Москва 2020

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
«Прикладная математика и информатика»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект на тему

«Реализация целых и десятичных чисел с фиксированной запятой с расширенной точностью в ClickHouse»

Выполнил студент группы БПМИ 166, 4 курса,
Соколов Андрей Игоревич

Руководитель ВКР:
приглашенный преподаватель, департамент больших данных и информационного поиска, Миловидов Алексей Николаевич

Аннотация: Существуют разные модели представления вещественных чисел в двоичном виде; для вычислений, требующих точности, как, например, вычисления связанные с финансовыми данными, используются десятичные вещественные числа с фиксированным количеством знаков после запятой. ClickHouse — столбцовая распределенная СУБД с открытым кодом, поддерживает соответствующий тип Decimal с максимальной точностью до 38 десятичных знаков. В этой работе рассматриваются подходы к представлению десятичных вещественных чисел, обозреваются реализации в популярных СУБД с открытым кодом. На основе рассмотренных моделей в СУБД ClickHouse реализуется тип Decimal256, поддерживающий точность до 77 десятичных знаков. Также в работе рассматриваются существующие библиотеки, реализующие целые числа большой размерности, в СУБД ClickHouse добавляются знаковые и беззнаковые типы размерности 128 и 256 бит.

Ключевые слова: ClickHouse, вещественные числа с фиксированной запятой, базы данных, большие целые числа, вещественные числа расширенной точности

Abstract: Different models for representing real numbers in binary form exist. In several fields, requiring exact computations and low error tolerance, such as operations on financial data, fixed-point decimal number representation is preferred over standard floating-point format. ClickHouse — open-source distributed column-oriented DBMS — has support for Decimal type with precision up to 38 decimal digits. This work considers different models for fixed-point decimal numbers, their implementations in popular open-source DBMSs are reviewed. Following up, Decimal256 type is introduced to ClickHouse, extending maximum precision to 77 decimal digits. In addition, this work reviews libraries implementing big integers and adds signed and unsigned 128-bit and 256-bit integer types to ClickHouse.

Оглавление

1. Введение

С самого первого своего появления электронно-вычислительные машины использовались для вычислений над вещественными числами. ЭНИАК, первый компьютер общего назначения, был создан в рамках работы над ядерной программой, включавшей в себя большое количество вычислений с вещественными числами. С развитием и эволюцией компьютерных технологий, были созданы и разработаны разнообразные способы представления вещественных чисел, постепенно проигравшие в популярности и распространенности двоичному представлению вещественных чисел. Это представление было формализовано и зафиксировано в 1985 году с принятием стандарта IEEE 754 [1], описывающем представление вещественного числа как тройку знак, мантиссы и экспоненты.

Однако в некоторых областях вычислений вышеописанное представление неудобно или невыгодно, в частности, в применениях, где важна точность вычислений. Одна из таких областей, являющаяся наиболее распространенной — вычисления, связанные с обработкой и хранением финансовых операций и данных. Мелкие ошибки во время вычисления, связанные с неточностями представления десятичных чисел, используемых в вычислениях, производимых людьми, или некорректные округления, могут привести к большим финансовым потерям. Многие страны, во избежание подобных проблем, издают юридические документы, регулирующие обработку и хранение финансовых данных — в частности, постановлением 1997 года генеральный директорат Европейской комиссии по экономическим и финансовым вопросам требует гарантировать точность не менее 6 знаков после запятой во всех вычислениях и в хранении финансовых данных, связанных с евро, на территории Европейского союза [2].

ClickHouse — столбцовая распределенная система управления базами данных с открытым исходным кодом [3], обеспечивающая быстрые распределенные вычисления с использованием SQL-подобного синтаксиса, используемая для аналитических вычислений. На момент написания текущий работы ClickHouse поддерживает тип Decimal, использующий двоичное представление вещественных чисел с фиксированной точностью. Однако максимальная точность, которую можно задать десятичной дроби, составляет 38 десятичных знаков, чего недостаточно для ряда задач, например, вычислений, связанных с переводом валют. В этой работе мы расширим максимальную точность типа Decimal за счет добавления целых чисел расширенной точности.

