Домашнее задание №7

Применение In-Memory СУБД.

Цель:
В результате выполнения ДЗ вы перенесете хранение одного из модулей вашего приложения в In-Memory СУБД.
В данном задании тренируются навыки:

- администрирование In-Memory СУБД;
- разработка хранимых процедур для In-Memory СУБД.

Тестовые данные

Диалоги генерировались по 10 диалогов на каждого пользователя с id от 1 до 1000, id второго собеседника на [1,10] больше первого, то есть например такие:

- 1, 2
- 1, 3
- ...
- 1, 10
- 2, 3
- ...
  Всего 10 * 1000 = 10 000 диалогов

Сообщения генерировались по 10 сообщений для каждого диалога.
Всего 10 000 * 10 = 100 000 сообщений.

Postgres

```
DO $$
    DECLARE
        dialog_record RECORD;
        message_id UUID;
        author INT;
        message_text TEXT;
        sent_at TIMESTAMP;
    BEGIN
        -- Цикл по всем диалогам в таблице dialogs
        FOR dialog_record IN SELECT id FROM dialogs LOOP
                -- Генерация 10 сообщений для каждого диалога
```

```
                FOR i IN 1..10 LOOP
                        -- Генерация уникального идентификатора для
сообщения
                        message_id := gen_random_uuid();

                        -- Случайный выбор автора сообщения (один из
участников диалога)
                        author := (SELECT (string_to_array(dialog_record.id,
','))[floor(random() * 2 + 1)]);

                        -- Генерация текста сообщения
                        message_text := md5(random()::text);

                        -- Установка времени отправки сообщения (случайное
время в пределах последнего месяца)
                        sent_at := NOW() - INTERVAL '1 day' * FLOOR(RANDOM()
* 30)
                            + INTERVAL '1 hour' * FLOOR(RANDOM() * 24)
                            + INTERVAL '1 minute' * FLOOR(RANDOM() * 60)
                            + (RANDOM() * INTERVAL '1 second');

                        -- Вставка сообщения в таблицу messages
                        INSERT INTO messages (id, dialog_id, author, text,
sent_at)
                        VALUES (message_id, dialog_record.id, author,
message_text, sent_at);
                    END LOOP;
            END LOOP;
    END $$;
```

Redis

Здесь диалоги не потребовались, достаточно было формировать ключ для списка сообщений в формате `messages:${dialogId}`, где `dialogId="${userId1}/${userId2}"` (userId1 всегда меньше userId2 для консистентного ключа при запросах от обоих пользователей).

```
private void generateDialogs(boolean generateMessages) {
    var random = new Random();
    int users = 1000;
    int dialogs = 100;
    int messages = 10;
    for (int i = 1; i <= users; i++) {
        generateDialogsWithMessages(dialogs, i, users, messages, random);
    }
```

```
}

private void generateDialogsWithMessages(int dialogs, int from, int users,
int messages, Random random) {
    for (int j = 0; j < dialogs; j++) {
        var to = from + j + 1;
        var id = "%d,%d".formatted(from, to);
        if (to > users) {
            to = to - users;
            id = "%d,%d".formatted(to, from);
        }
        for (int x = 0; x < messages; x++) {
            var msg = Message.builder()
                             .dialogId(id)
                             .author(random.nextInt(2) == 0 ? from : to)
                             .text(UUID.randomUUID().toString())

.sentAt(LocalDateTime.now()

.toInstant(UTC)

.toEpochMilli()
                                                              )
                             .build();
            dialogRepository.sendMessage(msg);
        }
    }
}
```

Планы нагрузки

Все нагрузки выполнялись с помощью Apache JMeter с такими параметрами:

- 1 thread group
- 200 threads (users)
- ramp-up period: 1 second
- duration: 60 seconds

Нагрузка на чтение

Для создания тестовой нагрузки на чтение использовался 1 API endpoint:
`GET /dialog/${from}/${to}`

- `${from}` – id пользователя, случайно сгенерированное число от 1 до 1000

- `${to}` – id пользователя, `${from} + ${random(1,10)}`

Нагрузка на запись

Для создания тестовой нагрузки на запись использовался 1 API endpoint: `POST /dialog/${from}/${to}`

- `${from}` – id пользователя, случайно сгенерированное число от 1 до 1000
- `${to}` – id пользователя, `${from} + ${random(1,10)}`

Перенос функций в Redis

Проверялись 2 основных функции:

1. `get_messages`: отправить сообщение от одного пользователя другому
2. `send_message`: получить список сообщений диалога между двумя пользователями

Исходная реализация на Postgres подразумевала логику на стороне приложения и транзакционность при работе с БД.

