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

import java.io.IOException;

import java.util.ArrayList;

import java.util.Collections;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.DirectoryReader;

import org.apache.lucene.index.IndexWriterConfig;

import org.apache.lucene.index.LeafReader;

import org.apache.lucene.index.LeafReaderContext;

import org.apache.lucene.store.Directory;

import com.twitter.common.collections.Pair;

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

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

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.facets.AbstractFacetCountingArray;

import com.twitter.search.core.earlybird.facets.FacetCountingArrayWriter;

import com.twitter.search.core.earlybird.facets.FacetIDMap;

import com.twitter.search.core.earlybird.facets.FacetLabelProvider;

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

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

import com.twitter.search.core.earlybird.index.extensions.EarlybirdIndexExtensionsData;

import com.twitter.search.core.earlybird.index.extensions.EarlybirdIndexExtensionsFactory;

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

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

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

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

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

/\*\*

\* Base class that references data structures belonging to an Earlybird segment.

\*/

public abstract class EarlybirdIndexSegmentData implements Flushable {

/\*\*

\* This class has a map which contains a snapshot of max published pointers, to distinguish the

\* documents in the skip lists that are fully indexed, and safe to return to searchers and those

\* that are in progress and should not be returned to searchers. See

\* "Earlybird Indexing Latency Design Document"

\* for rationale and design.

\*

\* It also has the smallestDocID, which determines the smallest assigned doc ID in the tweet ID

\* mapper that is safe to traverse.

\*

\* The pointer map and smallestDocID need to be updated atomically. See SEARCH-27650.

\*/

public static class SyncData {

private final Map<InvertedIndex, Integer> indexPointers;

private final int smallestDocID;

public SyncData(Map<InvertedIndex, Integer> indexPointers, int smallestDocID) {

this.indexPointers = indexPointers;

this.smallestDocID = smallestDocID;

}

public Map<InvertedIndex, Integer> getIndexPointers() {

return indexPointers;

}

public int getSmallestDocID() {

return smallestDocID;

}

}

private volatile SyncData syncData;

private final int maxSegmentSize;

private final long timeSliceID;

private final ConcurrentHashMap<String, QueryCacheResultForSegment> queryCacheMap =

new ConcurrentHashMap<>();

private final AbstractFacetCountingArray facetCountingArray;

private final boolean isOptimized;

private final ConcurrentHashMap<String, InvertedIndex> perFieldMap;

private final ConcurrentHashMap<String, ColumnStrideByteIndex> normsMap;

private final Map<String, FacetLabelProvider> facetLabelProviders;

private final FacetIDMap facetIDMap;

private final Schema schema;

private final DocValuesManager docValuesManager;

private final DeletedDocs deletedDocs;

private final DocIDToTweetIDMapper docIdToTweetIdMapper;

private final TimeMapper timeMapper;

static LeafReader getLeafReaderFromOptimizedDirectory(Directory directory) throws IOException {

List<LeafReaderContext> leaves = DirectoryReader.open(directory).getContext().leaves();

int leavesSize = leaves.size();

Preconditions.checkState(1 == leavesSize,

"Expected one leaf reader in directory %s, but found %s", directory, leavesSize);

return leaves.get(0).reader();

}

/\*\*

\* Creates a new SegmentData instance using the provided data.

\*/

public EarlybirdIndexSegmentData(

int maxSegmentSize,

long timeSliceID,

Schema schema,

boolean isOptimized,

int smallestDocID,

ConcurrentHashMap<String, InvertedIndex> perFieldMap,

ConcurrentHashMap<String, ColumnStrideByteIndex> normsMap,

AbstractFacetCountingArray facetCountingArray,

DocValuesManager docValuesManager,

Map<String, FacetLabelProvider> facetLabelProviders,

FacetIDMap facetIDMap,

DeletedDocs deletedDocs,

DocIDToTweetIDMapper docIdToTweetIdMapper,

TimeMapper timeMapper) {

this.maxSegmentSize = maxSegmentSize;

this.timeSliceID = timeSliceID;

this.schema = schema;

this.isOptimized = isOptimized;

this.facetCountingArray = facetCountingArray;

this.perFieldMap = perFieldMap;

this.syncData = new SyncData(buildIndexPointers(), smallestDocID);

this.normsMap = normsMap;

this.docValuesManager = docValuesManager;

this.facetLabelProviders = facetLabelProviders;

this.facetIDMap = facetIDMap;

this.deletedDocs = deletedDocs;

this.docIdToTweetIdMapper = docIdToTweetIdMapper;

this.timeMapper = timeMapper;

Preconditions.checkNotNull(schema);

}

public final Schema getSchema() {

return schema;

}

/\*\*

\* Returns all {@link EarlybirdIndexExtensionsData} instances contained in this segment.

