Выпускная квалификационная работа

Умные алгоритмы обработки строк в ClickHouse

Кутенин Данила Михайлович 155
Научный руководитель: Миловидов Алексей

Область и цели

- Все используют строки. Везде. Всегда. В любых рантайм сервисах.
- ClickHouse заточен под скорость.
- Поиск, обработка, разжатие -- всё это важно. Очень.

Задачи

- Поиск подстрок во всех ипостасях, hyperscan
- Приближенный поиск
- Строки, закодированные в UTF-8
- Оптимизация SSE, в том числе и под другими платформами
- Ускорение кодека LZ4 (не успеем поговорить)
- Аллокации кешей (не успеем поговорить)

Задачи. Что не вошло в диплом

- Проверка на валидность UTF-8, toValidUTF8
- Оптимизация concat функции
- Внедрение format функции
- Бенчмарки на все функции, которые внедрили
- Стресс и fuzzy тестирование всех функций, которые внедрили
- Внедрение нового аллокатора
- Упрощение SSE кода для поиска подстрок
- Фиксы external dictionaries обработки строк
- Переезд на GCC9, внедрение libc++, сравнение библиотек
- And many more...

Получил удовольствие от написания кода!!!

Строки

Любая последовательность байт в алфавите Σ.

Строки в ClickHouse

- **ColumnString. Chars + Offsets.**

Chars = “string\0anotherone\0andanotherone\0”
Offsets = [7, 18, 32]

- 16 байтовый паддинг до Chars и 15 байтовы паддинг после. Безопасное чтение 16 байтового регистра.

Строки в ClickHouse

- **ColumnFixedString. Chars + FixedConstant N.**

  Chars = “yandexgoogleamazon”
  N = 6

- 16 байтовый паддинг до Chars и 15 байтовы паддинг после. Безопасное чтение 16 байтового регистра.
- Также **ColumnConst** с типом String.
- Ещё **LowCardinality** (но это тема для отдельного разговора).

Поиск. Определения.

- **haystack (англ. сено) -- строка, в которой мы ищем.**
- **needle (англ. иголка) -- строка (или регулярное выражение), по которой мы ищем.**

Поиск. Алгоритмы

Один needle, один haystack

1. Knuth-Morris-Pratt
2. Boyer-Moore
3. Boyer-Moore-Horspool
4. BNDM (Backward Nondeterministic Dawg)
5. Two-way (memmem)
6. Rabin-Karp
7. Поиск по индексу
8. ...

Много needle, один haystack

1. Aho-Corasick
2. Поиск по индексу
3. Rabin-Karp(?)

Что использует ClickHouse?

Ничего из вышеперечисленного
Но я честно попробовал все варианты

Поиск. Алгоритмы

Volnitsky algorithm

```
SELECT count()
FROM hits_100m_single
WHERE URL LIKE '%yandex%'
SETTINGS max_threads = 1

┌─count()─┐
│ 18401331 │
└──────────┘

1 rows in set. Elapsed: 5.337 sec. Processed 100.00 million ro
ws, 9.38 GB (18.74 million rows/s., 1.76 GB/s.)
```

1. Сохраняем в хэш-таблицу с открытой адресацией оффсеты всех биграмм (два подряд идущих байта) строки needle (с конца).
2. Идём по haystack с шагом needle.size() - 1, просматриваем соответствующую биграмму в хэш-таблице и сравниваем куски памяти на равенство (memcmp или что-то похитрее).
3. Шаг needle.size() - 1 гарантирует, что если есть вхождение, мы всегда просмотрим хотя бы одну биграмму этого вхождения.
4. Добавление с конца гарантирует поиск первого вхождения (рассматриваем сначала большие оффсеты, потом маленькие).

