEXT\_CALLS.increment();

if (termSet == null) {

termSet = new TreeSet<>();

KeysSource keysource = index.getKeysSource();

keysource.rewind();

int numTerms = keysource.getNumberOfKeys();

for (int i = 0; i < numTerms; ++i) {

BytesRef ref = keysource.nextKey();

// we need to clone the ref since the keysource is reusing the returned BytesRef

// instance and we are storing it

termSet.add(ref.clone());

}

TERMS\_ENUM\_CREATE\_TERM\_SET.increment();

TERMS\_ENUM\_CREATE\_TERM\_SET\_SIZE.add(numTerms);

}

// Construct termIter from the subset.

if (termIter == null) {

termIter = termSet.tailSet(bytesRef, true).iterator();

}

if (termIter.hasNext()) {

bytesRef = termIter.next();

termID = index.lookupTerm(bytesRef);

} else {

termID = -1;

bytesRef = null;

}

return bytesRef;

}

@Override

public long ord() {

return termID;

}

@Override

public void seekExact(long ord) {

// Nullify termIter.

termIter = null;

if (ord < index.getNumTerms()) {

termID = (int) ord;

index.getTerm(termID, bytesRef);

}

}

@Override

public BytesRef term() {

return bytesRef;

}

@Override

public long totalTermFreq() {

return docFreq();

}

}

/\*\*

\* This TermsEnum use a {@link SkipListContainer} backed termsSkipList provided by

\* {@link InvertedRealtimeIndex} to supported ordered terms operations like

\* {@link TermsEnum#next()} and {@link TermsEnum#seekCeil}.

\*/

public static class SkipListInMemoryTermsEnum extends BaseTermsEnum {

private final InvertedRealtimeIndex index;

private int termID = -1;

private BytesRef bytesRef = new BytesRef();

private int nextTermIDPointer;

/\*\*

\* {@link #nextTermIDPointer} is used to record pointer to next termsID to accelerate

\* {@link #next}. However, {@link #seekCeil} and {@link #seekExact} may jump to an arbitrary

\* term so the {@link #nextTermIDPointer} may not be correct, and this flag is used to check if

\* this happens. If this flag is false, {@link #correctNextTermIDPointer} should be called to

\* correct the value.

\*/

private boolean isNextTermIDPointerCorrect;

private final SkipListContainer<BytesRef> termsSkipList;

private final InvertedRealtimeIndex.TermsSkipListComparator termsSkipListComparator;

private final int maxPublishedPointer;

/\*\*

\* Creates a new {@link TermsEnum} for a skip list-based sorted real-time term dictionary.

\*/

public SkipListInMemoryTermsEnum(InvertedRealtimeIndex index, int maxPublishedPointer) {

Preconditions.checkNotNull(index.getTermsSkipList());

this.index = index;

this.termsSkipList = index.getTermsSkipList();

// Each Terms Enum shall have their own comparators to be thread safe.

this.termsSkipListComparator =

new InvertedRealtimeIndex.TermsSkipListComparator(index);

this.nextTermIDPointer =

termsSkipList.getNextPointer(SkipListContainer.FIRST\_LIST\_HEAD);

this.isNextTermIDPointerCorrect = true;

this.maxPublishedPointer = maxPublishedPointer;

}

@Override

public int docFreq() {

return index.getDF(termID);

}

@Override

public PostingsEnum postings(PostingsEnum reuse, int flags) {

int postingsPointer = index.getPostingListPointer(termID);

return index.getPostingList().postings(postingsPointer, docFreq(), maxPublishedPointer);

}

@Override

public ImpactsEnum impacts(int flags) {

return new SlowImpactsEnum(postings(null, flags));

}

@Override

public SeekStatus seekCeil(BytesRef text) {

// Next term pointer is not correct anymore since seek ceil

// will jump to an arbitrary term.

isNextTermIDPointerCorrect = false;

// Doing precise lookup first.

termID = index.lookupTerm(text);

// Doing ceil lookup if not found, otherwise we are good.

if (termID == -1) {

return seekCeilWithSkipList(text);

} else {

index.getTerm(termID, bytesRef);

return SeekStatus.FOUND;

}

}

/\*\*

\* Doing ceil terms search with terms skip list.

\*/

private SeekStatus seekCeilWithSkipList(BytesRef text) {

int termIDPointer = termsSkipList.searchCeil(text,

SkipListContainer.FIRST\_LIST\_HEAD,

termsSkipListComparator,

null);

// End reached but still cannot found a ceil term.

if (termIDPointer == SkipListContainer.FIRST\_LIST\_HEAD) {

termID = HashTable.EMPTY\_SLOT;

return SeekStatus.END;

}

termID = termsSkipList.getValue(termIDPointer);

// Set next termID pointer and is correct flag.

nextTermIDPointer = termsSkipList.getNextPointer(termIDPointer);

isNextTermIDPointerCorrect = true;

// Found a ceil term but not the precise match.

index.getTerm(termID, bytesRef);

return SeekStatus.NOT\_FOUND;

}

/\*\*

\* {@link #nextTermIDPointer} is used to record the pointer to next termID. This method is used

\* to correct {@link #nextTermIDPointer} to correct value after {@link #seekCeil} or

\* {@link #seekExact} dropped current term to arbitrary point.

\*/

private void correctNextTermIDPointer() {

final int curTermIDPointer = termsSkipList.search(

bytesRef,

SkipListContainer.FIRST\_LIST\_HEAD,

termsSkipListComparator,

null);

// Must be able to find the exact term.

assert termID == HashTable.EMPTY\_SLOT

|| termID == termsSkipList.getValue(curTermIDPointer);

nextTermIDPointer = termsSkipList.getNextPointer(curTermIDPointer);

isNextTermIDPointerCorrect = true;

}

@Override

public BytesRef next() {

// Correct nextTermIDPointer first if not correct due to seekExact or seekCeil.

if (!isNextTermIDPointerCorrect) {

correctNextTermIDPointer();

}

// Skip list is exhausted.

if (nextTermIDPointer == SkipListContainer.FIRST\_LIST\_HEAD) {

termID = HashTable.EMPTY\_SLOT;

return null;

}

termID = termsSkipList.getValue(nextTermIDPointer);

index.getTerm(termID, bytesRef);

// Set next termID Pointer.

nextTermIDPointer = termsSkipList.getNextPointer(nextTermIDPointer);

return bytesRef;

}

@Override

public long ord() {

return termID;

}

@Override

public void seekExact(long ord) {

if (ord < index.getNumTerms()) {

termID = (int) ord;

index.getTerm(termID, bytesRef);

// Next term pointer is not correct anymore since seek exact

// just jump to an arbitrary term.

isNextTermIDPointerCorrect = false;

}

}

@Override

public BytesRef term() {

return bytesRef;

}

@Override

public long totalTermFreq() {

return docFreq();

}

}

@Override

public long getSumTotalTermFreq() {

return index.getSumTotalTermFreq();

}

@Override

public long getSumDocFreq() {

return index.getSumTermDocFreq();

}

@Override

public int getDocCount() {

return index.getNumDocs();

}

@Override

public boolean hasFreqs() {

return true;

}

@Override

public boolean hasOffsets() {

return false;

}

@Override

public boolean hasPositions() {

return true;

}

@Override

public boolean hasPayloads() {

return true;

}

}