## Сценарий

**Дата теста** 22.12.2023

1. Даем ощутимую нагрузку на метод /user/search до добавления индекса. БД заполнена предоставленными данными (1млн записей). Нагружаем и снимаем показания с помощью Jmeter. Ds
2. Измеряем результы с различными параметрами потоков/длительности. По результатам калибровки нагрузки выброна следующая методика тестирования: Делается 4 замера, в каждом общее количество запросов составляет 1000, отличается только количество одновременных потоков, в которые они кидаются (1, 10, 100, 1000) .
3. Добавляем «правильный» индекс. Поскольку по условию задания на вход метода подается префикс имени и фамилии, то можем использовать стандартный btree-индекс, который умеет работать с выражениями `LIKE a%`: *CREATE INDEX CONCURRENTLY users\_i1 ON social\_net.users (upper(first\_name) varchar\_pattern\_ops, upper(second\_name) varchar\_pattern\_ops);*

согласно документации Postgres, указание operator classes `varchar\_pattern\_ops` полезно в нашем случае, в случае если postgres не использует стандартную «C» locale. Добавление в условие индекса upper необходимо, чтобы сравнивать префиксы без учета регистра.

1. Повторяем тестирование по методике из п.2. Ожидаем увидеть «ускорение»
2. Проводим 1-кратное долгое тестирование утилитой hey с параметрами 1000 потоков по 100 запросов до и после добавления индекса. Здесь мы увидим распределение latency запросов, но не увидим график throughput (поэтому hey не использует как основной инструмент). Пункт добавлен, т. к. Jmeter в таком случае строит слишком длинный график, который не влезает в скриншот, а посмотреть на большую нагрузку «на дистанции» тоже хочется. Ну и 2-мя инструментами измерять надежнее.

## Основные выводы

* Тест по указанному сценарию успешен. По плану запроса видно уменьшение стоимости (cost), а также количество затронутых буферов при Bitmap Index scan по сравнению с Seq Scan:

***До создания индекса****:*

*social\_net=# explain (analyze, buffers) SELECT user\_id, first\_name, second\_name, birthdate, biography, city*

*FROM social\_net.users*

*WHERE upper(first\_name) LIKE upper('И%') AND upper(second\_name) LIKE upper('антоно%');*

*QUERY PLAN*

*------------------------------------------------------------------------------------------------------------------------*

*Gather (cost=1000.00..27326.83 rows=25 width=603) (actual time=54.548..80.041 rows=160 loops=1)*

*Workers Planned: 2*

*Workers Launched: 2*

*Buffers: shared hit=15074 read=2917*

*-> Parallel Seq Scan on users (cost=0.00..26324.33 rows=10 width=603) (actual time=50.817..73.782 rows=53 loops=3)*

*Filter: ((upper((first\_name)::text) ~~ 'И%'::text) AND (upper((second\_name)::text) ~~ 'АНТОНО%'::text))*

*Rows Removed by Filter: 333280*

*Buffers: shared hit=15074 read=2917*

*Planning Time: 0.203 ms*

*Execution Time: 80.079 ms*

*(10 строк)*

*==========================================*

***После создания индекса****:*

*social\_net=# explain (analyze, buffers) SELECT user\_id, first\_name, second\_name, birthdate,biography, city*

*FROM social\_net.users*

*WHERE upper(first\_name) LIKE upper('И%') AND upper(second\_name) LIKE upper('антоно%');*

*------------------------------------------------------------------------------------------------------------------------*

*Bitmap Heap Scan on users (cost=103.43..201.14 rows=25 width=603) (actual time=0.189..0.336 rows=160 loops=1)*

*Filter: ((upper((first\_name)::text) ~~ 'И%'::text) AND (upper((second\_name)::text) ~~ 'АНТОНО%'::text))*

*Heap Blocks: exact=46*

*Buffers: shared hit=88 read=4*

*-> Bitmap Index Scan on users\_i1 (cost=0.00..103.42 rows=25 width=0) (actual time=0.177..0.177 rows=160 loops=1)*