\* Since index extensions are optional, the returned map might be null or empty.

\*/

public abstract <S extends EarlybirdIndexExtensionsData> S getIndexExtensionsData();

public DocIDToTweetIDMapper getDocIDToTweetIDMapper() {

return docIdToTweetIdMapper;

}

public TimeMapper getTimeMapper() {

return timeMapper;

}

public final DocValuesManager getDocValuesManager() {

return docValuesManager;

}

public Map<String, FacetLabelProvider> getFacetLabelProviders() {

return facetLabelProviders;

}

public FacetIDMap getFacetIDMap() {

return facetIDMap;

}

/\*\*

\* Returns the QueryCacheResult for the given filter for this segment.

\*/

public QueryCacheResultForSegment getQueryCacheResult(String queryCacheFilterName) {

return queryCacheMap.get(queryCacheFilterName);

}

public long getQueryCachesCardinality() {

return queryCacheMap.values().stream().mapToLong(q -> q.getCardinality()).sum();

}

/\*\*

\* Get cache cardinality for each query cache.

\* @return

\*/

public List<Pair<String, Long>> getPerQueryCacheCardinality() {

ArrayList<Pair<String, Long>> result = new ArrayList<>();

queryCacheMap.forEach((cacheName, queryCacheResult) -> {

result.add(Pair.of(cacheName, queryCacheResult.getCardinality()));

});

return result;

}

/\*\*

\* Updates the QueryCacheResult stored for the given filter for this segment

\*/

public QueryCacheResultForSegment updateQueryCacheResult(

String queryCacheFilterName, QueryCacheResultForSegment queryCacheResultForSegment) {

return queryCacheMap.put(queryCacheFilterName, queryCacheResultForSegment);

}

/\*\*

\* Subclasses are allowed to return null here to disable writing to a FacetCountingArray.

\*/

public FacetCountingArrayWriter createFacetCountingArrayWriter() {

return getFacetCountingArray() != null

? new FacetCountingArrayWriter(getFacetCountingArray()) : null;

}

public int getMaxSegmentSize() {

return maxSegmentSize;

}

public long getTimeSliceID() {

return timeSliceID;

}

public void updateSmallestDocID(int smallestDocID) {

// Atomic swap

syncData = new SyncData(Collections.unmodifiableMap(buildIndexPointers()), smallestDocID);

}

private Map<InvertedIndex, Integer> buildIndexPointers() {

Map<InvertedIndex, Integer> newIndexPointers = new HashMap<>();

for (InvertedIndex index : perFieldMap.values()) {

if (index.hasMaxPublishedPointer()) {

newIndexPointers.put(index, index.getMaxPublishedPointer());

}

}

return newIndexPointers;

}

public SyncData getSyncData() {

return syncData;

}

public AbstractFacetCountingArray getFacetCountingArray() {

return facetCountingArray;

}

public void addField(String fieldName, InvertedIndex field) {

perFieldMap.put(fieldName, field);

}

public Map<String, InvertedIndex> getPerFieldMap() {

return Collections.unmodifiableMap(perFieldMap);

}

public InvertedIndex getFieldIndex(String fieldName) {

return perFieldMap.get(fieldName);

}

public Map<String, ColumnStrideByteIndex> getNormsMap() {

return Collections.unmodifiableMap(normsMap);

}

public DeletedDocs getDeletedDocs() {

return deletedDocs;

}

/\*\*

\* Returns the norms index for the given field name.

\*/

public ColumnStrideByteIndex getNormIndex(String fieldName) {

return normsMap == null ? null : normsMap.get(fieldName);

}

/\*\*

\* Returns the norms index for the given field name, add if not exist.

\*/

public ColumnStrideByteIndex createNormIndex(String fieldName) {

if (normsMap == null) {

return null;

}

ColumnStrideByteIndex csf = normsMap.get(fieldName);

if (csf == null) {

csf = new ColumnStrideByteIndex(fieldName, maxSegmentSize);

normsMap.put(fieldName, csf);

}

return csf;

}

/\*\*

\* Flushes this segment to disk.

