

Яндекс



Почему ClickHouse это модно?

Никита Михайлов, разработчик ClickHouse.

Обо мне

- › 4 курс ФПМИ (ФИВТ) ПМИ
- › Разрабатываю ClickHouse уже почти полгода
- › Интересуюсь распределенными системами
- › Езжу на конференции

Этим летом пошел стажироваться в команду ClickHouse и еще ни разу не пожалел об этом.

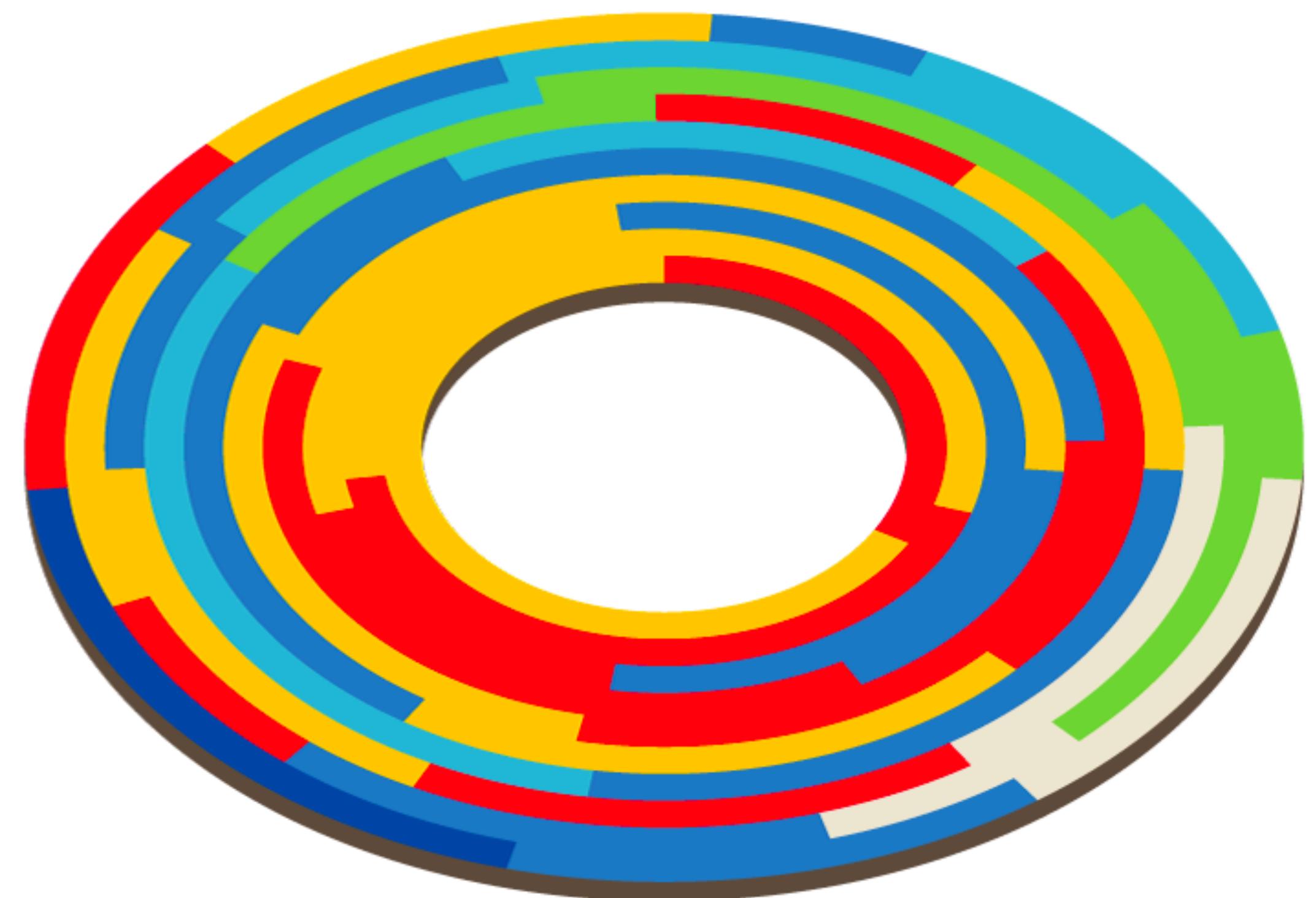
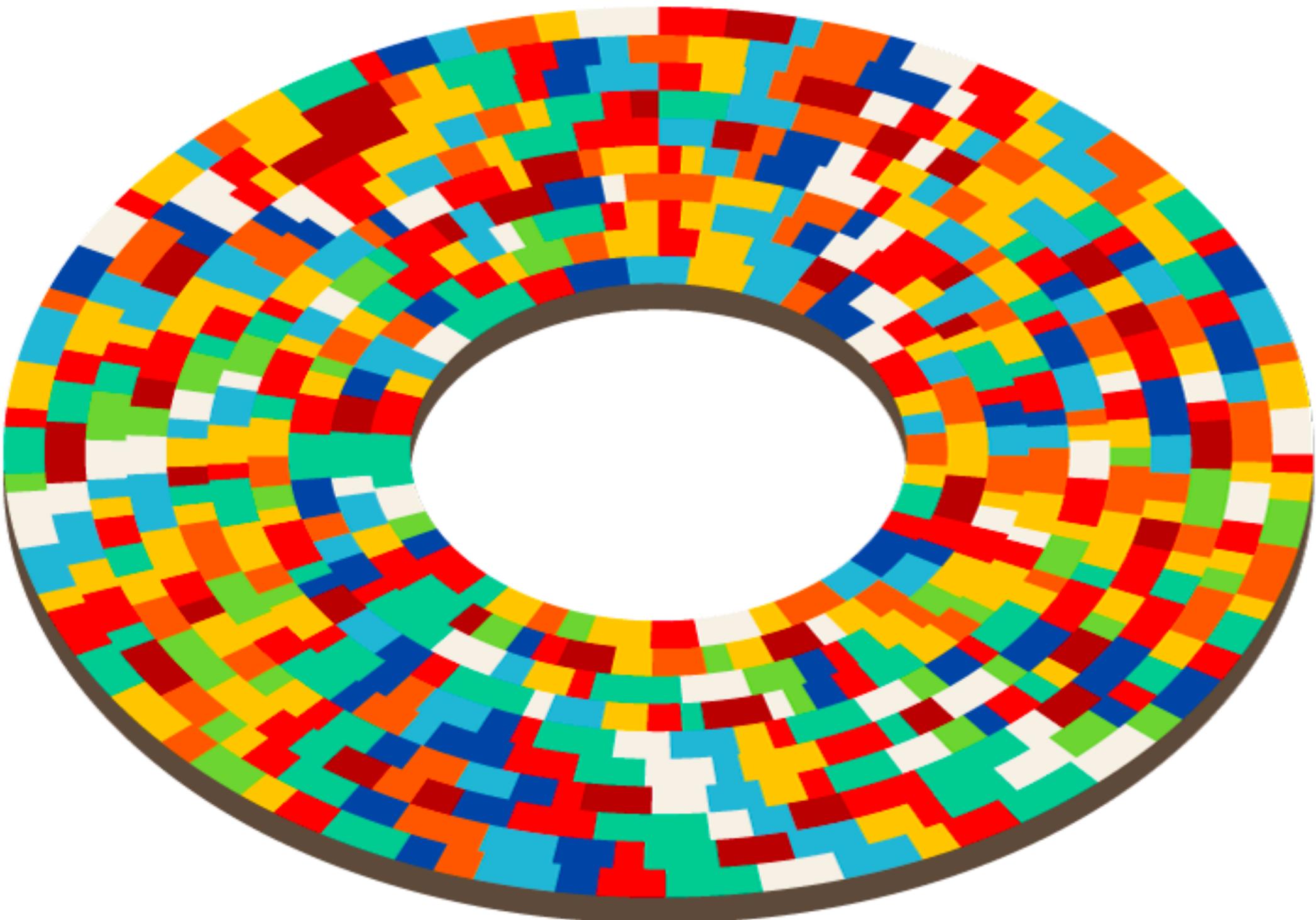
Что такое ClickHouse?