Помимо типа Decimal в рамках работы в СУБД ClickHouse для решения поставленной задачи будет добавлены большие целые числа — размером 128 и 256 бит. Одна из областей, где необходимы целые числа большой размерности — криптографически стойкие алгоритмы и криптографические приложения. Например, блокчейн-системы, такие как Биткоин или Ethereum, используют 256-битные целые числа как идентификаторы блоков. Результаты данной работы позволят использовать СУБД ClickHouse в качестве хранилища в системах, работающих с блокчейном.

2. Модели вещественных чисел с фиксированной запятой

За все время существования компьютерных технологий были предложены различные способы представления вещественных чисел с фиксированной точностью. В этой главе мы рассмотрим два наиболее распространенных варианта.

2.1. Двоично-кодированные десятичные числа с фиксированной точностью

Одна из классических моделей вещественных чисел — представление вещественного числа как массив десятичных цифр. Один из способов реализовать это представление — задавать каждую цифру от 0_{10} до 9_{10} двоичными числами от 0000_2 до 1001_2, по 4 бита на цифру. Так как современные компьютеры работают с байтами, часто встречается реализация, когда на каждую цифру отводят 1 байт, то есть 8 бит (например, реализация в СУБД PostgreSQL, рассмотренная далее в работе).

Положительным свойствам такого представления можно назвать простое преобразование чисел в строковое представление — каждый байт в числе однозначно соответствует цифре, и никаких дополнительных вычислений не требуется. Простота перевода важна, например, для калькуляторов и других вычислительных устройств с небольшой вычислительной мощностью.

0000	0	0100	4	1000	8	1100	
0001	1	0101	5	1001	9	1101	
0010	2	0110	6	1010		1110	
0011	3	0111	7	1011		1111	

Таблица 2.1.1. Десятичные цифры и их двоичное представление.

2.2. Двоичные вещественные числа с фиксированной точностью

Несмотря на описанные достоинства, у двоично кодированное модели есть важные недостатки. Первый важный недостаток — неоптимальное использование памяти. В случае, когда каждая цифра кодируется как 4-битное число, двоичные последовательности от 1010_2 до 1111_2 отмечаются как некорректные и не используются, тем самым имеют смысл только 60% возможных комбинаций (см. таблицу 2.1.1). В случае, если для представления

каждой цифры используется целый байт, использование памяти очевидным образом еще более неэффективно.

Второй важный недостаток — скорость работы. Современные процессоры спроектированы для быстрых операций с двоичными целыми числами, и плохо работают с кодированными последовательностями цифр. Двоичное представление представление использует целочисленную логику для операций, тем самым используя специализацию процессора.

В этом представлении вещественное число хранится как пара из данных (целого числа, value) и параметра scale:

$$x = value \cdot 10^{-scale}$$

2.3. Выводы

Были рассмотрены две распространенных модели представления десятичных вещественных типов с фиксированной запятой — двоично-кодированное и двоичное представление. Двоично-кодированное представление проще в реализации и часто применяется в задачах, требующих просто приведения к строковому виду. Двоичное представление сложнее, но показывает более эффективное использование памяти и лучшую производительность за счет использования оптимизаций по работе с целыми числами, существующими в современных процессорах.

Для СУБД ClickHouse скорость выполнения операций над данными приоритетнее простоты представления, таким образом, с теоретической точки зрения является разумным использования двоичного представления десятичных чисел для решения поставленной задачи. В следующей главе рассматриваются практические реализации десятичных чисел в различных популярных системах управления базами данных с открытым кодом.

3. Обзор существующих реализаций вещественных чисел с фиксированной запятой

Тип Decimal (и его синоним Numeric) был описан в стандарте ANSI SQL 1986 года [4]. Стандарт требует указания двух параметров при задании переменной — precision (общее количество десятичных знаков) и scale (количество десятичных знаков после запятой). Так как стандарт задает только интерфейс типа, его имплементация разнится между разными СУБД. В этой главе кратко рассматриваются реализации типа Decimal в двух популярных СУБД с открытым кодом, а также существующая реализация типа в СУБД ClickHouse.

3.1. Decimal в PostgreSQL

В СУБД PostgreSQL тип Decimal реализован через двоично-кодированное представление вещественного числа [5] — число хранится как массив цифр, каждая цифра типа unsigned char, занимает 1 байт. Операции над числами производятся посимвольно. PostgreSQL позволяет задавать precision (общую точность) до 131072 десятичных знаков и scale до 16383 знаков после запятой.

