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

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА
Программный проект
на тему
Генерация искусственных данных для тестирования заданных запросов. Обфускация запросов для тестирования ClickHouse.

Выполнил студент группы БПМИ175, 3 курса
Ильговский Роман Максимович

Руководитель КР:
Руководитель группы разработки ClickHouse, Яндекс.Технологии
Миловидов Алексей Николаевич

Куратор:
Кандидат технических наук, доцент
Сухорослов Олег Викторович

Содержание

Генерация искусственных данных для тестирования заданных запросов. Обфускация запросов для тестирования ClickHouse.

Аннотация

Автоматизация тестирования запросов в базу данных полезна для выявления ошибок. Она не только позволяет разработчику реализовывать функциональные тесты, но и поможет отловить ошибку при обращении клиента и наличии у него неработающего или неправильно работающего запроса. В данный момент разработчикам СУБД Clickhouse приходится вручную создавать таблицы и генерировать значения для тестирования определенного функционала, данная работа реализует автоматическую генерацию искусственной базы данных, которая будет учитывать тонкости запроса и покроет большое количество крайних случаев при его выполнении в коде. Используя существующие методы парсинга запросов в Clickhouse, алгоритм должен выявить структуру существующих таблиц, а также для каждой колонки подобрать набор значений, который покрывает условия запроса.

Automation of database queries testing is useful for identifying errors. It not only allows the developer to implement functional tests, but also helps to catch an error when a client has a broken or incorrectly working request. At the moment, developers of the Clickhouse DBMS have to manually create tables and generate values for testing certain functionality. This work implements automatic generation of an artificial database, which will take into account the subtleties of the query and cover a large number of corner cases when executing it in the code. Using existing query parsing methods in Clickhouse, the algorithm must identify the structure of existing tables, as well as select a set of values for each column that covers the query conditions.

Ключевые слова:
Clickhouse, тестирование баз данных, автоматическая генерация тестовых данных, тестирование СУБД.

1 Введение

ClickHouse является популярной СУБД для онлайн обработки аналитических запросов. Сейчас ClickHouse используется во многих сервисах Яндекса, в том числе для Яндекс.Метрики, второй крупнейшей в мире платформе для веб-аналитики. Поэтому точность работы сервиса очень важна и необходимо большое количество тестов для сохранения правильной работы сервиса и проверки новых версий на наличие ошибок.

Основным подходом для функционального тестирования сейчас является использование запросов на существующую таблицу, либо создание таблицы самостоятельно и заполнение ее данными. Генерация базы занимает часть времени разработчика, а также может привести к человеческой ошибке, из-за чего какой-то конкретный случай будет упущен из тестирования и функциональность не будет в полной мере покрыта тестами.

Данная работа предоставляет возможность генерировать базы данных по заданному запросу автоматически. Имея строку запроса или множество запросов, предложенный алгоритм определит, какую структуру имела исходная таблица, на которой данный запрос выполнялся. Также алгоритм должен определяет возможные значения столбцов в исходной таблице и генерирует данные, которыми будет заполнена тестовая база данных. Таким образом, по запросам алгоритм выдает структуру таблицы и данные, на которых исходный запрос должен успешно применяться, а также выдать непустой результат.

2 Обзор литературы

Основной статьей, на которую делается опора при генерации данных является [1]. Авторы рассматривают SQL запросы на базы данных, содержащие в себе логические выражения на поля таблиц и различные виды присоединений таблиц. Реализован алгоритм на языке Alloy, который по запросу и схемам таблиц генерирует данные для их заполнения. В статье авторы также используют критерий покрытия запроса SQLFpc на базу данных, приведенные в статье [2]. По строке запроса создается набор случаев, если в базе данных существуют комбинации записей для каждого случая, то база данных считается соответствующей данному критерию. Для каждой колонки выбирается множество ее значений, также для некоторых комбинаций колонок определяется набор значений, каждый их которых должен присутствовать в базе данных. Эта статья соответствует первой части выполняемой работы, единственной проблемой является наличие схемы базы данных, которая не предоставляется в нашем случае.

Работы релевантные к данной задаче недостаточно описывают применяемые алгоритмы, а выделяют только краткое описание приложения. Также большая часть существующих доступных решений не подходит конкретно для Clickhouse и генерирует так называемую “dummy data”, то есть массив данных который не подстраивается под запросы, а задает случайные данные.

