package com.twitter.search.common.encoding.features;

/\*\*

\* Util used to:

\* - Encode a positive Java float into a single byte float

\* - Decode a single byte into a positive Java float

\*

\* Configuration:

\* - Exponent: higher 4 bits, base 10.

\* - Mantissa: lower 4 bit, representing 1.0 to 9.0

\* - Exponent bias is 1.

\*

\* Formula:

\* Max(Mantissa, 9) \* 10 ^ (Exponent - 1)

\*

\* Smallest float: 0.0 (0000 0000)

\* Smallest positive float: 1.0 \* 10^-1 (0000 0001)

\* Largest float: 9.0 \* 10^13 (1110 1111)

\* Infinity: (1111 0000)

\* NaN: (1111 1000)

\*/

public final class SingleBytePositiveFloatUtil {

private SingleBytePositiveFloatUtil() { }

// 4 bits mantissa. Range [1.0, 10.0) is divided into 16 steps

public static final byte MAX\_BYTE\_VALUE = (byte) 0xEF;

public static final byte INFINITY = (byte) 0xF0;

public static final byte NOT\_A\_NUMBER = (byte) 0xF8;

private static final float STEP\_SIZE = 1.0f;

private static final int EXPONENT\_BIAS = 1;

private static final byte MIN\_EXPONENT = -EXPONENT\_BIAS;

private static final int MAX\_EXPONENT = 14 - EXPONENT\_BIAS;

private static final byte MANTISSA\_MASK = 0x0F;

/\*\*

\* Converts the given float into a single byte floating point number.

\* This is used in the updater and OK to be a bit slow.

\*/

public static byte toSingleBytePositiveFloat(float f) {

if (f < 0) {

throw new UnsupportedOperationException(

"Cannot encode negative floats into SingleBytePostiveFloat.");

}

if (Float.compare(f, Float.POSITIVE\_INFINITY) == 0) {

return INFINITY;

}

if (Float.compare(f, Float.NaN) == 0) {

return NOT\_A\_NUMBER;

}

int mantissa = 0;

int exponent = (int) Math.floor(Math.log10(f));

// Overflow (Number too large), just return the largest possible value

if (exponent > MAX\_EXPONENT) {

return MAX\_BYTE\_VALUE;

}

// Underflow (Number too small), just return 0

if (exponent < MIN\_EXPONENT) {

return 0;

}

int frac = Math.round(f / (float) Math.pow(10.0f, exponent) / STEP\_SIZE);

mantissa = fractionToMantissaTable[frac];

return (byte) (((exponent + EXPONENT\_BIAS) << 4) | mantissa);

}

/\*\*

\* Called in Earlybird per hit and needs to be fast.

\*/

public static float toJavaFloat(byte b) {

return BYTE\_TO\_FLOAT\_CONVERSION\_TABLE[b & 0xff];

}

// Table used for converting mantissa into a significant

private static float[] mantissaToFractionTable = {

// Decimal Matisa value

STEP\_SIZE \* 0, // 0000

STEP\_SIZE \* 1, // 0001

STEP\_SIZE \* 1, // 0010

STEP\_SIZE \* 2, // 0011

STEP\_SIZE \* 2, // 0100

STEP\_SIZE \* 3, // 0101

STEP\_SIZE \* 3, // 0110

STEP\_SIZE \* 4, // 0111

STEP\_SIZE \* 4, // 1000

STEP\_SIZE \* 5, // 1001

STEP\_SIZE \* 5, // 1010

STEP\_SIZE \* 6, // 1011

STEP\_SIZE \* 6, // 1100

STEP\_SIZE \* 7, // 1101

STEP\_SIZE \* 8, // 1110

STEP\_SIZE \* 9 // 1111

};

// Table used for converting fraction into mantissa.

// Reverse operation of the above

private static int[] fractionToMantissaTable = {

0, // 0

1, // 1

3, // 2

5, // 3

7, // 4

9, // 5

11, // 6

13, // 7

14, // 8

15, // 9

15, // 10 (Edge case: because we round the fraction, we can get 10 here.)

};

public static final byte LARGEST\_FRACTION\_UNDER\_ONE = (byte) (toSingleBytePositiveFloat(1f) - 1);

/\*\*

\* Converts the given byte to java float.

\*/

private static float toJavaFloatSlow(byte b) {

if (b == INFINITY) {

return Float.POSITIVE\_INFINITY;

}

if ((b & 0xff) > (INFINITY & 0xff)) {

return Float.NaN;

}

int exponent = ((b & 0xff) >>> 4) - EXPONENT\_BIAS;

int mantissa = b & MANTISSA\_MASK;

return mantissaToFractionTable[mantissa] \* (float) Math.pow(10.0f, exponent);

}

// Cached results from byte to float conversion

private static final float[] BYTE\_TO\_FLOAT\_CONVERSION\_TABLE = new float[256];

private static final double[] BYTE\_TO\_LOG2\_CONVERSION\_TABLE = new double[256];

private static final byte[] OLD\_TO\_NEW\_BYTE\_CONVERSION\_TABLE = new byte[256];

static {

LogByteNormalizer normalizer = new LogByteNormalizer();

for (int i = 0; i < 256; i++) {

byte b = (byte) i;

BYTE\_TO\_FLOAT\_CONVERSION\_TABLE[i] = toJavaFloatSlow(b);

BYTE\_TO\_LOG2\_CONVERSION\_TABLE[i] =

0xff & normalizer.normalize(BYTE\_TO\_FLOAT\_CONVERSION\_TABLE[i]);

if (b == 0) {

OLD\_TO\_NEW\_BYTE\_CONVERSION\_TABLE[i] = 0;

} else if (b > 0) {

OLD\_TO\_NEW\_BYTE\_CONVERSION\_TABLE[i] =

toSingleBytePositiveFloat((float) normalizer.unnormLowerBound(b));

} else {

// should not get here.

OLD\_TO\_NEW\_BYTE\_CONVERSION\_TABLE[i] = MAX\_BYTE\_VALUE;

}

}

}

/\*\*

\* Convert a normalized byte to the log2() version of its original value

\*/

static double toLog2Double(byte b) {

return BYTE\_TO\_LOG2\_CONVERSION\_TABLE[b & 0xff];

}

}