ClickHouse - распределенная колоночная аналитическая база данных с открытым исходным кодом

Умеет выполнять запросы в интерактивном режиме по неагрегированным данным, поступающим в систему в реальном времени.

Но может оказаться лучше других систем в рамках более широкой области применения.

Во время выполнения запроса с диска будут прочитаны только те колонки, которые участвуют в запросе



Где используется?

- › Аналитика веб приложений (Яндекс и другие)
- › Advertising Technology (LifeStreet)
- › Анализ операционных логов
- › Мониторинг логов безопасности
- › Аналитики DNS и HTTP запросов (Cloudflare)
- › Геокосмическая обработка (Carto)
- › Анализ бронирования отелей
- › Телеком компании
- › Мониторинг производственных процессов (Infinidat)
- › Блокчейн аналитика (Blockchair, Bloxy)
- › Биоинформатика

LIFESTREET



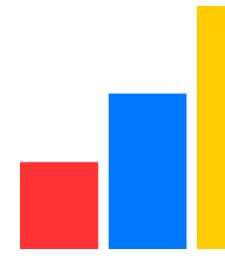
Spotify®



CLOUDFLARE



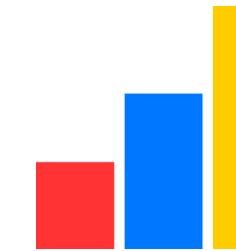
амаDEUS



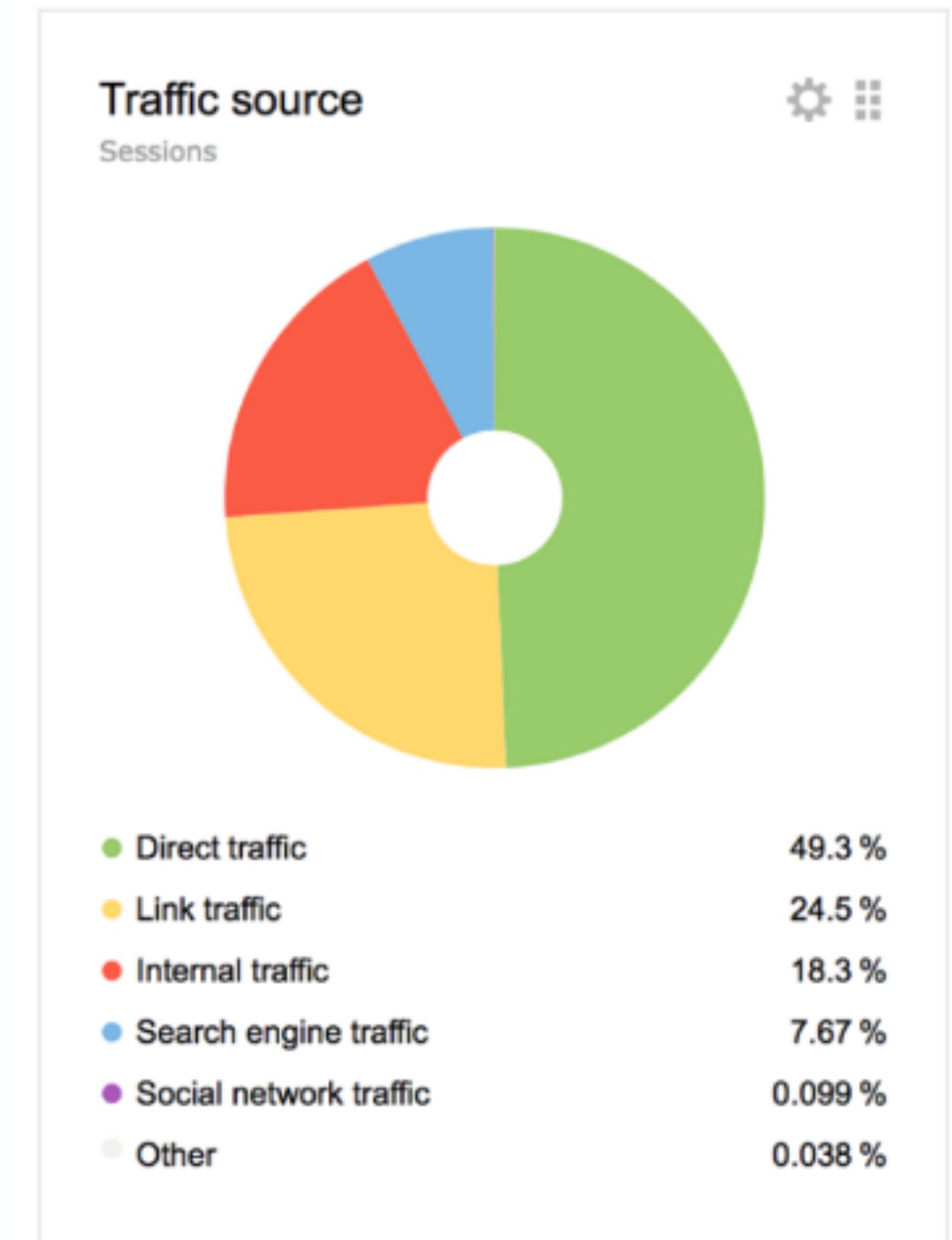
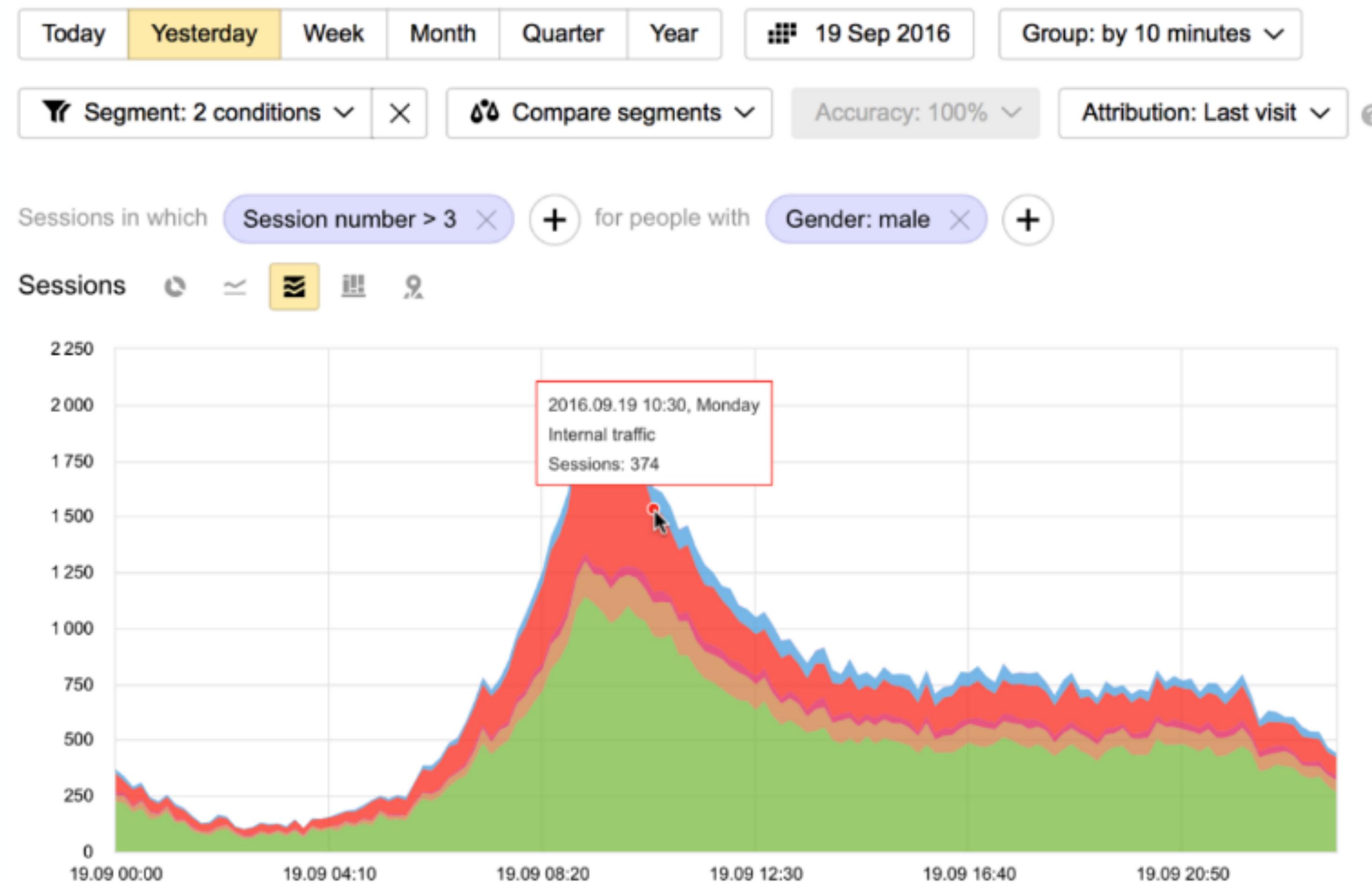
Яндекс.Метрика