Обфускация - возможность изменять данные таким образом, чтобы они не теряли свою структуру, но при этом по полученным данным нельзя было ничего сказать о реальных существующих данных. Для обфускации в ClickHouse уже есть готовая модель [3]. Автор применяет ее для преобразования всей базы данных, не теряя структуры, тем самым позволяя реализовать нагрузочные тесты на программу. Данная модель используется в ClickHouse для реализации нагрузочного тестирования, берется существующая база данных и к ней применяется обфускатор. Благодаря реализации обфускатора, преобразованная база сохраняет структуру предыдущей базы, но при этом в ней отсутствуют данные, позволяющие сделать какую-либо аналитику по оригинальным данным.

Основной сложностью при применении данного решения является его неадаптированность к малым количествам данных, в которых связь значений нельзя найти статически, а только синтаксически.

Статей по данной задаче найти не удалось, так как данная задача специфична для конкретной базы данных и ее решение необходимо исключительно разработчикам для отладки работы и реализации поддержки пользователей.

Данные задачи являются специфичными и не имеют явных аналогов. Большая часть существующих решений не подходят для применения в Clickhouse так как опираются на семантику определенного языка, а также не анализируют запрос для определения структур, а используют уже заданную структуру. Нам необходимо частное решение для задачи, встроенное в существующий код.

3 Основная часть

3.1 Анализ запроса в Clickhouse

В данной работе мы будем рассматривать только *SELECT* запросы, то есть те запросы, которые достают данные из таблиц.

Встроенный парсер Clickhouse принимает на вход строку, которая является запросом. В *SELECT* запросе выделяется 14 основных частей: *WITH*, *SELECT, TABLES, PREWHERE, WHERE, GROUP_BY, HAVING, ORDER_BY, LIMIT_BY_OFFSET, LIMIT_BY_LENGTH, LIMIT_BY, LIMIT_OFFSET, LIMIT_LENGTH, SETTINGS*. Мы будем рассматривать части *SELECT, TABLES, WHERE, GROUP_BY, HAVING, ORDER_BY* так как именно в них находятся основные данные, нужные нам для анализа структуры и выявления значений. После анализа запроса, парсер выдает нам древовидную структуру, где каждая вершина является определенной операцией выполнения запроса: функцией над значениями, константой, обозначением и тому подобное. Вершины также имеют свои поддеревья, в которых находятся их аргументы или подоперации. Обходя данное дерево, мы будем пытаться выявить необходимые нам данные.

3.2 Анализ схемы

По запросу необходимо определить возможные таблицы. Имея строку запроса можно понять, какие его части обозначают названия таблиц. Таким образом можно определить их количество и названия в нашей базе данных.

В парсере Clickhouse поддеревом запроса, отвечающим за таблицы из которых мы берем данные, является *TABLES* (Рисунок 1). В нем лежит основная таблица, из которой берутся колонки, а также операции *JOIN*, которые совершаются в запросе. Обходя все вершины в поддереве мы берем названия таблиц и баз данных в которых они лежат, а также их условные обозначения, то есть укороченные названия, выбранные автором запроса. Эти названия понадобятся нам для определения принадлежности колонки в дальнейшем.

Таким образом для запроса мы получаем набор баз данных, а также таблиц и их условных обозначений, по которым делается запрос.