В случае Redis я использовал встроенные функции на Lua. Они загружаются в Redis при старте приложения. В продакшн среде рекомендуется использовать другой подход с постоянным сохранением встроенных функций.

get_messages

```
local function get_messages(dialog_id)
    return redis.call("LRANGE", "messages:" .. dialog_id, 0, -1)
end
```

Поскольку тестовые данные ограничены 10 сообщениями в диалоге, здесь выполняется получение всего списка сообщений. В реальной среде количество сообщений в диалоге может исчисляться тысячами и даже больше, и в этом случае рекомендуется загружать данные постранично (например, по 50 или по 100 сообщений за один вызов).

send_message

```
local function send_message(dialog_id, author, text, sent_at)
    local message = cjson.encode({
            author = author,
            text = text,
            sent_at = sent_at
        })
```

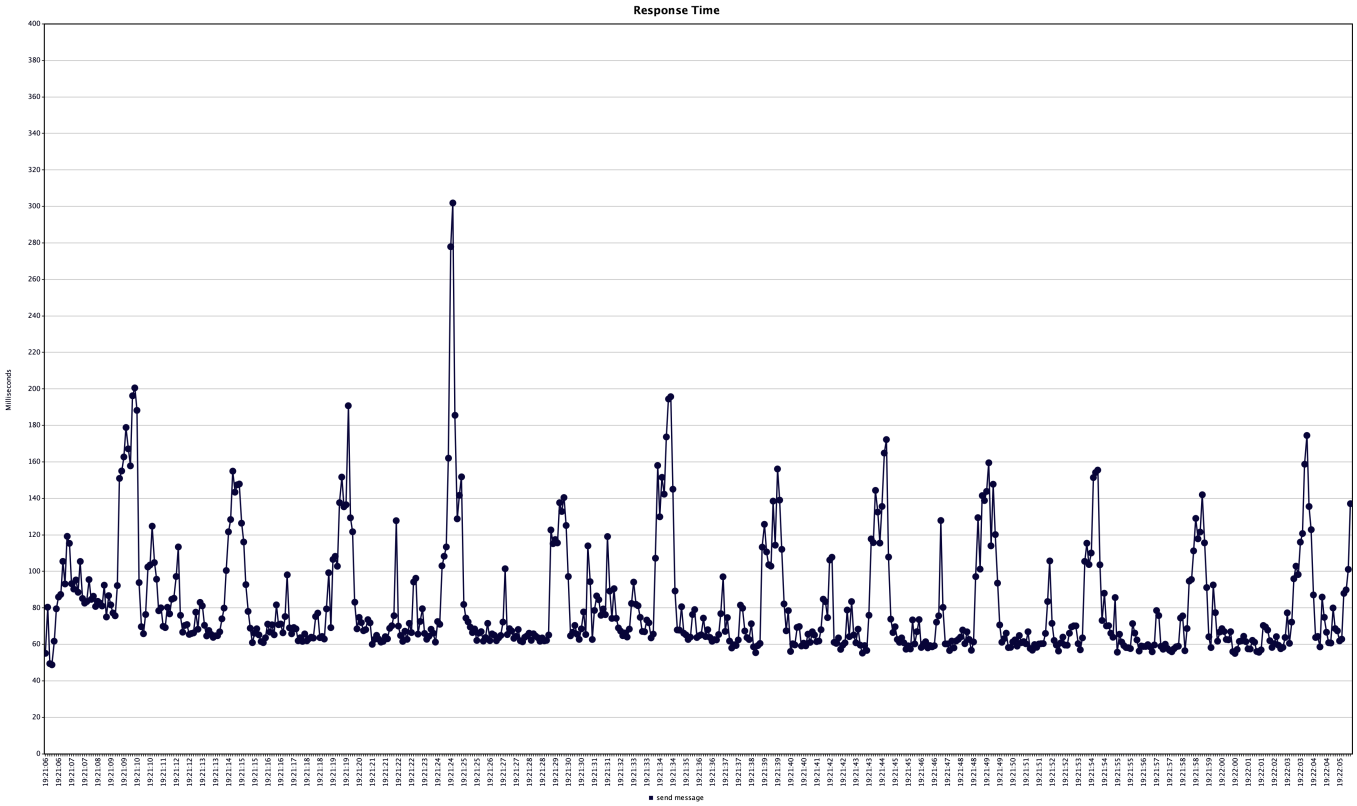
```
        redis.call("RPUSH", "messages:" .. dialog_id, message)
end
```

Все необходимые параметры для сообщения получены со стороны приложения, новое сообщение добавляется в конец списка диалога.