| **Яндекс.Метрика - в тройке крупнейших веб-аналитических систем по количеству сайтов.**

- › Более 20 млрд. событий в день
- › Более 1 млн. сайтов
- › Более 100 000 аналитиков каждый день
- › Более 1000 машин в кластере



Яндекс.Метрика



Впечатляет?

Яндекс Кликхаус

Помощь

О сервисе

Настройки

Как работает аналитика

Яндекс Кликхаус — это сервис веб-аналитики для сайтов и электронной коммерции. Он помогает анализировать:

- аудиторию сайта и поведение посетителей;
- выручку и конверсию сайта;

```
SELECT  
    EventDate,  
    Path,  
    X,  
    Y,  
    T  
FROM mouse_clicks_layer  
SAMPLE 32768  
WHERE (CounterID = 29761725) AND (URLNormalizedHash = 11352313218531117537)  
AND ((EventDate >= '2016-05-28') AND (EventDate <= '2016-06-03'))
```

Почему?

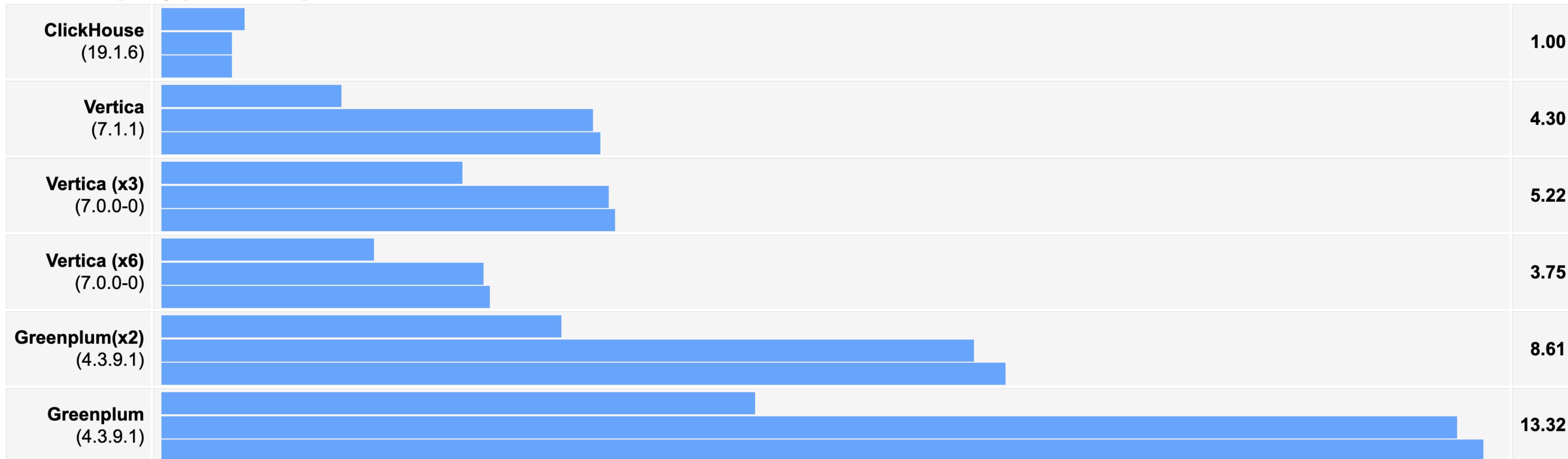
Если вы аналитик - то скорее всего используете ClickHouse.

Compare: [ClickHouse](#) [Vertica](#) [Vertica \(x3\)](#) [Vertica \(x6\)](#) [InfiniDB](#) [MonetDB](#) [Infobright](#) [Hive](#) [MySQL](#) [MemSQL](#) [Greenplum\(x2\)](#)
[Greenplum](#)

Dataset size: [10 mln.](#) [100 mln.](#) [1 bn.](#)

Run number: [first \(cold cache\)](#) [second](#) [third](#)