*Index Cond: ((upper((first\_name)::text) ~>=~ 'И'::text) AND (upper((first\_name)::text) ~<~ 'Й'::text) AND (upper((second\_name)::text) ~>=~ 'АНТОНО'::text) AND (upper((second\_name)::text) ~<~ 'АНТОНП'::text))*

*Buffers: shared hit=46*

*Planning:*

*Buffers: shared hit=17 read=1*

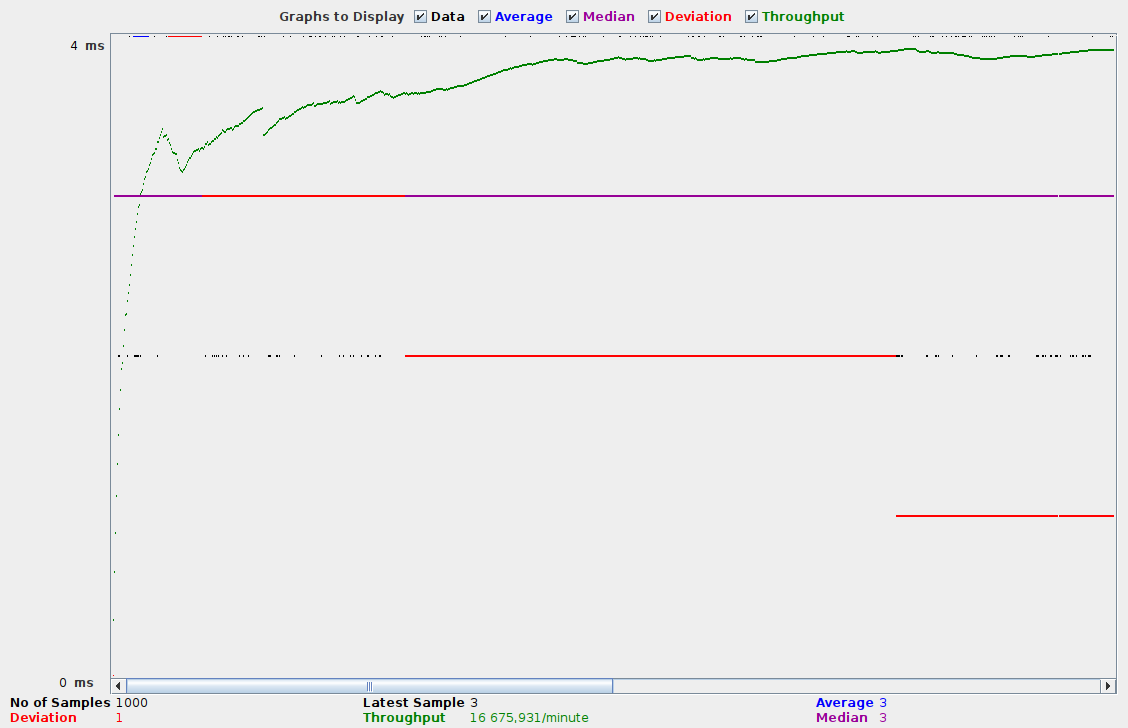
*Planning Time: 0.331 ms*

*Execution Time: 0.352 ms*

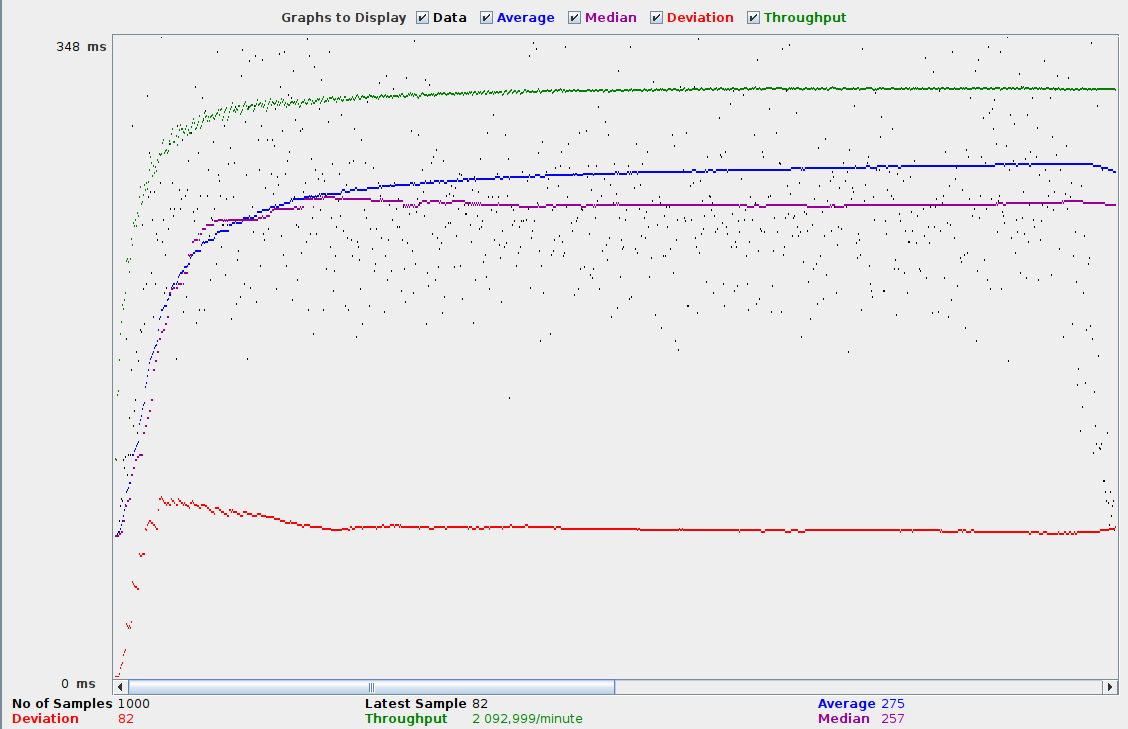
*(11 строк)*

* По времени отклика наблюдается заметно улучшение ситуации как на нагрузке в 1 поток (среднее время отклика снизилось с 73 мс до 3 мс), так и при одновременном подключении 1000 потоков (среднее время отклика снизилось с катастрофических 14 секунд до вполне приемлемых 6 мс). Остальные показатели приведены на графиках 1 — 8
* Тестирование длительной нагрузки показало схожие результаты (графики 9-10). Добавление индекса сократило время теста с 570 с до 4с, пропускная способность возросла с 18rps до 2000 rps, а 95%-персентиль latency сократился с 6 секунд до 50 мс.
* Общий вывод — индекс выбран «адекватно», позволил на порядки снизить время отклика и пропускную способность метода /user/search

