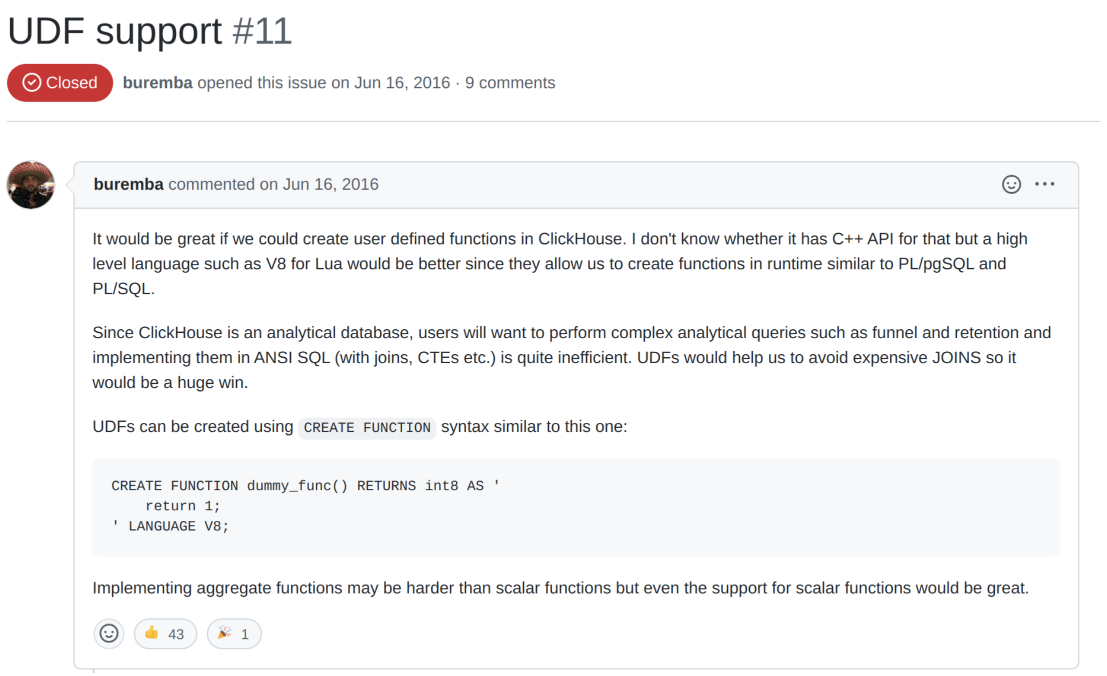
UDF in ClickHouse

About me

Maksim, developer of ClickHouse.

SQL UDF



UDF support #11

Closed buremba opened this issue on Jun 16, 2016 · 9 comments

buremba commented on Jun 16, 2016

It would be great if we could create user defined functions in ClickHouse. I don't know whether it has C++ API for that but a high level language such as V8 for Lua would be better since they allow us to create functions in runtime similar to PL/pgSQL and PL/SQL.

Since ClickHouse is an analytical database, users will want to perform complex analytical queries such as funnel and retention and implementing them in ANSI SQL (with joins, CTEs etc.) is quite inefficient. UDFs would help us to avoid expensive JOINS so it would be a huge win.

UDFs can be created using `CREATE FUNCTION` syntax similar to this one:

```
CREATE FUNCTION dummy_func() RETURNS int8 AS '
    return 1;
' LANGUAGE V8;
```

Implementing aggregate functions may be harder than scalar functions but even the support for scalar functions would be great.