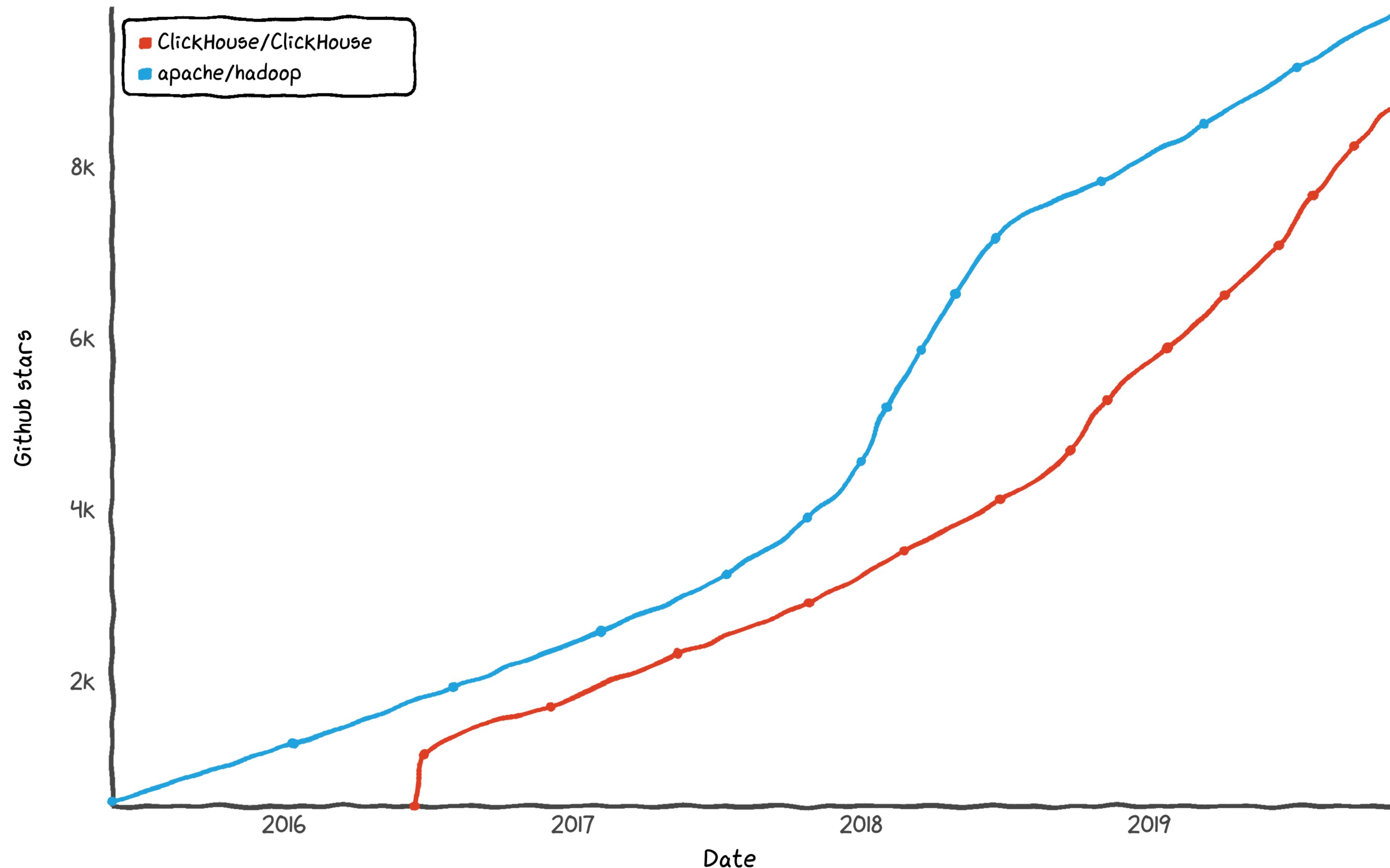
Relative query processing time (lower is better):



Независимые тесты

- › [ClickHouse: New Open Source Columnar Database](#) by Percona
- › [Column Store Database Benchmarks](#) by Percona
- › [1.1 Billion Taxi Rides on ClickHouse & an Intel Core i5](#) by Mark Litwintschik
- › [1.1 Billion Taxi Rides: 108-core ClickHouse Cluster](#) by Mark Litwintschik
- › [ClickHouse vs Amazon RedShift Benchmark](#) by Altinity
- › [Geospatial processing with Clickhouse](#) by Carto
- › ClickHouse and Vertica comparison by zhtsh
- › [ClickHouse and InfiniDB comparison](#) by RamboLau

Star history



Митапы (осень 2019)

- › Мюнхен - 17 сентября
- › Париж - 3 октября
- › Гонконг - 17 октября
- › Шэньчжэнь - 20 октября
- › Шанхай - 27 октября
- › Токио - 14 ноября
- › Стамбул - 19 ноября
- › Анкара - 21 ноября
- › Сингапур - 23 ноября





Карта митапов, проходивших осенью 2019 г. 14

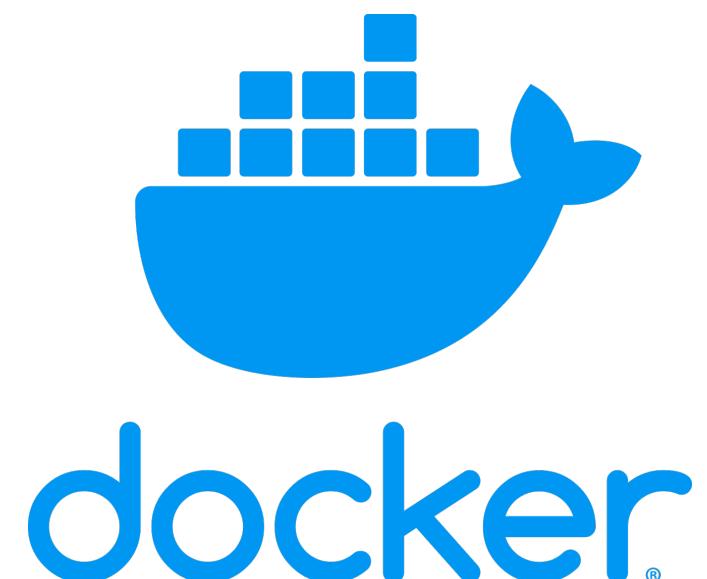
Почему ClickHouse обретает популярность?



Стек технологий

Используем самые «модные» технологии и библиотеки

- › Основная кодовая база: C++17, boost, clang + sanitizers
- › Распределенная синхронизация: ZooKeeper
- › jdbc-driver: guava, Mockito, lombok
- › Тестирование: docker, vagrant, kazoo, minio



Алгоритмы и структуры данных.