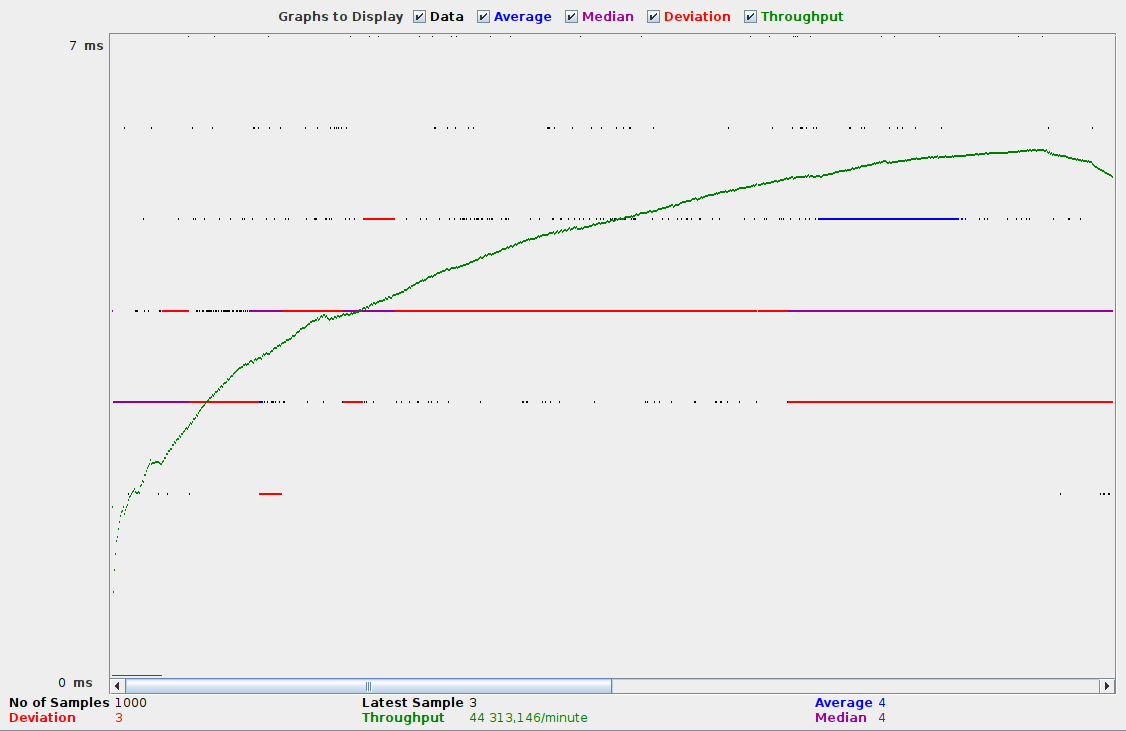
**

*График 1. Показатели 1000 запросов в 1 поток до индекса*

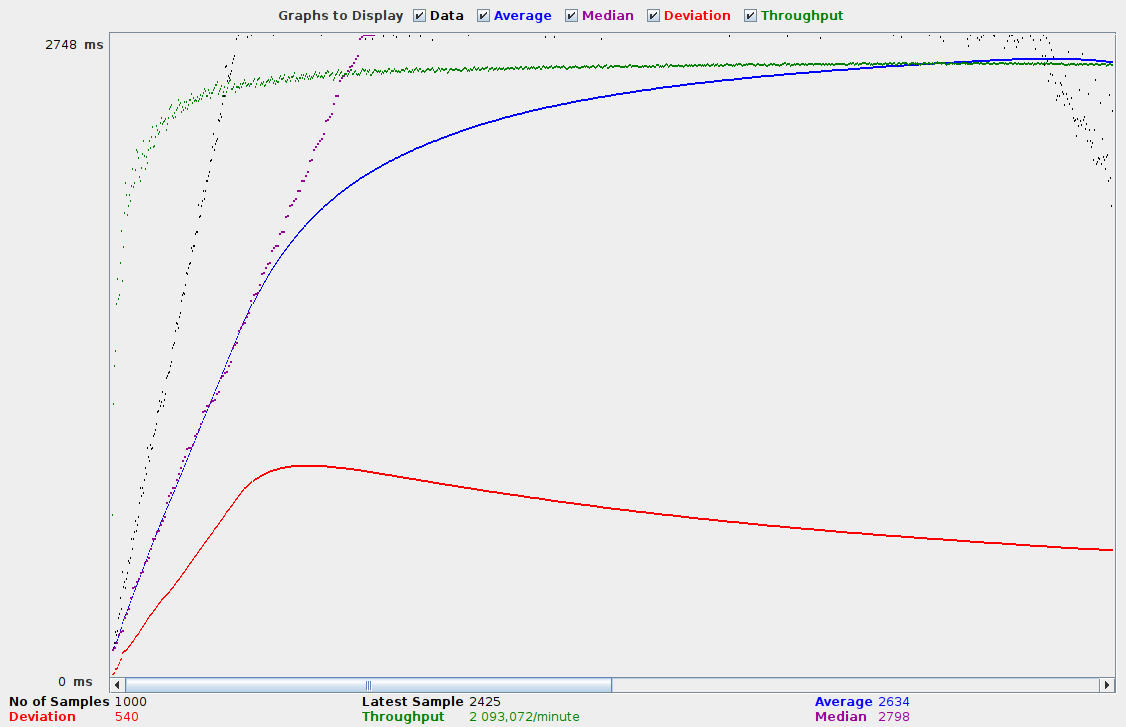
*График 2. Показатели 1000 запросов в 1 поток после индекса*

