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

/\*\*

\* Pre-computed shifts, mask, and start int indices used by

\* {@link IntBlockPoolPackedLongsReader} to decode packed values from

\* {@link IntBlockPool}.

\*

\* The purpose of this class is for decoding efficiency and speed. This class is thread-safe since

\* all its usages are read-only.

\*

\* Packed ints are stored from LOWEST bits for HIGHEST bits in an int.

\*

\* Here are 3 different situations when a packed value spans 1, 2, and 3 ints:

\*

\* - A packed value spans 1 int:

\* [High Bits ................................. Low Bits]

\* int[n] = [possible\_other\_data|packed\_value|possible\_other\_data]

\*

\* To decode, 1 shift right and 1 mask are needed:

\* \* shift - {@link #allLowBitsRightShift}

\* \* mask - dynamically computed based on bitsPerValue (in decoded slice).

\*

\* - A packed value spans 2 ints:

\* The data is stored as:

\* [High Bits .................. Low Bits]

\* int[n] = [low\_bits\_of\_packed\_value | other\_data]

\* int[n+1] = [other\_data| high\_bits\_of\_packed\_value]

\*

\* To decode, 1 shift right, 1 shift left, and 2 masks are needed:

\* \* 1 shift right {@link #allLowBitsRightShift} and 1 mask (computed on the fly) to compute

\* low\_bits\_of\_packed\_value

\* \* 1 mask {@link #allMiddleBitsMask} and 1 shift left {@link #allMiddleBitsLeftShift} to

\* compute high\_bits\_of\_packed\_value

\* \* 1 OR to combine `high\_bits\_of\_packed\_value | low\_bits\_of\_packed\_value`

\*

\* - A packed value spans 3 ints:

\* The data is stored as:

\* [High Bits .................. Low Bits]

\* int[n] = [low\_bits\_of\_packed\_value | other\_data]

\* int[n+1] = [ ... middle\_bits\_of\_packed\_value ... ]

\* int[n+2] = [other\_data| high\_bits\_of\_packed\_value]

\*

\* To decode, 1 shift right, 2 shift left, and 3 masks are needed:

\* \* 1 shift right {@link #allLowBitsRightShift} and 1 mask (computed on the fly) to compute

\* low\_bits\_of\_packed\_value

\* \* 1 shift left {@link #allMiddleBitsLeftShift} and 1 mask {@link #allMiddleBitsMask} to

\* compute middle\_bits\_of\_data

\* \* 1 shift left {@link #allHighBitsLeftShift} and 1 mask {@link #allHighBitsMask} to compute

\* high\_bits\_of\_data

\* \* 1 OR to combine `low\_bits\_of\_data | middle\_bits\_of\_data | high\_bits\_of\_data`

\*

\* Example usage:

\* @see HighDFPackedIntsDocsEnum

\* @see HighDFPackedIntsDocsAndPositionsEnum

\*/

public final class PackedLongsReaderPreComputedValues {

private final int[][] allLowBitsRightShift;

private final int[][] allMiddleBitsLeftShift;

private final int[][] allMiddleBitsMask;

private final int[][] allHighBitsLeftShift;

private final int[][] allHighBitsMask;

/\*\*

\* 2D int arrays containing pre-computed start int indices; the 2 dimensions are

\* int[numBitsPerPackedValue][packedValueIndex].

\*

\* For a given number bits per packed value and a given packed value index, this is the first

\* int in the subsequent of ints that contains the packed value with the given packed value index.

\*/

private final int[][] allStartIntIndices;

/\*\*

\* Sole constructor.

\*

\* @param maxBitsPerValue max possible number of bits of packed values that will be decoded

\* @param maxNumValues max number of values are encoded back to back

\* @param maxNumInts max number of ints are used to store packed values

\* @param needStartIntIndex for optimization: whether start int indices are needed

\*/

PackedLongsReaderPreComputedValues(

int maxBitsPerValue,

int maxNumValues,

int maxNumInts,

boolean needStartIntIndex) {

assert maxBitsPerValue <= Long.SIZE;

if (needStartIntIndex) {

this.allStartIntIndices = new int[maxBitsPerValue + 1][maxNumValues];

} else {

this.allStartIntIndices = null;

