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

Исследовательский проект на тему

Пользовательские функции в ClickHouse

Выполнил студент группы БПМИ166, 4 курса,
Минеев Игорь Евгеньевич

Руководитель ВКР:
Руководитель группы разработки ClickHouse
Миловидов Алексей Николаевич

Москва 2020

Оглавление

1 Аннотация

1.1 Аннотация

ClickHouse — это высокопроизводительная база данных, разработанная Яндексом для онлайн-обработки запросов на огромных объемах данных. В этой работе мною представлен обзор существующих решений для создания пользовательских функций для базы данных. Мною предлагается решение в форме реализации для СУБД ClickHouse. Разработки в области хранения и обработки большого количества информации не только актуальны, но и будут требовать все больше и больше интеллектуальной работы. Каждый день объем хранимых данных растет в геометрической прогрессии. Предприятия в значительной степени используют базы данных для получения большей прибыли от своих клиентов. Результат этой работы позволит ClickHouse быть более гибким и удобным для разработки и использования.

1.2 Annotation

ClickHouse is a high-performance database developed by Yandex for online processing queries on huge amounts of data. In this paper I present an overview of existing solutions for creating user-defined database functions. I aggregate the results and propose a solution to this issue in the form of implementation for ClickHouse. Developments in the field of storage and processing of a large amount of information are not only relevant now but will continue to require a lot of mental work. Every day, the amount of stored data is growing exponentially. Businesses use databases to a large extent to get more value from their customers. The result of this work will allow ClickHouse to be more flexible and convenient for development.

1.3 Ключевые слова

ClickHouse; СУБД; User Defined Functions; Python UDF; Безопасность

2 Введение

2.1 ClickHouse

Несколько лет назад Яндексом была разработана колоночная система управления базами данных (СУБД) ClickHouse [1] для нужд Яндекс Метрики - системы веб-аналитики третьей по популярности в мире. За счет удобной работы сразу со многими нужными колонками таблицы, она позволяет быстро выполнять аналитические запросы с большими данными, которые пользователи описывают на собственном языке ClickHouse, близком к SQL. К сожалению, в этом языке есть ограниченное количество функций, хотя сейчас всё больше и больше компаний из разных сфер используют ClickHouse. Не столь большое количество всевозможных функций, а также отсутствие UDF на данный момент обусловлено следующими проблемами:

Во-первых, СУБД должно отвечать высоким гарантиям надежности. Никакая компания не станет использовать базу данных, которая допускает ошибки и неточности в работе, не гарантирует целостность данных и не гарантирует устойчивость при длительной работе. Поэтому разработчикам ClickHouse приходится подтверждать ее работоспособность единственным разумным способом - написанием тестов. Наличие большого количества функций и их совместное применение может приводить к комбинаторному взрыву количества тестов, что недопустимо ввиду усложнения разработки и скорости тестирования БД.

Во-вторых, тривиальная реализация UDF подразумевает доступ пользовательской функции ко всей памяти основного процесса СУБД, что в свою очередь недопустимо при наличии политик изоляции данных между различными группами лиц. Это может приводить не только к нарушению политики безопасности (что является допустимым, если базой пользуются только доверенные лица, то есть существует внешняя гарантия безопасности), но и к некорректной работе базы данных, в том числе и к изменению самих хранимых данных.

2.2 Цели и задачи

- Исследовать возможность простого внедрения пользовательских функций в ClickHouse.
- Предложить реализацию Python UDF.
- Предложить реализацию C(C++) UDF.
- Предложить политики и реализации, реализующие безопасность выполнения пользовательских функций.

2.3 Актуальность

Описание задачи и первые pull requests [2] (запрос на изменение исходного кода) вызвали много обсуждений и предложений на GitHub. Сейчас компаниям приходится нанимать сотрудников хорошо разбирающихся в ClickHouse для написания дополнительной, необходимой этим компаниям, функциональности. Миграция между версиями порой требует повторного тестирования и уточнения совместимости функций. Введение UDF позволит огромному количеству пользователей ClickHouse получать результат сравнительно меньшими трудозатратами за короткое время, а также не испытывать проблем с проверкой совместимости в некоторых случаях.

