Федеральное государственное автономное образовательное учреждение высшего образования

«**Университет ИТМО»**

Факультет ПИиКТ

Дисциплина: Хранилища и базы данных

Лабораторная работа № 4

Выполнили:

Камышанская Ксения Васильевна

Гурин Евгений Иванович

Преподаватель: Королёва Юлия Александровна

Группа: P4116

Санкт-Петербург 2023г.

Цель работы: Нагрузочное тестирование базы данных

Задание

Развернуть несколько, например, три docker-контейнера как кластер, внутри каждого контейнера развернуть выбранную БД. Ограничить оперативную память 2 ГБ и память жёсткого диска 5 ГБ для каждого контейнера. Сымитировать «стрессовую» нагрузку, активно писать, а после запрашивать данные из БД. Фиксировать время выполнения запросов. Определить критический момент «падения» одного, двух и всех узлов нашего кластера. До «падения» последнего узла чтение должно быть корректным.

Ссылка на репозиторий https://github.com/GulDilin/itmo/tree/master/magistracy/storage-and-databases/load-tarantool-cluster

Выполнение

Для выполнения задания была выбрана БД Tarantool. Для реализации кластера Tarantool предлагает решение в виде Cartridge https://www.tarantool.io/en/doc/latest/book/cartridge/ . Данное решение является не просто кластером, но и содержит встроенные возможности для шардирования и создания веб-приложений на базе Tarantool. Такой подход является крайне перспективных и удобным для разработки хранилищ.

На базе WSL было установлено ПО cartridge-cli со всеми необходимыми зависимостями

```
sudo apt-get update
sudo apt-get install -y unzip cmake make git gcc
sudo apt-get install cartridge-cli
```

Далее было создано cartridge приложение

```
cartridge create --name cluster
cd cluster
cartridge build
```

Для настройки кластера используется набор конфигурационных файлов в формате yml и ряд скриптов и модулей на lua. Для инициализации необходимых настроек в репозитории создан ряд конфигураций и python скрипт для их установки в кластер.

Старт кластера

```
cartridge start
```

replicasets.yml – данный файл является конфигурацией наборов реплик. В данном случае создано 3 набора реплик (1 – роутер, который обеспечивает доступ к хранилищам и два хранилища). Для обеспечения возможности выполнения write команд в случае отказа мастер но все ноды были указаны как read write.

```
router:
  instances:
  - router
  roles:
  - failover-coordinator
  - vshard-router
  all_rw: false
s-1:
  instances:
  - s1-master
  - s1-replica
  roles:
  - vshard-storage
  - app.roles.storage
  weight: 1
  all_rw: true
  vshard_group: default
s-2:
  instances:
  - s2-master
  - s2-replica
  roles:
  - vshard-storage
  - app.roles.storage
  weight: 1
  all_rw: true
  vshard_group: default
```

Применение реплик (на работающем кластере)

```
cartridge start
```

В конфигурации реплик используются ссылки к объявленным в `instances.yml` экземплярам. Данный конфиг определяет 6 экземпляров (роутер, мастер 1, реплика 1, мастер 2, реплика 2 и экземпляр для перевыбора лидеров)

```
---
cluster.router:
  advertise_uri: localhost:3301
  http_port: 8081

cluster.s1-master:
  advertise_uri: localhost:3302
```

```
  http_port: 8082

cluster.s1-replica:
  advertise_uri: localhost:3303
  http_port: 8083

cluster.s2-master:
  advertise_uri: localhost:3304
  http_port: 8084

cluster.s2-replica:
  advertise_uri: localhost:3305
  http_port: 8085

cluster-stateboard:
  listen: localhost:4401
  password: passwd
```

Tarantool поддерживает автоматический перевыбор лидеров при отказах в процессе работы. Для этого используется `failover.yml`. В приведенной конфигруации используется статический перевыбор мастера на следующие реплики.

```
mode: stateful
state_provider: stateboard
stateboard_params:
   uri: localhost:4401
   password: passwd
failover_timeout: 15
```