👍 43 🎉 1

https://github.com/ClickHouse/ClickHouse/issues/11

SQL UDF

Syntax

```
CREATE FUNCTION name AS (parameter0,...) ->
expression
```

1. Recursive functions are not allowed.

2. All identifiers used by a function must be specified in its parameter list.

3. The name of a function must be unique among user defined and system functions.

SQL UDF

```
CREATE FUNCTION contains AS (string, value) ->
position(string, value) > 0;
```

```
SELECT contains('Test', 'T') AS result
```

```
┌─result─┐
│      1 │
└────────┘
```

SQL UDF persistence

Stored in configuration_path/user_defined folder as SQL script

```
CREATE FUNCTION a_plus_b AS (a, b) -> a + b;
```

```
cat ../user_defined/function_a_plus_b.sql
CREATE FUNCTION a_plus_b AS (a, b) -> (a + b)
```

SQL UDF introspection

```
CREATE FUNCTION a_plus_b AS (a, b) -> a + b;
```

```
SELECT name, create_query FROM system.functions
WHERE origin = 'SQLUserDefined'
```

```
┌─name─────┬─create_query─────────────────────────────────┐
│ a_plus_b │ CREATE FUNCTION a_plus_b AS (a, b) -> (a + b) │
└──────────┴──────────────────────────────────────────────┘
```

SQL UDF optimizations

SQL UDF is syntax level optimization

```
SELECT a_plus_b(a, b) + c FROM test_table
WHERE b + c > 5;
```

Translated into:

```
SELECT a + b + c FROM test_table WHERE b + c > 5;
```

Optimizations will apply:

1. JIT Compilation.

2. Equal expression optimization.

Executable script

1. Run child process and execute script.
2. Sending data to its stdin using pipe, reading result from stdout.
3. Data is serialized and deserialized using native formats (TabSeparated, ...).

Executable script Bash

Example:

```
#!/bin/bash

while read read_data;
    do printf "Key $read_data\n";
done
```

Executable script Python

Example:

```
#!/usr/bin/python3

import sys

if __name__ == '__main__':
    for line in sys.stdin:
        print("Key " + line, end='')
```

Executable script C++

Example. Option send_chunk_header is true:

```
int main(int argc, char **argv)
{
    char value[4096]; size_t rows = 0;

    std::cin.tie(nullptr); std::cin.sync_with_stdio(false);
    std::cout.tie(nullptr); std::cout.sync_with_stdio(false);

    while (std::cin >> rows) {
        for (size_t i = 0; i < rows; ++i) {
            std::cin >> value;
            std::cout << "Key " << value << "\n";
        }
        std::cout.flush();
    }
    return 0;
}
```

ExecutableDictionary

Example:

```
<dictionary>
    <name>executable_dictionary</name>
    <source>
        <executable>
            <format>TabSeparated</format>
            <command>user_scripts/script_name</command>
        </executable>
    </source>
    <layout><complex_key_direct/></layout>
    <structure>
        <key>
            <attribute><name>key</name><type>String</type></attribute>
        </key>
        <attribute><name>result</name><type>String</type>>/attribute>
    </structure>
</dictionary>
```

ExecutableDictionary example

```
SELECT dictGet('executable_dictionary', 'result', '1')
as result
```

```
┌─result─┐
│ Key 1  │
└────────┘
```

ExecutableDictionary benchmark

```
clickhouse-benchmark --query="SELECT
dictGet('dictionary', 'result', toString(number))
FROM system.numbers LIMIT 1000000 FORMAT Null"
--concurrency=3
```

Dictionary executable Bash:	16.112 MiB/s
Dictionary executable Python:	196.691 MiB/s
Dictionary executable C++:	264.827 MiB/s

ExecutablePool

ClickHouse process data in blocks.

Overhead of script creation (fork + exec) on each block of data is significant.

Script can have state, that need to be created on startup.

