## OTUS – Highload Architecture

Автор: Дмитрий Авдонин

## Задача

Выполнить нагрузочное тестирование сервиса social\_nerwork для оценки влияния индексов в БД на время отклика (latency) и пропускную способность (throughput).

## Тестовые сценарии

Выполнение GET-запроса поиска пользователя /user/search с параметрами first\_name=’Алекс%’ и second\_name=’Ив%’ в БД, содержащей не менее 1.000.000 записей. В процессе тестирования измеряем время отклика и пропускную способность для трех вариантов нагрузки:

1. Кол-во одновременных пользователей 1. Кол-во запросов 100
2. Кол-во одновременных пользователей 10. Кол-во запросов 100
3. Кол-во одновременных пользователей 100. Кол-во запросов 100

Тестовый сценарий выполнялся в двух условиях:

1. Без индексов в БД
2. С индексами в БД

**Использовались следующие индексы**

-- Создание обычных B-tree индексов

CREATE INDEX IF NOT EXISTS idx\_users\_first\_name ON users(first\_name);

CREATE INDEX IF NOT EXISTS idx\_users\_second\_name ON users(second\_name);

-- Создание индексов с оператором pattern\_ops для поиска с LIKE 'текст%'

CREATE INDEX IF NOT EXISTS idx\_users\_first\_name\_pattern ON users(first\_name varchar\_pattern\_ops);

CREATE INDEX IF NOT EXISTS idx\_users\_second\_name\_pattern ON users(second\_name varchar\_pattern\_ops);

-- Создание составного индекса для обоих полей

CREATE INDEX IF NOT EXISTS idx\_users\_names\_combined ON users(first\_name, second\_name);

**Результат explain analyze**

social\_network=# EXPLAIN ANALYZE

social\_network-# SELECT id, first\_name, second\_name, birthdate, biography, city

social\_network-# FROM users

social\_network-# WHERE first\_name LIKE 'Алекс%' AND second\_name LIKE 'Ив%'

social\_network-# ORDER BY id;

QUERY PLAN

---------------------------------------------------------------------------------------------------------------------------------------------

Sort (cost=8.46..8.47 rows=1 width=167) (actual time=0.023..0.024 rows=0 loops=1)

Sort Key: id

Sort Method: quicksort Memory: 25kB

-> Index Scan using idx\_users\_second\_name\_pattern on users (cost=0.42..8.45 rows=1 width=167) (actual time=0.020..0.020 rows=0 loops=1)

Index Cond: (((second\_name)::text ~>=~ 'Ив'::text) AND ((second\_name)::text ~<~ 'ИЯ'::text))

Filter: (((first\_name)::text ~~ 'Алекс%'::text) AND ((second\_name)::text ~~ 'Ив%'::text))

Planning Time: 0.535 ms

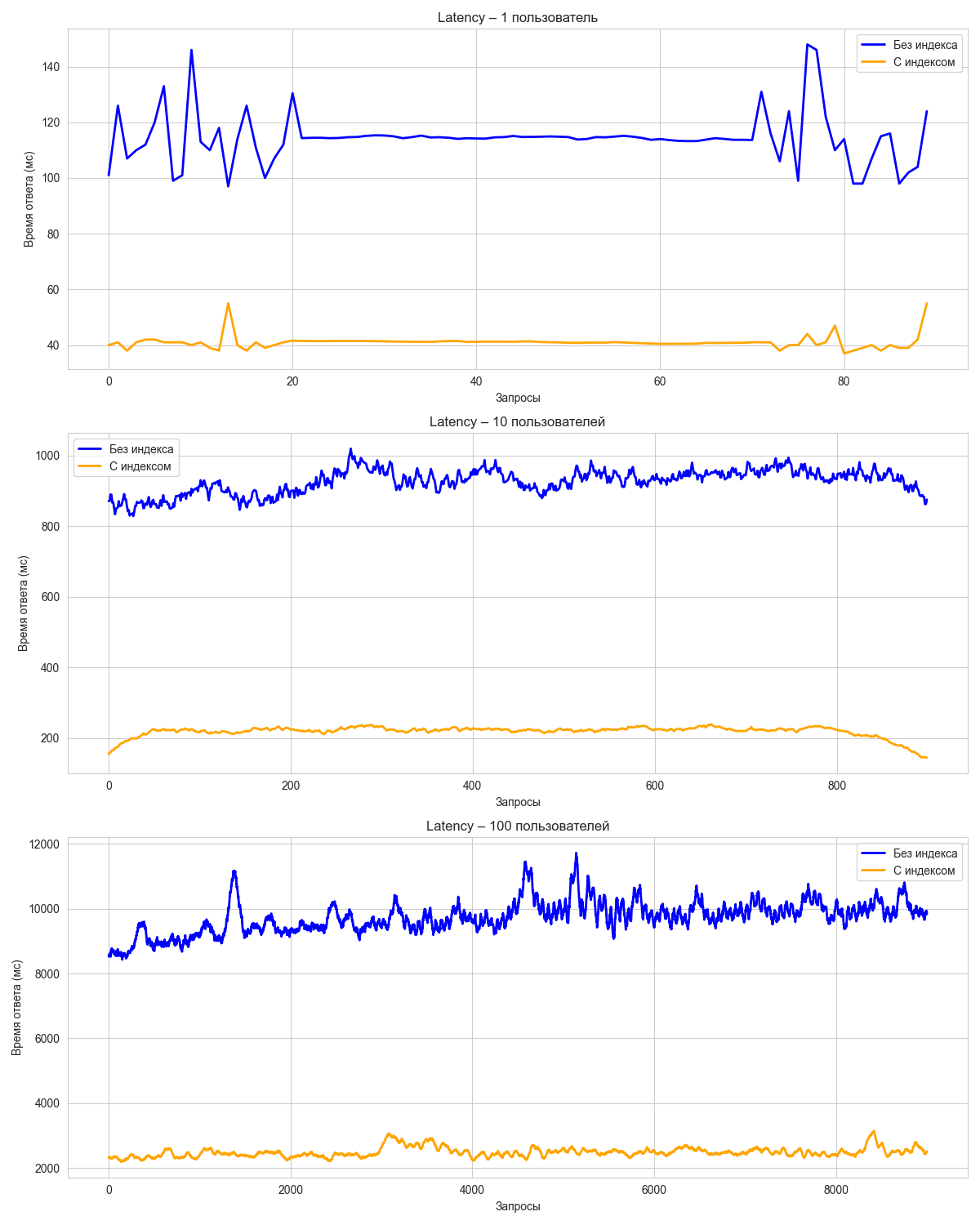
Execution Time: 0.043 ms

**Любопытное наблюдение**

Несмотря на наличие составного индекса idx\_users\_names\_combined по полям first\_name и second\_name, PostgreSQL использует индекс idx\_users\_second\_name\_pattern, который создан только для поля second\_name. Очевидно, оптимизатор PostgreSQL решил, что использование этого индекса будет эффективнее для данного запроса.

## Результаты тестов

**Время отклика**



## **Пропускная способность**

## 

## Основные выводы

Как видно из графиков, добавление индекса дает сокращение времени отклика в 3-5 раз, и 2-3 раза увеличение пропускной способности. При чем эффект от использования индексов увеличивается с ростом количества одновременных пользователей.

Самым эффективным оказался индекс с оператором pattern\_ops, т.к. он оптимален для префиксного поиска вида LIKE 'текст%'