Отчёт о нагрузочном тестировании приложения

Сергей Герасимов

2023-01-22

Содержание

1 Подготовка

Наиболее надёжным способом с точки зрения воспроизводимости счёл следующий вариант:

- Сгенерируем 1 миллион пользователей по API (через `/user/register` ручку)
- Создадим дамп базы, загрузим его на сервер
- Добавим отдельный `docker-compose.yml`, формирующий тестовое окружение
- При запуске тестового окружения, если дамп ещё не развёрнут, скачаем и развернём его

Для замера *throughput* и *latency* использованы **Prometheus** и **Grafana**. Настройки дашбордов графаны, а также скрипт для генерации тестовых данных указаны в репозитории в `src/test/resources/`

2 Поиск в отсутствие индексов

2.1 Единичный запрос

Без нагрузки запрос выполняется порядка 500ms. Запрос вида

```
EXPLAIN SELECT u.id as id, first_name, second_name, age, t.tag_value as biography, city
        FROM users AS u
                 JOIN user_tags AS ut ON ut.user_id = u.id
                 JOIN tags AS t ON t.id = ut.tag_id AND t.tag_type = 'bio'
                 JOIN location AS l ON l.user_id = u.id
        WHERE  u.first_name LIKE 'Си%' AND u.second_name LIKE 'Ди%' ORDER BY id
```

2.2 Нагрузочное тестирование

Агрегирующие графики следующие:

Видим, что значения *throughput* и *latency* резко увеличились и продолжали такими сохраняться на протяжении всего тестирования. Общее время тестирования: *27:01*

Профиль нагрузки: 256 потоков, отправка поочереёдно пакетов из 1, 10, 150, 250, 750, 2000, 5000, 10000 запросов

3 Добавление индексов

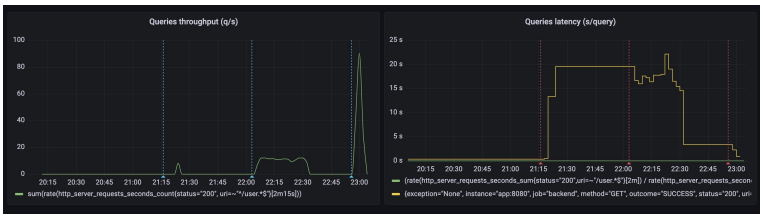
Добавление индекса по двум полям, по которым идёт поиск пользователя:

```
CREATE INDEX name_surname_index ON users(first_name, second_name)
```

4 Поиск при наличии индексов

4.1 Нагрузочное тестирование

Скорость значительно увеличилась. *Throughput* с 10 увеличилась в среднем до 40, а *latency* уменьшилась со средних 20 секунд до диапазона 0.9-2 секунды. Само тестирование проходит примерно в 6 раза быстрее, за *3:50*



Профиль нагрузки: 256 потоков, отправка поочереёдно пакетов из 1, 10, 150, 250, 750, 2000, 5000, 10000 запросов

5 Планы запросов

Приведены выжимки из планов запросов (левая часть с повторяющимися данными удалена)

```
+-------+--------------------------+------+--------+-----------+
|id.....|ref                       |rows  |filtered|Extra      |
+--.....+--------------------------+------+--------+-----------+
|1 .....|NULL                      |992801|1.23    |Using where|
|1 .....|highload_social.u.id      |1     |100     |Using where|
|1 .....|highload_social.u.id      |1     |100     |NULL       |
|1 .....|highload_social.ut.tag_id|1     |100     |Using where|
+-------+--------------------------+------+--------+-----------+
```

```
+--+----+--------------------------+-----+--------+----------------------+
|id|....|ref                       |rows |filtered|Extra                 |
+--+....+--------------------------+-----+--------+----------------------+
|1 |....|NULL                      |19688|11.11   |Using index condition;|
|  |    |                          |     |        |Using filesort        |
|1 |....|highload_social.u.id      |1    |100     |Using where           |
|1 |....|highload_social.u.id      |1    |100     |NULL                  |
|1 |....|highload_social.ut.tag_id|1    |100     |Using where           |
+--+----+--------------------------+-----+--------+----------------------+
```

Видно, что количество рядов для фильтрации значительно уменьшено, т.к ненужные ряды отброшены индексом.

6 Выводы

Индексы значительно увеличивают производительность приложения, однако нужно принимать в расчёт сценарии использования базы. Например, индексом поддерживается только wildcard как суффикс, в случае префиксного wildcard (%name% или %name) такой индекс как выше окажется бесполезным (но поможет, например, *trigram index*)