Поиск. Алгоритмы

Пусть будет строка haystack – **abacabaac** и needle равна **aaca**. Хэш-таблица {aa → 0, ac → 1, ca → 2}.

```
0 1 2 3 4 5 6 7 8 9
a b a c a b a a c a
    ^ - курсор изначальный здесь

0 1 2 3 4 5 6 7 8 9
a b a c a b a a c a
  a a c a

0 1 2 3 4 5 6 7 8 9
a b a c a b a a c a
          ^ - курсор теперь здесь

0 1 2 3 4 5 6 7 8 9
a b a c a b a a c a
                ^ - курсор теперь здесь
```

```
0 1 2 3 4 5 6 7 8 9
a b a c a b a a c a
            a a c a

Нашли совпадение.
```

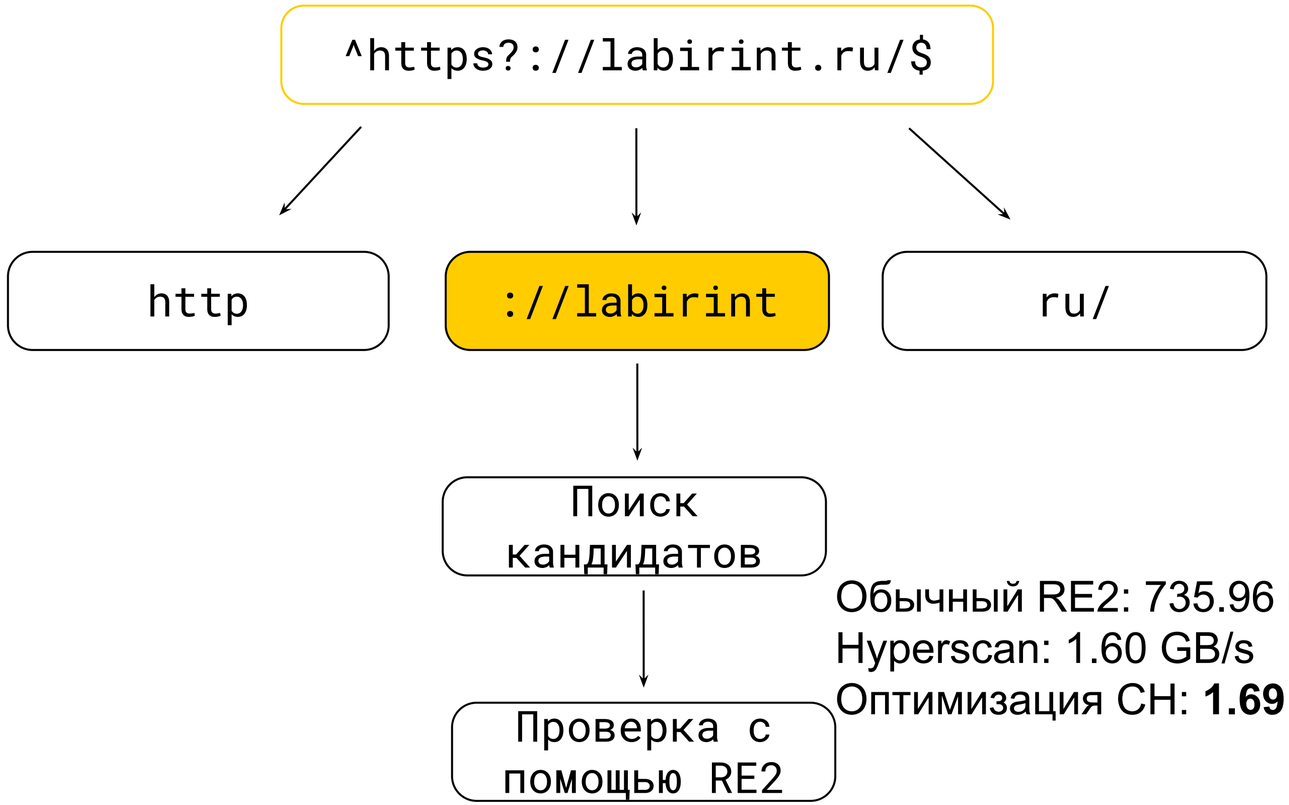
Реализация в ClickHouse

Поиск. Алгоритмы

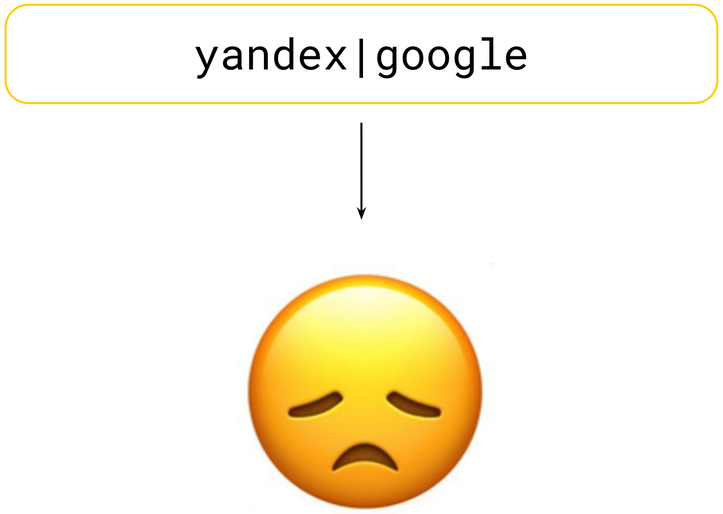
Алгоритм/Запрос	SELECT sum(position(URL, 'google')) FROM hits_100m_single settings max_threads=1	SELECT count() FROM hits_100m_single WHERE URL LIKE '%metrika%' settings max_threads=1	SELECT sum(position(URL, 'yandex')) FROM hits_100m_single settings max_threads=1	SELECT SearchPhrase, any(URL), any(Title), count() AS c, uniq(UserID) FROM hits_100m_single WHERE (Title LIKE '%Яндекс%') GROUP BY SearchPhrase ORDER BY c DESC LIMIT 10 settings max_threads=1
memmem	655 Mb/s	710 MB/s	637 MB/s	578 MB/s
BM	865 MB/s	940 MB/s	780 MB/s	600 MB/s
BMH	920 MB/s	930 MB/s	790 MB/s	687 MB/s
KMP	600 MB/s	653 MB/s	580 MB/s	540 MB/s