2.4 Основные термины

1. **Процесс-рабочий** - worker - единица потока, выполняющего вычислительную логику.
2. **СУБД** - Система Управления Базами Данных.
3. **UDF** - User Defined Functions - функции, заданные пользователем СУБД. В отличие от других функций, могут быть заданы во время работы СУБД.
4. **Factory** - объект, умеющий регистрировать и выдавать объекты по имени. Обычно является SingleTon или статичной переменной, доступной из любой точки программы.
5. **Case Sensitive/Insensitive** - чувствительность имени к регистру. Иными словами, возможность идентифицировать объект по имени без или с учётом регистра.
6. **Заголовок функции** - описание, содержащее имя, а также аргументы (имена и/или типы) функции.
7. **Окружение в Python** - применимо к Python переменной. Область видимости, ограниченная данным местом в данном коде Python.
8. **UNIX** - семейство переносимых, многозадачных и многопользовательских операционных систем.
9. **Аллокация/деаллокация** - функция выделения / указания неиспользуемости области памяти.
10. **COW** - Copy On Write - механизм оптимизации работы с памятью. Идея заключается в использовании общей копии в режиме чтения, если же приложение изменяет данные, то прежде оно создает копию объекта.

3 Обзор источников

3.1 MonetDB

Популярная в узких кругах, быстро развивающаяся, легковесная колоночная база данных. MonetDB [3] предоставляет возможность использования Python и C UDF.

Инициализация Python глобальна и происходит в функции PyAPIprelude, где кроме прочего импортируется модуль Numpy, а также вспомогательный модуль Marshal, используемый для преобразования данных между Python Object и непосредственным типом, с которым работает БД.

3.1.1 Python Scalar UDF

MonetDB предоставляет возможность создавать функции напрямую в пользовательском запросе. Входной формат полностью повторяет строение Python функции за исключением заголовка - в нем представлен “Create Function” SQL запрос. При этом для каждой созданной функции создаётся отдельный Python модуль с телом функции из запроса.

Выполнение функции происходит по требованию пользователя (при вызове такой функции по имени). Также возможно многопоточное выполнение.

3.1.2 Python Table UDF

Данная функция создаёт таблицу. Формат возвращаемого значения - Python словарь с необходимыми столбцами. Словарь отображает имя столбца в колонку с данными. Проверка корректности данных происходит только после выполнения UDF. Запрос не отличается от Scalar, кроме типа возвращаемого значения - таблицы.

3.1.3 Python Aggregate и Filter UDF

Благодаря тому, что возвращаемое значение из функции может быть любым (в том числе иметь любой размер) MonetDB поддерживает возможность

фильтрации и агрегирования данных. Функция фильтра абсолютно не отличается от Scalar, в то же время Aggregate function имеет собственный SQL тип Aggregate.

3.1.4 Внутреннее строение Python UDF

MonetDB, благодаря своей модели данных, реализует изящное решение, которое обходит Python GIL (подробнее в 3.5): при использовании нескольких потоков вызывается fork (раздвоение исполняемой программы) основного потока со взятой блокировкой GIL. Все входные столбцы остаются COW, при этом каждый дочерний процесс имеет внутри свой интерпретатор и может использовать свою блокировку независимо. Запись результата происходит в shared mmap область памяти, выделенную заблаговременно.

Данный подход приводит к большому количеству ошибок доступа записи к страницам памяти сразу после fork из-за COW, а также имеет жесткие ограничения по размеру виртуальной памяти. Тем не менее Python UDF реализован просто и эффективно.

3.1.5 Другие UDF

MonetDB предоставляет возможность написать пользовательские функции на языке C. Пользовательские функции состоят из двух файлов.

Первый - непосредственный C код функции, принимающий на вход указатели на типы, объявленные в заголовочном файле udf.h, будь то входные или выходные аргументы (при этом выходные аргументы идут до входных).

Второй - файл с расширением “mal”, описывающий соответствующий “Create Function” SQL запрос, с помощью которого и происходит инициализация и поиск соответствующего метода в скомпилированном из исходного кода объекте.

В такой реализации сложно гарантировать безопасность ввиду прямого исполнения произвольного клиентского кода на языке C в неизолированной среде.

3.2 Postgres

Одна из известнейших баз данных, ставших классическими. Предоставляет возможность создавать функции на собственном SQL совместимом языке PostgreSQL [4]. Запрос не отличается от подобного в MonetDB. Обусловлено это тем, что именно Postgres является первоначальным создателем стандарта функций на процедурном PostgreSQL языке.