Solution:

Executable Pool. Create pool of running processes and reuse them during queries.

https://en.wikipedia.org/wiki/FastCGI

ExecutablePool

1. Pool size. If pool size == 0 then there is no size restrictions.

2. Command termination timeout. Default 10 seconds.

```
<source>
    <executable_pool>
        <format>TabSeparated</format>
        <command>user_scripts/test_input.sh</command>
        <pool_size>16</pool_size>
        <send_chunk_header>1<send_chunk_header>
    </executable_pool>
</source>
```

ExecutablePoolDictionary benchmark

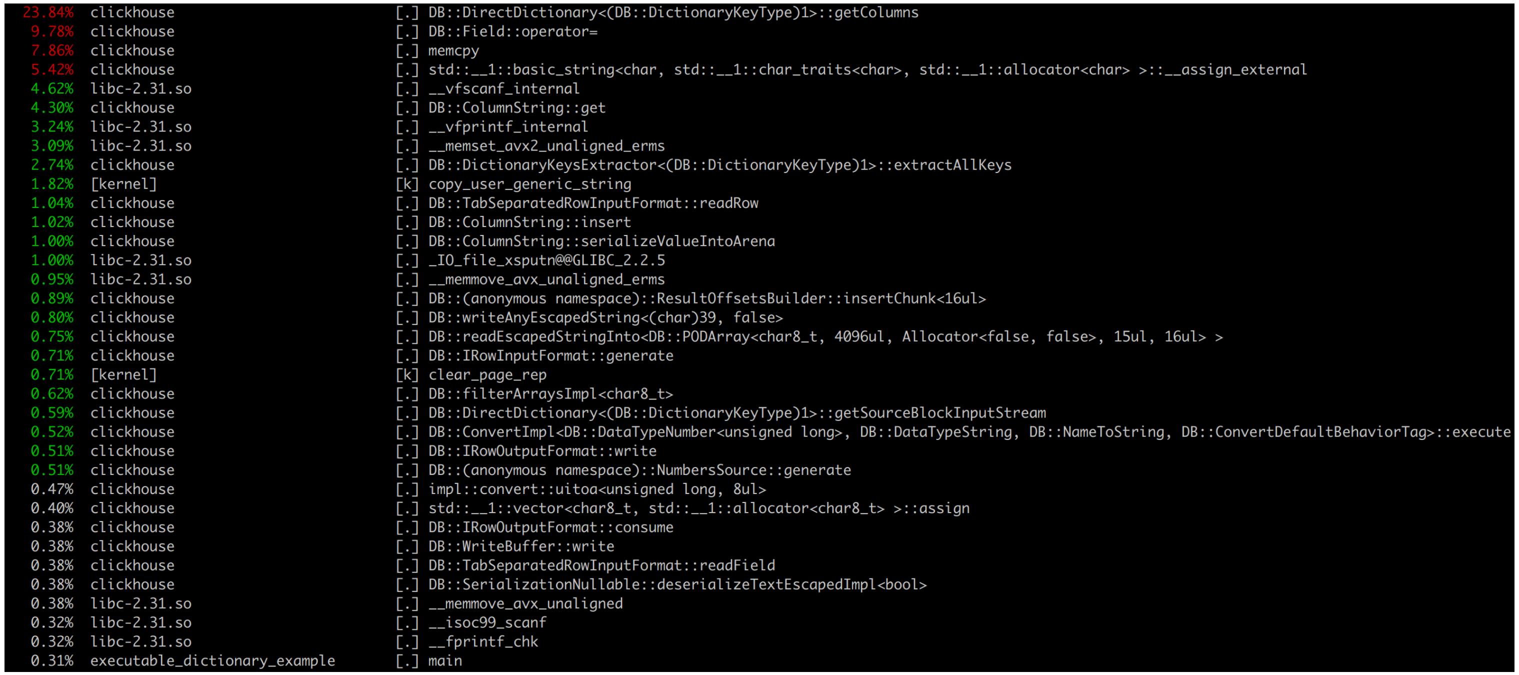
```
clickhouse-benchmark --query="SELECT
dictGet('dictionary', 'result', toString(number))
FROM system.numbers LIMIT 1000000 FORMAT Null"
--concurrency=32
```

Dictionary executable C++:	264.827 MiB/s
Dictionary executable pool C++:	305 MiB/s