Профили нагрузки

Отправка сообщения

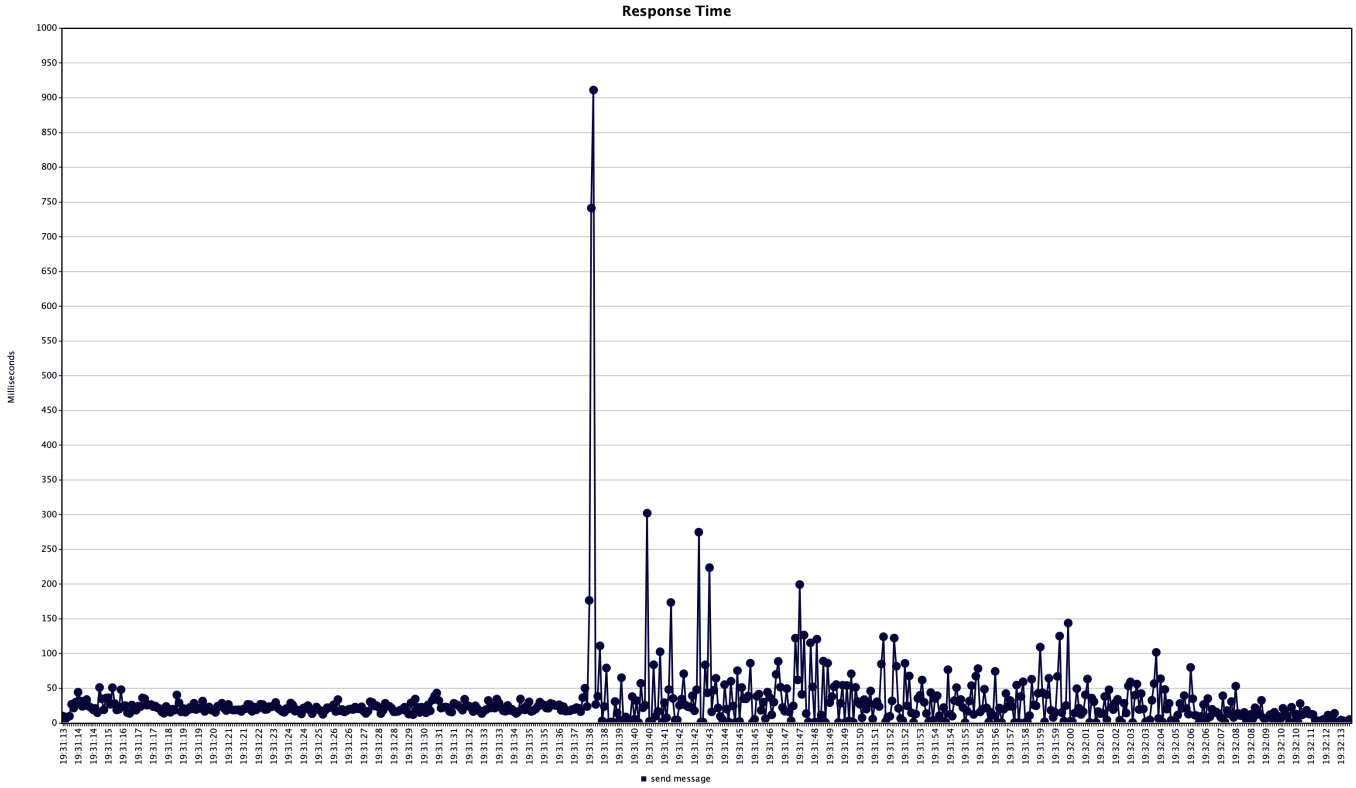
Postgres



Время ответа сервиса:

- min: 50 мс
- max: 300 мс
  Много запросов колеблется в диапазоне 60-80мс, однако наблюдаются частые пики до 100-200мс (в отдельном случае до 300 мс).

