Floating-Point Rounding & BSON Limit

QE-ID-FP-1

June 28, 2024

1 Handling Rounding Errors in Floating-Point

SERVER-91788 identifies an occurrence of floating-point rounding that can lead to correctness issues in OSTType, Edge and Mincover. This is due to the fact that double and decimal128 cannot represent all possible values in their domain exactly. For example, double has an 11-bit exponent that allows it to represent values between 10^{-308} and 10^{308}, but the value $2^{53}+1$ cannot be represented exactly as even though it belongs to the domain. Instead, $2^{53}+1$ is rounded to 2^{53}.

Effect on QE ranges. In QE Ranges, one can either use default encodings or more efficient custom encodings for bounded domains. For the latter, the customer must provide three parameters: a lower bound lb on the domain, an upper bound on the domain ub and a (query) precision prc that determines the precision of the range queries. To determine whether to use a default encoding or a custom encoding, we need to compute the following quantity

$$\texttt{logdomsize}(\mathsf{lb}, \mathsf{ub}, \mathsf{prc}) = \lceil \log_2 \left((\mathsf{ub} - \mathsf{lb} + 1) \cdot 10^{\mathsf{prc}} \right) \rceil \tag{1}$$

and check if it is strictly less than 64 (in the case of double) and if so, we use a custom encoding. In the case of decimal128 we check if it is strictly less than 128.

logdomsize is also used by the Edges and Mincover and, in particular, controls the length of the values' binary encodings which, in turn, impact the Edge and Mincover algorithms. Being off by 1, here, will lead to incorrect results.

For example, as pointed out in SERVER-91788 if $\mathsf{lb} = 0$, $\mathsf{ub} = 2^{53}$ and $\mathsf{prc} = 0$, then $(\mathsf{ub} - \mathsf{lb} + 1) \cdot 10^{\mathsf{prc}}$ will be rounded to 2^{53} so the $\log_2$ will be computed on 2^{53} instead of on $2^{53}+1$ due to rounding. The final computation (after applying the ceiling) will therefore result in 53 instead of 54.

Solutions. In the following, we propose two solutions to handle this issue. We prefer the first solution for reasons we explain below, but are OK with using the second as well if any technical challenges arise with the first one.

- `logdomsize`(lb, ub, prc):
    1. compute $s_0 := \lceil \log_2((\mathsf{ub} - \mathsf{lb} + 1) \cdot 10^{\mathsf{prc}}) \rceil$ as a `double`;
    2. if $s_0 \geq 64$, return s_0;
    3. parse ub as $a.a_1a_2a_3 \ldots a_n$ and lb as $b.b_1b_2 \ldots b_m$;
    4. if $\max(n, m) > \mathsf{prc}$, return error and ask for new parameters;
    5. compute $s_1 := (\mathsf{ub} - \mathsf{lb} + 1) \cdot 10^{\mathsf{prc}}$ as a `double`; [a]
    6. compute $s_2 :=$ `double-to-int64`(s_1);
    7. return $s_3 :=$ `ceilLog`(s_2);

[a] Note that s_1 is a non-negative integer.

Figure 1: *Preferred* domain size computation when given bounds and precision

1.1 First and *Preferred* Solution

Before we can describe our first solution, we introduce some terminology. Given a number represented using either binary or decimal formats we refer to the digits before the binary/decimal point as its *integral digits* and to the digits after the binary/decimal point as its *fractional digits*. So, for example, given a number 123.987, we refer to the digits 1, 2 and 3 as its integer digits and to the digits 9, 8 and 7 as its fractional digits.

The first solution is based on the assumption that the customer-provided *lower bound* lb *and upper bound* ub *have a number of fractional digits that are at most* prc. We will refer to this as the *fractional precision constraint*. In this solution, if the fractional precision constraint is violated (i.e., if the customer provides either lb or ub with more than prc fractional digits) then we abort and output an error that asks the customer to provide new (lb, ub, prc) that verify the constraint. The reason we suggest aborting and returning an error if the fractional precision constraint is violated is because if a customer does provide a lower and/or upper bound that has more fractional digits than the precision, then the customer has probably misunderstood how to tune the parameters.

Pseudocode for `double`. With this in mind, we describe in Figure 2 the steps to compute `logdomsize` as in Equation (1) for the case of `double`. We assume the existence of a function `double-to-int64` that converts `double`s to `int64`s and of a function `ceilLog` that we describe in Section 2.

Pseudocode for `decimal128`. The pseudocode for `decimal128` is the same as for `double` except that the constant 64 is replaced with 128 in Steps 2 and the function `double-to-int64` is replaced with a function `double-to-int128`.