Volnitsky	1.66 GB/s	1.80 GB/s	1.56 GB/s	880 MB/s

Поиск. Где используется алгоритм.

- **position(s0, s1), -UTF8, -CaseInsensitive, -CaseInsensitiveUTF8**
- **LIKE ‘%somestring%’**
- **Оптимизация поиска регулярных выражений, в т.ч. LIKE**



Обычный RE2: 735.96 MB/s
Hyperscan: 1.60 GB/s
Оптимизация CH: **1.69 GB/s**



Но мы придумали решение и тут!

Поиск. Алгоритмы

MultiVonitsky algorithm

```
SELECT sum(multiSearchAny(URL, ['yandex', 'google', 'yahoo']))
FROM hits_100m_single
SETTINGS max_threads = 1

┌─sum(multiSearchAny(URL, ['yandex', 'google', 'yahoo']))─┐
│                                                 18441259 │
└──────────────────────────────────────────────────────────┘

1 rows in set. Elapsed: 7.747 sec. Processed 100.00 million ro
ws, 9.38 GB (12.91 million rows/s., 1.21 GB/s.)
```

1. Сохраняем в хэш-таблицу с открытой адресацией оффсеты **и id строки** всех биграмм (два подряд идущих байта) **всех строк** needle (с конца).
2. Идём по haystack с шагом **step=min(needle[i].size() - 1) for all i**, просматриваем соответствующую биграмму в хэш-таблице и сравниваем куски памяти.
3. Шаг **step** гарантирует, что если есть вхождение, мы всегда просмотрим хотя бы одну биграмму этого вхождения.

