

Towards complete, conformant and performant Decimal computations In Apache DataFusion Core

Evgenii Seliverstov // Bloomberg

About me

Senior Software Engineer in Bloomberg

PhD and academic research on parallel computing systems and GPUs

Worked on proprietary distributed databases, query engines, PostgreSQL

Apache DataFusion contributor

GitHub <https://github.com/theirix>

@theirix



Opinions expressed in this talk are solely my own and do not express the views or opinions of my employer.

Decimal support in Apache DataFusion

Structure of the talk

- Decimal types
- Decimal model in DataFusion
- Performance

Areas of support in DataFusion

- Core type support
- Execution
- UDFs
- Performance

Who needs decimals?

Financial applications

Scientific applications

General numerical computation

Feature parity with databases and analytical engines

Functional benchmarks (TPC-H)

How DataFusion handles decimals

How does it work?

Logical level:

- Precision (total digits)

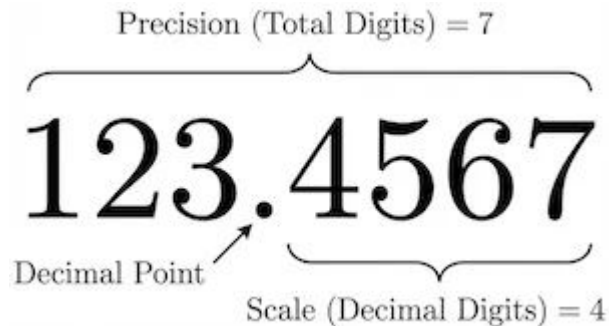
- Scale (decimal digits)

Physical level:

- Uses Apache Arrow datatype

- Physical underlying type

- Fused to type name (Decimal128 means 128-bit storage)



How DataFusion handles decimals

Old flow

Numerics are parsed as floats in SQL (`parse_float_as_decimal=false`)

Decimals are supported as a type `42::decimal(38, 0)`

Decimals are coerced to floats in UDFs

Return type is decimal

Type information is lost

How DataFusion handles decimals

Decimal-native flow

- Decimals are first-class citizens

- Types are preserved

- Wider range of values with exact arithmetic

- Special handling of decimal operations

Decimal model

Types, types, types

```
// arrow::datatypes::Decimal128Type
impl DecimalType for Decimal128Type {
    const BYTE_LENGTH: usize = 16;
    const MAX_PRECISION: u8 = DECIMAL128_MAX_PRECISION;
    const MAX_SCALE: i8 = DECIMAL128_MAX_SCALE;
    const DEFAULT_TYPE: DataType =
        DataType::Decimal128(DECIMAL128_MAX_PRECISION, DECIMAL_DEFAULT_SCALE);
}

impl ArrowPrimitiveType for Decimal128Type {
    type Native = i128;
    const DATA_TYPE: DataType = <Self as DecimalType>::DEFAULT_TYPE;
}
```

```
// arrow_schema
pub enum DataType {
    Int32,
    Float64,
    ...
    Decimal64(u8, i8),
    Decimal128(u8, i8),
}

// datafusion_common::scalar::ScalarValue
pub enum ScalarValue {
    Int32(Option<i32>),
    Decimal64(Option<i64>, u8, i8),
    Decimal128(Option<i128>, u8, i8),
}
```

Decimal model

Class of types

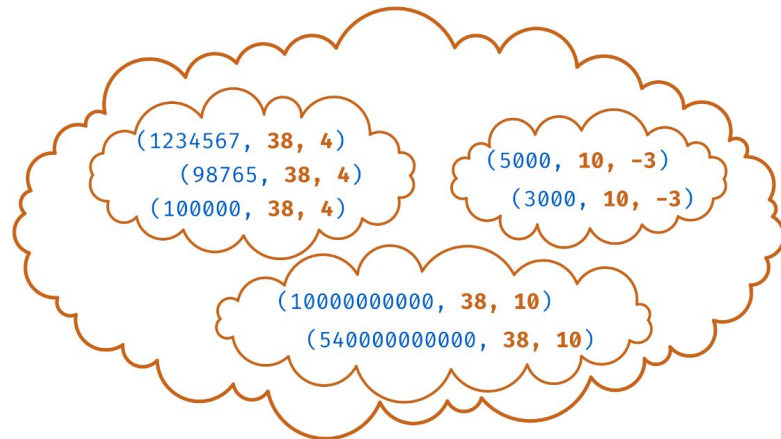
Int32



Utf8



Decimal128(p, s)



