package com.twitter.search.earlybird.index;

import java.io.IOException;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.LeafReader;

import org.apache.lucene.index.NumericDocValues;

import org.apache.lucene.search.DocIdSetIterator;

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

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

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.DocIDToTweetIDMapper;

import com.twitter.search.core.earlybird.index.column.ColumnStrideFieldIndex;

/\*\*

\* A few caveats when using this class:

\* - Before actually using this class, one must call prepareToRead() with a Lucene AtomicReader

\* - prepareToRead() will load docID to tweetID mapping into memory, if not already done.

\*/

public class DocValuesBasedTweetIDMapper extends TweetIDMapper implements Flushable {

private LeafReader reader;

private ColumnStrideFieldIndex docValues;

/\*\*

\* When indexing finishes, this method should be called with a index reader that

\* can see all documents.

\* @param leafReader Lucene index reader used to access TweetID to internal ID mapping

\*/

public void initializeWithLuceneReader(LeafReader leafReader, ColumnStrideFieldIndex csf)

throws IOException {

reader = Preconditions.checkNotNull(leafReader);

docValues = Preconditions.checkNotNull(csf);

NumericDocValues onDiskDocValues = reader.getNumericDocValues(

EarlybirdFieldConstants.EarlybirdFieldConstant.ID\_CSF\_FIELD.getFieldName());

for (int i = 0; i < reader.maxDoc(); ++i) {

Preconditions.checkArgument(onDiskDocValues.advanceExact(i));

docValues.setValue(i, onDiskDocValues.longValue());

}

// In the archive, tweets are always sorted in descending order of tweet ID.

setMinTweetID(docValues.get(reader.maxDoc() - 1));

setMaxTweetID(docValues.get(0));

setMinDocID(0);

setMaxDocID(reader.maxDoc() - 1);

setNumDocs(reader.maxDoc());

}

@Override

public int getDocID(long tweetID) throws IOException {

int docId = DocValuesHelper.getFirstDocIdWithValue(

reader,

EarlybirdFieldConstants.EarlybirdFieldConstant.ID\_FIELD.getFieldName(),

SortableLongTermAttributeImpl.copyIntoNewBytesRef(tweetID));

if (docId == DocIdSetIterator.NO\_MORE\_DOCS) {

return ID\_NOT\_FOUND;

}

return docId;

}

@Override

protected int getNextDocIDInternal(int docID) {

// The doc IDs are consecutive and TweetIDMapper already checked the boundary conditions.

return docID + 1;

}

@Override

protected int getPreviousDocIDInternal(int docID) {

// The doc IDs are consecutive and TweetIDMapper already checked the boundary conditions.

return docID - 1;

}

@Override

public long getTweetID(int internalID) {

if (internalID < 0 || internalID > getMaxDocID()) {

return ID\_NOT\_FOUND;

}

return docValues.get(internalID);

}

@Override

protected int addMappingInternal(long tweetID) {

throw new UnsupportedOperationException(

"ArchiveTweetIDMapper should be written through Lucene instead of TweetIDMappingWriter");

}

@Override

protected final int findDocIDBoundInternal(long tweetID,

boolean findMaxDocID) throws IOException {

// TermsEnum has a seekCeil() method, but doesn't have a seekFloor() method, so the best we can

// do here is ignore findLow and always return the ceiling if the tweet ID cannot be found.

// However, in practice, we do a seekExact() in both cases: see the inner classes in

// com.twitter.search.core.earlybird.index.inverted.RealtimeIndexTerms.

int docId = DocValuesHelper.getLargestDocIdWithCeilOfValue(

reader,

EarlybirdFieldConstants.EarlybirdFieldConstant.ID\_FIELD.getFieldName(),

SortableLongTermAttributeImpl.copyIntoNewBytesRef(tweetID));

if (docId == DocIdSetIterator.NO\_MORE\_DOCS) {

return ID\_NOT\_FOUND;

}

// The docId is the upper bound of the search, so if we want the lower bound,

// because doc IDs are dense, we subtract one.

return findMaxDocID ? docId : docId - 1;

}

@Override

public DocIDToTweetIDMapper optimize() {

// DocValuesBasedTweetIDMapper instances are not flushed or loaded,

// so their optimization is a no-op.

return this;

}

@Override

public Flushable.Handler<DocValuesBasedTweetIDMapper> getFlushHandler() {

// EarlybirdIndexSegmentData will still try to flush the DocValuesBasedTweetIDMapper

// for the respective segment, so we need to pass in a DocValuesBasedTweetIDMapper instance to

// this flusher: otherwise, Flushable.Handler.flush() will throw a NullPointerException.

return new FlushHandler(new DocValuesBasedTweetIDMapper());

}

// Full archive earlybirds don't actually flush or load the DocValuesBasedTweetIDMapper. This is

// why doFlush() is a no-op, and doLoad() returns a new DocValuesBasedTweetIDMapper instance

// (initializeWithLuceneReader() will be called at load time to initialize this new

// DocValuesBasedTweetIDMapper instance).

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

public FlushHandler() {

super();

}

public FlushHandler(DocValuesBasedTweetIDMapper objectToFlush) {

super(objectToFlush);

}

@Override

protected void doFlush(FlushInfo flushInfo, DataSerializer out) {

}

@Override

protected DocValuesBasedTweetIDMapper doLoad(FlushInfo flushInfo, DataDeserializer in) {

return new DocValuesBasedTweetIDMapper();

}

}

}