**

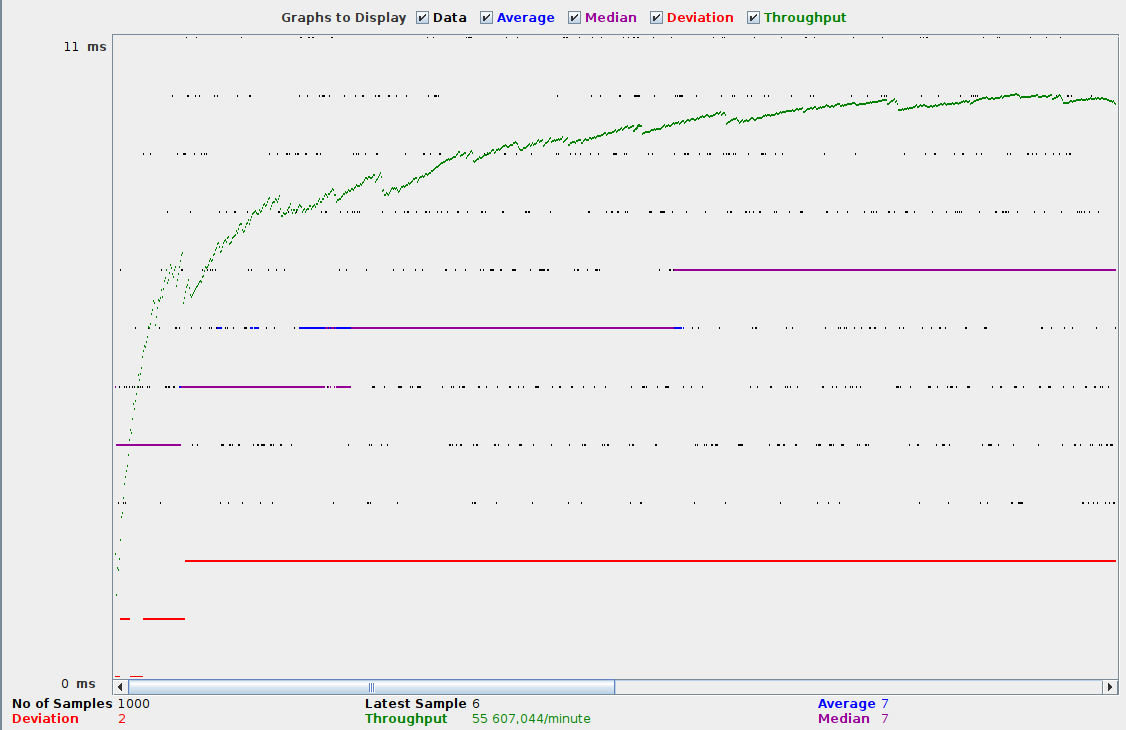
*График 3. Показатели 1000 запросов в 10 потоков до индекса*