+16% performance improvement for script with zero startup cost

ExecutablePoolDictionary issues

305 MB/s is too slow. Just for copying data beetween processes.



```
 23.84%  clickhouse                     [.] DB::DirectDictionary<(DB::DictionaryKeyType)1>::getColumns
  9.78%  clickhouse                     [.] DB::Field::operator=
  7.86%  clickhouse                     [.] memcpy
  5.42%  clickhouse                     [.] std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::__assign_external
  4.62%  libc-2.31.so                   [.] __vfscanf_internal
  4.30%  clickhouse                     [.] DB::ColumnString::get
  3.24%  libc-2.31.so                   [.] __vfprintf_internal
  3.09%  libc-2.31.so                   [.] __memset_avx2_unaligned_erms
  2.74%  clickhouse                     [.] DB::DictionaryKeysExtractor<(DB::DictionaryKeyType)1>::extractAllKeys
  1.82%  [kernel]                       [k] copy_user_generic_string
  1.04%  clickhouse                     [.] DB::TabSeparatedRowInputFormat::readRow
  1.02%  clickhouse                     [.] DB::ColumnString::insert
  1.00%  clickhouse                     [.] DB::ColumnString::serializeValueIntoArena
  1.00%  libc-2.31.so                   [.] _IO_file_xsputn@@GLIBC_2.2.5
  0.95%  libc-2.31.so                   [.] __memmove_avx_unaligned_erms
  0.89%  clickhouse                     [.] DB::(anonymous namespace)::ResultOffsetsBuilder::insertChunk<16ul>
  0.80%  clickhouse                     [.] DB::writeAnyEscapedString<(char)39, false>
  0.75%  clickhouse                     [.] DB::readEscapedStringInto<DB::PODArray<char8_t, 4096ul, Allocator<false, false>, 15ul, 16ul> >
  0.71%  clickhouse                     [.] DB::IRowInputFormat::generate
  0.71%  [kernel]                       [k] clear_page_rep
  0.62%  clickhouse                     [.] DB::filterArraysImpl<char8_t>
  0.59%  clickhouse                     [.] DB::DirectDictionary<(DB::DictionaryKeyType)1>::getSourceBlockInputStream
  0.52%  clickhouse                     [.] DB::ConvertImpl<DB::DataTypeNumber<unsigned long>, DB::DataTypeString, DB::NameToString, DB::ConvertDefaultBehaviorTag>::execute
  0.51%  clickhouse                     [.] DB::IRowOutputFormat::write
  0.51%  clickhouse                     [.] DB::(anonymous namespace)::NumbersSource::generate
  0.47%  clickhouse                     [.] impl::convert::uitoa<unsigned long, 8ul>
  0.40%  clickhouse                     [.] std::__1::vector<char8_t, std::__1::allocator<char8_t> >::assign
  0.38%  clickhouse                     [.] DB::IRowOutputFormat::consume
  0.38%  clickhouse                     [.] DB::WriteBuffer::write
  0.38%  clickhouse                     [.] DB::TabSeparatedRowInputFormat::readField
  0.38%  clickhouse                     [.] DB::SerializationNullable::deserializeTextEscapedImpl<bool>
  0.38%  libc-2.31.so                   [.] __memmove_avx_unaligned
  0.32%  libc-2.31.so                   [.] __isoc99_scanf
  0.32%  libc-2.31.so                   [.] __fprintf_chk
  0.31%  executable_dictionary_example  [.] main
```

Executable

Executable, ExecutablePool dictionaries.

Executable, ExecutablePool engines. Executable table function.

Executable user defined functions.

Executable table function

Syntax:

```
executable(script_name_optional_arguments,
            format,
            structure,
            input_queries)
```

Data is processed in streaming fashion.

ClickHouse process input queries and sending their results into process stdin. And simualteneosly read data from process stdout.

If more than one input query is created clickhouse creates pipes for file descriptors starting from 3.