\*/

public void flushSegment(FlushInfo flushInfo, DataSerializer out) throws IOException {

getFlushHandler().flush(flushInfo, out);

}

public final boolean isOptimized() {

return this.isOptimized;

}

/\*\*

\* Returns a new atomic reader for this segment.

\*/

public EarlybirdIndexSegmentAtomicReader createAtomicReader() throws IOException {

EarlybirdIndexSegmentAtomicReader reader = doCreateAtomicReader();

EarlybirdIndexExtensionsData indexExtension = getIndexExtensionsData();

if (indexExtension != null) {

indexExtension.setupExtensions(reader);

}

return reader;

}

/\*\*

\* Creates a new atomic reader for this segment.

\*/

protected abstract EarlybirdIndexSegmentAtomicReader doCreateAtomicReader() throws IOException;

/\*\*

\* Creates a new segment writer for this segment.

\*/

public abstract EarlybirdIndexSegmentWriter createEarlybirdIndexSegmentWriter(

IndexWriterConfig indexWriterConfig) throws IOException;

public abstract static class AbstractSegmentDataFlushHandler

<S extends EarlybirdIndexExtensionsData>

extends Flushable.Handler<EarlybirdIndexSegmentData> {

protected static final String MAX\_SEGMENT\_SIZE\_PROP\_NAME = "maxSegmentSize";

protected static final String TIME\_SLICE\_ID\_PROP\_NAME = "time\_slice\_id";

protected static final String SMALLEST\_DOCID\_PROP\_NAME = "smallestDocID";

protected static final String DOC\_ID\_MAPPER\_SUBPROPS\_NAME = "doc\_id\_mapper";

protected static final String TIME\_MAPPER\_SUBPROPS\_NAME = "time\_mapper";

public static final String IS\_OPTIMIZED\_PROP\_NAME = "isOptimized";

// Abstract methods child classes should implement:

// 1. How to additional data structures

protected abstract void flushAdditionalDataStructures(

FlushInfo flushInfo, DataSerializer out, EarlybirdIndexSegmentData toFlush)

throws IOException;

// 2. Load additional data structures and construct SegmentData.

// Common data structures should be passed into this method to avoid code duplication.

// Subclasses should load additional data structures and construct a SegmentData.

protected abstract EarlybirdIndexSegmentData constructSegmentData(

FlushInfo flushInfo,

ConcurrentHashMap<String, InvertedIndex> perFieldMap,

int maxSegmentSize,

S indexExtension,

DocIDToTweetIDMapper docIdToTweetIdMapper,

TimeMapper timeMapper,

DataDeserializer in) throws IOException;

protected abstract S newIndexExtension();

protected final Schema schema;

protected final EarlybirdIndexExtensionsFactory indexExtensionsFactory;

private final Flushable.Handler<? extends DocIDToTweetIDMapper> docIdMapperFlushHandler;

private final Flushable.Handler<? extends TimeMapper> timeMapperFlushHandler;

public AbstractSegmentDataFlushHandler(

Schema schema,

EarlybirdIndexExtensionsFactory indexExtensionsFactory,

Flushable.Handler<? extends DocIDToTweetIDMapper> docIdMapperFlushHandler,

Flushable.Handler<? extends TimeMapper> timeMapperFlushHandler) {

super();

this.schema = schema;

this.indexExtensionsFactory = indexExtensionsFactory;

this.docIdMapperFlushHandler = docIdMapperFlushHandler;

this.timeMapperFlushHandler = timeMapperFlushHandler;

}

public AbstractSegmentDataFlushHandler(EarlybirdIndexSegmentData objectToFlush) {

super(objectToFlush);

this.schema = objectToFlush.schema;

this.indexExtensionsFactory = null; // factory only needed for loading SegmentData from disk

this.docIdMapperFlushHandler = null; // docIdMapperFlushHandler needed only for loading data

this.timeMapperFlushHandler = null; // timeMapperFlushHandler needed only for loading data

}

@Override

protected void doFlush(FlushInfo flushInfo, DataSerializer out)

throws IOException {

EarlybirdIndexSegmentData segmentData = getObjectToFlush();

Preconditions.checkState(segmentData.docIdToTweetIdMapper instanceof Flushable);

((Flushable) segmentData.docIdToTweetIdMapper).getFlushHandler().flush(

flushInfo.newSubProperties(DOC\_ID\_MAPPER\_SUBPROPS\_NAME), out);

if (segmentData.timeMapper != null) {

segmentData.timeMapper.getFlushHandler()

