1. Сгенерировать любым способ 1,000,000 анкет. Имена и Фамилии должны быть реальными (чтобы учитывать селективность индекса)

С помощью гема https://github.com/ffaker/ffaker который поможет нам сгенерировать уникальные имена напишем такой метод который за раз пушит в базу 1000 записей и останется всего лишь вызвать его 1000 раз

```ruby
def insert_data
  sql = String.new("
    insert into users (name,surname,email, gender, age, city, created_at, updated_at) values
")

  x = []
  1000.times do |i|
    x << "('#{FFaker::Name.first_name}', '#{FFaker::Name.last_name.gsub("'", '')}', '#{FFaker::Internet.email}', #{[1, 2, 3
  end

  sql += x.join( separator ",\n")

  ActiveRecord::Base.connection.execute(sql)
end

ActiveRecord::Base.connection.execute("truncate table friend_requests;")
ActiveRecord::Base.connection.execute("truncate table users;")

1000.times do
  insert_data
end
```

2. Реализовать функционал поиска анкет по префиксу имени и фамилии (одновременно) в вашей социальной сети (запрос в форме firstName LIKE ? and secondName LIKE ?). Сортировать вывод по id анкеты. Использовать InnoDB движок.

Искать будем в лоб, добавим только лимит в 10 чтобы число записей возвращаемых не портило нам замеры производительности самого запроса в БД, что является целью этого урока :

select * from users where lower(name) like 'a%' AND lower(surname) like 'ma%' order by id desc limit 10

В интерфейсе воткнем формочку как в лучшие годы vkontakte - прямо сверху (только у нас будет 2 поля ввода)

name prefix:
a
surname prefix:
ma
Search

Users

Name	Surname	Age	Gender	
Alexandria	Mayert	54	female	Show
Antony	Mayert	40	female	Show
Ada	Mante	40	male	Show
Avis	Mann	103	other	Show
Antoine	Macejkovic	73	male	Show
Amalia	Mann	89	female	Show
Alvin	Mayer	64	other	Show
Anastasia	Mann	40	female	Show
Alena	Mayert	105	male	Show
Alpha	Mann	27	female	Show
Alyssa	Mayert	39	female	Show
Ahmed	Mayer	97	male	Show
Afton	Mayert	66	male	Show

3. С помощью wrk провести нагрузочные тесты по этой странице. Поиграть с количеством одновременных запросов. 1/10/100/1000.

Тут я не очень понял почему в курсе был jmeter так подробно расписан а в итоге тестируем мы с помощью wrk про который до этого ни слова не говорили, но видимо чтобы сами разобрались и в итоге было уже 2 инструмента в навыках.

Так как нам надо держать 1000 соединений, а рельса в этом плане не то чтобы топчик, попробуем запустить пуму вот так

```
 DATABASE_NAME=highload1_development WEB_CONCURRENCY=4 SECRET_KEY_BASE=fsaasfasf123
RAILS_ENV=production rails s
```

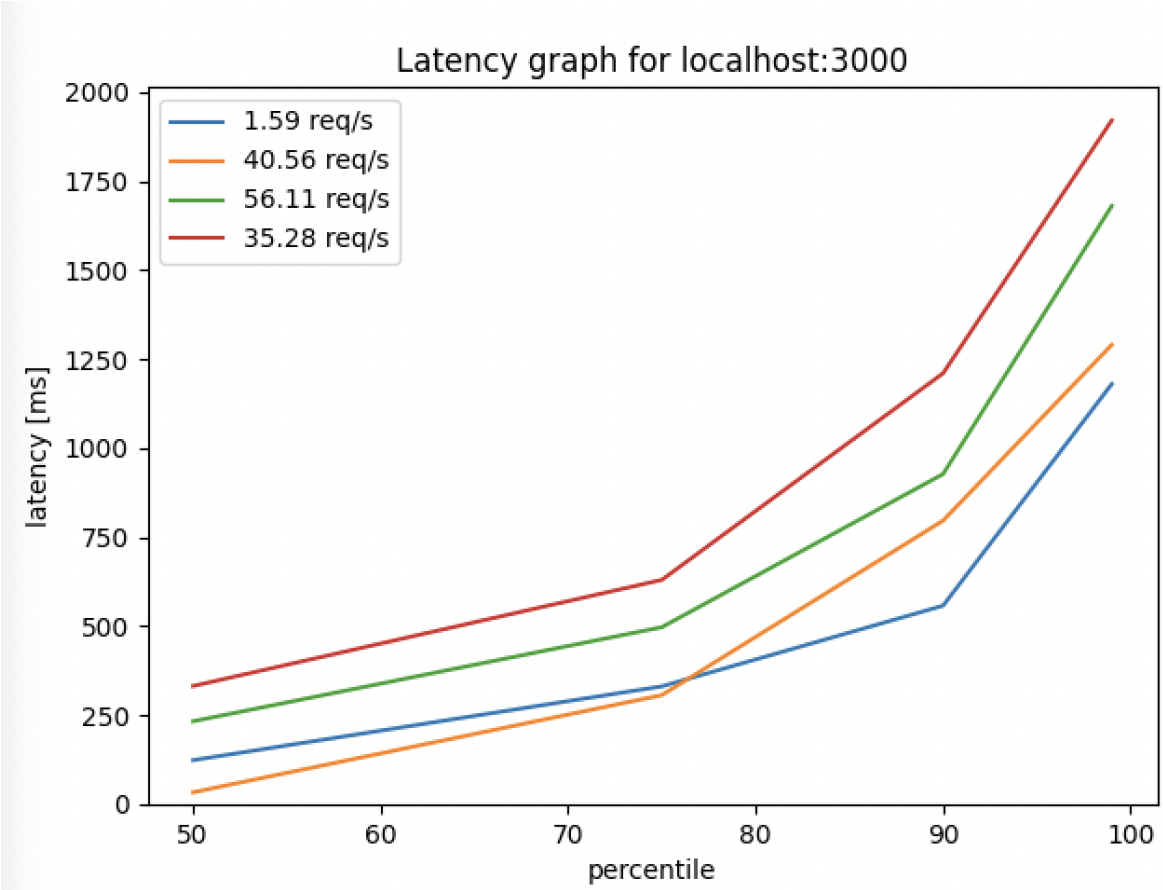
Чтобы помешать базе кэшировать ответы рандомизируем наш запрос к /users/search - для этого напишем свой скрипт для wrk

```
request = function()
  path = "/users/search?name_prefix=" .. string.char(math.random(97,122)) .. "&surname_prefix=" ..
string.char(math.random(97,122))
  return wrk.format("GET", path)
end
```