1.2 Second Solution

In this section, we describe a second possible solution that does not force customers to use (lb, ub, prc) parameters that meet the fractional precision constraint.

- `logdomsize`($\mathsf{lb}, \mathsf{ub}, \mathsf{prc}$):
  1. compute $s_0 := \lceil \log_2((\mathsf{ub} - \mathsf{lb} + 1) \cdot 10^{\mathsf{prc}}) \rceil$ as a `double`;
  2. if $s_0 \geq 64$, return s_0;
  3. compute $s_1 := (\mathsf{ub} - \mathsf{lb} + 1) \cdot 10^{\mathsf{prc}}$ as a `double`;
  4. compute $s_2 := \lfloor s_1 \rfloor$;
  5. compute $s_3 :=$ `double-to-int64`(s_2);
  6. return $s_4 :=$ `ceilLog`(s_3);

Figure 2: Alternative domain size computation when given bounds and precision

2 Computing Ceiling of Base-2 Logarithm

Computing $\lfloor \log_2 \rfloor$. Given an integer $s \in \{0, 1, \ldots, 2^{64} - 1\}$, we denote by $s_{\mathbf{2}}$ its Big Endian binary representation. We can compute the floor of its base-2 logarithm as

$$\lfloor \log_2(s) \rfloor = \mathsf{pfbs}_0(s_{\mathbf{2}}),$$

where pfbs_0 is the position of the first bit set to 1 using 0-indexing. As an example, if $s = 9$, then $\mathsf{pfbs}_0(s_{\mathbf{2}}) = 3$.

Computing $\lceil \log_2 \rceil$. Given an integer $s \in \{0, 1, \ldots, 2^{64} - 1\}$, we can compute the ceiling of its base-2 logarithm as

$$\lceil \log_2(s) \rceil = \lfloor \log_2(s) \rfloor + \mathbb{1}\{(s-1)_{\mathbf{2}} \wedge s_{\mathbf{2}} \neq 0\},$$

where $\wedge$ is the bitwise `AND` operation. The intuition is the following. First, notice that if $\log_2(s)$ is a power of 2 then

$$\lceil \log_2(s) \rceil = \lfloor \log_2(s) \rfloor + 0,$$

whereas if $\log_2(s)$ is not a power of 2 then

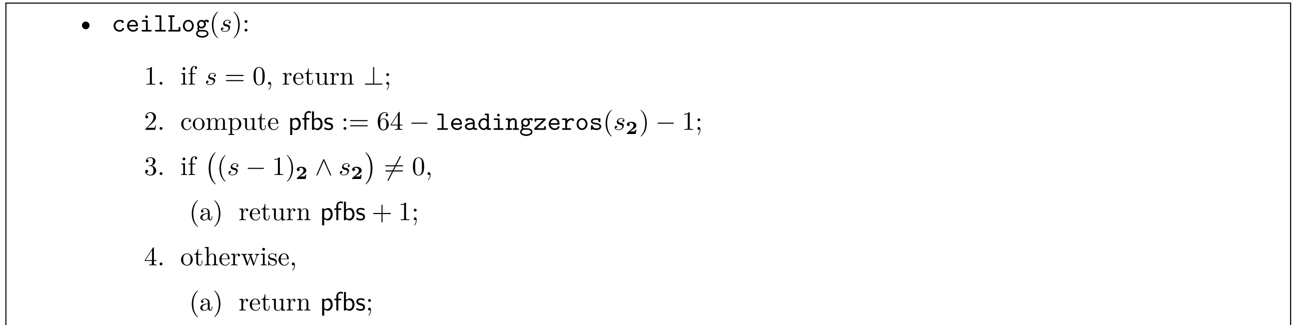
$$\lceil \log_2(s) \rceil = \lfloor \log_2(s) \rfloor + 1$$

The expression

$$\mathbb{1}\{(s-1)_{\mathbf{2}} \wedge s_{\mathbf{2}} \neq 0\}$$

outputs 0 if and only if s is a power of 2. Specifically, if s is a power of 2 then $s_{\mathbf{2}}$ will have a single bit set, $s - 1$ will flip $s_{\mathbf{2}}$'s bit to 0 and all the following bits to 1 and, therefore, their bitwise and will be the zero bit string. On the other hand, if s is not a power of two then $s_{\mathbf{2}}$ has at least 2 bits set and it will have at least one of its post-pfbs bits in common with $(s-1)_{\mathbf{2}}$. Because of this the bitwise `AND` will have at least one bit set and will be different from the zero bit string.