Executable table function example

```
SELECT * FROM executable('test_input.sh',
            'TabSeparated',
            (SELECT 1))
```

```
┌─value─┐
│ Key 1 │
└───────┘
```

Executable table engine

```
CREATE TABLE test_table (value String)
ENGINE=Executable('test_input.sh',
    'TabSeparated',
    (SELECT 1));
```

```
SELECT * FROM test_table;
```

```
┌─value─┐
│ Key 1 │
└───────┘
```

ExecutablePool table engine

```
CREATE TABLE test_table (value String)
ENGINE=ExecutablePool('test_input.sh',
    'TabSeparated',
    (SELECT 1));
```

```
SELECT * FROM test_table;
```

```
┌─value─┐
│ Key 1 │
└───────┘
```

ExecutableEngine example

```
#!/usr/bin/python3

import sys

from essential_generators import DocumentGenerator

if __name__ == '__main__':
    length = int(sys.argv[1]);

    gen = DocumentGenerator()

    for i in range(0, length):
        print(gen.sentence())
```

ExecutablePool table engine

```
SELECT
    length(tokens(sentence)) AS token_length,
    length(sentence)
FROM executable('sentence_generator.py 10000',
    'TabSeparated',
    'sentence String')
ORDER BY token_length DESC LIMIT 5;

┌─token_length─┬─length(sentence)─┐
│           22 │              116 │
│           21 │              110 │
│           20 │              109 │
│           20 │               85 │
│           19 │              112 │
└──────────────┴──────────────────┘
```

Executable UDF

```
<function>
    <type>executable/executable_pool</type>
    <name>test_function</name>
    <return_type>String</return_type>
    <argument>
        <type>String</type>
    </argument>
    <format>TabSeparated</format>
    <command>user_scripts/test_input.sh</command>
</function>
```

Executable UDF

```
SELECT test_function('1')
```

```
┌─test_function('1')─┐
│ Key 1              │
└────────────────────┘
```

Executable UDF Introspection

```sql
SELECT name FROM system.functions
WHERE origin = 'ExecutableUserDefined'
```

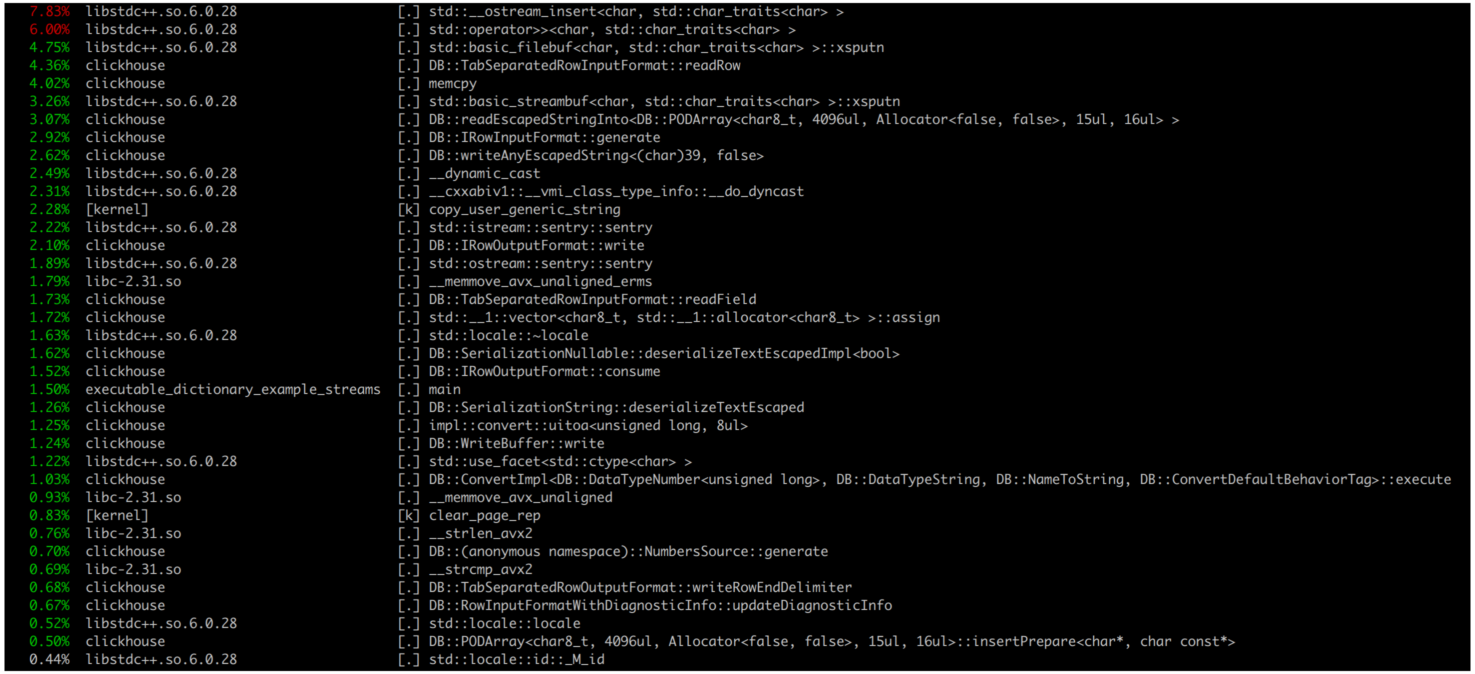
```
┌─name──────────┐
│ test_function │
└───────────────┘
```