Decimal support for UDFs - type system

A finite set of UDFs

Generic code over `num_traits` traits

Move as much logic as possible to `num_traits`

Supports `i32`, `i64`, `i128` out of box

Adding `arrow_buffer::i256` support explicitly

```
abs, any_value, approx_distinct,
approx_percentile_cont_with_weight,
array_agg, avg, ceil, degrees,
radians, cot, exp, first_value,
last_value, floor, gcd, greatest,
isnan, iszero, least, lcm, median,
log, min, max, nth_value,
percentile_cont, round, sqrt, cbrt,
sum, trunc
```

```
use num_traits::{
    Bounded, CheckedAdd, CheckedDiv, CheckedMul, CheckedNeg, CheckedRem, CheckedShl, CheckedShr,
    CheckedSub, ConstOne, ConstZero, FromPrimitive, MulAdd, MulAddAssign, Num, One, SaturatingAdd,
    SaturatingMul, SaturatingSub, Signed, ToPrimitive, WrappingAdd, WrappingMul, WrappingNeg,
    WrappingShl, WrappingShr, WrappingSub, Zero, cast::AsPrimitive,
};
```

Decimal support for UDFs - unified implementation

Full `num_traits` support for `Decimal256` is blocked

`num_traits::float::Float` cannot be implemented

Requires `num_traits::NumCast`, which is conflicting with `std::convert`

Unified implementation is possible for `floats` + `Decimal32` + `Decimal128` + `Decimal256`

As a dedicated `i256` implementation...

```
// num_cast::NumCast
pub trait NumCast: Sized + ToPrimitive {
    fn from<T: ToPrimitive>(n: T) -> Option<Self>;
}

// std::convert::From
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}
```

Decimal support for UDFs - dispatch

Entry point — `ScalarUDFImpl::invoke_with_args`

Dispatch via `data_type()` and downcasting from `ArrayRef`

For every type, dispatch statically to a concrete function or a generic function

Decimal types also carry type information (precision and scale)

Operation must carry type information

```
match value.data_type() {  
  DataType::Int64 => {apply_op::<Int64Type>(value) },  
  DataType::Decimal32(precision, scale) => {  
    apply_decimal_op::<Decimal32Type, _>(value, precision, scale)  
  }  
}
```

Decimal support for UDFs - computation

Arrow kernels operate on a native type

If operation is scale preserving

There is no need to know precision, scale

Works for numeric scalars (pure integers and floats)

If operation can change scale

All decimals

Examples: pow, log, ceil

Precision and scale are passed to operation

Operation is aware of **both** input and output types

```
// arrow::datatypes::Decimal128Type
impl ArrowPrimitiveType for Decimal128Type
{
    type Native = i128;
}
```

Decimal support for UDFs - casting

Problem with Arrow

By default, Arrow casts to `0 : :DATA_TYPE`

Which is decimal of default precision (e.g. 38 for 128-bit)