Запрос имеет вид:

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr
] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
} ...
```

И описывает создание или замену функции на новую с заданным именем, языком, аргументами, возвращаемым значением и телом функции.

3.3 Sybase IQ

Колоночная реляционная база данных [5], используемая в бизнес-аналитике.

Предоставляет многие виды UDF функций, таких как Scalar, Table и Aggregate [6]. Все UDF распространяются в виде уже готовых скомпилированных shared библиотек для динамической линковки. Библиотеки линкуются напрямую в рабочие процессы. Такой способ позволяет реализовывать UDF на любом языке программирования, который можно вызвать с помощью C Calling Convention [7], то есть любой современный компилируемый или интерпретируемый язык программирования, при условии написания необходимой прослойки на C или C++.

Безопасность не может быть полностью гарантирована ввиду исполнения произвольного кода динамической библиотеки.

3.4 ClickHouse

3.4.1 Структура функций

Для того чтобы понять, как именно лучше всего сделать пользовательские функции, необходимо подробно изучить строение существующих функций.

Все функции в ClickHouse делятся на три основные глобальные группы [1]:

1. **Functions** - обычные scalar функции.

Унаследованы от базового класса IFunction, предоставляющего интерфейс для получения имени функции, типа возвращаемого значения, исходя из типов входных аргументов и их порядка, и, непосредственно, самого вызова функции.

Многие функции на современных процессорах можно выполнить быстрее, если выполнять сразу несколько однотипных базовых инструкций, а точнее, если использовать специальные соответствующие инструкции, которые могут выполнить одновременно одну операцию с несколькими значениями. Во многих операциях используются не все данные, а только некоторые столбцы, а значит оптимальнее хранить данные не построчно, а поколоночно: именно таким способом избегается лишняя обработка незадействованных данных. ClickHouse предназначен для обработки Петабайт, а значит не имеет возможности загрузить в память сразу все данные. Именно поэтому метод для вызова функции принимает на вход Block - структуру, содержащую некоторый отрезок строк всех необходимых входных (обрабатываемые данные) и выходных (результат) колонок.

2. **Aggregate Functions** - функции агрегирования данных.

Унаследованы от класса IAggregateFunction, предоставляющего структуру для хранения агрегированного состояния; аналогичный IFunction метод,

возвращающий тип итогового значения; метод add, обновляющий состояние новыми записями.

Современные процессоры умеют выполнять более одного программного потока одновременно, что позволяет распараллелить и тем самым ускорить обработку. Необходимость в отказоустойчивости и достаточной скорость работы вынуждают архитекторов баз данных использовать кластер из компьютеров. Именно по этим двум причинам, IAggregateFunction также предоставляет метод merge, объединяющий результаты работы с нескольких потоков и/или компьютеров.

3. **Table Functions** - функции, предоставляющие таблицы.

Унаследованы от ITableFunction. Такие функции используются, чтобы выполнить некоторую задачу на другой машине, на таблице, которой может не существовать в виде табличной структуры данных (например, случайные величины или нулевые значения) и для других целей.

3.4.2 Фабрики функций

Для каждого типа функций существует своя фабрика (Factory) функций. Она позволяет регистрировать, удалять и находить функции по имени.

Фабрика принимает Creator - объект, умеющий создавать Resolver - объект, который создает непосредственно функцию.

К сожалению, текущая реализация фабрики не умеет работать с несколькими потоками одновременно. Все Resolvers инициализируются на старте программы из одного потока и лишь затем используются процессами-работниками.

Resolver позволяет компилировать некоторые функции на языке C++ на лету благодаря встроенному в ClickHouse компилятору clang.

3.4.3 Типы значений

ClickHouse предоставляет множество различных типов данных, в том числе и комбинации из них (например Vector<String> - вектор строк в качестве

значения ячейки таблицы). Работа с типами организована посредством соответствующей DataTypeFactory, которая умеет конструировать тип по определяющей его строке.

3.4.4 Текущее состояние UDF

ClickHouse не поддерживает пользовательские функции.

3.4.5 Обработка запроса пользователем

Каждый запрос преобразуется в Abstract Syntax Tree (AST), оптимизируется и выполняется советующими вызовами методов объектов, отождествляемых таблицами.

3.5 Python

3.5.1 CPython

Современный высокоуровневый интерпретируемый язык программирования, написанный на си [8]. Благодаря огромному количеству библиотек Python позволяет написать программу легче и быстрее, чем его конкуренты. Легкость синтаксиса позволяет обучить большое количество программистов в короткий срок. К сожалению, данный язык известен и своей плохой производительностью в угоду вышеописанных свойств.

Стандартная CPython библиотека предоставляет доступ ко всем необходимым компонентам интерпретатора. Модель данных в Python не позволяет в правильной мере делать приложения с несколькими потоками, так как работа с памятью, а именно выделение объектов и сбор мусора, должны выполняться однопоточно — это называется Global Interpreter Lock (GIL). Данная архитектура работы интерпретатора была выбрана задолго до появления мощных многопоточных процессоров [9] и по сей день остаётся неизменной в угоду обратной совместимости, а также скорости работы в однопоточном режиме.

Существуют и другие реализации Python, не содержащие GIL, такие как IronPython или PyPy-STM. К сожалению, их реализации не поддерживают некоторые важные библиотеки, а также производительность при некоторых обстоятельствах оставляет желать лучшего.

3.5.2 Numpy

Один из самых распространенных Python-модулей для обработки данных [10]. Предоставляет интерфейс для работы с низкоуровневыми массивами, представляющими из себя непрерывные куски памяти, а также всеми производными от них (многоуровневые массивы, массивы Python-объектов и другие). Numpy Array - идеальное представление данных из ClickHouse в Python, предоставляющее большое количество эффективных scalar и aggregate функций над ними.

Numpy Array умеют работать достаточно корректно без GIL в многопоточной среде. На самом деле, реализация не полностью избавлена от GIL, тем не менее, многие операции над уже выделенными в памяти объектами не требует блокировки.

3.5.3 Boost.Python

Удобная обертка на CPython [11], предоставляющая C++ интерфейс для выполнения скриптов, хранения и использования Python объектов.

4 Описание предлагаемого алгоритма

4.1 Function Factory

Усовершенствуем FunctionFactory и AggregateFunctionFactory для поддержки многопоточности. Для этого в каждой из фабрик был создан новый User Defined Function Map (отображающий имя функции в UDF Creator, который в точности повторяет обычный Creator). В фабрики были добавлены

мьютексы, ограничивающие доступ к этой конкретной Map, так как пользовательские функции будут добавляться во время работы процессов-рабочих. Для удобства будем считать все имена UDF функций как case sensitive.

4.2 Python Scalar UDF

4.2.1 Типы данных

Для всех типов DataType создадим метод, возвращающий соответствующий ему тип представления в Numpy.

4.2.2 Тип функций

Создадим класс FunctionPython унаследованный от IFunction. Его конструктор принимает имя, тело функции, тип возвращаемого значения в виде DataTypePtr, а также список из пар DataTypePtr с именем - аргументы функции.

getReturnTypeImpl - метод, который сравнивает входные типы аргументов с хранимыми типами аргументов. В случае неудачи выбрасывается исключение. В случае успешного выполнения возвращаем тип возвращаемого значения.

execImpl метод блокирует Python GIL и состоит из трех шагов:

1. Преобразование входных аргументов в соответствующие Numpy объекты.

Объекты, представляемые непрерывным участком памяти, преобразуются напрямую через указатель. Прочие объекты или создают соответствующие новые Numpy типы или не поддерживаются в качестве аргументов Python функции. Данные, хранящиеся в этих массивах (array) не могут быть изменены, так как не выставлен соответствующий флаг Writeable (флаг, разрешающий запись в Numpy array).

2. Вызов функции.

Тело функции представляет из себя Python функцию без заголовка и подставляется в следующий шаблон:

```
def udf():
  # FunctionBody

result = udf()
```

Полученный таким образом python код исполняется с помощью boost::python::exec. В качестве глобального и локального окружения используется Python Dict, содержащий необходимые при работе стандартные модули (numpy, sys, time и другие) с соответствующими именами, а также Numpy Arrays со входными данными с заданными пользователем именами.

3. Обработка возвращаемого значения.

После выполнения в объекте окружения появляется переменная с именем result, которая и содержит итоговые значения. Данные значения копируются с помощью функции memcpy в буффер, отождествляющий возвращаемое значение. Данного копирования нельзя избежать ввиду того, что переменная result появляется с помощью интерпретатора Python. Если же создать переменную result заранее, то присвоение в нее значения, возвращаемого из пользовательской функции, затрёт её и в таком случае все равно потребуется копирование.

4.3 Хранение функций

Для хранения функций заведем специальную таблицу SystemStoragePythonFuncionsSources. Каждая запись в этой таблице будет содержать:

1. Name - полное имя функции
2. Return Type - строка, задающая тип возвращаемого значения
3. Arguments Types - массив из пар, задающих тип переменной (в виде строки) и её имя, используемое в качестве аргументов Python функции

4. Body - тело функции

Как уже известно из пункта 3.4.1, необходимо для функции знать тип входных аргументов, а также тип возвращаемого значения. DataTypeFactory позволяет получить указатель на тип с помощью имени типа, а уже из указателя на тип получить представление в Numpy для аргументов скрипта, поэтому все типы хранятся строкой.

В отличие от C++, Python не позволяет иметь несколько объектов (в том числе функций) с одинаковым именем, поэтому используется стандартный OverloadResolver, не представляющий возможность перегрузки функций.

4.3.1 Восстановление функций при запуске

Данная таблица хранится на диске в бинарном построчном виде. При инициализации обращаемся к FunctionFactory и восстанавливаем все функции соответствующим конструктором класса FunctionPython.

Указатели на типы данных получаются с помощью DataTypeFactory из отождествляющих эти типы строк.

4.3.2 Создание новой функции

Воспользуемся стандартным способов, описанным в Postgres и реализованным в MonetDB - создадим новый тип SQL запроса:

```
CREATE FUNCTION name
        (
        name1 type1,
        name2 type2,
        …
        )
RETURNS type LANGUAGE lang_name {
        function_body...
}
```

, где name - строка - имя функции,

name1, name2, … - строки - имена аргументов функции,

type1, type2, … - строки - типы соответствующих аргументов,

type - строка - тип возвращаемого значения,

lang_name - строка - имя языка,

function_body - строка между “{“ и “}” - тело функции.

ClickHouse AST позволяет задать такой тип запроса. При исполнении используем ту же функцию, что и при загрузке Python UDF таблицы, подставляя соответствующие аргументы.

4.3.3 Пример функции

Создадим таблицу с двумя колонками и заполним ее какими-то значениями:

```
CREATE TABLE example (id int, name String) ENGINE = MergeTree() PARTITION BY id ORDER BY (name, id);
INSERT INTO example VALUES (1, 'A'), (2, 'B');
```

Создадим функцию python_example, принимающую на вход два столбца с числами, и возвращающую столбец с числами:

```
CREATE FUNCTION python_example ( i int, j int ) RETURNS int LANGUAGE Python {
return i + j * 2;
}
```

Выполним функцию, в качестве аргументов используя одну и ту же колонку таблицы:

```
SELECT python_example(id, id) FROM example;
```

Результат работы - колонка типа int со значением (3, 6) или (6, 3).

4.4 Python Aggregate UDF

4.4.1 Тип функции

Создадим класс AggregateFunctinoPython, унаследованный от IAggregateFunction. Внутри будем хранить экземпляр Python Object (user_aggregate_python_object), содержащего следующие методы:

1. add - принимающий на вход колонку аналогично передаче аргументов в execImpl.
2. merge - принимающий на вход экземпляр Python объекта того же класса.

Методы add и merge IAggregateFunction будут вызывать соответствующие Python методы user_aggregate_python_object.

Проверка getReturnTypeImpl происходит полностью аналогично PythonFunction.

4.4.2 Создание, хранение, восстановление при запуске

Creator хранит внутри себя исходный код класса, введенный пользователем. При создании функции вызывается инстанциация Python класса.

Хранятся функции аналогично PythonFunction в той же таблице.

Для поддержки разделения агрегатных и обычных функций был добавлен еще один столбец, содержащий тип функции - Aggregate или Scalar.

Имя класса, которое создает пользователь не должно отличаться от имени функции.

Пользовательский запрос также переиспользуем от scalar Python UDF, добавив ключевое слово Aggregate

4.4.3 Пример

Воспользуемся таблицей из примера 4.3.3

Создадим функцию python_aggregate_example, принимающую на вход два столбца с числами, и возвращающую сумму частных:

```
CREATE AGGREGATE FUNCTION python_aggregate_example( i int,
j int ) RETURNS int LANGUAGE Python {
class python_aggregate_example:
    def __init__(self):
        self.result  = 0;
    def add(self, i, j):
        self.result += sum(i / j)
    def merge(self, other):
        self.result += other.result
}
```

Выполним функцию, в качестве аргументов используя одну и ту же колонку таблицы:

```
SELECT python_aggregate_example(id, id) FROM example;
```

Результат работы - int со значением 2

4.5 Альтернативные решение для UDF Python функций

4.5.1 Сложность имплементации

К сожалению, предложенный способ достаточно сложен для реализации, так как требует от каждого типа возможность быстрого преобразования целой колонки. Для каждого из десятка различных существующих типов необходимо написать специальную функцию, преобразующую массив.

4.5.2 Альтернативный способ

Гораздо проще и эстетичнее было бы преобразовывать ровно одно значение вместо массива. Переделаем интерфейс PythonUDF.

PythonFunction будет принимать на вход не массивы из объектов, а лишь по одному экземпляру каждого объекта. Для пользователя запрос и хранение остаются неизменными. Изменяется лишь шаблон подстановки функции:

```
def udf(/* ArgNamesSepparatedByComma */):
      # FunctionBody

for i in range(rows_num):
      result[i] = udf(*(data[i]))[i]
```

rows_num - переменная, представляющая input_rows_count из exec.

ArgNamesSepparatedByComma - разделенные запятой имена аргументов.

result - Numpy Array, с выставленным флагом WRITABLE (то есть с возможностью записи в место), соответствующий результату.

data - Экземпляр класса, отождествляющий входные данные и дающий доступ с преобразованием.

4.5.3 Результаты

Благодаря гибкости Python пример 4.3.3 остается актуальным, избежание копирования итогового значения позволяет слегка ускорить выполнение, а более сложная Python архитектура слегка его замедлит.

4.5.4 Aggregate Python UDF

Аналогично изменению в Scalar метод add пользовательского класса теперь принимает не колонки, а одну строку значений.

4.6 Shared Library UDF

4.6.1 Основной алгоритм

Рассмотрим блок-схему, приведенную на рисунке 3.1.

Основные рабочие потоки ClickHouse (Worker Threads) производят чтение, запись и обработку пользовательских запросов. Эти потоки имеют доступ ко всему содержимому Базы Данных (Data Storages) и их некорректная работа может приводить к потере или неправильному изменению данных, остановки работы всего приложения.

Unix-подобные системы умеют разделять оперативную память между различными процессами, в том числе и различными пользователями. Тонкая настройка прав каждого пользователя или группы пользователей позволяет

ограничить доступ к большинству компонент системы: дискам, сети, памяти других приложений, определенным системным вызовам.

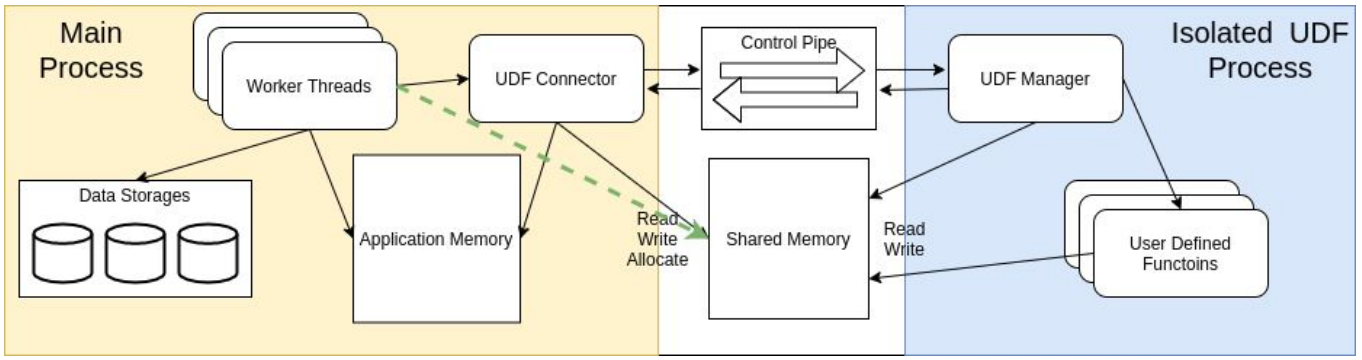

Рис. 3.1: Блок-схема приложения с Shared Library UDF.

Желтым выделены потоки и данные в основном процессе, синим - в изолированном процессе для UDF, белым - данные, которые доступны и основному и изолированному процессу. Блоками обозначены основные компоненты. Стрелки обозначают доступ одного компонента к методам другого.

Создадим изолированный процесс (Isolated UDF Process), в котором и будет происходить обработка всех пользовательских функций.

Создадим общую память для обоих процессов (Shared Memory), а также две однонаправленные Unix Pipe (Control Pipe).

Данные таблиц и результат передаются через Shared Memory, так как Pipe требует, как минимум одного лишнего копирования, что недопустимо ввиду большого объёма самих данных. Control Pipe используется для передачи управляющих команд и их результатов, так как имеет удобное API, позволяющее проверять наличие, блокироваться до поступления, а также быстро обмениваться небольшими командами.

В качестве Control Pipe будем использовать стандартные потоки ввода и вывода.

Пользовательские функции распространяются в виде готовой библиотеки для динамической линковки (shared library) в формате файла “.so”. Каждая библиотека может содержать несколько функций.

При загрузке библиотека сообщает своё имя. Имена библиотек и функций не могут содержать символ “_” в названии. Имя UDF функции имеет

следующий формат: UDF + “_” + lib_name + “_” + func_name, где lib_name и func_name - имена библиотеки и функции соответственно.

4.6.2 UDF Manager

Часть программы, отвечающая за исполнение пользовательских функций в изолированном процессе, называется UDF Manager. Он выполняет следующие функции:

1. Читает из Control Pipe команды (Control Command) и обрабатывает их.
2. Загружает во внутреннюю память и управляет пользовательскими библиотеками функций.

Пользовательская библиотека обернута в класс UDFLib, содержащий методы загрузки, выгрузки и поиска функций exec и getReturnType по имени.

UDF Manager хранит map, отображающий имя библиотеки в UDFLib.

4.6.3 UDF Connector

UDF Connector используется для управления изолированным потоком. Может быть вызван из любого рабочего потока.

Управление всей Shared памятью происходит из Connector. В качестве общей памяти используется boost::interprocess::shared_memory (ссылка), который умеет выделять в памяти область заданного размера.

Идентификатором первичного запроса назовем глобальный порядковый номер каждого непосредственного вызова exec или getReturnType функции.

Connector запускает UDF Manager в отдельном процессе по команде БД.

4.6.4 UDF Control Command API