| Для каждой задачи используется уникальный алгоритм, а иногда и несколько

- › Очень быстрые хэш-таблицы

Их много, каждая - под свою задачу

- › Четыре разных алгоритма для поиска подстроки в строке

<https://habr.com/en/company/yandex/blog/466183/>

- › Thompson Sampling для оптимизации разжатия LZ4

<https://habr.com/en/company/yandex/blog/452778/>

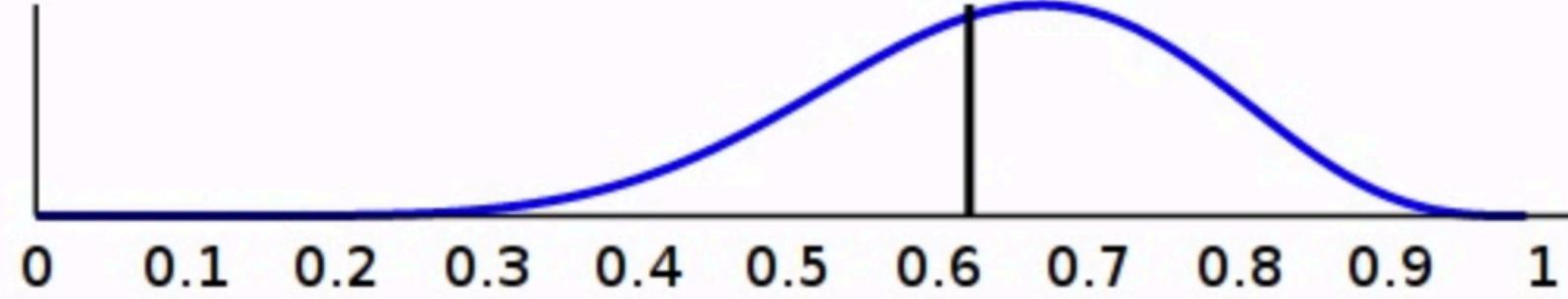
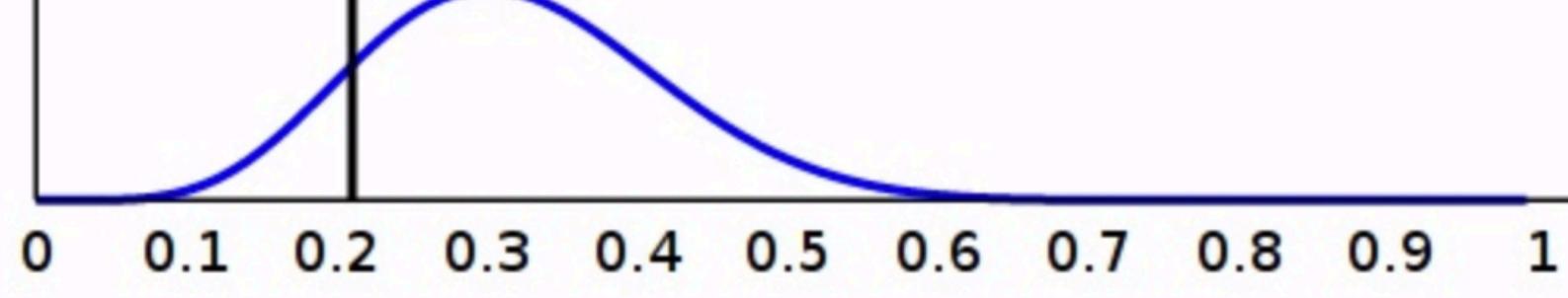
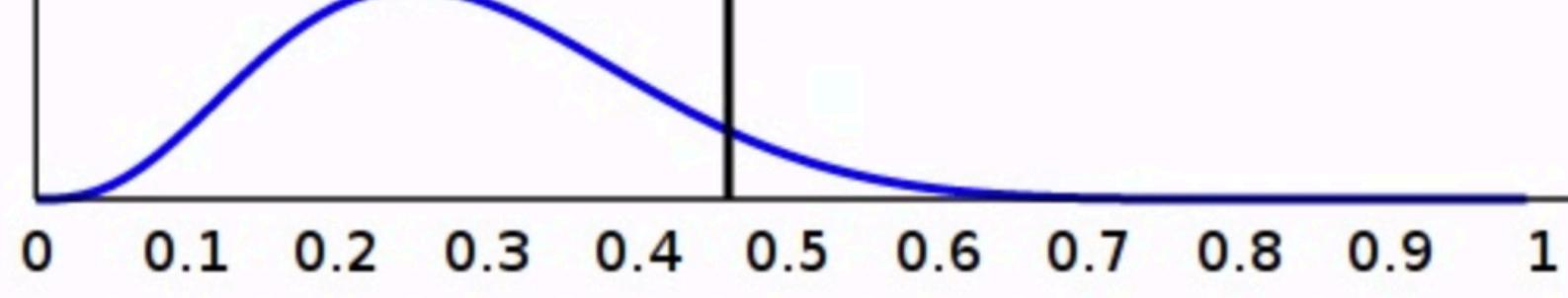
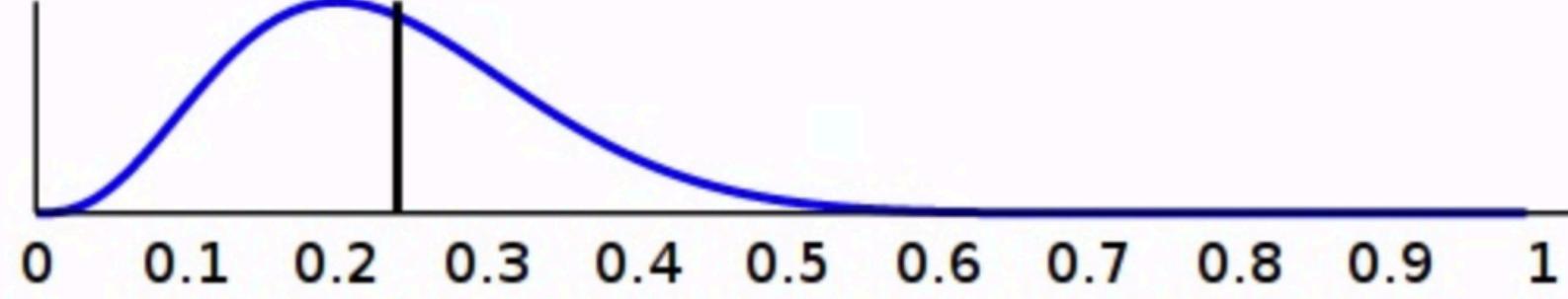
Поиск подстроки в строке

| Используется несколько алгоритмов:

- › алгоритм **Volnitsky** для константных подстрок
- › SIMD вариант brute-force алгоритма для неконстантных подстрок
- › алгоритм **Volnitsky** для небольшого множества константных подстрок
- › re2 и hyperscan для регулярных выражений

Алгоритмы подстраиваются под данные

Cumulative Regret: 15.8 after 59 pulls (minimum regret is 0)

Bandit	Actual Probability <i>Enter new value or drag mouse in input box to change</i>	Posterior Distribution <i>Black bar is the bandit's actual probability of success</i>	Hits	Misses	Total Pulls
1	0.62	 A blue beta distribution curve on a scale from 0 to 1. A vertical black bar marks the center at approximately 0.62. The x-axis is labeled with values 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.	8 (67%)	4	12
2	0.21	 A blue beta distribution curve on a scale from 0 to 1. A vertical black bar marks the center at approximately 0.21. The x-axis is labeled with values 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.	6 (30%)	14	20
3	0.46	 A blue beta distribution curve on a scale from 0 to 1. A vertical black bar marks the center at approximately 0.46. The x-axis is labeled with values 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.	3 (25%)	9	12
4	0.24	 A blue beta distribution curve on a scale from 0 to 1. A vertical black bar marks the center at approximately 0.24. The x-axis is labeled with values 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.	3 (20%)	12	15

Что сделал
непосредственно я?



Поговорим про логи



Логи - это важно!

Предположим, что мы аналитики. Или разработчики, у которых все сломалось.

Как понять, что случилось?

- › Читать логи глазами?
- › «Грепать»?
- › Писать скрипт на питоне?

Нет! Все вышеперечисленное - **очень медленно**. А никто и никто тормозить не должны.

clickhouse-local

| Лучшие фичи ClickHouse'а непосредственно с одним файлом.

```
$ clickhouse-local  
--file ~/hits_v1.tsv  
--structure 'WatchID UInt64, JavaEnable UInt8, ...'  
--query 'SELECT UserID, count() FROM table GROUP BY UserID, SearchPhrase'
```

```
Read 8873898 rows, 7.88 GiB in 5.208 sec., 1704038 rows/sec., 1.51 GiB/sec.  
UserID          count()  
8410854169855355129    3
```

А если надо посмотреть
логи ClickHouse?



system.text_log

| Сколько ошибок было на сервере сегодня?

```
SELECT count(*)  
FROM system.text_log  
WHERE (level = 'Error') AND (event_date = today())
```

| Ни одной! Потому что это ClickHouse.

system.metric_log

| **Self-monitoring.**

С настраиваемым интервалом собирает все внутренние метрики ClickHouse, буферизирует и складывает в системную таблицу.

| **Удобно строить графики состояний сервера.**

Логи в системных таблицах

| **Мы очень любим логи.**

- › system.part_log
- › system.query_log
- › system.query_thread_log
- › system.trace_log
- › system.text_log
- › system.metric_log

Описание других системных таблиц доступно по ссылке ниже.

Что еще полезного?

Параллельный парсинг
форматов данных
(но не всех)



Почему нужно ускорять?

При парсинге упираемся в CPU, а не в IO.

Bottleneck - это

- › последовательный разбор файла на токены
- › выведение типов
- › проверка на корректность относительно формата данных и схемы таблицы
- › вставку значений по умолчанию вместо пропущенных

| **То есть нужно запускать парсинг частей файла параллельно.**

Почему задача нетривиальная?

| Как разбить файл на куски?

```
int fseek( FILE *stream, long offset, int origin );
```

- › Sets the file position indicator for the file stream `stream` to the value pointed to by `offset`.

