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

import java.io.IOException;

import java.util.Arrays;

import org.apache.lucene.store.DataInput;

import org.apache.lucene.store.DataOutput;

import org.apache.lucene.util.ArrayUtil;

import org.apache.lucene.util.ByteBlockPool;

import org.apache.lucene.util.BytesRef;

import static org.apache.lucene.util.RamUsageEstimator.NUM\_BYTES\_OBJECT\_REF;

/\*\*

\* Base class for BlockPools backed by byte[] arrays.

\*/

public abstract class BaseByteBlockPool {

/\*\*

\* The extra object with final array is necessary to guarantee visibility to

\* other threads without synchronization/using volatile.

\*

\* From 'Java Concurrency in practice' by Brian Goetz, p. 349:

\*

\* "Initialization safety guarantees that for properly constructed objects, all

\* threads will see the correct values of final fields that were set by the con-

\* structor, regardless of how the object is published. Further, any variables

\* that can be reached through a final field of a properly constructed object

\* (such as the elements of a final array or the contents of a HashMap refer-

\* enced by a final field) are also guaranteed to be visible to other threads."

\*/

public static final class Pool {

public final byte[][] buffers;

public Pool(byte[][] buffers) {

this.buffers = buffers;

}

public byte[][] getBlocks() {

return buffers;

}

}

public Pool pool = new Pool(new byte[10][]);

// The index of the current buffer in pool.buffers.

public int bufferUpto = -1;

// The number of bytes that have been written in the current buffer.

public int byteUpto = ByteBlockPool.BYTE\_BLOCK\_SIZE;

// The current buffer, i.e. a reference to pool.buffers[bufferUpto]

public byte[] buffer;

// The total number of bytes that have been used up to now, excluding the current buffer.

public int byteOffset = -ByteBlockPool.BYTE\_BLOCK\_SIZE;

// The one and only WriteStream for this pool.

private WriteStream writeStream = new WriteStream();

protected BaseByteBlockPool() { }

/\*\*

\* Used for loading flushed pool.

\*/

protected BaseByteBlockPool(Pool pool, int bufferUpto, int byteUpTo, int byteOffset) {

this.pool = pool;

this.bufferUpto = bufferUpto;

this.byteUpto = byteUpTo;

this.byteOffset = byteOffset;

if (bufferUpto >= 0) {

this.buffer = pool.buffers[bufferUpto];

}

}

/\*\*

\* Resets the index of the pool to 0 in the first buffer and resets the byte arrays of

\* all previously allocated buffers to 0s.

\*/

public void reset() {

if (bufferUpto != -1) {

// We allocated at least one buffer

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

// Fully zero fill buffers that we fully used

Arrays.fill(pool.buffers[i], (byte) 0);

}

// Partial zero fill the final buffer

Arrays.fill(pool.buffers[bufferUpto], 0, byteUpto, (byte) 0);

bufferUpto = 0;

byteUpto = 0;

byteOffset = 0;

buffer = pool.buffers[0];

}

}

/\*\*

\* Switches to the next buffer and positions the index at its beginning.

\*/

public void nextBuffer() {

if (1 + bufferUpto == pool.buffers.length) {

byte[][] newBuffers = new byte[ArrayUtil.oversize(pool.buffers.length + 1,

NUM\_BYTES\_OBJECT\_REF)][];

System.arraycopy(pool.buffers, 0, newBuffers, 0, pool.buffers.length);

pool = new Pool(newBuffers);

}

buffer = pool.buffers[1 + bufferUpto] = new byte[ByteBlockPool.BYTE\_BLOCK\_SIZE];

bufferUpto++;

byteUpto = 0;

byteOffset += ByteBlockPool.BYTE\_BLOCK\_SIZE;

}

/\*\*

\* Returns the start offset of the next data that will be added to the pool, UNLESS the data is

\* added using addBytes and avoidSplitting = true

\*/

public int getOffset() {

return byteOffset + byteUpto;

}

/\*\*

\* Returns the start offset of b in the pool

\* @param b byte to put

\*/

public int addByte(byte b) {

int initOffset = byteOffset + byteUpto;

int remainingBytesInBuffer = ByteBlockPool.BYTE\_BLOCK\_SIZE - byteUpto;

// If the buffer is full, move on to the next one.

if (remainingBytesInBuffer <= 0) {

nextBuffer();

}

buffer[byteUpto] = b;

byteUpto++;

return initOffset;

}

/\*\*

\* Returns the start offset of the bytes in the pool.