Manager и Connector обмениваются командами на исполнения (Control Commands) и их результатами (Control Commands Result).

Команда содержит имя (идентификатор) в виде строки, список из аргументов строкового вида, а также уникальный идентификатор запроса (порядковый номер).

Результат содержит код ошибки в виде числа (0 в случае успеха), сообщение об ошибке в виде строки, опциональное поле произвольного типа, а также идентификатор запроса соответствующей команды

Существуют следующие команды:

1) InitLib.

Посылается от Connector к Manager. В качестве аргумента содержит имя файла для загрузки.

При получении данной команды Manager читает библиотеку из файла и отправляет в качестве результата список полных имен функций этой библиотеки (в формате, представленном в пункте 1), запоминает указатели на функции (подробнее изложено в пункте 3.4.5.).

Connector при получении проверяет формат имен функций и добавляет в Фабрику. Добавление происходит с помощью специального класса FunctionUDF, которые хранит имя функции и указатель на Connector. При вызове getReturnTypeImpl и execImpl FunctionUDF вызывает соответствующие методы Connector, который в свою очередь отправляет эти команды далее.

2) GetReturnType.

Посылается от Connector к Manager. В качестве аргумента содержит имя функции, а также список типов, где каждый тип задан строкой.

При получении данной команды Manager разбивает строку по символам “_”, находит имя библиотеки, имя функции согласно формату, описанному в пункте 4.6.1, получает UDFLib, вызывает соответствующий метод. В качестве возвращаемого значения отправляется указатель на структуру DataType, содержащуюся в Shared Memory.

3) ExecFunc.

Посылается от Connector к Manager. В качестве аргумента содержит имя функции, количество строк, а также список аргументов, где каждый аргумент задан в виде указателя IColumn, отождествляющего колонку таблицы.

При получении данной команды Manager вызывает соответствующий метод аналогично GetReturnType. В качестве возвращаемого значения отправляется указатель на структуру с результатом IColumn, содержащуюся в Shared Memory.

4) Stop.

Посылается от Connector к Manager для завершения работы потока. Аналогично работа завершается при закрытии Control Pipe.

5) Allocate.

Посылается от Manager к Connector. Первый аргумент - идентификатор первичного запроса пользовательской функции. Второй аргумент - необходимый размер.

В случае отсутствия необходимого места выбрасывается исключение. Менеджеру сообщается о прекращении выполнения. Все аллоцированные на тот момент данные для этого первичного запроса высвобождаются.

4.6.5 UDF API

Было рассмотрено два варианта:

1) Аналогичное Sybase IQ.

Данное решение позволяет легко интегрировать множество различных языков, благодаря C API. В то же время, необходимо уметь переводить колонки с данными в некоторое C-представление, что возможно не для всех типов и требует большого количества дополнительного кода.

2) Использовать вызовы, аналогичные IFunction::execImpl и IFunction::getReturnType.

Такой способ не полностью совместим с C, так как требует C++ заголовочные файлы для работы, но прост в разработке и долговременной поддержке, так как исходный код для всех типов данных не меняется и используется в качестве UDF API.

4.6.6 Allocator

Реализация выбранного второго способа из пункта 4.6.5 требует предоставить возможность выделять память не в Application Memory, а в Shared Memory (изображение 3.1). Тем самым Worker может напрямую работать с shared memory, что позволяет избежать копирования (зеленая стрелка на рисунке 1).

Практически всё выделение памяти происходит в PODArray. Добавим в качестве дополнительных аргументов аргумент аллокации, а именно указатель на Allocator, умеющий выделять, удалять и реаллоцировать память. Если Allocator не указан - используем стандартные (то есть использовавшиеся до этого).

Также необходимо добавить возможность выделять DataType в SharedMemory. Для этого сделаем аргумент, аналогичный PODArray.

В качестве упрощения задачи возможно передавать DataType в виде строки, пригодной для DataTypeFactory

5 Заключение

5.1 Python UDF

5.1.1 Простота использования

