Fast Fixed-Point Decimals

Final Presentation

Jekyeom Jeon
Taoxi Jiang
Yuttapichai (Guide) Kerdcharoen
Re-iterate the motivation...

Source: https://www.youtube.com/watch?v=rhxd_xaeMPU
What we did (in a slide)

**LIBFIXEYPOINTY**
CMU-DB’s standalone 128-bit fixed-point decimal

- Documented `libfixeypointy`
- Hardened `libfixeypointy`
- Improved `libfixeypointy`’s multiplication and division performance
- Evaluated `libfixeypointy` against other standalone decimal implementations

**FIXEYPOINTY**
`libfixeypointy` as PostgreSQL’s UDT

- Integrated `libfixeypointy` as PostgreSQL user-defined types, `fxypy`
  - Including arithmetic and relative operators
  - Basic aggregators (sum, min, max, count)
- Evaluated `fxypy` against PostgreSQL NUMERIC (its built-in fixed-point decimal type) and DOUBLE, REAL (its floating-point type)
All the code are documented

Doxygen Comment

Complex Code Explanation

```c
/**
 * Calculate product of two unsigned integers of arbitrary
 * 64-bit chunks.
 * @param a The first operand.
 * @param b The second operand.
 * @return The product of a and b.
 * @brief Add two fxypt objects.
 * @param a The pointer to the first fxypt object.
 * @param b The pointer to the second fxypt object.
 * @return The sum of the two fxypt objects.
 * extern "C" void *fxypt_add(void *a, void *b) {
 *  FxyPty_Decimal *wrapped_a = (FxyPty_Decimal *)a;
 *  FxyPty_Decimal *wrapped_b = (FxyPty_Decimal *)b;
 *  assert(wrapped_a->scale == wrapped_b->scale);
 *  FxyPty_Decimal *result = (FxyPty_Decimal *)malloc(sizeof(FxyPty_Decimal));
 *  result->scale = wrapped_a->scale;
 *  try {
 *    libfixeypointy::Decimal tmp{_pack128(wrapped_a)};
 *  }
 *  check(result, wrapped_a, wrapped_b);
 *  return result;
 */
*/
```

```c
uint128_t half_words_magic_result[8];
```
More libfixeypointy boundary cases are handled

**LIBFIXEYPINTY**

```cpp
// Decimal::Divide(const Decimal &denominator, const ScaleType &scale)
// 1. Multiply the dividend with is (the denominator scale), with overflow.
// 2. If overflow, divide by the denominator with multi-word 256-bit.
// 3. If no overflow, divide by the denominator with magic numbers.
// Moreover, the result is in the numerator's scale for technical reasons.

// If the result were to be in the denominator's scale, the first step
// is 10^(2*denominator scale - numerator scale) which requires 256-bit

// If value is 0
(denominator.ToNative() == 0) {
  throw std::runtime_error("Divided by 0");
}

// Decimal::Add(const Decimal &other) {
// If both values are positive, it is possible to get overflowed
// (value_ > 0 && other.value_ > 0) {
// Compute the maximum and safe value for other
// E.g., 63 - 62 = 1
// So, if other = 1, 62 + 1 = 63 (safe)
// But if other = 2 (> 1), 62 + 2 = 64 (overflowed)
int128_t other_bound = std::numeric_limits<__int128>::max() - value;
if (other.value_ > other_bound) {
  throw std::runtime_error("Result overflow > 128 bits");
}
```
The performance bottleneck is grade-school multiply

Observation

- Both multiplication and division (by magic number) use 128-bit grade-school multiplication
- An existing grade-school multiply implementation contains a number of loops and potential bubbles

Assumption

- Unwinding the loop and manually reordering instructions (to avoid bubbles) could improve multiplication and division performance
Optimizing by unwinding loops and reordering instructions

1. Let m and n = 2 and unwind loops

```c
void _CalculateMultiWordProduct128_2_2(const uint128_t *const half_words, uint128_t *half_words_result) {
    constexpr const uint128_t bottom_mask = (uint128_t{1} << 64) - 1;
    uint128_t t2 = (half_words_a[1] * half_words_b[0]);
    uint128_t t1 = (half_words_a[0] * half_words_b[0]);
    t2 += (t1 >> 64);
    half_words_result[0] = t1 & bottom_mask;
    uint128_t t3 = (t2 >> 64);
    t3 += (half_words_a[1] * half_words_b[1]);
    t2 &= bottom_mask;
    t2 += (half_words_a[0] * half_words_b[1]);
    t3 += (t2 >> 64);
    half_words_result[3] = (t3 >> 64);
    half_words_result[1] = t2 & bottom_mask;
    half_words_result[2] = t3 & bottom_mask;
}
```