Redis

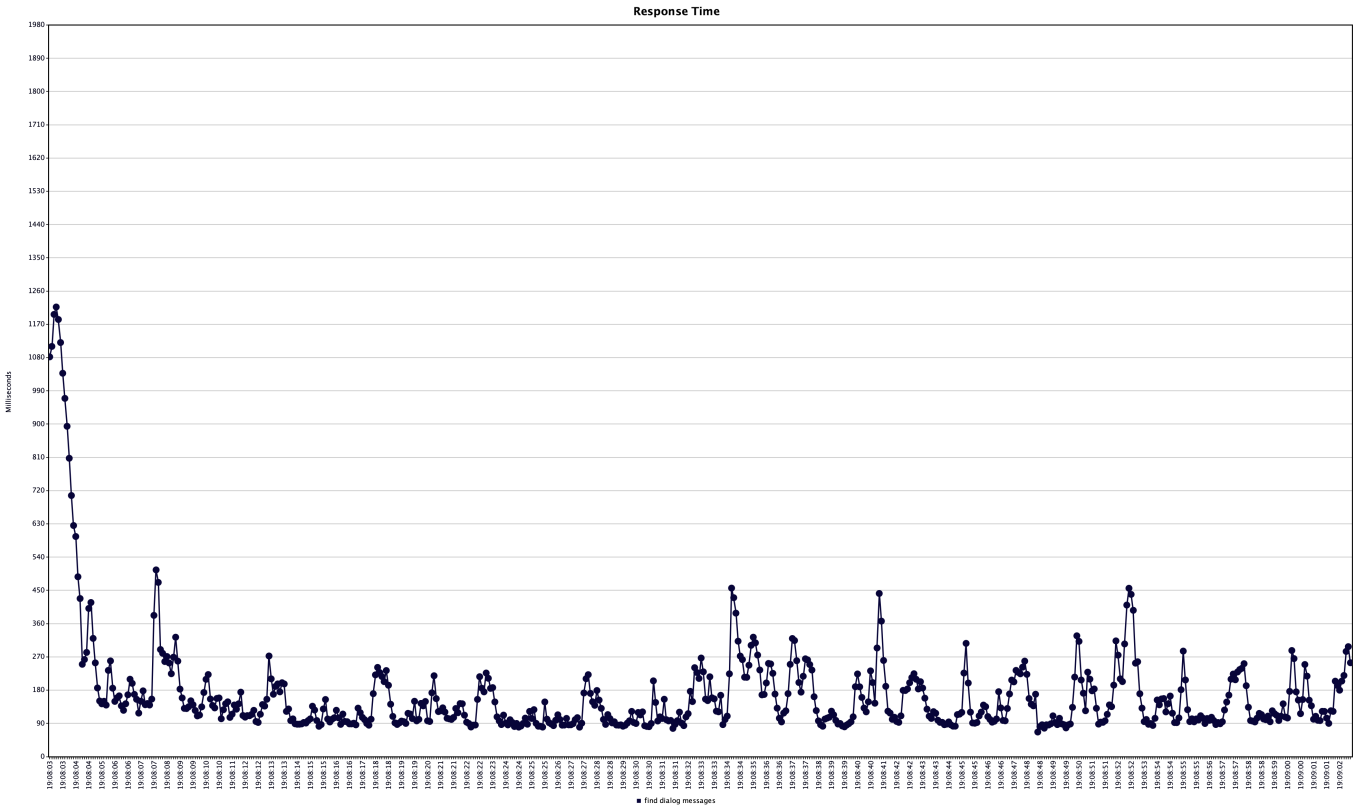


Время ответа сервиса:

- min: 10 мс
- max: 1 000 мс
  В начале нагрузки запросы выполняются менее чем за 50мс. Однако примерно через полминуты после подачи нагрузки случился мощный пик до 1 секунды. После этого запросы стали работать менее стабильно, появились более частые пики до 150-300 мс.