**

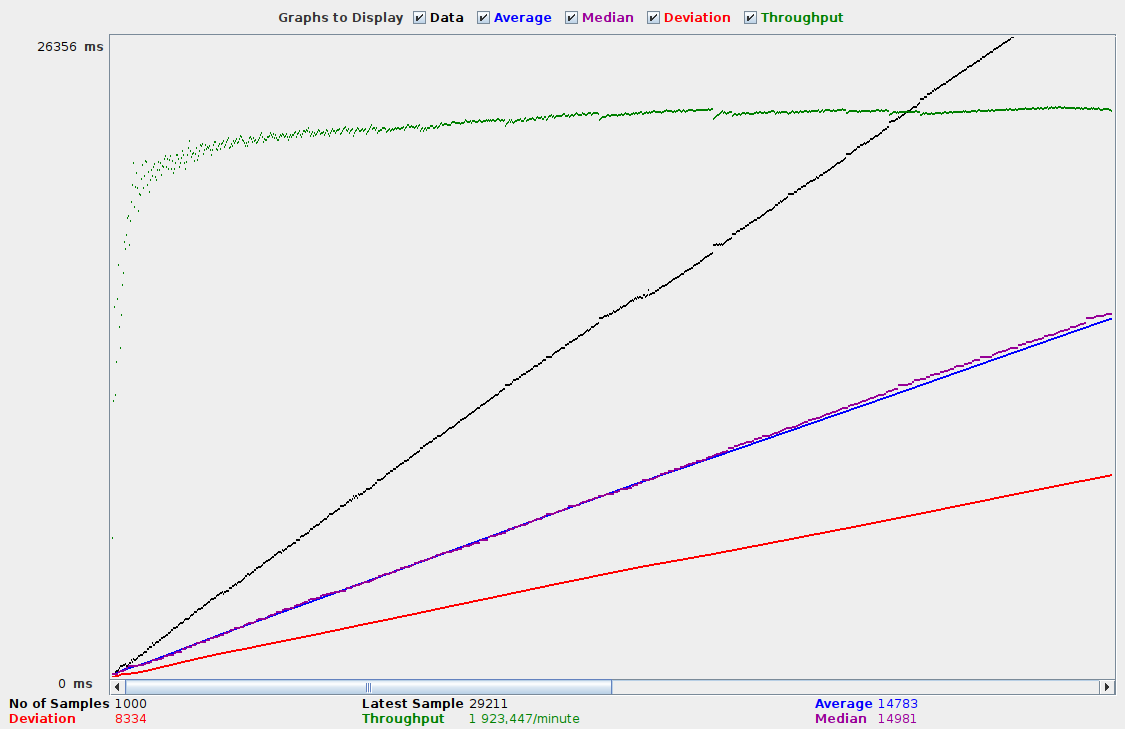
*График 4. Показатели 1000 запросов в 10 потоков после индекса*