Executable UDF Benchmark

```
./clickhouse-benchmark
--query="SELECT test_func(toString(number))
FROM system.numbers LIMIT 100000 FORMAT Null"
--concurrency=32
```

ClickHouse concat('Key', toString(number)):	MiB/s: 3829.216
Function Bash:	MiB/s: 20.964
Function Python:	MiB/s: 174.635
Function executable C++:	MiB/s: 574.620
Function executable pool C++:	MiB/s: 859.483

Executable UDF Benchmark



```
   7.83%  libstdc++.so.6.0.28                  [.] std::__ostream_insert<char, std::char_traits<char> >
   6.00%  libstdc++.so.6.0.28                  [.] std::operator>><char, std::char_traits<char> >
   4.75%  libstdc++.so.6.0.28                  [.] std::basic_filebuf<char, std::char_traits<char> >::xsputn
   4.36%  clickhouse                           [.] DB::TabSeparatedRowInputFormat::readRow
   4.02%  clickhouse                           [.] memcpy
   3.26%  libstdc++.so.6.0.28                  [.] std::basic_streambuf<char, std::char_traits<char> >::xsputn
   3.07%  clickhouse                           [.] DB::readEscapedStringInto<DB::PODArray<char8_t, 4096ul, Allocator<false, false>, 15ul, 16ul> >
   2.92%  clickhouse                           [.] DB::IRowInputFormat::generate
   2.62%  clickhouse                           [.] DB::writeAnyEscapedString<(char)39, false>
   2.49%  libstdc++.so.6.0.28                  [.] __dynamic_cast
   2.31%  libstdc++.so.6.0.28                  [.] __cxxabiv1::__vmi_class_type_info::__do_dyncast
   2.28%  [kernel]                             [k] copy_user_generic_string
   2.22%  libstdc++.so.6.0.28                  [.] std::istream::sentry::sentry
   2.10%  clickhouse                           [.] DB::IRowOutputFormat::write
   1.89%  libstdc++.so.6.0.28                  [.] std::ostream::sentry::sentry
   1.79%  libc-2.31.so                         [.] __memmove_avx_unaligned_erms
   1.73%  clickhouse                           [.] DB::TabSeparatedRowInputFormat::readField
   1.72%  clickhouse                           [.] std::__1::vector<char8_t, std::__1::allocator<char8_t> >::assign
   1.63%  libstdc++.so.6.0.28                  [.] std::locale::~locale
   1.62%  clickhouse                           [.] DB::SerializationNullable::deserializeTextEscapedImpl<bool>
   1.52%  clickhouse                           [.] DB::IRowOutputFormat::consume
   1.50%  executable_dictionary_example_streams [.] main
   1.26%  clickhouse                           [.] DB::SerializationString::deserializeTextEscaped
   1.25%  clickhouse                           [.] impl::convert::uitoa<unsigned long, 8ul>
   1.24%  clickhouse                           [.] DB::WriteBuffer::write
   1.22%  libstdc++.so.6.0.28                  [.] std::use_facet<std::ctype<char> >
   1.03%  clickhouse                           [.] DB::ConvertImpl<DB::DataTypeNumber<unsigned long>, DB::DataTypeString, DB::NameToString, DB::ConvertDefaultBehaviorTag>::execute
   0.93%  libc-2.31.so                         [.] __memmove_avx_unaligned
   0.83%  [kernel]                             [k] clear_page_rep
   0.76%  libc-2.31.so                         [.] __strlen_avx2
   0.70%  clickhouse                           [.] DB::(anonymous namespace)::NumbersSource::generate
   0.69%  libc-2.31.so                         [.] __strcmp_avx2
   0.68%  clickhouse                           [.] DB::TabSeparatedRowOutputFormat::writeRowEndDelimiter
   0.67%  clickhouse                           [.] DB::RowInputFormatWithDiagnosticInfo::updateDiagnosticInfo
   0.52%  libstdc++.so.6.0.28                  [.] std::locale::locale
   0.50%  clickhouse                           [.] DB::PODArray<char8_t, 4096ul, Allocator<false, false>, 15ul, 16ul>::insertPrepare<char*, char const*>
   0.44%  libstdc++.so.6.0.28                  [.] std::locale::id::_M_id
```