Сравнение алгоритмов для поиска

	Multi Volnitsky	Volnitsky n раз	Aho Corasick	RE2	Hyperscan
'yandex', 'google'	1.49 GB/s	1.18 GB/s	851.32 MB/s	329.39 MB/s	788.21 MB/s
'yandex', 'google', 'yahoo', 'pikabu'	1.27 GB/s	762.13 MB/s	780.65 MB/s	303.58 MB/s	748.03 MB/s
'yandex', 'google', 'http'	1.69 GB/s	1.00 GB/s	1.19 GB/s	773.28 MB/s	626.18 MB/s
'Honda', 'Хонд', 'HONDA'	900.07 MB/s	741.97 MB/s	730.20 MB/s	168.26 MB/s	814.30 MB/s
'yandex', 'google', 'facebook', 'wikipedia', 'reddit'	1.22 GB/s	677.39 MB/s	813.57 MB/s	267.00 MB/s	757.16 MB/s
'news.ngs.ru', 'she.ngs.ru', 'afisha.ngs.ru', 'business.ngs.ru', '//ngs.ru/', '//m.ngs.ru/'	1.35 GB/s	681.91 MB/s	890.13 MB/s	220.24 MB/s	850.83 MB/s

Сравнение алгоритмов для поиска

	Multi Volnitsky	Volnitsky n раз	Aho Corasick	RE2	Hyperscan
'newFlat=YES', 'newbuilding', 'siteId=', 'novostrojka', 'nb.phone.show', 'nb.show', 'pik/'	906.08 MB/s	584.09 MB/s	737.94 MB/s	252.77 MB/s	724.39 MB/s
'kvartiry', 'nedvizhimost', 'kommercheskaya_nedvizhimost', 'garazhi_i_mashinomesta', 'doma_dachi_kottedzhi', 'zemelnye_uchastki', 'komnaty', 'nedvizhimost_za_rubezhom'	1.05 GB/s	644.00 MB/s	757.43 MB/s	250.75 MB/s	775.68 MB/s
'ут', 'утк', 'утко', 'утконос', 'enrjyjc', 'utkonos', 'enrjyjc', 'www', 'http', 'enrfyjc', 'гелщтщы'	500.44 MB/s	254.45 MB/s	453.21 MB/s	134.45 MB/s	532.78 MB/s
'бэбиблок', 'бэбиблог', 'бебиблок', 'бебиблог', 'blog', 'беби блог', 'бэби блог', 'бб', ',t,b,kju', 'бейбиблог', 'бейбиблог.ру', ',\',b,kju', 'бэйбиблог', 'бейби блог', 'бэби блок', 'бэбибл', 'бебиб', 'бебибло'	106.74 MB/s	74.21 MB/s	394.98 MB/s	139.00 MB/s	511.22 MB/s

Сравнение алгоритмов для поиска

	Multi Volnitsky	Volnitsky n раз	Aho Corasick	RE2	Hyperscan
'fitnes-kluby', 'sportivnoe-oborudovanie-atributika', 'krytye-sportivnye-ploshchadki', 'pejntbol-strajk-i-hard-bol', 'strelkovye-kluby-tiry', 'sportivnye-organizatsii', 'basseyny-plavatelnye', 'otkrytye-sportivnye-ploshchadki-bazy', 'gornolyzhnye-sklony', 'pryzhki-s-parashyutom', 'sportivnye-sektsii', 'yakht-kluby', 'joga-centry-i-instruktory', 'bukmekerskie-kontory', 'skalolazanie-voskhozhdenie-v-gory', 'fekhtovalnye-kluby', 'drugoj-sport-i-fitnes', 'boevye-iskusstva'	889.12 MB/s	198.91 MB/s	640.33 MB/s	195.09 MB/s	642.42 MB/s

Сравнение алгоритмов для поиска

	Multi Volnitsky	Volnitsky n раз	Aho Corasick	RE2	Hyperscan
'chelyabinsk.74.ru', 'doctor.74.ru', 'transport.74.ru', 'm.74.ru', '//74.ru/', 'chel.74.ru', 'afisha.74.ru', 'diplom.74.ru', 'chelfin.ru', '//chel.ru', 'chelyabinsk.ru', 'cheldoctor.ru', '//mychel.ru', 'cheldiplom.ru', '74.ru/video', 'market', 'poll', 'mail', 'conference', 'consult', 'contest', 'tags', 'feedback', 'pages', 'text'	418.14 MB/s	230.13 MB/s	413.58 MB/s	183.31 MB/s	639.61 MB/s
'p17266p66989p97b7', 'p17266p66988p285b', 'p17266p66986pa4e8', 'p15926p65809pbab6', 'p15926p65810p2672', 'p15926p65811p9afa', 'p15926p65812p97e3', 'p15926p65813pd214', 'p15926p65813pd214', 'p15926p65815p350b', 'p15926p65816pe52c', 'p15926p65814p0cc9', 'p15926p65817p4cea', 'p15926p65818p9b20', 'p15926p65860p4435', 'p15926p65861p1f1e', 'p15926p65862p6f5b', 'p15926p65864pef1c', 'p15926p64433p9fca', 'p15926p64435p9eb7', 'p15926p64436p5e2c', 'p14762p59496p5de8', 'p14762p67782pfc4a'	1.20 GB/s	330.57 MB/s	990.87 MB/s	524.60 MB/s	1.10 GB/s