**

*График 5. Показатели 1000 запросов в 100 потоков до индекса*

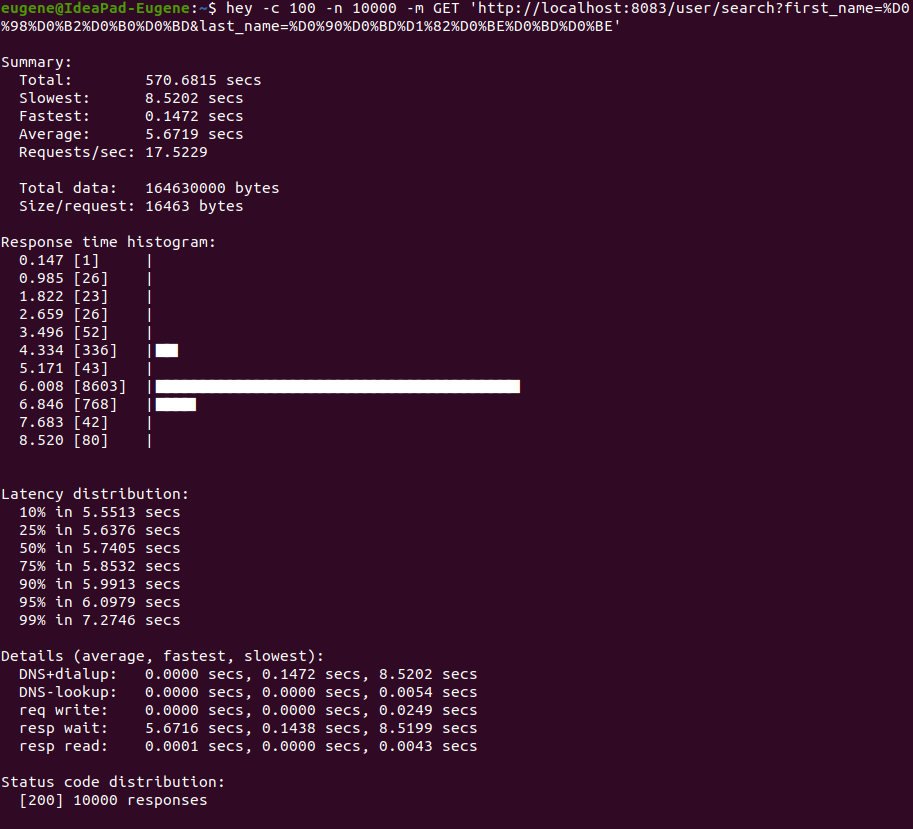
**

*График 6. Показатели 1000 запросов в 100 потоков после индекса*

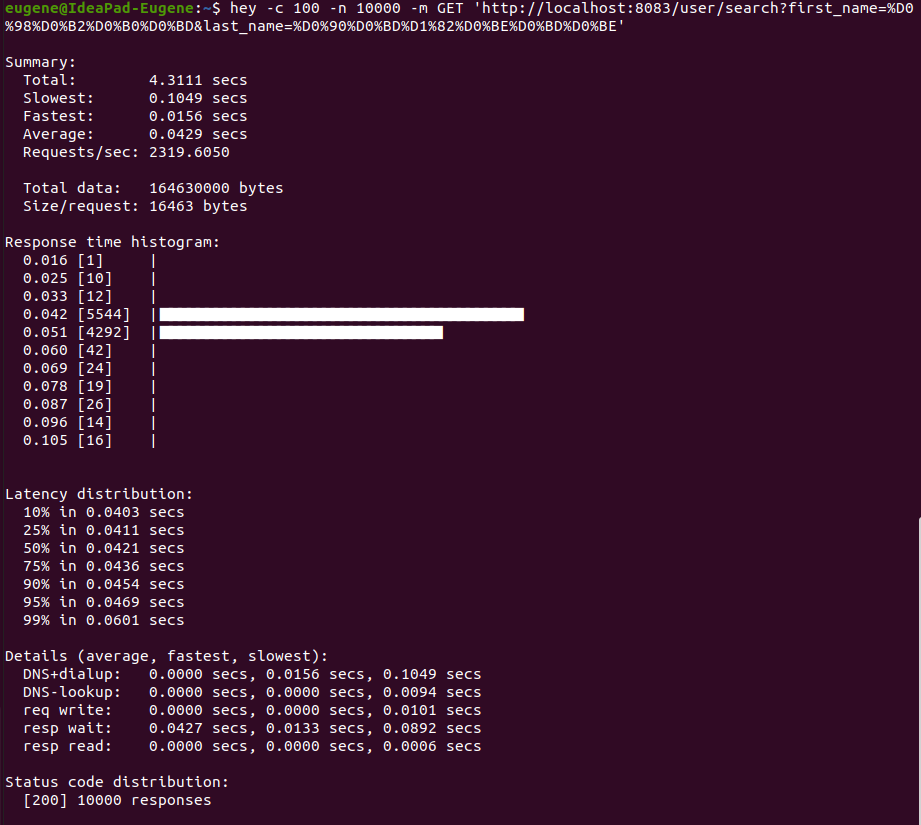
**

*График 7. Показатели 1000 запросов в 1000 потоков до индекса*

*График 8. Показатели 1000 запросов в 1000 потоков после индекса*

**

*График 9. Показатели 10000 запросов в 100 потоков до индекса*

*График 10. Показатели 10000 запросов в 100 потоков после индекса*