3.2. Decimal в MySQL

В ранних версиях MySQL тип Decimal был реализован так же, как и в PostgreSQL, однако с версии MySQL 5.0, вышедшей в 2005 году, имплементация была изменена для увеличения производительности. На момент написания работы MySQL реализует тип Decimal используя смесь из двух описанных в предыдущей главе моделей [6]. Число хранится как набор блоков по 9 цифр в каждом, каждый блок — беззнаковое 32-битное целое от 0 до 999 999 999, тем самым занимая по 4 байта на блок. Крайние блоки (слева и справа) оптимизируются, чтобы занимать меньше места, если цифр меньше чем девять.

Операции производятся поблоково, каждый блок приводится к 64-битному целому для избежания переполнений. MySQL позволяет задавать точность до 65 десятичных цифр.

3.3. Существующие Decimal меньшей точности в ClickHouse

В ClickHouse используется двоичное представление вещественных чисел, основанное на знаковом целом различной разрядности. В настоящее время доступны реализации на 32-битных, 64-битных и 128-битных целых числах, максимально допустимая точность составляет, соответственно, 9, 18 и 38 знаков после запятой.

3.4. Выводы

Имплементация Decimal в PostgreSQL позволяет хранить и обрабатывать значительно большие числа, чем реализация MySQL или ClickHouse, жертвуя при этом производительностью. MySQL, начиная с версии 5.0, использует более производительную реализацию, однако ограничивает точность типа 65 десятичными знаками.

Стремясь расширить существующую в ClickHouse имплементацию, мы будем реализовывать тип Decimal, используя тот же подход, что используется для существующих. Помимо лучшей производительности, такой подход позволит в дальнейшем относительно просто продолжать расширять точность, если такая необходимость появится.

Для вышеописанной реализации типа Decimal256 необходимо добавить в ClickHouse знаковые 256-битные целые. Максимально возможная точность нового типа составит 77 десятичных знаков. В следующей главе рассматриваются и сравниваются подходы по добавлению больших целых

чисел в ClickHouse, в частности, готовые библиотеки с открытым кодом, реализующие логику больших целых чисел.

4. Реализация больших целых чисел

СУБД ClickHouse написана на языке программирования C++. Для дальнейшей удобной работы у класса больших целых чисел должны быть реализованы арифметические операции и константы, необходимые для интеграции в ClickHouse. Для того, чтобы не повторять уже существующие проверенные и оптимизированные решения, были рассмотрены ряд библиотек для работы с большими целыми числами, описанных далее.

4.1. GNU Multiprecision Library

Библиотека GMP активно используется в различных научных и криптографических приложениях, например, активно используется в программном пакете Wolfram Mathematica. GMP реализует знаковые и беззнаковые целые, как фиксированной, так и нефиксированной точности.

```
mpz_t x, y;
mpz_init_set_ui(x, 0xFFFFF323523DD);
mpz_init_set_str(y, "25565", 10);

mpz_mul(x, x, y);
mpz_out_str(stdout, 10, x);
mpz_clear(x);
mpz_clear(y);
```

Листинг 4.1.1. Пример работы с библиотекой GNU Multiprecision Library.

К сожалению, поскольку библиотека написана на языке C, ее интерфейс был сочтен неудобным для интеграции в ClickHouse — перед использованием переменных необходимо вручную инициализировать переменные и

аллоцировать память, отсутствуют перегруженные операторы. У библиотеки есть заголовочный файл, перегружающий операторы и предоставляющий интерфейс для C++, однако авторы библиотеки не дают гарантий о стабильности API и отсутствии потенциальных изменений. ClickHouse активно развивается, и нестабильный интерфейс внутренней библиотеки усложнит работу и принесет неудобства в будущем.

Помимо этого, библиотека рассчитана на работу с очень большими числами, и при размере целого до 1024 бит проигрывает по производительности другим альтернативам. По этим причинам было решено не использовать библиотеку GMP в данной работе.