Решение - давайте для каждого отдельного потока с номером `k` сместим курсор на `1000k` байт от начала, далее - по циклу.

- › Такая реализация обречена на провал - получившийся кусок может быть невалидным с точки зрения выбранного формата.

Подумаем еще

Для форматов данных TSV, CSV, JSONEachRow разделителем является \n.

Ок. Давайте делать **fseek** на k^*1000 , идти вперед до ближайшего разделителя.

- › В случае TSV и CSV разделитель может экранироваться (входные данные могут содержать строки с переносами).
- › В случае JSONEachRow такая же ситуация + поле может быть типа JSONEachRow. Получаем вложенную иерархию JSON'ов. Понять, на каком уровне иерархии мы находимся - невозможно. Для этого нужно пересчитывать скобочный баланс.

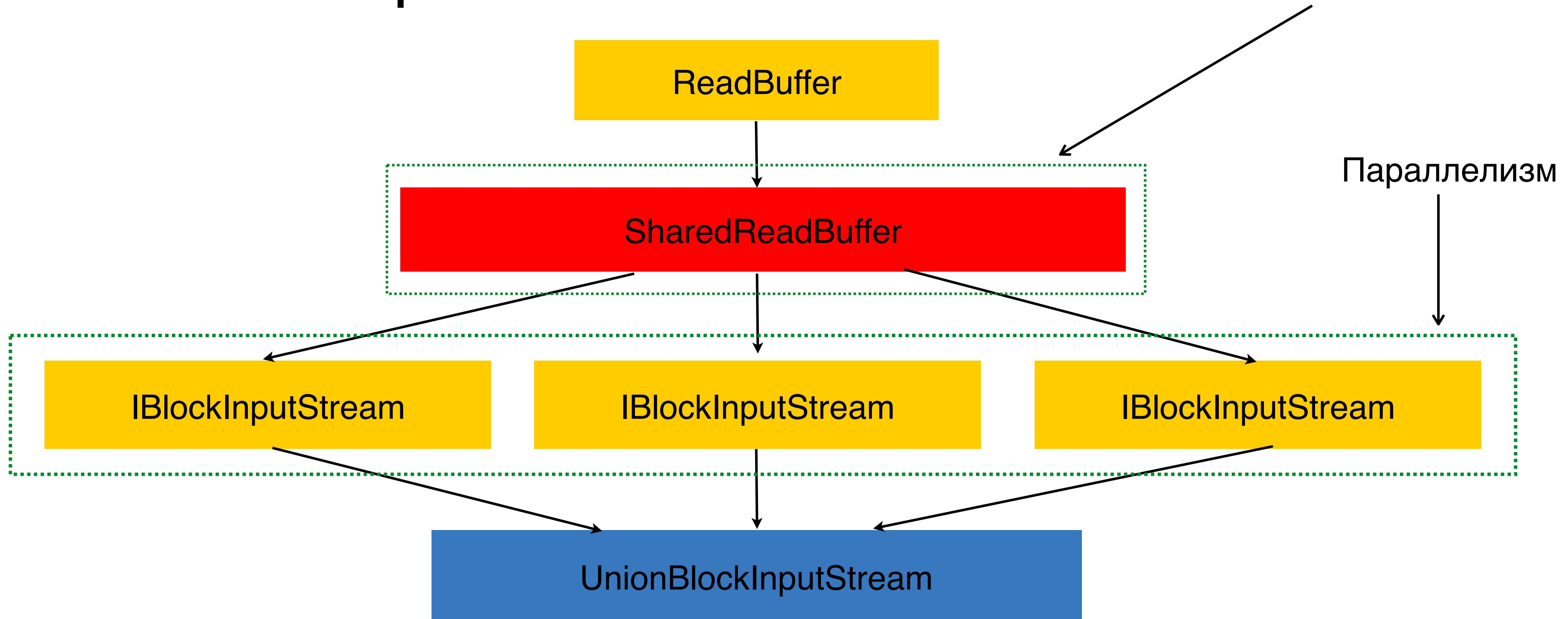
Вывод - разбиение файла на сегменты должно быть однопоточным и очень быстрым.

Введение:

- › ReadBuffer - абстрактный класс для буфферизованного чтения
- › IRowInputStream - интерфейс потока для чтения по строкам
- › IBlockInputStream - интерфейс потока для чтения по блокам

| **В каждом из интерфейсов есть виртуальная функция nextImpl().**

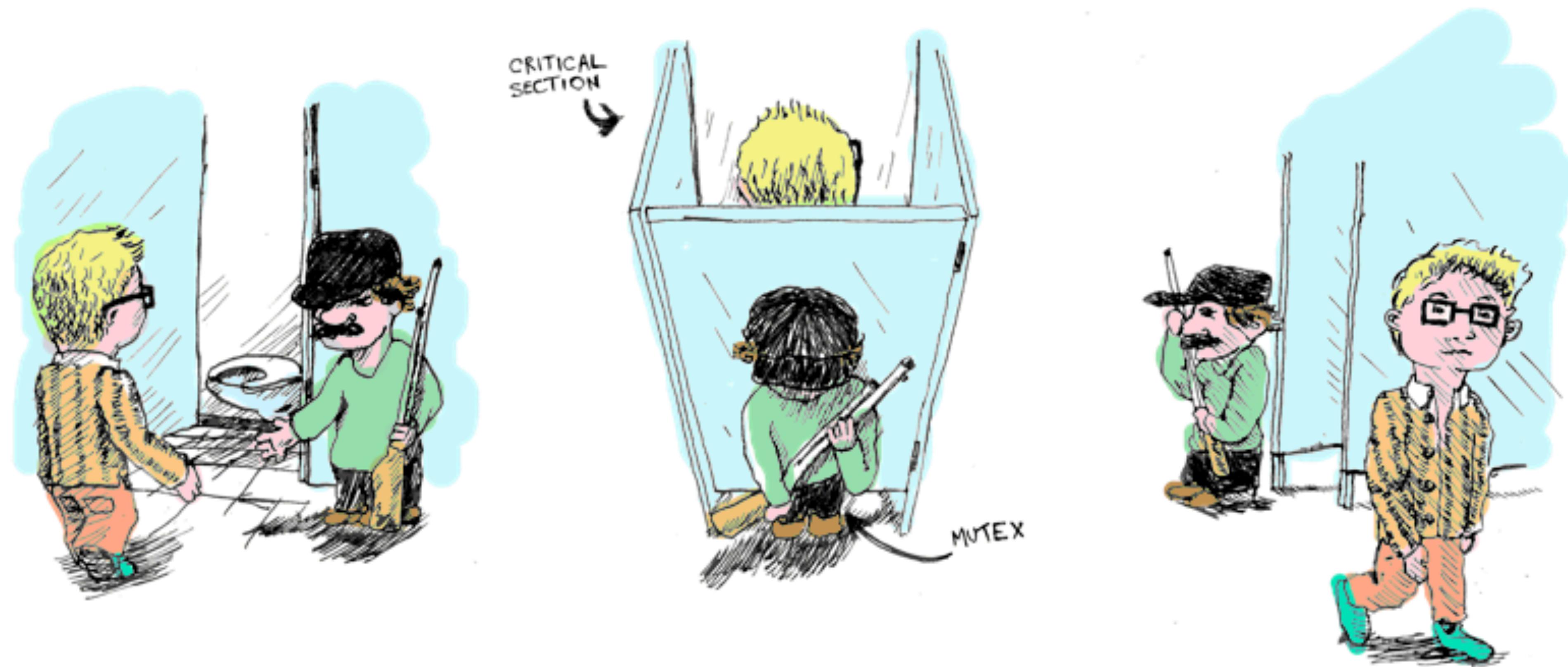
Что было раньше?



std::mutex

| **Примитив синхронизации, который обеспечивает эксклюзивный доступ к разделяемым данным в многопоточной среде**

Класс, реализующий mutex, имеет методы .lock() и .unlock()



Что с этим не так?