Сравнение алгоритмов для поиска

	Multi Volnitsky	Volnitsky n раз	Aho Corasick	RE2	Hyperscan
'вуман','вумен ру женский журнал','вумен форум', 'вуманжурнал ру','воман','www.woman.ru','вуман ру', 'woman ru','женский журнал','форум вумен','devty', 'вумен.ру','вумен ру форум','вуманжурнал','женский форум', 'журнал вумен','цщьфт','женский журнал вумен','women', 'woman форум','devty he','вуман.ру','женский форум вумен', 'women.ru женский сайт','форум вумен ру','вум','сайт вумен', 'воменс. ру','devfy','вомен','woman.ru журнал','woman.ru форум', 'вуменру','вуман форум','цщьфтюкг','devfy he', 'wom','вумен форум новое', 'вумен форум новые','вумэн','форум вуман	42.19 MB/s	31.10 MB/s	380.23 MB/s	140.07 MB/s	441.12 MB/s

Сравнение алгоритмов для поиска

- До 10-15 строк MultiVolnitsky обыгрывает всех (97% запросов).
- Деградация при большом количестве маленьких или похожих строк.
- Ускорение от большой минимальной длины строки.

Новые фичи ClickHouse (19.5)

- **multiSearchAny** -- есть ли хоть одно вхождение из needle
- **multiSearchFirstPosition** -- самая левая позиция haystack, найденного каким-нибудь needle
- **multiSearchFirstIndex** -- самый левый индекс найденного needle
- **multiSearchAllPositions** -- поиск всех первых позиций
- **Суффиксы -UTF8, -CaseInsensitive, -CaseInsensitiveUTF8**

Все алгоритмы используют MultiVolnitsky

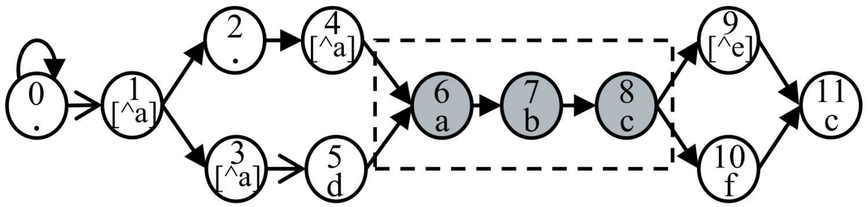
Сравнение алгоритмов для поиска множества регулярных выражений

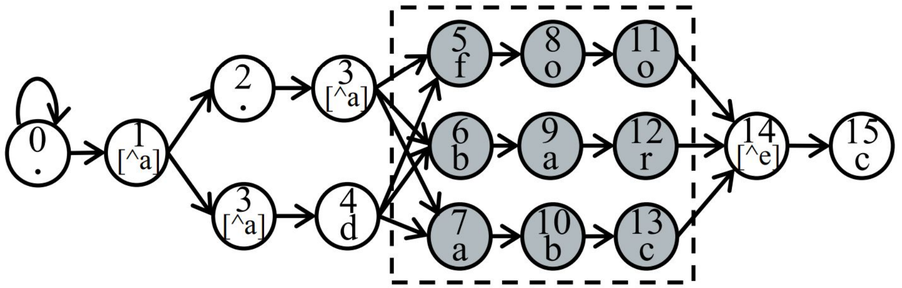
- **Добавили поддержку Hyperscan.**

USENIX Статья 2019 года. 5 версия вышла во время наших исследований. Более 12 лет разработки. Мы его смогли обогнать на 97% обычных текстовых запросах.