4.2. Boost.Multiprecision

Библиотека реализует большие числа классом из двух компонент — фронтендом и бекендом. Компонент-фронтенд `boost::multiprecision::number<backend>` представляет собой интерфейс над компонентом-бекендом, предоставляющий перегруженные операторы и незначительные оптимизации производительности. Библиотека также включает в себя примеры различных компонент-бекендов, для нашей задачи мы выбрали предложенный бекенд `boost::multiprecision::cpp_int_backend<>`, реализующий целые числа фиксированной точности, знаковые и беззнаковые. Данные хранятся в виде массива 64-битных беззнаковых целых.

```
boost::multiprecision::int256_t x{0xFFFFF323523DD};
x <<= 65;
x += 1;

boost::multiprecision::int256_t y;
std::cin >> y;
std::cout << x * y << std::endl;
```

Листинг 4.2.1. Пример работы с библиотекой Boost.Multiprecision

4.3. Предложение по включению в стандарт C++ больших целых чисел

В 2017 году появилось предложение по добавлению в стандарт языка C++ и библиотеку Standard Template Library шаблонного класса `wide_integer` [7]. Класс должен реализовывать большие целые числа заданной двоичной разрядности, заданной в качестве шаблонного аргумента. В качестве внутреннего представления данных предлагается использовать беззнаковые 8-битные числа. Авторы аргументируют свой выбор тем, что блоки размером в 1 байт позволят сделать новый класс максимально гибким, практически не жертвуя при этом производительностью и скоростью выполнения на современных процессорах.

На момент написания работы (май 2020 г.) предложение и код, прилагаемый к предложению, находились в состоянии доказательства жизнеспособности концепции, не были приняты в стандарт C++20 и будут приняты в стандарт не ранее редакции C++23.

4.4. Выводы

Был рассмотрен подход к реализации целых чисел, предлагаемый к включению в стандарт языка C++. К сожалению, на момент написания работы работа над предложением не была завершена, в связи с чем было принято решение не использовать код, сопровождающий предложения на выполнения поставленной задачи.

Также были рассмотрены две библиотеки, реализующие работу с большими целыми числами — GNU Multiprecision Library (GMP) и Boost.Multiprecision. GMP является библиотекой, написанной на языке программирования C, и, как следствие, предоставляет неудобный для

поставленной задачи интерфейс (ClickHouse написан на языке C++ и активно использует такие средства объектно-ориентированного программирования, как классы и шаблоны). Помимо этого библиотека GMP рассчитана на работу с очень большими числами (десятки и сотни тысяч двоичных знаков), и показывает проигрыш в производительности на числах до 1024 двоичных знаков.

В свою очередь, Boost.Multiprecision предоставляет удобный интерфейс и реализует почти всю необходимую логику. По итогам исследования подходов и библиотек было принято решение для выполнения поставленной задачи использовать библиотеку Boost.Multiprecision.

5. Реализация новых типов в СУБД ClickHouse

После рассмотрения возможных вариантов реализации больших целых чисел в ClickHouse, было принято решение добавлять большие целые и тип Decimal 256-битной разрядности согласно следующему плану:

1. Добавление больших целых чисел (типы Int128, UInt128, Int256, UInt256) на основе библиотеки Boost.Multiprecision.
2. Интеграция больших целых чисел с набором операций над данными, реализованными в ClickHouse
3. Исследовать производительность созданной функциональности в сравнении с целыми числами меньшей размерности
4. Реализация десятичных дробей расширенной точности (тип Decimal256) на основе добавленных больших целых.
5. Интеграция нового типа с существующими операциями
6. Исследование производительности типа Decimal256 и сравнение с производительностью типов Decimal меньшего размера.

Одной из первых подзадач работы было задание для новых type traits — констант, характеризующих типы данных и позволяющих предпосчитать часть операций на этапе компиляции программы, например, знаковость типа, максимальное и минимальное значение, является ли тип целым или нет, приведение знакового типа к соответствующему беззнаковому и наоборот. Для большинства типов эти константы заданы в стандартной библиотеке языка C++, однако попытки расширения этих констант ведут к потенциальным проблемам, и, как следствие, type traits переопределяются отдельно, оставляя возможность расширять константы для новых типов.

```
template <typename T>
struct make_unsigned
{
   typedef std::make_unsigned_t<T> type;
};

template <> struct make_unsigned<bInt128>  { typedef bUInt128 type; };
template <> struct make_unsigned<bUInt128> { typedef bUInt128 type; };
template <> struct make_unsigned<bInt256>  { typedef bUInt256 type; };
template <> struct make_unsigned<bUInt256> { typedef bUInt256 type; };

template <typename T>
using make_unsigned_t = typename make_unsigned<T>::type;
```

Листинг 5.1. Пример переопределения type traits.