- › Вся синхронизация - один единственный mutex.
- › Потоки встают в очередь на захват блокировки (mutex) неявно.
- › Большой контеншн или конкуренция.

| Потерян порядок вставки блоков в таблицу.

- › Пользователь не ожидает такого поведения.
- › Что, если поступающие данные уже отсортированы?

Как сделать хорошо?

Подумать.



Выделим три роли:

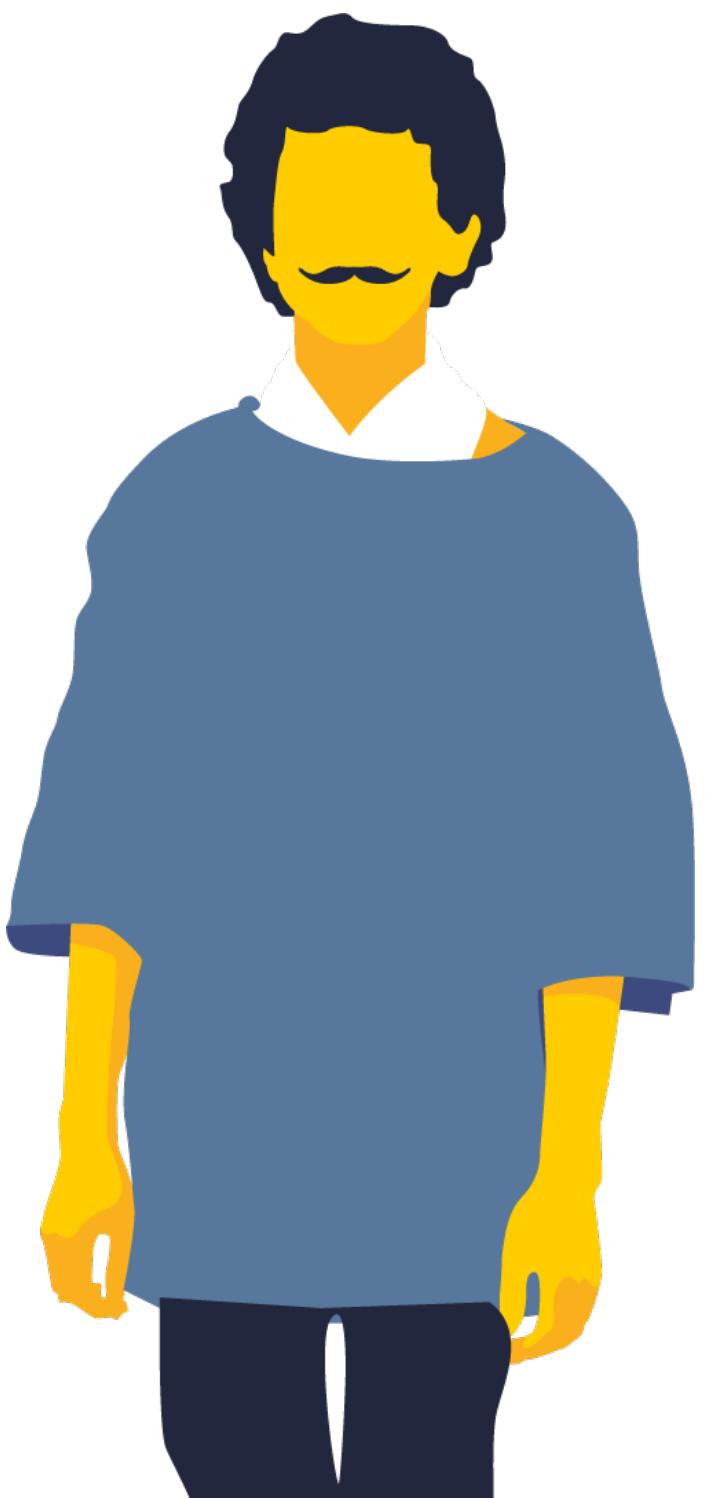
"Сегментатор"



"Парсер"



"Читатель"



Для чего?

- > "Читатель" - реализует функцию `nextImpl()`.
- > "Парсер" - превращает сегмент памяти с данными в `Block`.
- > "Сегментатор" - идет по файлу с огромной скоростью, разбивая его на куски.

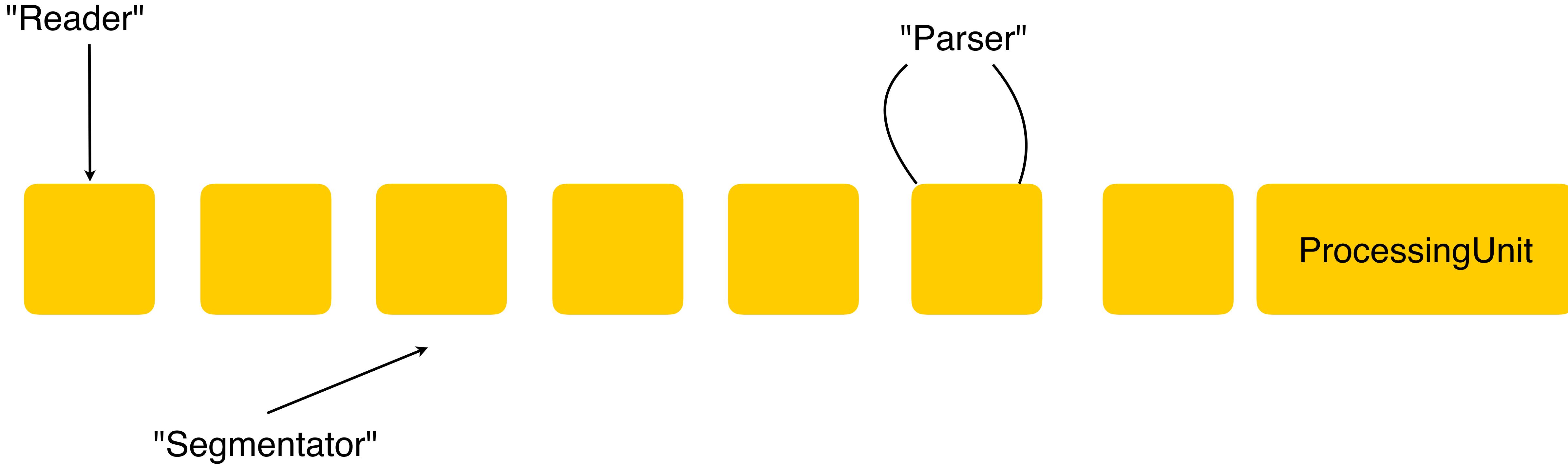
Вспомогательные структуры

```
enum ProcessingUnitStatus
{
    READY_TO_INSERT,
    READY_TO_PARSE,
    READY_TO_READ
}
```

```
struct ProcessingUnit
{
    Block block;
    Memory<> segment;
    ProcessingUnitStatus status;
    bool is_last;
}
```

| Все ProcessingUnit сложим в контейнер, назовем его WorkingField.

Схема взаимодействия.



| Потоки «ходят» WorkingField в одну сторону для сохранения порядка, но с разной скоростью.
Нужна синхронизация!