Имеет ряд ограничений. Длина строки должна быть меньше 2^32. Потребляет недокументированное количество памяти.

- **Зато быстрый, как мы уже видели.**





Сравнение алгоритмов для поиска

	RE2	Hyperscan
'/t[0-9]+-', '/questions/7{9}[0-9]+'	565.51 MB/s	743.88 MB/s
'ножниц.*вырубн', 'ножниц.*рычажн', 'ножниц.*гильотин'	155.72 MB/s	505.31 MB/s
'f[ae]b[ei]rl', 'ф[иаэе]б[еэи][рпл]', 'афиукд', 'a[ft],th', '^ф[аиеэ]?б?[еэи]?$', 'берлик', 'fab', 'фа[беьв]+е?[рлко]'	183.32 MB/s	422.65 MB/s
'/questions/q*', '/q[0-9]*/', '/questions/[0-9]*'	524.58 MB/s	936.12 MB/s
'//ngs.ru/$', '//m.ngs.ru/$', '//news.ngs.ru/$', '//m.news.ngs.ru/$', '//ngs.ru/\\?', '//m.ngs.ru/\\?', '//news.ngs.ru/\\?', '//m.news.ngs.ru/\\?'	543.41 MB/s	619.82 MB/s
'[ми][аеэпви][нм][асзи][иус]*', '[mn][aeauo][nm]s[yyi]*', 'ru', 'v[ft']v[cp][be]', 'www', 'ьфьын', 'маиси', 'mam', 'amsy', 'маммси', 'амси', 'vfvc'	141.17 MB/s	463.87 MB/s

Новые фичи ClickHouse (19.5)

- **multiMatchAny** -- возвращает один, если хотя бы одно регулярное выражение подошло под haystack и ноль иначе.
- **multiMatchAnyIndex** -- возвращает индекс любого вхождения регулярного выражения в haystack

В функциях используется честный hyperscan.

Приближённый поиск

- **Хотим искать похожие строки между собой. (яндекс ~ индекс)**
  Надо определиться с метрикой. Левенштейн -- квадратично тяжело и “сильно” легче быть не может.
- **Тем не менее, сделали подсказки опечаток в запросах по Левенштейну.**

```
SELECT sum(multiSearch(URL, ['yandex', 'google', 'yahoo']))
FROM hits_100m_single
SETTINGS max_threads = 1

Received exception from server (version 19.9.1):
Code: 46. DB::Exception: Received from localhost:9000, ::1. DB::Exception: Unk
nown function multiSearch. Maybe you meant: ['multiSearchAny'].

0 rows in set. Elapsed: 0.002 sec.
```

```
CREATE TABLE stored_aggregates
(
    `d` Date,
    `Uniq` AggregateFunction(uniq, Uint64)
)
ENGINE = MergeTree(d, d, 8192)

Received exception from server (version 19.9.1):
Code: 50. DB::Exception: Received from localhost:9000, ::1. DB::Exception: Unk
nown data type family: Uint64. Maybe you meant: ['UInt64','Int64'].
```

Приближённый поиск

Есть расстояние Хэмминга. Есть алгоритм O(|Σ|n log n).

1. Пусть есть две строки размера n и m, где m < n. Мы хотим найти из всех наложений минимальное отличающееся. Будем считать массив отличающихся и выбирать наименьшее число.
2. Можно посчитать ответ посимвольно через битовые маски, где 1 -- есть символ A, а 0 -- нет символа A.
3. Далее это скалярные произведения таких битовых масок. Это можно посчитать через Быстрое Преобразование Фурье за O(n log n) (скалярные произведения очень похожи на свёртки).

Не самый практичный способ, так как |Σ| может быть до 256.
Даже 30 уже много.