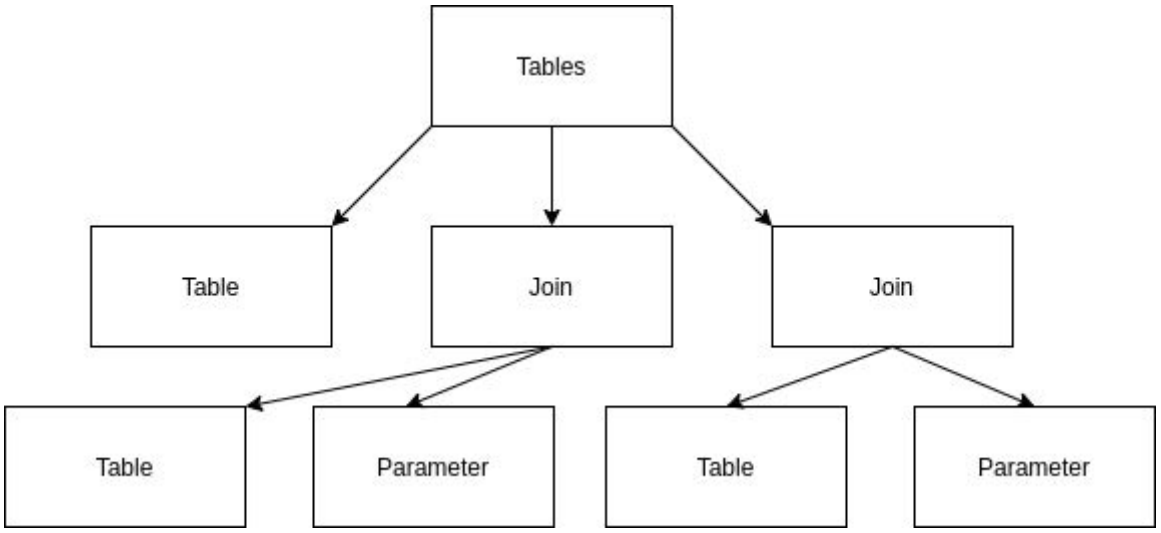

Рисунок 1: дерево Tables

Затем нам необходимо определить множество столбцов, которые присутствуют в запросе и таблицы, к которым они могут относиться. Во время исполнения запроса уже известно множество столбцов в каждой таблице, поэтому при исполнении программа автоматически связывает столбец и таблицу. Однако в нашем случае нельзя однозначно трактовать принадлежность столбца к определенной таблице, например в следующем запросе: “S*ELECT column1, column2, column3 FROM table1 JOIN table2 on table1.column2 = table2.column3*”. Здесь мы однозначно можем сказать, к какой таблице относятся колонки column2 и column3, однако column1 может принадлежать как первой, так и ко второй таблице. Для однозначности трактовки таких случаев, мы будем относить такие неопределенные колонки к основной таблице, по которой делается запрос. Например, в данном случае, это будет таблица table1.

Все столбцы в дереве лежат в вершинах типа *INDENTIFIER*, которые находятся в поддеревьях *SELECT, TABLES, WHERE, GROUP_BY, HAVING, ORDER_BY*. Рекурсивно обходя поддеревья, мы формируем множество всех таблиц. Затем мы разделяем колонку на составляющие: таблица (если она явно указана через точку) и само название. Так как таблица может являться условным обозначением, мы заменяем это обозначение на оригинальное название таблицы. Для столбцов без таблиц определяем основную таблицу запроса.

Теперь у нас есть список всех столбцов и таблиц, к которым они относятся.

3.3 Анализ столбцов

Продолжением является точное определение типов данных для столбцов, у которых в запросе присутствует значение. Примером являются логические условия *WHERE*, в которых у определенного набора атрибутов проверяется логическое выражение. Если в запросе указано “*column > 5”*, то можно сделать вывод, что в данном столбце содержится численное значение, либо если на атрибут применяется выражение *LIKE*, то атрибут представляет собой строковый тип.

В данной части необходимо научится вычленять из запроса все такие выражения и сопоставлять типы данных для тех столбцов, для которых это возможно сделать. При этом понятно, что из присутствующих значений не всегда можно сделать однозначное решение о типе конкретного атрибута, например “*column > 5”* может означать множество численных типов таких как *UINT8, UINT32, INT32, INT64* и тому подобных. Здесь нужно определиться с трактовкой определенных значений, так как перебор всех возможных может быть достаточно большим, а поэтому занимать продолжительное время.

Для числовых значений было решено использовать *INT64*(целочисленный тип 64 битности) для целочисленных значений и *FLOAT64*(число с плавающей точкой 64 битности) для нецелых значений. Также используются типы *STRING* для строковых значений, *DATE* для дат, *DATETIME* для времени. Стоит заметить, что существует еще тип *ARRAY*, который является надстройкой над предыдущими типами и представлять собой массив из значений определенного типа.

Определить значения столбцов мы можем используя логический, арифметические и другие функции над значениями столбцов, которые указаны в запросе. Такие функции лежат в поддеревьях *SELECT* и *WHERE*. Параметром функции может быть константа, колонка либо другая функция (Рисунок 2). Таким образом для понимания типа колонки могут помочь следующие параметры: 1) Типы аргументов, которые может принимать функция, например функция *TOSTARTOFMINUTE*(округляет время до кратного 5 минутам вниз) может принимать только *DATETIME*, таким образом если аргументом данной функции является колонка, то данная колонка имеет тип *DATETIME*. 2) типы остальных аргументов в

данной функции, например функция *EQUALS* (равенство), она подразумевает собой равенство типов ее аргументов, таким образом если в данной функции присутствует константа и столбец, то мы можем определить тип столбца как тип константы.