Executable UDF Benchmark

```
ReadBufferFromFileDescriptor read_buffer(0);
WriteBufferFromFileDescriptor write_buffer(1);
size_t rows = 0;
char dummy;

while (!read_buffer.eof()) {
    readIntText(rows, read_buffer);
    readChar(dummy, read_buffer);

    for (size_t i = 0; i < rows; ++i) {
        readString(buffer, read_buffer);
        readChar(dummy, read_buffer);

        writeString("Key ", write_buffer);
        writeString(buffer, write_buffer);
        writeChar('\n', write_buffer);
    }

    write_buffer.next();
}
```

Executable UDF Benchmark

```
./clickhouse-benchmark
--query="SELECT test_func(toString(number))
FROM system.numbers LIMIT 100000 FORMAT Null"
--concurrency=32
```

ClickHouse concat('Key', toString(number)):	MiB/s: 3829.216
Function executable pool C++:	MiB/s: 859.483
Function executable pool C++ ClickHouse buffers:	MiB/s: 1124.672

+31% performance improvement over basic script

Executable UDF Example

```
#!/usr/bin/python3
import sys
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer

if __name__ == '__main__':
    sentiment_analyzer = SentimentIntensityAnalyzer()

    # Read chunk length
    for number in sys.stdin:
        length = int(number)

        # Read lines from chunk
        for _ in range(0, length):
            line = sys.stdin.readline()
            score = sentiment_analyzer.polarity_scores(line)
            print(str(score['compound']) + '\n', end='')

        # Flush results to stdout
        sys.stdout.flush()
```

Executable UDF Example

```xml
<function>
    <type>executable_pool</type>
    <name>sentenceScore</name>
    <return_type>Double</return_type>
    <argument>
        <type>String</type>
    </argument>
    <format>TabSeparated</format>
    <command>user_scripts/sentence_analyzer</command>
    <send_chunk_header>1</send_chunk_header>
</function>
```

Executable UDF Example

```
SELECT sentenceScore('ClickHouse is fast') as score
```

```
┌─score─┐
│     0 │
└───────┘
```

```
SELECT avg(sentenceScore(sentence)) AS avg_sentence_score
FROM executable('sentence_generator.py 10000',
    'TabSeparated',
    'sentence String')
```

```
┌───────────avg_score─┐
│ 0.030663238759543694 │
└──────────────────────┘
```

Questions?