Несмотря на то, что библиотека Boost определяет почти все арифметические операции, конструктор типа помечен как `explicit` — вызываемый явно, а это означает, что все неявные приведения целых типов, существующие в ClickHouse в рамках работы необходимо было провалидировать и указать как явное приведение. Помимо этого, в процессе выполнения работы было выяснено, что поддержка относительно нового типа

`char8_t` еще не была добавлена в библиотеку Boost, и, вследствие, необходимы промежуточные приведения.

После проведения базовой интеграции в ClickHouse было необходимо интегрировать новый тип в работу с колонками. Для этого в ClickHouse задан класс-интерфейс IColumn, от которого наследуются все другие классы с колонками. На этом этапе основной сложностью работы было задание новых контейнеров и работы с сериализацией/десериализаций в двоичный формат. До этого момента целые типы были plain old data (POD) — в относительно простом представлении и с гарантиями, предоставляемыми C++, позволяющими, например, копирование данных напрямую из памяти с помощью команд memcpy и memset. Добавляемые же типы не являются POD и требуют вызова конструктора для создания переменной и аллокации памяти, что потребовало правок в существующих методах, копирующих данные.

Следующим этапом работы стала интеграция новых типов в математические функции с колонками и данными в них, уже реализованные в ClickHouse. Для определения типа результата функции используется логика на основании трех параметров типа — знаковость числа, признак целости/вещественности числа и двоичная разрядность. Логика реализована на шаблонных константах и функциях.

```
template <typename A, typename B> struct ResultOfAdditionMultiplication
{
    using Type = typename Construct<
        is_signed_v<A> || is_signed_v<B>,
        std::is_floating_point_v<A> || std::is_floating_point_v<B>,
        nextSize< is_signed_v<A> || is_signed_v<B> >(
            max(sizeof(A), sizeof(B))
      )>::Type;
};
```

Листинг 5.2. Пример вычисления типа результата сложения двух типов.

Помимо требуемой функциональности были добавлены тесты на производительность и корректность работы новых типов данных.

По итогам работы был создан pull request — предложение по поправкам в существующую кодовую базу — с объемом добавлений более 2,5 тысяч новых строк кода, затрагивающих больше 80 файлов. Pull request был опубликован в сервисе GitHub (приложение А).

6. Тесты производительности

6.1. Методология

Для поддержания качества существующей кодовой базы, командой разработки ClickHouse в релизный цикл была внедрена методология Continuous Integration — «непрерывной интеграции» — впервые описанная Греди Бутчем в 1991 году [8]. Согласно этой методологии, любые предлагаемые изменения в кодовой базе перед тем, как оказываются в основной ветке разработки проекта и отправляются в релиз, должны пройти набор тестов, проверяющих корректность существующей логики и потенциальную деградацию в производительности, связанную с предлагаемыми изменениями. Таким образом, предложенные изменения были протестированы на безопасность и отсутствие ухудшения производительности в уже реализованной функциональности.

Однако вышеописанные тесты не проверяют новую функциональность, и для проверки были проведены синтетические тесты, проверяющие производительность новых типов.

Тесты проводились на виртуальной машине сервиса «Яндекс.Облако», с техническими характеристиками:

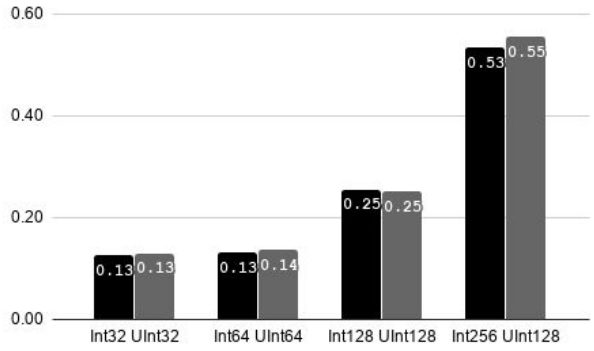
- 8 cores, Intel Cascade Lake (эквивалент процессора Intel Xeon Gold 6230)
- 16 ГБ RAM

На машине был запущен процесс-сервер, для запросов использовался процесс-клиент, запускаемый как:

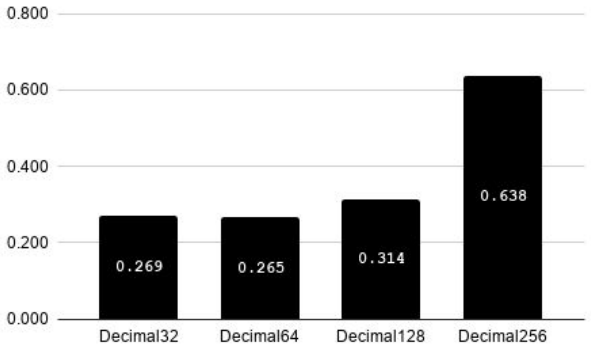
```
clickhouse-client --query “<query>” --time
```

Для измерения времени выполнения запроса использовался встроенный в процесс-сервер таймер, с указанным параметром командной строки `--time` клиент ClickHouse выводит результат измерения таймера в `stderr` — стандартный поток вывода ошибки. В качестве результата бралось среднее из 5 запусков, время указано в секундах.

6.2. Производительность при конвертации в строковое представление

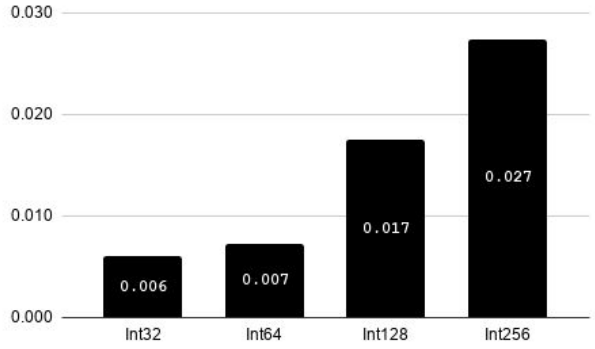


Изображение 6.2.1. Результаты тесты производительности конвертации в строковое представление, целые числа, секунды (Приложение Б)

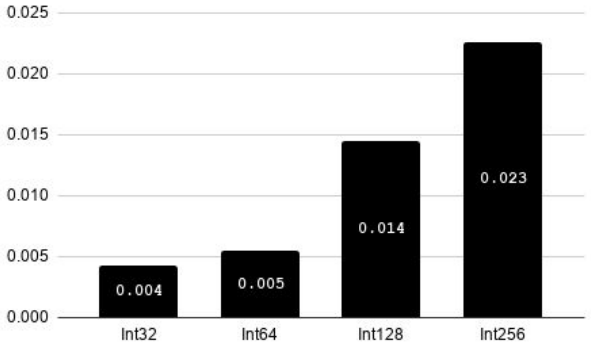


Изображение 6.2.2. Результаты тесты производительности конвертации в строковое представление, десятичные числа, секунды (Приложение В)

6.3. Производительность целых чисел в агрегирующих функциях



Изображение 6.3.1. Результаты тестов производительности на агрегирующих функциях `min, max, argMin, argMax`, целые числа, секунды (Приложение Г)



Изображение 6.3.2. Результаты тестов производительности на агрегирующих функциях `avg, sum, sumWithOverflow`, целые числа, секунды (Приложение Г)

6.4. Выводы

В проведенных тестах для целых чисел можно наблюдать, что начиная с 64-битного целого время выполнения растет пропорционально увеличению разряда числа — операции над 64-битными целыми примерно в два раза быстрее операций над 128-битными, а над 128-битными примерно в два раза быстрее, чем над 256-битными. Поскольку время выполнения пропорционально объему памяти, будет верным заметить, что накладные расходы на работу с большими целыми достаточно малы, и, в целом, реализация больших целых чисел в библиотеке Boost.Multiprecision достаточно хорошо оптимизирована.

7. Заключение

Десятичные вещественные числа используется в областях вычислений, где важна точность, например, в финансовых данных. В этой работе были рассмотрены две возможных модели представления десятичных чисел с фиксированной запятой - двоично-кодированное и двоичное представление.

Двоично-кодированное представление чисел проще в реализации и обоснованно в тех случаях, где важно быстро переводить число в строковое представление. Однако на современных процессорах такая модель проигрывает по скорости выполнения операций и требуемой оперативной памяти двоичному представлению.

Тип десятичных чисел с фиксированной точкой задается стандартом ANSI SQL как необходимый (Decimal или, как синоним, Numeric). Оба вышеописанных представления используются в популярных базах данных - в работе были рассмотрены СУБД PostgreSQL, где тип Decimal представлен как двоично-кодированный, и СУБД MySQL, где с развитием базы данных перешли с двоично-кодированного на двоичное представление.