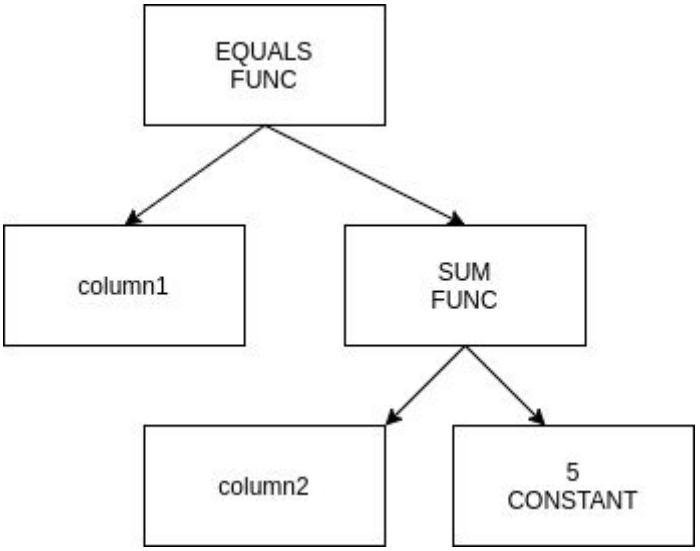

Рисунок 2: дерево функции

Таким образом, для каждой функции мы определяем возможные типы аргументов, тип возвращаемого значения, а также параметр, являются ли аргументы функции одинакового типа. Рекурсивный обработчик функций будет определять возможные типы столбцов использующихся в данных функциях по значениям аргументов и возвращать возможные типы результата выполнения функции.

Теперь для каждого столбца мы имеем множество возможных типов его значений. Для однозначной трактовки запроса мы выберем один конкретный тип из этого множества.

3.4 Определение значений столбцов

На этом этапе мы уже имеем определенную структуру таблиц базы данных, нам необходимо заполнить эту таблицу значениям. Нам необходимо понять, какие столбцы зависят друг от друга при исполнении функции (например по двум столбцами делается join, значит они должны иметь одинаковые значения), а также какие значения должны принимать столбцы, чтобы выполнялись различные условия при исполнении.

Для достижения цели ищем все операции сравнения в нашем запросе, если аргументами операции являются два столбца, то мы считаем

их связанными, если аргументами являются столбец и значение, то присваиваем данное значение возможным значением данного столбца, а также добавляем данное значение + определенный шум. Для числового типа шумом является случайное число, для даты - случайное количество дней и т.п. При этом для каждой операции сравнения необходим свой обработчик этой операции, который генерирует хотя бы два значения, одно из которых условие операции, а другое нет. Например, для операции "*column1 > 5"*, *column1* должно присваиваться значение большее 5 и меньшее, либо равное 5, аналогично для операции "*column2 LIKE some%string*", столбцу *column2* должно присваиваться значение удовлетворяющее выражение, а также не удовлетворяющее.

Теперь для некоторых колонок мы имеем множество связанных с ними колонок и множество значений. Мы знаем, что связность колонок симметрична, но для полноценного определения связности колонок нам необходимо добавить транзитивность, т.к если "*column1 = column2*" и "*column2 = column3*", то "*column1 = column3*", но это не вытекает из построения. Соответственно нам необходимо распространить связность по всем колонкам. Затем мы для каждой колонки объединяем множество ее значений со значениями всех связанных с ней. Теперь если у нас остались колонки без значений, мы просто генерируем случайные значения.

3.5 Генерация записей

Теперь у нас есть полноценной представление схемы базы данных, а также множество значений каждой таблицы. Мы будем генерировать данные посредством декартова произведения множества значений каждого столбца для определенной таблицы. Таким образом мы получаем для каждой таблицы множество, состоящее из множеств значений каждого столбца.

По этим данным мы начинаем генерировать запросы, создающие данную таблицу и заполняет ее данными. По структуре таблицы и типам ее столбцов мы генерируем *CREATE QUERY*, которая создает данную таблицу. Затем по множеству значений мы генерируем *INSERT QUERY*, которая заполняет данную таблицу данными.

3.6 Тестирование

Для тестирования полученного алгоритма были использованы запросы из существующих тестов в Clickhouse. Для каждого запроса применялся данный алгоритм, затем использовались сгенерированные запросы создания базы данных. Для данной базы применялся запрос, проверялось что запрос успешно выполняется. Успешность выполнения запроса говорит о правильно определенной схеме базы данных. Также важно, чтобы запрос возвращал хотя бы одну строку, что означает правильность определения возможных значений столбцов.

4 Заключение

4.1 Результат

Таким образом мы научились по заданному пользователем запросу определять структуру баз данных и таблиц, использующихся в этом запросе, а также заполнять ее определенными данными. Алгоритм описывает каждую колонку. *Type* отвечает за тип данных в колонке, I - *integer*, F - *float*, D - *date,* DT - *datetime*, ARR - массив. В параметре *values* описаны предлагаемые значения колонки. В параметре *equal* описаны связанные с данной колонки, то есть колонки, которых должны иметь общие значения с данной. Затем алгоритм генерирует записи посредством декартового произведения значений values в колонках.