Приближённый поиск

1. Посмотрели немного в биоинформатику, нашли n-граммное расстояние.
2. Сделали ngramDistance (19.5, фиксы в 19.8)
3. Чем ближе к нулю возвращаемое значение, тем строки более похожи.
4. По факту это 4 граммное расстояние (симметрическая нормализованная на количество n-грамм разность между мульти-множествами двухбайтных CRC32 хэшей).

```
SELECT DISTINCT SearchPhrase
FROM hits_100m_single
ORDER BY ngramDistance(SearchPhrase, 'clickhouse') ASC
LIMIT 20
```

SearchPhrase
tickhouse
clockhouse
house
clickhomecyprus
1click
uhouse
teakhouse.ru
teakhouse.com
madhouse
icehouse
doghouse
funhouse
dollhouse
houses.ru
bighouses
uhouse.ru
tic house
dog house
luk house
house m.d

Приближённый поиск

Реализация ngramDistance

1. Мы будем использовать 4-граммы и хэшировать их в 16 битное число с помощью инструкции crc32q, а также заведём 16-битный массив из 2^16 элементов, куда будем складывать абсолютную разницу количества этих хэшей у haystack и needle.

2. Мы заведём буффер из 19 кодовых точек, чтобы можно было копировать 3 кодовые точки из конца в начало и читать по 16 кодовых точек в буффер. Такие числа позволяют быстрее читать данные из-за SIMD инструкций, а именно получается по 13 кодовых точек за раз;

Приближённый поиск

- **Реализация ngramDistance**

```
|a0|a1|a2|a3|a4|a5|a6|a7|a8|a9|a10|a11|a12|a13|a14|a15|a16|a17|a18|
                                  Мы копируем ^^^^^^^^^^^^^^^^
                                              эти кодовые точки в начало

Теперь у нас массив выглядит так:
|a13|a14|a15|a16|a4|a5|a6|a7|a8|a9|a10|a11|a12|a13|a14|a15|a16|a17|a18|
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             И мы сюда копируем 16 кодовых точек из входного буффера
```

- **Строки длины больше 2^15 никогда не похожи друг на друга**
- **С наивной имплементации за 150 MB/s мы смогли получить 250 MB/s на один поток**

Приближённый поиск

- **ngramDistance(haystack, needle)**
- **-UTF8, -CaseInsensitive, -CaseInsensitiveUTF8**

CaseInsensitiveUTF8 имеет грязный хак -- работает только для русских и английских букв. Мы зануляем 5-й бит каждого байта и нулевой бит нулевого байта, если байтов больше одного.

- **ngramSearch(haystack, needle) (19.8)** -- приближенный поиск подстрок, идея та же. Пока в экспериментальном режиме -- надо доделать мелочь.

Приближённый поиск

- **multiFuzzyMatchAny(haystack, distance, [pattern_1, ..., pattern_n])**
- **multiMatchAnyIndex(haystack, distance, [pattern_1, ..., pattern_n])**

Это приближённый поиск hyperscan. Создаются distance+1 конечных автоматов (слои) и проводятся соответствующие переходы между соседними слоями, означающие “штраф”.

Работает долго, так как автомат усложняется и добавляются eps-переходы. Нет UTF-8 поддержки.

UTF-8 кодировка

Кодовые точки	Первый байт	Второй байт	Третий байт	Четвёртый байт
U+0000 ... U+007F	00...7F			
U+0080 ... U+07FF	C2...DF	80...BF		
U+0800 ... U+0FFF	E0	**A0**...BF	80...BF	
U+1000 ... U+CFFF	E1...EC	80...BF	80...BF	
U+D000 ... U+D7FF	ED	80...**9F**	80...BF	
U+E000 ... U+FFFF	EE...EF	80...BF	80...BF	
U+10000 ... U+3FFFF	F0	**90**...BF	80...BF	80...BF
U+40000 ... U+FFFFF	F1...F3	80...BF	80...BF	80...BF
U+100000 ... U+10FFFF	F4	80...**8F**	80...BF	80...BF