\* If avoidSplitting is false, this is guaranteed to return the same value that would be

\* returned by getOffset()

\* @param bytes source array

\* @param length number of bytes to put

\* @param avoidSplitting if possible (the length is less than ByteBlockPool.BYTE\_BLOCK\_SIZE),

\* the bytes will not be split across buffer boundaries. This is useful for small data

\* that will be read a lot (small amount of space wasted in return for avoiding copying

\* memory when calling getBytes).

\*/

public int addBytes(byte[] bytes, int offset, int length, boolean avoidSplitting) {

// The first time this is called, there may not be an existing buffer yet.

if (buffer == null) {

nextBuffer();

}

int remainingBytesInBuffer = ByteBlockPool.BYTE\_BLOCK\_SIZE - byteUpto;

if (avoidSplitting && length < ByteBlockPool.BYTE\_BLOCK\_SIZE) {

if (remainingBytesInBuffer < length) {

nextBuffer();

}

int initOffset = byteOffset + byteUpto;

System.arraycopy(bytes, offset, buffer, byteUpto, length);

byteUpto += length;

return initOffset;

} else {

int initOffset = byteOffset + byteUpto;

if (remainingBytesInBuffer < length) {

// Must split the bytes across buffers.

int remainingLength = length;

while (remainingLength > ByteBlockPool.BYTE\_BLOCK\_SIZE - byteUpto) {

int lengthToCopy = ByteBlockPool.BYTE\_BLOCK\_SIZE - byteUpto;

System.arraycopy(bytes, length - remainingLength + offset,

buffer, byteUpto, lengthToCopy);

remainingLength -= lengthToCopy;

nextBuffer();

}

System.arraycopy(bytes, length - remainingLength + offset,

buffer, byteUpto, remainingLength);

byteUpto += remainingLength;

} else {

// Just add all bytes to the current buffer.

System.arraycopy(bytes, offset, buffer, byteUpto, length);

byteUpto += length;

}

return initOffset;

}

}

/\*\*

\* Default addBytes. Does not avoid splitting.

\* @see #addBytes(byte[], int, boolean)

\*/

public int addBytes(byte[] bytes, int length) {

return addBytes(bytes, 0, length, false);

}

/\*\*

\* Default addBytes. Does not avoid splitting.

\* @see #addBytes(byte[], int, boolean)

\*/

public int addBytes(byte[] bytes, int offset, int length) {

return addBytes(bytes, offset, length, false);

}

/\*\*

\* Reads one byte from the pool.

\* @param offset location to read byte from

\*/

public byte getByte(int offset) {

int bufferIndex = offset >>> ByteBlockPool.BYTE\_BLOCK\_SHIFT;

int bufferOffset = offset & ByteBlockPool.BYTE\_BLOCK\_MASK;

return pool.buffers[bufferIndex][bufferOffset];

}

/\*\*

\* Returns false if offset is invalid or there aren't these many bytes

\* available in the pool.

\* @param offset location to start reading bytes from

\* @param length number of bytes to read

\* @param output the object to write the output to. MUST be non null.

\*/

public boolean getBytesToBytesRef(int offset, int length, BytesRef output) {

if (offset < 0 || offset + length > byteUpto + byteOffset) {

return false;

}

int currentBuffer = offset >>> ByteBlockPool.BYTE\_BLOCK\_SHIFT;

int currentOffset = offset & ByteBlockPool.BYTE\_BLOCK\_MASK;

// If the requested bytes are split across pools, we have to make a new array of bytes

// to copy them into and return a ref to that.

if (currentOffset + length <= ByteBlockPool.BYTE\_BLOCK\_SIZE) {

output.bytes = pool.buffers[currentBuffer];

output.offset = currentOffset;

output.length = length;

} else {

byte[] bytes = new byte[length];

int remainingLength = length;

while (remainingLength > ByteBlockPool.BYTE\_BLOCK\_SIZE - currentOffset) {

int lengthToCopy = ByteBlockPool.BYTE\_BLOCK\_SIZE - currentOffset;