4. Построить графики и сохранить их в отчет

```
./wrk  -t 1 -c 1  -s my_script.lua --latency http://localhost:3000 > 1
./wrk  -t 6 -c 10  -s my_script.lua --latency http://localhost:3000 > 10
./wrk  -t 6 -c 100 -s my_script.lua --latency http://localhost:3000 > 100
./wrk  -t 6 -c 1000  -s my_script.lua --latency http://localhost:3000 > 1000
cat 1 10 100 1000 | wrk2img output.png
```

По мере выполнения эксперимента видно что в основном потеет mysql - 80% всех CPU на моей машине съел



Сверху вниз в легенде 1-10-100-1000

Видно что максимальная производительность была достигнута при 100 одновременных соединений а при 10000 уже что-то начало отхлебывать

5. Сделать подходящий индекс.

Тут честно говоря я вообще без идей гуглил сидел особо ничего не помогало хоть как-то оптимизировать запрос вида

explain format=json select * from users where name like 'z%' and surname like 'm%' order by id desc;

По логике лучшее что должно работать это индекс B-Tree (name, surname) тут должен задействоваться ICP так как по сути like это range то мы должны по первой части индекса легко отфильтровать по name like ? часть записей а дальше по данным хранимым в индексе отфильтровать по surname like ? ну и так как мы в индексе храним primary key то вообще шик - order by id desc тоже делаем без похода на диск. Но на деле все вообще не так


```
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "7319.16"
    },
    "ordering_operation": {
      "using_filesort": true,
      "cost_info": {
        "sort_cost": "1178.10"
      },
      "table": {
        "table_name": "users",
        "access_type": "range",
        "possible_keys": [
          "idx_name_surname5"
        ],
        "key": "idx_name_surname5",
        "used_key_parts": [
          "name"
        ],
        "key_length": "1536",
        "rows_examined_per_scan": 10604,
        "rows_produced_per_join": 1178,
        "filtered": "11.11",
        "index_condition": "((`highload1_development`.`users`.`name` like 'z%') and (`highload1_development`.`users`.`surname`
like 'm%'))",
        "cost_info": {
          "read_cost": "6023.25",
          "eval_cost": "117.81",
          "prefix_cost": "6141.06",
          "data_read_per_join": "4M"
        },
        "used_columns": [
          "id",
          "name",
```

```
        "email",
        "surname",
        "age",
        "gender",
        "password_hash",
        "city",
        "about",
        "created_at",
        "updated_at"
      ]
    }
  }
}
```

Spatial не создать на 2 колонки, Full text search только для слов - тоже не подходит. R-tree бы конечно неплохо подошел но как я понял нельзя для текстовых полей.

