Note: this is a copy of the original document from Mehant Baid (mehant@apache.org)

1. Introduction:

This document covers the design specification for the decimal data type, specifically the decimal data type with precision > 18. We discuss how the data would be encoded in the vectors and the corresponding holder representation.

One important idea for Decimal data type moving forward is to be able to take advantage of the native (c++) libraries to perform mathematical operations which will be faster than the BigDecimal implementation. In order to require no rework of the functions we need to define a standard api which the Drill function would invoke and we can move to the native implementation seamlessly.

This document also covers how the scale and precision of the output of various mathematical operations are computed and how we will handle overflows. Finally we cover how the native implementation will work.

2. Encoding:

Parquet uses 128 bit 2's complement (big endian) to store the unscaled decimal value. In addition it stores the scale and precision on a per column basis. Drill's internal representation of decimal in Vectors would match that of parquet. The Holder classes will comprise of a Drillbuf containing the unscaled decimal value, the precision and the scale.

1. Vectors: We will store 2's complement, big endian, unscaled value in the vectors. MajorType within the vector would contain the scale and precision. This would allow Drill to read decimal values from Parquet into its value vectors efficiently with no added conversion cost.

1. Holders: Below is a stub of the Decimal38Holder

```
public class Decimal38Holder implements ValueHolder{
        public static final int WIDTH = 16;
        public int start;
```

```
        public DrillBuf buffer;
        public int scale;
        public int precision;
}
```

Note: Since we store the unscaled value in the vectors we have scale and precision in the individual holders for performing math operations.

3. Functions:

There are two approaches to implementing decimal functions. A simpler implementation using Java BigDecimal and a higher performance version using a native C++ implementation via JNI (GCC has native support for operating on 128 bit numbers which will be more efficient than using BigDecimal).

Since there will be two implementations (native and in Java using BigDecimal) it is necessary to have one common API to interact with the actual implementations. Drill function would invoke the actual implementation via the API, the first implementation would be in Java using BigDecimal. The native implementation would be available later only on platforms that support it.

1. API:

```
Unary API:
// examples: round(), trunc() ...
public static void API_CALL(
 DrillBuf buf1, int start1, int scale1, int precision1,
 DrillBuf outputBuf, int outputStart, int expectedScale, int expectedPrecision);

Binary API:
// examples :add, subtract, multiply ...
public static void API_CALL(
 DrillBuf buf1, int start1, int scale1, int precision1,
 DrillBuf buf2, int start2, int scale2, int precision2,
 DrillBuf outputBuf, int outputStart, int expectedScale, int expectedPrecision);
```

Parameters:

buf1: input1's buffer containing unscaled decimal value

start1: start index into the buffer

scale1: decimal scale
precision1: decimal precision
buf2: input2's buffer containing unscale decimal value
start2: start index into the buffer
scale2: decimal scale
precision2: decimal precision
outputBuf: pointer to output buffer, where the unscaled value will be copied
outputStart: start index into the output buffer
expectedScale: maximum scale beyond which rounding would occur
expectedPrecision: maximum precision beyond which an overflow error be raised

Note about Rounding
Rounding and overflow handling is done by the implementation, not by the Drill function. Even though the actual decimal math will be done as part of the function implementation we pass it an expected scale and precision. The reason being that, in cases where we want to round (if computed scale > expectedScale) then it's best to do it within this function rather in the Drill function, otherwise we will have to reconstruct the BigDecimal from the output buffer again, whereas if we were to do it here we would already have the BigDecimal. (also see the overflow handling section and code samples of an example Drill function and its implementation)

1. Codegen: We use a single scale and precision per column, so for any given operation we can compute the output's scale and precision upfront to be set in the output holder (for chained expressions). The setup() method in the codegen framework is an ideal place for this computation as its invoked once at the beginning of the operation, however we need to modify codegen to detect that we have a decimal function and populate the input holder's scale and precision value, for us to be able to perform the computation. We cannot do this with the current codegen framework because the holders aren't populated/ created until the eval() method.

1. Scalar Functions:
    a. equals
    b. less than
    c. less than equals
    d. greater than
    e. greater than equals

f. compare_to
g. subtract
h. multiply
i. divide
j. round
k. trunc
l. mod
m. abs
n. sign
o. floor
p. hash

1. Aggregate Functions:
    a. Count
    b. Min
    c. Max
    d. Sum
    e. Avg

1. Cast Functions:
    a. cast from decimal9
    b. cast from decimal18
    c. cast from varchar
    d. cast from int
    e. cast from bigint
    f. cast to decimal9
    g. cast to decimal18
    h. cast to varchar
    i. cast to int
    j. cast to bigint
    k. cast to double

4. Scale & Precision:

We are going to use the below table to compute the output type's scale and precision. Part of this table is taken from SQL server's documentation indicating how they compute the scale and precision.