System.arraycopy(pool.buffers[currentBuffer], currentOffset, bytes,

length - remainingLength, lengthToCopy);

remainingLength -= lengthToCopy;

currentBuffer++;

currentOffset = 0;

}

System.arraycopy(pool.buffers[currentBuffer], currentOffset, bytes, length - remainingLength,

remainingLength);

output.bytes = bytes;

output.length = bytes.length;

output.offset = 0;

}

return true;

}

/\*\*

\* Returns the read bytes, or null if offset is invalid or there aren't these many bytes

\* available in the pool.

\* @param offset location to start reading bytes from

\* @param length number of bytes to read

\*/

public BytesRef getBytes(int offset, int length) {

BytesRef result = new BytesRef();

if (getBytesToBytesRef(offset, length, result)) {

return result;

} else {

return null;

}

}

/\*\*

\* get a new readStream at a given offset for this pool.

\*

\* Notice that individual ReadStreams are not threadsafe, but you can get as many ReadStreams as

\* you want.

\*/

public ReadStream getReadStream(int offset) {

return new ReadStream(offset);

}

/\*\*

\* get the (one and only) WriteStream for this pool.

\*

\* Notice that there is exactly one WriteStream per pool, and it is not threadsafe.

\*/

public WriteStream getWriteStream() {

return writeStream;

}

/\*\*

\* A DataOutput-like interface for writing "contiguous" data to a ByteBlockPool.

\*

\* This is not threadsafe.

\*/

public final class WriteStream extends DataOutput {

private WriteStream() { }

/\*\*

\* Returns the start offset of the next data that will be added to the pool, UNLESS the data is

\* added using addBytes and avoidSplitting = true

\*/

public int getOffset() {

return BaseByteBlockPool.this.getOffset();

}

/\*\*

\* Write bytes to the pool.

\* @param bytes source array

\* @param offset offset in bytes of the data to write

\* @param length number of bytes to put

\* @param avoidSplitting same as {link ByteBlockPool.addBytes}

\* @return the start offset of the bytes in the pool

\*/

public int writeBytes(byte[] bytes, int offset, int length, boolean avoidSplitting) {

return addBytes(bytes, offset, length, avoidSplitting);

}

@Override

public void writeBytes(byte[] b, int offset, int length) throws IOException {

addBytes(b, offset, length);

}

@Override

public void writeByte(byte b) {

addByte(b);

}

}

/\*\*

\* A DataInput-like interface for reading "contiguous" data from a ByteBlockPool.

\*

\* This is not threadsafe.

\*

\* This does not fully implement the DataInput interface - its DataInput.readBytes method throws

\* UnsupportedOperationException because this class provides a facility for no-copy reading.

\*/

public final class ReadStream extends DataInput {

private int offset;

private ReadStream(int offset) {

this.offset = offset;

}

public BytesRef readBytes(int n) {

return readBytes(n, false);

}

/\*\*

\* read n bytes that were written with a given value of avoidSplitting

\* @param n number of bytes to read.

\* @param avoidSplitting this should be the same that was used at writeBytes time.

\* @return a reference to the bytes read or null.

\*/

public BytesRef readBytes(int n, boolean avoidSplitting) {

int currentBuffer = offset >>> ByteBlockPool.BYTE\_BLOCK\_SHIFT;

int currentOffset = offset & ByteBlockPool.BYTE\_BLOCK\_MASK;

if (avoidSplitting && n < ByteBlockPool.BYTE\_BLOCK\_SIZE

&& currentOffset + n > ByteBlockPool.BYTE\_BLOCK\_SIZE) {

++currentBuffer;

currentOffset = 0;

offset = currentBuffer << ByteBlockPool.BYTE\_BLOCK\_SHIFT;

}

BytesRef result = getBytes(offset, n);

this.offset += n;

return result;

}

@Override

public byte readByte() {

return getByte(offset++);

}

@Override

public void readBytes(byte[] b, int off, int len) throws IOException {

throw new UnsupportedOperationException("Use the no-copies version of ReadBytes instead.");

}

}

}