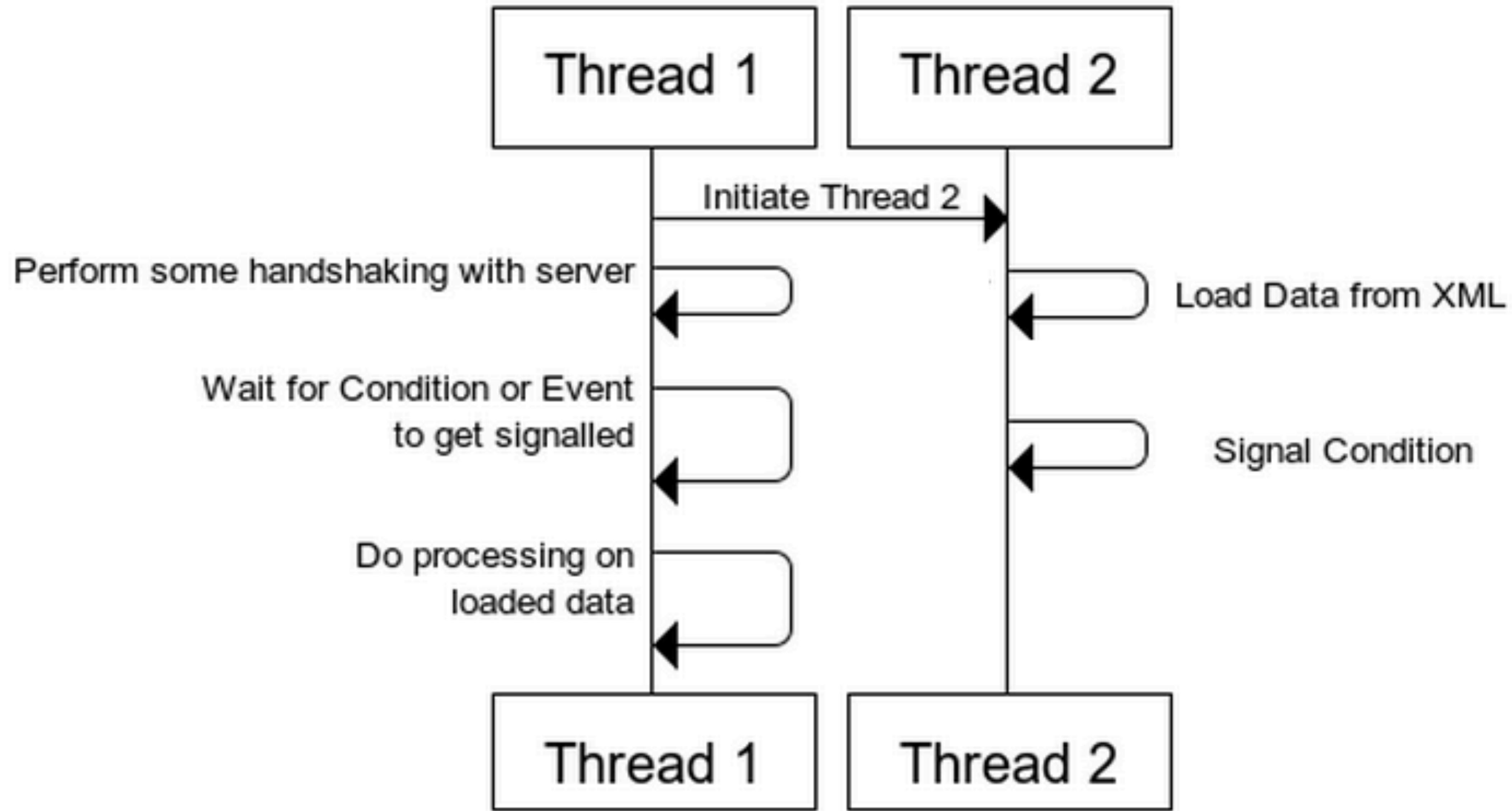
`std::condition_variable`

| **Примитив синхронизации, который используется для блокирования множества потоков до наступления одного из событий.**

- › будет получено извещение из другого потока
- › тайм-аут
- › произойдет ложное пробуждение

У класса, реализующего `condition_variable` есть методы `.wait`, `.notify_all`, `.notify_one`

Условные переменные неразрывно связаны с мьютексами



Сегментатор

Парсер

Читатель

Получаем Unit

Ждем, пока в Unit
можно вставлять

Вставляем

Запускаем поток

Парсим

Говорим, что можно
читать

Уведомляем об это
Читателя

Ждем, пока из Unit
можно читать

Читаем, возвращаем
ответ наружу

Говорим, что можно
вставлять в Unit

Уведомляем об это
Сегментатора

Сегментатор

Парсер

Читатель

Как выглядят потоки?

```
void ParallelParsingBlockInputStream::segmentatorThreadFunction()
{
    while (!finished)
    {
        auto unit = getNextUnit();
        {
            std::unique_lock lock(mutex);
            segmentator_condvar.wait(lock,
                [&]{ return unit.status == READY_TO_INSERT || finished; })
        }

        getFileSegment();

        unit.status = READY_TO_PARSE;
        scheduleParserThreadForCurrentUnit();
    }
}
```

ThreadPool

Очень полезный паттерн, который обеспечивает фиксированным количеством потоков, готовых выполнять полезную работу асинхронно.

- › Количество потоков задается в конструкторе
- › Потоки заранее инициализируются
- › Позволяет сосредоточиться на задаче, а не на синхронизации потоков

Какие были сложности?

Сложности с C++.

- › Отсутствие move и copy конструктора у ProcessingUnit. Какой контейнер выбрать для WorkingField?
- › Битовое сжатие std::vector<bool>. Неупорядоченный доступ к ячейке памяти из разных потоков - data race.

Сложности в схеме взаимодействия:

- › Как выбрасывать исключение при парсинге из другого потока?
- › Как завершить работу класса в случае исключения?
- › Какого размера сделать WorkingField?

Баги

Еще немного про баги

| **Performance тесты не завершаются в CI. Почему? Локально не воспроизводится.**

```
$ docker ps -a // Узнаем hash контейнера
$ docker exec -u root -it <hash> bash
$ ps aux | grep clickhouse // Узнаем PID процесса clickhouse-server
$ sudo -u clickhouse gdb -p <PID>
$ thread apply all backtrace
```

| **Оказалось, что один поток заблокировался на condition_variable на бесконечное время.
Другой поток бесконечно ждет первый.**

Самое главное - зачем?

SELECT * FROM table_TabSeparated	x3.50 (0.389 s.)	(0.111 s.)	OK
SELECT * FROM table_TabSeparatedWithNames	x1.01 (0.409 s.)	(0.406 s.)	OK
SELECT * FROM table_TabSeparatedWithNames	(0.393 s.)	x1.04 (0.409 s.)	OK
SELECT * FROM table_CSV	x4.44 (0.590 s.)	(0.133 s.)	OK
SELECT * FROM table_CSVWithNames	x1.01 (0.587 s.)	(0.580 s.)	OK
SELECT * FROM table_Values	(0.370 s.)	x1.03 (0.381 s.)	OK
SELECT * FROM table_JSONEachRow	x1.86 (0.776 s.)	(0.418 s.)	OK
SELECT * FROM table_TSKV	x4.37 (0.625 s.)	(0.143 s.)	OK

Теперь ClickHouse не тормозит еще больше.

Спасибо!

Никита Михайлов
Разработчик ClickHouse



jakalletti@yandex-team.ru