Operation	Result precision	Result scale *
e1 + e2	max(s1, s2) + max(p1-s1, p2-s2) + 1	max(s1, s2)
e1 - e2	max(s1, s2) + max(p1-s1, p2-s2) + 1	max(s1, s2)
e1 * e2	p1 + p2 + 1	s1 + s2
e1 / e2	p1 - s1 + s2 + max(6, s1 + p2 + 1)	max(6, s1 + p2 + 1)
e1 % e2	min(p1-s1, p2 -s2) + max(s1,s2)	max(s1, s2)
Aggregate sum(e1)	38	s1
Aggregate avg(e1)	38	max(6, s1)

1. Overflow Handling:

Drill will not aggressively chop off the number of digits in the scale unless there is an overflow in which case we will raise an error. This is different from our existing behavior. Let's look at an example to illustrate the current behavior vs the proposed future behavior

Consider we are multiplying two decimals with precision and scale as 30 and 2 respectively

in1(30, 2) * in2(30, 2) ⇒ out(60, 4) ⇒ out(38, 0)

Ideally the output type should have a precision and scale of 60 and 4 respectively. However Drill only supports a maximum of 38 decimal digits so we make the precision 38. Coming to scale we notice that there is a possibility that the total number of non-fractional digits may be greater than 38 hence there is no space to store the fractional digits which is why we make the scale as 0.

Moving forward the computation would change so that we don't chop off any part of the scale and in the case where we overflow we would simply raise an exception, so the same operation would be resolved as below

in1(30, 2) * in2(30, 2) ⇒ out(60, 4) ⇒ out(38, 4)

Note: Overflow checking would only happen in functions related to the Decimal38 type and not in Decimal9 or Decimal18, reason being that if the smaller type (decimal9 or 18) weren't enough to store the result then we should've up-promoted the output type to be of Decimal38.

5. Native Implementation:

The native implementation will be a thin Java wrapper around a native library. Details of the implementation are out of the scope of this document, below are the two options under consideration.

The standard multi-precision library is MPFR which is designed to provide arbitrary precision for floating point values. Using this library has the challenge that the representation does not match the representation of decimal in Parquet and Drill. There would be the cost of some conversion. The other issue is that the library, being for floating point types, does not accept scale in the APIs. The implementation would have to take care of that.

The second option is to use 128 bit integer arithmetic and implement scale and rounding natively.

Example Drill Func (add function):

```
public class Decimal38Functions {
@FunctionTemplate(name = "add", scope =
FunctionTemplate.FunctionScope.DECIMAL_ADD_SCALE,
   nulls = FunctionTemplate.NullHandling.NULL_IF_NULL)
public static class Decimal38Decimal38Add implements DrillSimpleFunc {

  @Param  Decimal38Holder input1;
  @Param  Decimal38Holder input2;
  @Inject DrillBuf buffer;
  @Output Decimal38Holder output;

  @Override
  public void setup(RecordBatch incoming) {
    buffer = buffer.reallocIfNeeded(Decimal38Holder.WIDTH);
  }

  @Override
  public void eval() {
    output.start = 0;
    output.buffer = buffer;

org.apache.drill.exec.expr.fn.impl.Decimal38FunctionImplementation.addDecimal38Decimal38(
        input1.buffer, input1.start, input1.scale, input1.precision,
        input2.buffer, input2.start, input2.scale, input2.precision,
        output.buffer, output.start, 2 /* for prototyping purposes hard coded scale */, 38);
  }
}
}
```

Example of Function Implementation:

```
public class Decimal38FunctionImplementation {

private static final int WIDTH = Decimal38Holder.WIDTH;



public static void addDecimal38Decimal38(
    DrillBuf buf1, int start1, int scale1, int precision1,
    DrillBuf buf2, int start2, int scale2, int precision2,
    DrillBuf outputBuf, int outputStart, int expectedScale, int expectedPrecision
    ) {
  BigDecimal in1 =
org.apache.drill.exec.util.DecimalUtility.getBigDecimalFromVector(buf1, start1, scale1);
  BigDecimal in2 =
org.apache.drill.exec.util.DecimalUtility.getBigDecimalFromVector(buf2, start2, scale2);

  BigDecimal output = in1.add(in2);

  // Check for overflow
  if (output.precision() > expectedPrecision) {
    throw new DrillRuntimeException("Overflow error");
  }

  // convert BigDecimal into two's complement representation to store in the vector
  DecimalUtility.writeTwosComplement(output, outputBuf, outputStart, WIDTH);
}
        }
```

BigDecimal from Vector:

```
public static BigDecimal getBigDecimalFromVector(DrillBuf bytebuf, int start, int scale) {
byte[] value = new byte[16];
bytebuf.getBytes(start, value, 0, 16);
BigInteger unscaledValue = new BigInteger(value);
return new BigDecimal(unscaledValue, scale);
```

```
}
```