Pseudocode. We provide the pseudo-code of the `ceilLog` function in Figure 3. It takes as input an integer $s \in \{0, 1, \ldots, 2^{64} - 1\}$ and outputs a value $\ell \in \{0, 1, \ldots, 64\}$. Recall that $\wedge$ is the bitwise `AND` operation (i.e., & in `C`).



- `ceilLog`(s):
  1. if $s = 0$, return $\perp$;
  2. compute $\mathsf{pfbs} := 64 - \mathtt{leadingzeros}(s_{\mathbf{2}}) - 1$;
  3. if $\big((s-1)_{\mathbf{2}} \wedge s_{\mathbf{2}}\big) \neq 0$,
     (a) return $\mathsf{pfbs} + 1$;
  4. otherwise,
     (a) return pfbs;

Figure 3: The `ceilLog` function.

3 BSON Limit Bounds

When executing a QE range query, the client generates a cover cvr composed of many edges that, together, make up the range query. The cover cvr is computed by the $\mathsf{Mincover}$ algorithm. When the trimming and sparsity optimizations are not used, i.e., when the trimming factor $\mathtt{tf} = 0$ and the sparsity factor $\mathtt{sp} = 1$, then the maximum size of the cover is $2 \cdot \log_2(n) - 1$ where n is the size of the domain. With higher values of the trimming and sparsity factors, the size of the cover can get significantly large. The size of the cover matters because the client needs to insert the cover in a single `find` query which is stored as a BSON document. As such, we need to make sure that the size of the cover does not exceed the BSON limit; otherwise this will result in a correctness error.

Size of the cover. In order to understand whether a cover can fit in a document or not, it is first important to understand how big the cover can be as function of the trimming factor $\mathtt{tf} \in \mathbb{N}$, the sparsity $\mathtt{sp} \in \{1, 2, 3, 4\}$, and the size of the domain n. More precisely, the the cover size is upper bounded as follows,

$$\#\mathsf{cvr} \leq \min\left(n, 2^{\mathtt{sp}-1} \cdot \left(2^{\mathtt{tf}} + 2\log_2(n) - 1\right)\right).$$

To avoid the correctness error due to the BSON limit, one should check that

$$\min\left(n, 2^{\mathtt{sp}-1} \cdot \left(2^{\mathtt{tf}} + 2\log_2(n) - 1\right)\right) < \mathtt{CBSON} \tag{2}$$

where `CBSON` is the size of the largest cover that can fit in a BSON document. To implement this check a domain size n and a `CBSON` value must be chosen. From previous discussions with Server Security, we assume that $\mathtt{CBSON} = 300,000$ and we discuss how to set the domain size next.

Size of the domain. The size of the domain n depends on the numerical data type we are working with and can also depend of the lower bound lb, the upper bound ub, and the precision prc. In the following, we describe how to set the domain size for each numerical type:

1. if the data type is $\mathsf{sint32}$, $\mathsf{sint64}$ or $\mathsf{int64}$ with $\mathsf{lb} = \mathsf{ub} = \perp$, then n is equal to 2^{32}, 2^{64} and 2^{64}, respectively;

2. if the data type is $\mathsf{sint32}$, $\mathsf{sint64}$ or $\mathsf{int64}$ with $\mathsf{ub} \neq \perp$ and $\mathsf{lb} \neq \perp$, then $n = 2^{\lceil \log_2(\mathsf{ub}-\mathsf{lb}+1) \rceil}$;
3. if the data type is double or $\mathsf{decimal128}$ with $\mathsf{lb} = \mathsf{ub} = \perp$, then $n = 2^{64}$ or $n = 2^{128}$, respectively;
4. if the data type is double or $\mathsf{decimal128}$ with $\mathsf{lb} \neq \perp$, $\mathsf{ub} \neq \perp$ and $\mathsf{prc} \neq \perp$, then $n = \min(2^{64}, 2^{\lceil \log_2((\mathsf{ub}-\mathsf{lb}+1)\cdot 10^{\mathsf{prc}}) \rceil})$ or $n = \min(2^{128}, 2^{\lceil \log_2((\mathsf{ub}-\mathsf{lb}+1)\cdot 10^{\mathsf{prc}}) \rceil})$, respectively;

Observe that the highest domain size is $n = 2^{128}$.

Default setting. We assume $\mathtt{CBSON} = 300,000$ and $n = 2^{128}$. QE Range sets $\mathtt{tf} = 6$ and $\mathtt{sp} = 2$ as defaults, so Inequality 2 is verified since

$$\min(n, 2^{\mathtt{sp}-1} \cdot (2^{\mathtt{tf}} + 2\log_2(n) - 1)) = \min(2^{128}, 2 \cdot (2^6 + 2 \cdot 128 - 1)) = 638 \ll 300,000$$