ClickHouse использует двоичное представление десятичных чисел; исходя из стремления поддерживать единообразную кодовую базу и приоритета в скорости выполнения вычислений было решено дополнить уже существующие типы. Для этого необходимо было добавить большие целочисленные типы — размерностью 128 и 256 бит. Были рассмотрены две библиотеки на языке программирования C++, реализующие большие целые числа — GNU Multiprecision Library (GMP) и Boost.Multiprecision. По итогам сравнения было решено использовать библиотеку Boost как более удобную в интеграции в СУБД ClickHouse.

Используя библиотеку Boost.Multiprecision, в ClickHouse были добавлены типы (U)Int128 и (U)Int256. После этого, основываясь на Int256, был добавлен тип Decimal256, расширяя максимальную возможную точность типов Decimal до 77 десятичных знаков. Были проведены тесты производительности, в которых новые типы показали линейную зависимость времени исполнения от объема занимаемой памяти, тем самым позволяя сделать вывод о достаточной оптимизированности целых чисел из библиотеки Boost.Multiprecision.

8. Библиография

[1] IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754, 1985.

[2] European Commission Directorate General II Economic and Financial Affairs, “The Introduction of the Euro and the Rounding of Currency Amounts”, 1997, C. 29.

[3] “ClickHouse - open source distributed column-oriented DBMS”, URL: https://clickhouse.tech/ (дата обращения 20.05.2020)

[4] Information processing systems - Database language - SQL, ISO 9075:1987, 1987.

[5] Исходный код PostgreSQL, версия 12.0, файл src/interfaces/ecpg/include/pgtype_numeric.h. URL: https://github.com/postgres/postgres (дата обращения 20.05.2020)

[6] Исходный код MySQL, версия 8.0, файл strings/decimal.cc. URL: https://github.com/mysql/mysql-server (дата обращения 20.05.2020)

[7] I. Klevanets, A. Polukhin. “A Proposal to add wide int Template Class”. URL: https://cerevra.github.io/int/ (дата обращения 20.05.2020)

[8] G. Booch. “Object Oriented Design: With Applications”, Benjamin Cummings, 1991, C. 209.

9. Приложения

А. Исходный код

Исходный код доступен в системе GitHub по ссылке: https://github.com/ClickHouse/ClickHouse/pull/10388

Б. Тест Integer Parse

```
SELECT count() FROM zeros(10000000)
```

```
     WHERE NOT ignore(toInt32OrZero(toString(rand() % 10000)));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toInt64OrZero(toString(rand() % 10000)));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toInt128OrZero(toString(rand() % 10000)));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toInt256OrZero(toString(rand() % 10000)));

SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toUInt32OrZero(toString(rand() % 10000)));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toUInt64OrZero(toString(rand() % 10000)));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toUInt128OrZero(toString(rand() % 10000)));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toUInt256OrZero(toString(rand() % 10000)));
```

B. Тест Decimal Parse

```
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toDecimal32OrZero(toString(rand() % 10000), 5));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toDecimal64OrZero(toString(rand() % 10000), 5));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toDecimal128OrZero(toString(rand() % 10000), 5));
SELECT count() FROM zeros(10000000)
     WHERE NOT ignore(toDecimal256OrZero(toString(rand() % 10000), 5));
```

Г. Тест Integer Aggregates

```
CREATE TABLE t (x UInt64, i32 Int32, i64 Int64, i128 Int128, i256 Int256)
ENGINE = Memory;
```

```
INSERT INTO t SELECT number AS x, x AS i32, x AS i64, x AS i128, x AS
i256 FROM numbers(1000000);

SELECT min(i32), max(i32), argMin(x, i32), argMax(x, i32) FROM t;
SELECT min(i64), max(i64), argMin(x, i64), argMax(x, i64) FROM t;
SELECT min(i128), max(i128), argMin(x, i128), argMax(x, i128) FROM t;
SELECT min(i256), max(i256), argMin(x, i256), argMax(x, i256) FROM t;

SELECT avg(i32), sum(i32), sumWithOverflow(i32) FROM t;
SELECT avg(i64), sum(i64), sumWithOverflow(i64) FROM t;
SELECT avg(i128), sum(i128), sumWithOverflow(i128) FROM t;
SELECT avg(i256), sum(i256), sumWithOverflow(i256) FROM t;
```