Чтение списка сообщений

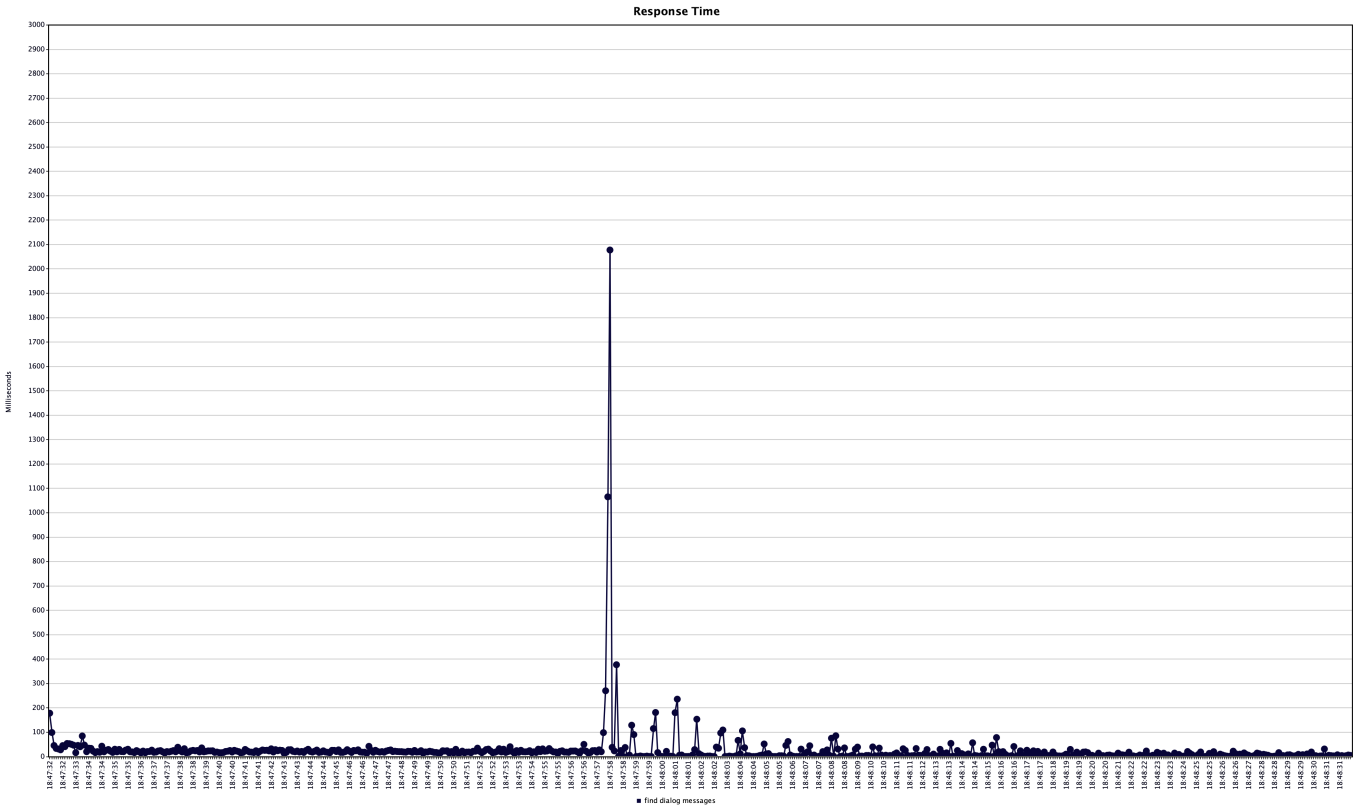
Postgres



Время ответа сервиса:

- min: 80 мс
- max: 1 200 мс
  Высокий пик в начале можно объяснить холодным стартом приложения – пул коннектов к БД ещё не был прогрет. Далее запросы выполнялись в диапазоне от 80 мс до 200 мс, с пиками до 500 мс.

Redis



Время ответа сервиса:

- min: 30 мс
- max: 2 100 мс
  Большинство запросов отрабатывало менее чем за 50 мс, что можно назвать очень хорошим результатом. Однако здесь наблюдается большой пик более чем в 2 секунды, с последующими более мелкими пиками. К концу нагрузки профиль стабилизировался и вернулся к значениям < 50 мс.

Сравнение

При сравнении профилей нагрузки Postgres показал большее время ответа и более плавающий профиль нагрузки: значения менялись волнами с большой амплитудой > 100% – время ответа могло увеличиваться более чем в 2 раза.

Redis показал более стабильный профиль нагрузки. При отсутствии проблем на ровных участках профиля наблюдаются скачки с амплитудой ~30-50%. Однако есть и очень высокие пики до 1-2 секунд. Postgres не деградировал до таких значений.

Проблемы при работе Redis

Судя по логам, в варианте с Redis были проблемы при работе с коннектами. Я использовал библиотеку Jedis, и полагаю, что базовые настройки пула коннектов JedisPool не позволяют поддерживать оптимальную производительность для такой нагрузки. Я добавил ретраи на обращения к Redis, что и привело к большим пикам – однако это позволило не терять сообщения и избавило от ошибок 500 internal server error.

Если в реальной системе допустимы ошибки в этих вызовах – можно перенести ретраи с сервера на клиент. Тогда сервер будет сразу отвечать 500-й ошибкой, и пики пропадут. Однако это не будет являться решением проблемы – нужно детальнее разбираться, почему возникает проблема с коннектами Jedis, и исправлять.

Вывод

В целом Redis как In-Memory хранилище показал более высокую производительность по сравнению с Postgres. Использование UDF позволяет выполнять всю логику внутри Redis, что в случае более сложных запросов (когда может потребоваться несколько раз обращаться к БД из приложения) позволяет сократить издержки на сетевые обращения, что в свою очередь положительным образом сказывается на производительности и доступности системы.