}

this.allLowBitsRightShift = new int[maxBitsPerValue + 1][maxNumValues];

this.allMiddleBitsLeftShift = new int[maxBitsPerValue + 1][maxNumValues];

this.allMiddleBitsMask = new int[maxBitsPerValue + 1][maxNumValues];

// Packed value could use up 2 ints.

if (maxBitsPerValue > Integer.SIZE) {

this.allHighBitsLeftShift = new int[maxBitsPerValue + 1][maxNumValues];

this.allHighBitsMask = new int[maxBitsPerValue + 1][maxNumValues];

} else {

this.allHighBitsLeftShift = null;

this.allHighBitsMask = null;

}

compute(maxBitsPerValue, maxNumValues, maxNumInts);

}

/\*\*

\* Compute masks, shifts and start indices.

\*/

private void compute(int maxBitsPerValue, int maxNumValues, int maxNumInts) {

// For each possible bits per packed value.

for (int bitsPerPackedValue = 0; bitsPerPackedValue <= maxBitsPerValue; bitsPerPackedValue++) {

int[] startIntIndices =

allStartIntIndices != null ? allStartIntIndices[bitsPerPackedValue] : null;

int[] lowBitsRightShift =

allLowBitsRightShift[bitsPerPackedValue];

int[] middleBitsLeftShift =

allMiddleBitsLeftShift[bitsPerPackedValue];

int[] middleBitsMask =

allMiddleBitsMask[bitsPerPackedValue];

int[] highBitsLeftShift =

allHighBitsLeftShift != null ? allHighBitsLeftShift[bitsPerPackedValue] : null;

int[] highBitsMask =

allHighBitsMask != null ? allHighBitsMask[bitsPerPackedValue] : null;

int shift = 0;

int currentIntIndex = 0;

int bitsRead;

int bitsRemaining;

// For each packed value.

for (int packedValueIndex = 0; packedValueIndex < maxNumValues; packedValueIndex++) {

if (startIntIndices != null) {

startIntIndices[packedValueIndex] = currentIntIndex;

}

// Packed value spans to the 1st int.

lowBitsRightShift[packedValueIndex] = shift;

bitsRead = Integer.SIZE - shift;

bitsRemaining = bitsPerPackedValue - bitsRead;

if (bitsRemaining >= 0) {

// Packed value spans to the 2nd int.

currentIntIndex++;

if (currentIntIndex == maxNumInts) {

break;

}

middleBitsLeftShift[packedValueIndex] = bitsRead;

middleBitsMask[packedValueIndex] =

bitsRemaining >= Integer.SIZE ? 0xFFFFFFFF : (1 << bitsRemaining) - 1;

// Packed value spans to the 3rd int.

bitsRead += Integer.SIZE;

bitsRemaining -= Integer.SIZE;

if (bitsRemaining >= 0) {

currentIntIndex++;

if (currentIntIndex == maxNumInts) {

break;

}

assert highBitsLeftShift != null;

assert highBitsMask != null;

highBitsLeftShift[packedValueIndex] = bitsRead;

highBitsMask[packedValueIndex] =

bitsRemaining >= Integer.SIZE ? 0xFFFFFFFF : (1 << bitsRemaining) - 1;

}

}

shift += bitsPerPackedValue;

shift = shift % Integer.SIZE;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Getters of Pre-computed Values: returns should NEVER be modified \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int[] getStartIntIndices(int numBitsPerValue) {

return allStartIntIndices == null ? null : allStartIntIndices[numBitsPerValue];

}

int[] getLowBitsRightShift(int numBitsPerValue) {

return allLowBitsRightShift[numBitsPerValue];

}

int[] getMiddleBitsLeftShift(int numBitsPerValue) {

return allMiddleBitsLeftShift[numBitsPerValue];

}

int[] getMiddleBitsMask(int numBitsPerValue) {

return allMiddleBitsMask[numBitsPerValue];

}

int[] getHighBitsLeftShift(int numBitsPerValue) {

return allHighBitsLeftShift == null ? null : allHighBitsLeftShift[numBitsPerValue];

}

int[] getHighBitsMask(int numBitsPerValue) {

return allHighBitsMask == null ? null : allHighBitsMask[numBitsPerValue];

}

}