4.2 Пример работы

Дан запрос:

SELECT integer, second.float, arrayjoin(array)
FROM db.table1
JOIN db.table2 AS second ON db.table1.sim_value1 = second.sim_value2
WHERE integer > 5 AND second.float > 100/3 AND date = yesteray() - 5;

В результате работы алгоритма получаем следующую структуру таблиц и значения колонок:
Table: *db.table2*
Columns:

float (*db.table2.float*)
type: F
values: 100 / 3, 100 / 3 + 1.166667, 100 / 3 - 1.235294
equal: db.table2.float

s*im_value2* (*db.table2.sim_value2*)
type: I
values: -1454026639, 1136820438, 1832565398
equal: *db.table1.sim_value1, db.table2.sim_valu2*

Table: *db.table1*
Columns:

array (*db.table1.array*)
type: I, ARR
values: [-648416411, -1030677187, 2087194344, 896488399, -1400453402, -958290909], [-850244462, 1449933719, 1492774445], [1406660063, 1693510466, 1410773259, 168168917, -1333650859, 1824173584, -310271577, 1750986740]
equal: *db.table1.array*

date(db.table1.date)
type: DT
values: yesterday() - 5, toDateTime(yesterday() - 5) + 142, toDateTime(yesterday() - 5) - 7930
equal: *db.table1.date*

integer(db.table1.integer)
type: I
values: 5, 5 + 1, 5 - 4
equal: *db.table1.integer*

sim_value1(db.table1.sim_value1)
type: I
values: -1454026639, 1136820438, 1832565398
equal: *db.table1.sim_value1*, *db.table2.sim_value2*

Также получаем запросы, создающие базы данных:

CREATE DATABASE IF NOT EXISTS db;

CREATE TABLE IF NOT EXISTS db.table2 (
float Float64,

```
sim_value2 Int64
) ENGINE = Log;

INSERT INTO db.table2
(float, sim_value2) VALUES
(100 / 3, -1454026639),
...

CREATE DATABASE IF NOT EXISTS db;

CREATE TABLE IF NOT EXISTS db.table1 (
array Array(Int64),
date DateTime,
integer Int64,
sim_value1 Int64
) ENGINE = Log;

INSERT INTO db.table1
(array, date, integer, sim_value1) VALUES
([-648416411, -1030677187, 2087194344, 896488399, -1400453402,
-958290909], yesterday() - 5, 5, -1454026639),
...
```

4.3 Проблемы и дальнейшая работа

Данный алгоритм требует описания обработчиков функций, которые помогли бы лучше определять значения столбцов. Проблемами данного решения являются чрезмерное заполнение таблицы данными, некоторые из которых не являются нужными для покрытия случаями. Например для определения равенства двух колонок с несколькими значениями у каждой генерируется количество записей, равное произведению количества значений у каждой колонки, хотя достаточно было сгенерировать две записи. Также проблемой данного решения является отсутствие возможности разрешения сложных функций, аргументами которых является столбец и функция от другого столбца, Например “*column1 = column2 + column3”*, данное решение не позволяет генерировать данные,

которые позволили со стопроцентной вероятностью иметь значения, выполняющие и не выполняющие данное равенство.

Доработкой алгоритма является поддержка вложенных структур для баз данных. Сейчас вложенные структуры не поддерживаются. Также необходимо добавить обработчики на некоторые функции, которые сейчас не поддерживаются данным алгоритмом.

5 Список источников

[1] Claudio de la Riva, María José Suárez-Cabal, Javier Tuya, AST '10: Proceedings of the 5th Workshop on Automation of Software Test, May 2010 Pages 67–74, Constraint-based Test Database Generation for SQL Queries

[2] Javier Tuya , María José Suárez Cabal , Claudio de la Riva , Software Testing, Verification and Reliability, 20(3) 237-288, September 2010, Full predicate coverage for testing SQL database queries

[3] [Электронный ресурс]. – Режим доступа: https://habr.com/en/company/yandex/blog/457354/ свободный – (03.02.2020) .

[4] Bruno, N., Chaudhuri, S. 2005. Flexible database generators. In Proceedings of the 31st International Conference on Very Large Data Bases. VLDB Endowment, 1097-1107

[5] Bining, C., Kossmad, D., Lo, E. 2008. MultiRQP - Generating test databases for the functional testing of OLTP applications. In Proceedings of the 1st International Workshop on Testing Database Systems. DBTest'08. ACM New York, NY, 1-6