2. Reorder/remove instructions

```c
uint128_t k, t;
uint32_t i, j;
constexpr const uint128_t bottom_mask = (uint128_t{1} << 64) - 1;
// Initialize first m chunks with 0
half_words_result[0] = 0;
half_words_result[1] = 0;
// For each chunk in b
for (j = 0; j < n; j++) {
    k = 0;
    // Match with all chunks i in a
    for (i = 0; i < m; i++) {
        // Product + Old Value (all)
        t = half_words_a[i] * half_words_b[0];
        // Take only bottom 64 bits
        half_words_result[i + j] = t & bottom_mask;
        // Carry
        k = t >> 64;
    }
    half_words_result[j + m] = k;
}
```
Grade-school multiply is no longer bottleneck

Apply loop unwinding and instruction reordering

1.45x speedup
Division

- Specify custom predefined magic number to speed up division (and multiplication)
- Not too many of them (depending on the operations you normally want to do)
- No magic number -> predicate small -> hot path taking the branch (good)
- A few magic number -> hot path not taking the branch -> predicate small (good)
- Lots of magic number -> predicate big -> access pattern uniform anyway, doesn't make sense to add those magic number (bad)
- Generate and cache all seen magic numbers? -> will test in the future

Todo: Macro to convert static hashtable lookup to compiled predicates

… div by zero / power of 2 check

```c
// 2. If not possible, regular division.
{
    if (MAGIC_CUSTOM_128BIT_CONSTANT_DIVISION.count(constant) == 0) {
        value = static_cast<uint128_t>(value_) / constant;
        return;
    }
}
```

… Magic Number Division
Verification

- **Python decimal (hybrid of fixed point and float)**
  - Random $\text{op1} \pm \ast / \text{op2}$, repeated millions of times
  - Compare rounded off values

- **Java BigDecimal**
  - Long random chain $\text{op1} \pm \ast / \text{op2} \pm \ast / \text{op3}$ ...
  - Compare error handling behavior (overflow), report and revert to previous value when error encountered
  - Compare exact values
  - Results exactly match
Evaluation (Standalone Lib performance)

Operations on decimals stored in 32bit size, most digits before decimal point (scale is small)
Dataset fits in L3 cache. Larger dataset results will come in future.
Evaluation (Standalone Lib performance)

32bit Op Time (10000 line, 1000 iteration, avg 10 runs, skip first 2)

128bit Op Time (10000 line, 1000 iteration, avg 5 runs, skip first 2)

BigDecimal optim for small scale decimals
 Libfixeypointy does not, runtime consistent
Evaluation (Standalone Lib performance)

32bit Op Time (10000 line, 1000 iteration, avg 10 runs, skip first 2)

128bit Op Time (10000 line, 1000 iteration, avg 5 runs, skip first 2)

BigDecimal change both storage and mult algo for bigger decimals
If we force BigDecimal to use our fixed max precision (28) and then round to scale, it becomes ~10x slower.
Integrating libfixeypointy as PostgreSQL's UDT
Evaluation (PostgreSQL operation performance)

Op Time (4M tuples, 2 columns, 5 runs avg)

Fixed-Point
- Libfixeypointy(24B)
- NUMERIC(Variable Size)
- DOUBLE(8B)
- REAL(4B)

Floating-Point

⇒ All Data Types have different sizes
Evaluation (PostgreSQL operation performance)

Op Time (4M tuples, 2 columns, 5 runs avg)

Libfixeypointy(24B) NUMERIC(Variable Size) DOUBLE(8B) REAL(4B)

pgfixeypointy is faster than NUMERIC in division
Evaluation (PostgreSQL operation performance)

Op Time (4M tuples, 2 columns, 5 runs avg)

- pgfixeypointy is faster than NUMERIC in division
- pgfixeypointy is slower than NUMERIC (ToString() is slow)
Evaluation (PostgreSQL operation performance)

Op Time (4M tuples, 2 columns, 5 runs avg)

- **Libfixeypointy(24B)**
- **NUMERIC(Variable Size)**
- **DOUBLE(8B)**
- **REAL(4B)**

pgfixeypointy performs better for aggregation
Evaluation (PostgreSQL operation performance)

Op Time (4M tuples, 2 columns, 5 runs avg)

Libfixeypointy(24B) NUMERIC(Variable Size) DOUBLE(8B) REAL(4B)

pgfixeypointy performs better for aggregation. But, this is result when we're using a single worker
Future Work

➢ Support variable size: store a small decimal in a 64-bit or 32-bit
➢ Parallel aggregation to improve throughput
➢ Running perf to find an opportunity for optimizing multiplication performance
➢ Improve result writing (probably, string conversion) performance
➢ More Aggregator support: AVG, STD, VAR
➢ Type Casting: Operations between different types (e.g., double+libfixeypointy)
➢ More realistic workloads
Resources

- libfixeypointy - https://github.com/cmu-db/libfixeypointy/tree/develop
- pgfixeypointy - https://github.com/pnxguide/pgfixeypointy
Prompt: a fast stream of 128-bit fixed-point decimals