It doesn't match UDF result type from the planner

Solved problem by introducing schema-aware kernels

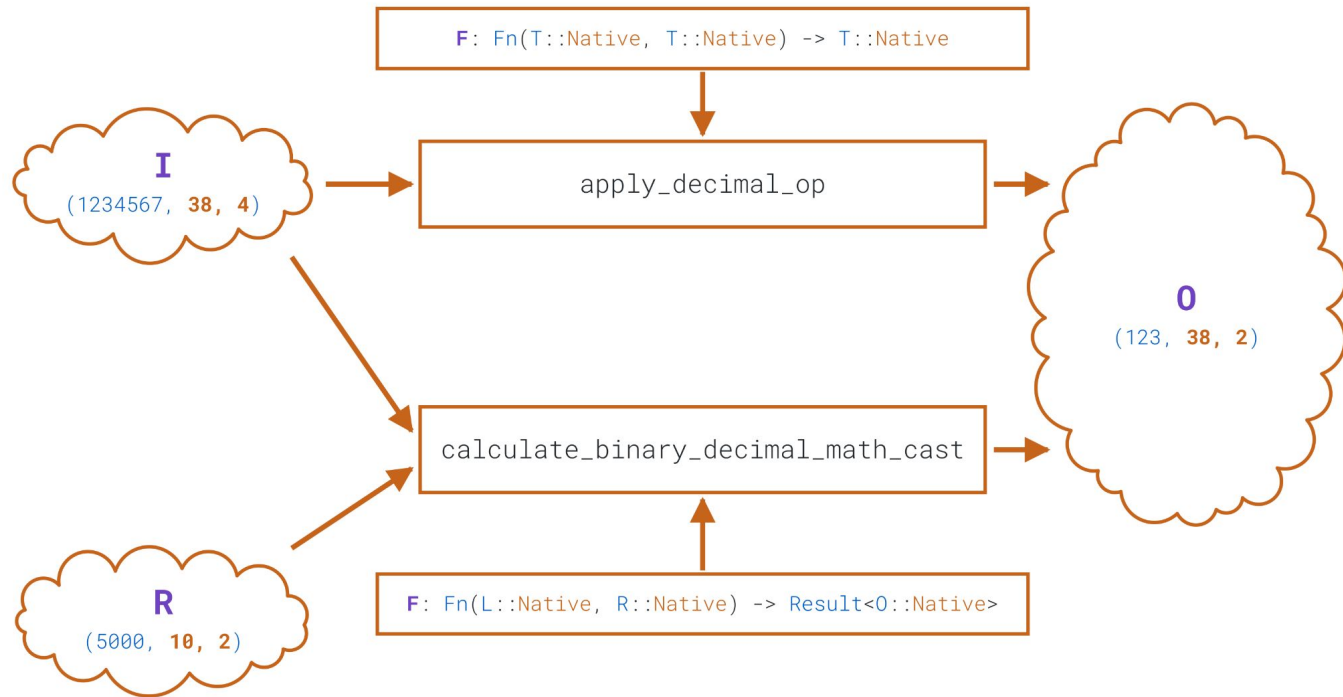
Decimal support for UDFs - casting

Decimal helpers

mirrors Arrow kernels

`compute::unary`

`compute::binary`



Decimal support for UDFs - specialisation

Some mathematical operations can be natively supported for Decimals

`log10(value)` when a value is decimal

`pow(value, exp)` when a value is decimal and exp is integer

Add or remove zeroes

Problems:

- Planner and execution inconsistency
- DataFusion planner announces a return type
- For Arrow array bases and exponents can be different
- Effective type is different from a planned type for different elements

Solution: omit dependent types for pow UDF

Decimal support for UDFs - state

GitHub Epic [Support Decimal for User Defined Functions](#)

A very good state

Almost complete!

Goals:

1. Theoretical possibility of decimal support
2. Technical feasibility of decimal support
3. Implementation state

Legend:

-  - all good, implemented
-  - not supported or not applicable
-  - DataFusion implementation conforms to possible support, nothing to follow-up

UDF	Is implemented	Could Support Decimal	Comments
<code>abs</code>			Native support, #17808
<code>any_value</code>			Native support by design
<code>approx_distinct</code>			Native support, #23190
<code>approx_median</code>			Coerce to floats as with majority of DBs, #21074
<code>approx_percentile_cont</code>			See <code>approx_median</code>
<code>approx_percentile_cont_with_weight</code>			See <code>approx_median</code>
<code>array_agg</code>			Native support by design
<code>avg</code>			Native support, #22713
<code>bit_and</code> / <code>bit_or</code> / <code>bit_xor</code>			N/A, bit domain
<code>bool_and</code> / <code>bool_or</code>			N/A, boolean domain
<code>ceil</code>			Native support, #18979
<code>corr</code>			Coerced. Native support is hard. #3572 , #3481 , #19463

Conformance

Let's model the DataFusion implementation after another database?