Для использования Python кода с созданными UDF совсем не требуется знать C++, а также специфику работы ClickHouse, поэтому пользователи смогут легко написать собственные методы, зная лишь Python.

Создать новую функцию очень просто - достаточно написать SQL запрос к базе данных. Удаление и замена функций не предусмотрены.

5.1.2 Сложность реализации

Сложным в реализации является необходимость определить дополнительно метод преобразования в Python объект для каждого из десятков типов данных, которые поддерживает ClickHouse. На первом этапе были реализованы переводы ключевых необходимых типов - целочисленных и строковых.

5.1.3 Скорость работы

Сравним реализацию суммы двух колонок, уже существовавшую в ClickHouse с аналогичной, написанной на питоне.

На малых данных имеем не отличимое от погрешности различие, что означает, что дополнительные расходы на инициализацию и хранение UDF небольшие.

На данных среднего и большого размера имеем ухудшение производительности в 2 раза, что допустимо ввиду использования Python.

Более сложные функции могут вызвать большее замедление ввиду того, что оптимизатор не всегда сможет правильно предугадать поведение функции. Также эффективность зависит и от самой реализации функции пользователем.

5.2 C / C++ UDF

5.2.1 Простота использования

UDF на C и C++ сложнее к реализации, так как требуется знание этих языков, а также особенностей типов данных/колонок и работы с ними. Можно сказать, что писать функции также сложно, как и писать новые функции в сам ClickHouse.

Также для использования требуется компиляция исходного кода с заголовочными файлами соответствующей версии БД, а также распространение shared библиотек в виде файла между серверами ClickHouse.

5.2.2 Сложность реализации

Реализация UDF не требует больших временных затрат на имплементацию. Поддержание API в актуальном состоянии не требуется ввиду использования в качестве типов данных и колонок тех же заголовочных файлов, что и для основного ClickHouse.

5.2.3 Скорость работы

Сложность использования компенсируется скоростью работы. Текущая реализация UDF уступает встроенным функциям на 10-15% из-за необходимости выделять память в определенной области. В дальнейшем возможно заменить boost::allocator на другой, который даст чуть больше производительности.

5.2.4 Безопасность

Выбранный способ позволяет полностью обезопасить сохранность данных и работу приложения при правильном использовании UDF функций. Изолированный процесс гарантирует безопасность неиспользуемых при UDF данных, гарантирует, что пользовательский код не сможет каким-то образом навредить системе и процессам ClickHouse.

6 Ссылки

1. Yandex. СУБД ClickHouse [Электронный ресурс] // GitHub: [сайт]. URL: http://github.com/ClickHouse/ClickHouse (дата обращения: 30.4.2020).
2. Community UDF discussion and proof of concept [Электронный ресурс] // GitHub: [сайт]. URL: https://github.com/ClickHouse/ClickHouse/pull/6991 (дата обращения: 30.04.2020).
3. MonetDB B.V. Official Monet DB documentations [Электронный ресурс] URL: https://www.monetdb.org/Documentation (дата обращения: 30.4.2020).
4. The PostgreSQL Global Development Group. Официальная документация об Procedural Language [Электронный ресурс] URL: https://www.postgresql.org/docs/current/xplang.html (дата обращения: 30.04.2020).
5. Sybase Technical. Sybese IQ UDF [Электронный ресурс] [2011]. URL:

http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc01034.1540/doc/html/jfo1253821858770.html (дата обращения: 30.4.2020).
6. Hannes M., Mark R. Vectorized UDFs in Column-Stores 2016.
7. Ferrari A., Batson A., Lack M., Jones A. The 64 bit x86 C Calling Convention URL: https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf (дата обращения: 30.4.2020).
8. Python. Python official GitHub [Электронный ресурс] URL: https://github.com/python/cpython (дата обращения: 30.4.2020).
9. Pulleyn I. Embedding Python in Multi-Threaded C/C++ Applications 2000. URL: https://www.linuxjournal.com/article/3641 (дата обращения: 30.4.2020).
10. Numpy official GitHub [Электронный ресурс] URL: https://github.com/numpy/numpy (дата обращения: 30.4.2020).
11. Boost Community. Official Boost Python Documentation [Электронный ресурс] URL: https://www.boost.org/doc/libs/1_73_0/libs/python/doc/html/index.html (дата обращения: 30.4.2020).