.flush(flushInfo.newSubProperties(TIME\_MAPPER\_SUBPROPS\_NAME), out);

}

flushInfo.addBooleanProperty(IS\_OPTIMIZED\_PROP\_NAME, segmentData.isOptimized());

flushInfo.addIntProperty(MAX\_SEGMENT\_SIZE\_PROP\_NAME, segmentData.getMaxSegmentSize());

flushInfo.addLongProperty(TIME\_SLICE\_ID\_PROP\_NAME, segmentData.getTimeSliceID());

flushInfo.addIntProperty(SMALLEST\_DOCID\_PROP\_NAME,

segmentData.getSyncData().getSmallestDocID());

flushIndexes(flushInfo, out, segmentData);

// Flush cluster specific data structures:

// FacetCountingArray, TweetIDMapper, LatLonMapper, and TimeMapper

flushAdditionalDataStructures(flushInfo, out, segmentData);

}

private void flushIndexes(

FlushInfo flushInfo,

DataSerializer out,

EarlybirdIndexSegmentData segmentData) throws IOException {

Map<String, InvertedIndex> perFieldMap = segmentData.getPerFieldMap();

FlushInfo fieldProps = flushInfo.newSubProperties("fields");

long sizeBeforeFlush = out.length();

for (Map.Entry<String, InvertedIndex> entry : perFieldMap.entrySet()) {

String fieldName = entry.getKey();

entry.getValue().getFlushHandler().flush(fieldProps.newSubProperties(fieldName), out);

}

fieldProps.setSizeInBytes(out.length() - sizeBeforeFlush);

}

@Override

protected EarlybirdIndexSegmentData doLoad(FlushInfo flushInfo, DataDeserializer in)

throws IOException {

DocIDToTweetIDMapper docIdToTweetIdMapper = docIdMapperFlushHandler.load(

flushInfo.getSubProperties(DOC\_ID\_MAPPER\_SUBPROPS\_NAME), in);

FlushInfo timeMapperFlushInfo = flushInfo.getSubProperties(TIME\_MAPPER\_SUBPROPS\_NAME);

TimeMapper timeMapper =

timeMapperFlushInfo != null ? timeMapperFlushHandler.load(timeMapperFlushInfo, in) : null;

final int maxSegmentSize = flushInfo.getIntProperty(MAX\_SEGMENT\_SIZE\_PROP\_NAME);

ConcurrentHashMap<String, InvertedIndex> perFieldMap = loadIndexes(flushInfo, in);

return constructSegmentData(

flushInfo,

perFieldMap,

maxSegmentSize,

newIndexExtension(),

docIdToTweetIdMapper,

timeMapper,

in);

}

// Move this method into EarlybirdRealtimeIndexSegmentData (careful,

// we may need to increment FlushVersion because EarlybirdLuceneIndexSegmentData

// currently has the 'fields' subproperty in its FlushInfo as well)

private ConcurrentHashMap<String, InvertedIndex> loadIndexes(

FlushInfo flushInfo, DataDeserializer in) throws IOException {

ConcurrentHashMap<String, InvertedIndex> perFieldMap = new ConcurrentHashMap<>();

FlushInfo fieldProps = flushInfo.getSubProperties("fields");

Iterator<String> fieldIterator = fieldProps.getKeyIterator();

while (fieldIterator.hasNext()) {

String fieldName = fieldIterator.next();

EarlybirdFieldType fieldType = schema.getFieldInfo(fieldName).getFieldType();

FlushInfo subProp = fieldProps.getSubProperties(fieldName);

boolean isOptimized = subProp.getBooleanProperty(

OptimizedMemoryIndex.FlushHandler.IS\_OPTIMIZED\_PROP\_NAME);

final InvertedIndex invertedIndex;

if (isOptimized) {

if (!fieldType.becomesImmutable()) {

throw new IOException("Tried to load an optimized field that is not immutable: "

+ fieldName);

}

invertedIndex = (new OptimizedMemoryIndex.FlushHandler(fieldType)).load(subProp, in);

} else {

invertedIndex = (new InvertedRealtimeIndex.FlushHandler(

fieldType, TermPointerEncoding.DEFAULT\_ENCODING))

.load(subProp, in);

}

perFieldMap.put(fieldName, invertedIndex);

}

return perFieldMap;

}

}

public int numDocs() {

return docIdToTweetIdMapper.getNumDocs();

}

}