Easier for other parts of the system

Almost entirely different decimal implementation and behaviour for major database engines

PostgreSQL, ClickHouse, DuckDB, Apache Spark, Databricks, Snowflake, Oracle, MySQL

Conformance

DataFusion usually follows PostgreSQL

`numeric` type everywhere

it's a lie — not a decimal

but an infinite-precision numeric type

“without any precision or scale creates an “unconstrained numeric” column in which numeric values of any length can be stored, up to the implementation limits”

`abs ( numeric_type ) → numeric_type`

Absolute value

`abs(-17.4) → 17.4`

`cbrt(double precision) → double precision`

Cube root

`cbrt(64.0) → 4`

`ceil(numeric) → numeric`

`ceil(double precision) → double precision`

Nearest integer greater than or equal to argument

`ceil(42.2) → 43`

`ceil(-42.8) → -42`

`ceiling(numeric) → numeric`

`ceiling(double precision) → double precision`

Nearest integer greater than or equal to argument (same as ceil)

`ceiling(95.3) → 96`

Conformance

ClickHouse is optimised mostly for performance

- Opinionated correctness on decimals
- Relaxed overflow check
- [Related DF discussion](#)
- No logical type range checks
- Precision validation only during string conversion

Conformance

DuckDB is the closest analogue

It is also based on Apache Arrow (C++ implementation)

Decimal performance

DataFusion secret sauce — vectorized execution on top of Arrow compute

Native for Decimals — i32, i64, i128 and i256

Performance tip: reuse operations for native types

SQL parser level — improvements in Arrow 60

Arrow vectorisation

SIMD vectorisation of decimal operations

Arrow uses LLVM auto-vectorisation

Not `core::simd / arch::simd`

Decimal32 is fully vectorized (32-bit lanes)

Decimal64 is fully vectorized (64-bit lanes)

Decimal128 is not vectorized (no 128-bit vector types in LLVM IR)

Decimal256 support is software-only ([arrow_buffer::i256](#))

Arrow vectorisation

Fallible Arrow kernels (`try_binary`, `try_unary`) are not auto-vectorized yet

- Only basic read operations are vectorized
- Not caused by Arrow or DataFusion
- Mostly LLVM issue

Rewrite as infallible if possible

If handling nulls, just split to values and nulls

```
PrimitiveArray::new(values: ScalarBuffer<T::Native>,  
                    nulls: Option<NullBuffer>) -> Self
```

Decimal fallible kernels

Why decimal kernels are fallible?

Maintain decimal correctness

Avoid decimal overflows

Always verify the output value

```
// DecimalType  
fn validate_decimal_precision(  
    value: Self::Native,  
    precision: u8, scale: i8,  
) -> Result<(), ArrowError>;
```

Decimal performance

Improve performance with special case handling in UDF

Optional arguments

Edge cases

```
fn invoke_with_args(&self, args: ScalarFunctionArgs) -> Result<ColumnarValue> {  
    let arg = &args.args[0];  
  
    // Fast path for scalars  
    if let ColumnarValue::Scalar(scalar) = arg { ... }  
  
    let result: ArrayRef = match arg.data_type() {  
        DataType::Decimal64(precision, scale) => {  
            // arg is array  
            apply_decimal_op:::<Decimal64Type, _>(  
                &arg.to_array(N), precision, *scale, kernel,  
            )?  
        }  
    }  
}
```

Difference between engines

DataFusion is an extensible engine

Supports Postgres-compatible mode, Spark mode

Behaviour is verified by integration tests via SLT files (query-result)

Spark is an opinionated implementation (provided by core)

Core UDFs are a fast-moving target

Spark UDFs piggyback on DataFusion implementation

Still a lot of work to do to harmonise implementations

Future plans

Feature-parity is done

Improving type coercion support with `TypeSignature::Coercible`

Improving test suite in SLTs

- Growing with decimal test cases

- SLTs are easier to support and verify

Substrait

Unifying Spark and DataFusion core

Cross-referencing with other database engines

Benchmarking and fuzzing implementation