В общем лучшее до чего я смог дойти это сделать 3 индекса

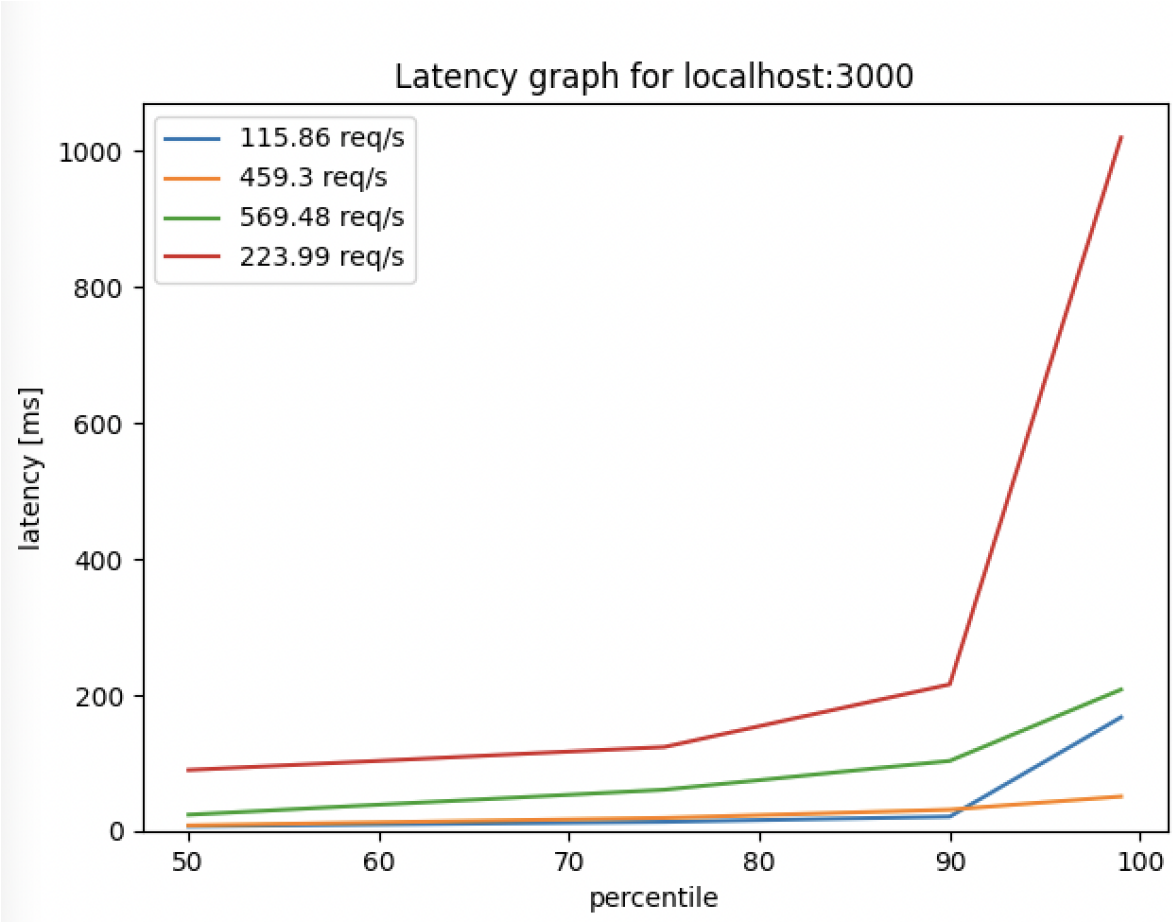
```sql
create index idx_name_surname4 on users(surname, name);

create index idx_name_surname5 on users(name, surname);

create index idx_name_surname6 on users(name(1), surname(1));
```

Тогда у нас в худшем кейсе (когда указана только по 1 букве из имени и фамилии) будет подхватываться индекс `on users(name(1), surname(1))`

А в остальных случаях (когда хотя бы в 1 параметре больше 1 символа) индекс по одному полю будет отсекать уже довольно много данных

Честно скажу решение выглядит сомнительным костылем и наверняка я что-то упускаю.

6. Повторить пункт 3 и 4.



Как видно на 1/10/100 значительно снизилась latency 95/99 percentile

Throughput тоже вырос в 5-10 раз

1000 конкарент соединений моя локальная установка совсем не хочет тянуть.


```
{
 "query_block": {
  "select_id": 1,
  "cost_info": {
   "query_cost": "755.39"
  },
  "ordering_operation": {
   "using_filesort": true,
   "cost_info": {
    "sort_cost": "421.00"
   },
   "table": {
```

```
      "table_name": "users",
      "access_type": "range",
      "possible_keys": [
        "idx_name_surname4",
        "idx_name_surname5",
        "idx_name_surname6"
      ],
      "key": "idx_name_surname6",
      "used_key_parts": [
        "name",
        "surname"
      ],
      "key_length": "12",
      "rows_examined_per_scan": 421,
      "rows_produced_per_join": 421,
      "filtered": "100.00",
      "cost_info": {
        "read_cost": "292.29",
        "eval_cost": "42.10",
        "prefix_cost": "334.39",
        "data_read_per_join": "1M"
      },
      "used_columns": [
        "id",
        "name",
        "email",
        "surname",
        "age",
        "gender",
        "password_hash",
        "city",
        "about",
        "created_at",
        "updated_at"
      ],
      "attached_condition": "((`highload1_development`.`users`.`name` like 'z%') and
(`highload1_development`.`users`.`surname` like 'm%'))"
    }
   }
  }
} |
```


Как видно радостно используем индекс по первым буквам который мы создали