UTF-8 кодировка. Вычисление длины

```
1  inline size_t countCodePoints(const UInt8 * data, size_t size)
2  {
3      size_t res = 0;
4      const auto end = data + size;
5
6  #ifdef __SSE2__
7      constexpr auto bytes_sse = sizeof(__m128i);
8      const auto src_end_sse = data + size / bytes_sse * bytes_sse;
9
10     const auto threshold = _mm_set1_epi8(0xBF);
11
12     for (; data < src_end_sse; data += bytes_sse)
13         res += __builtin_popcount(_mm_movemask_epi8(
14             _mm_cmpgt_epi8(_mm_loadu_si128(reinterpret_cast<const __m128i *>(data)), threshold)));
15 #endif
16
17     for (; data < end; ++data) /// Skip UTF-8 continuation bytes.
18         res += static_cast<Int8>(*data) > static_cast<Int8>(0xBF);
19
20     return res;
21 }
22
```

UTF-8 кодировка. Вычисление длины

Вход:

0x00 0xE0 0x80 0xF4 0x80 0xBF 0xBF 0x7A 0xEE 0x82 0x8A 0xE1 Ox80 0x80 0x0E 0x0A

0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF 0xBF

↓ _mm_cmpgt_epi8 (pcmpgtb)

0xFF 0xFF 0x00 0xFF 0x00 0x00 0x00 0x00 0xFF 0x00 0x00 0xFF 0x00 0x00 0xFF 0xFF

↓ _mm_movemask_epi8 (pmovmskb)

0b1101000010010011

↓ __builtin_popcount (popcnt)

7

UTF-8 кодировка. Проверка на валидность

- **isValidUTF8(string) (19.7)** -- проверяет, является ли строка корректно UTF-8 закодированной

  Наивная имплементация выдавала 900 MB/s. Мы, конечно, хотим больше.

- **Использовали алгоритм “диапазонов” за основу и адаптировали под нас. Source: https://github.com/cyb70289/utf8/**

  Соптимизировали до **1.22 GB/s**

Последние приятности (после диплома)

- **toValidUTF8(string)** -- заменяем некорректные символы UTF-8 на символ `&#xFFFD;`. Все идущие подряд некорректные символы схлопываются в один заменяющий символ (19.8). No rocket science.
- **format(pattern, strings...)** -- новая функция (19.8). Сильно SSE оптимизирована.
- **concat(strings...)** -- соптимизировали до 40% через format (19.8).

```
SELECT format('{1} {0} {1}', 'World', 'Hello')

┌─format('{1} {0} {1}', 'World', 'Hello')─┐
│ Hello World Hello                       │
└─────────────────────────────────────────┘

SELECT format('{} {}', 'Hello', 'World')

┌─format('{} {}', 'Hello', 'World')─┐
│ Hello World                       │
└───────────────────────────────────┘
```

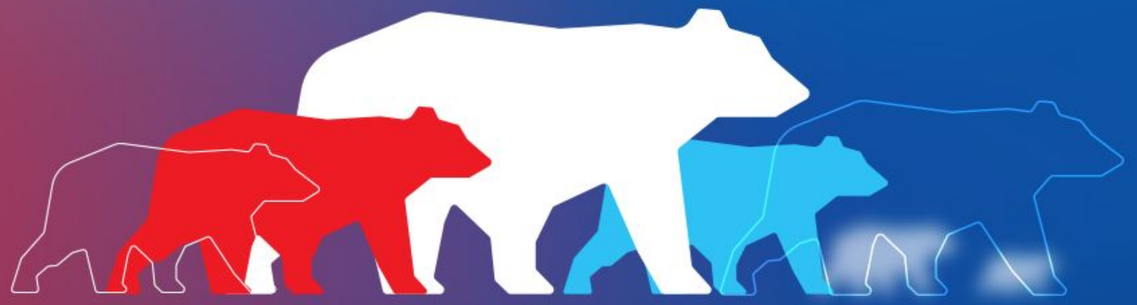
```
SELECT toValidUTF8('\x61\xF0\x80\x80\x80b')

┌─toValidUTF8('a????b')─┐
│ a?b                   │
└───────────────────────┘
```

Умные алгоритмы обработки строк в ClickHouse

Данила Кутенин

HighLoad++
Siberia 2019

Профессиональная конференция
для разработчиков высоконагруженных
систем



Яндекс

Спасибо

Данила Кутенин

 danlark@yandex-team.ru

 @Danlark