Так как Tarantool использует шардирование для реплик, то просто подключиться к кластеру как к обычной базе не выйдет. Необходимо написать функции-обертки для операций и инициализировать схемы спейсов (аналог таблиц). Обертки для операций должны использовать id для выбора шарда и передавать нужному шарду вызов функции. Это должно происходить на уровне роутера.

```
router:
  instances:
  - router
  roles:
  - failover-coordinator
  - vshard-router
  all_rw: false
s-1:
  instances:
  - s1-master
  - s1-replica
  roles:
  - vshard-storage
  - app.roles.storage
  weight: 1
```

```
  all_rw: true
  vshard_group: default
s-2:
  instances:
  - s2-master
  - s2-replica
  roles:
  - vshard-storage
  - app.roles.storage
  weight: 1
  all_rw: true
  vshard_group: default
```

Функции-обертки операций над шардами (для Tarantool код пишется на языке lua). Были реализрованы функции добавления и чтения элемента.

```
local function customer_add(customer)
    local bucket_id = vshard.router.bucket_id(customer.customer_id)
    customer.bucket_id = bucket_id
    local _, error = err_vshard_router:pcall(
        vshard.router.call,
        bucket_id,
        'write',
        'customer_add',
        {customer}
    )
    return _, error
end


local function customer_lookup(customer_id)
    local bucket_id = vshard.router.bucket_id(customer_id)
    local customer, error = err_vshard_router:pcall(
        vshard.router.call,
        bucket_id,
        'read',
        'customer_lookup',
        {customer_id}
    )
    return customer, error
end
```

Часть кода модуля имитации нагрузки

```
class TarantoolClient:
    def __init__(self, spaces: dict, store_connection = False) -> None:
        self.connection = get_connection()
        self.spaces = spaces
        self.store_connection = store_connection

    def select_customer(self):
        idx = random_integer()
        try:
```

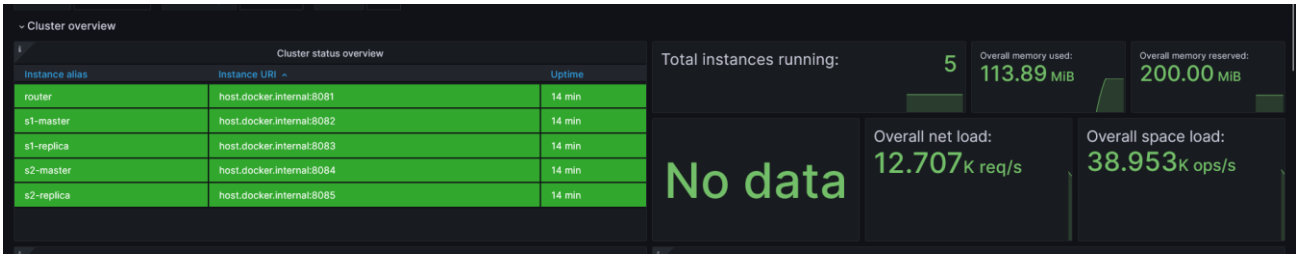
```
            if self.store_connection:
                selected = self.connection.call('customer_lookup', (idx,))
            else:
                selected = get_connection().call('customer_lookup', (idx,))
            print(f'{selected=}')
        except Exception as e:
            print(f'SELECT FAILED: {e}')

    def insert_customer(self):
        space_cols = self.spaces['customer']
        data = {
            key: generators[col_type]()
            for key, col_type in space_cols.items()
        }
        try:
            if self.store_connection:
                inserted = self.connection.call('customer_add', (data,))
            else:
                inserted = get_connection().call('customer_add', (data,))
            print(f'{inserted=}')
        except Exception as e:
            print(f'INSERT FAILED: {e}')

def thread_insert_3(store_connection):
    global is_running
    print('Start insert thread')
    t = TarantoolClient(config.spaces, store_connection)
    while is_running:
        time.sleep(0.001)
        t.insert_customer()


def thread_select_3(store_connection):
    global is_running
    print('Start select thread')
    t = TarantoolClient(config.spaces, store_connection)
    while is_running:
        time.sleep(0.001)
        t.select_customer()


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--threads',
        type=int,
        default=10,
        help="threads amount"
    )
    parser.add_argument(
        '--store-connection',
        default=False,
        action='store_true'
    )
    args = parser.parse_args()
    threads = []
```

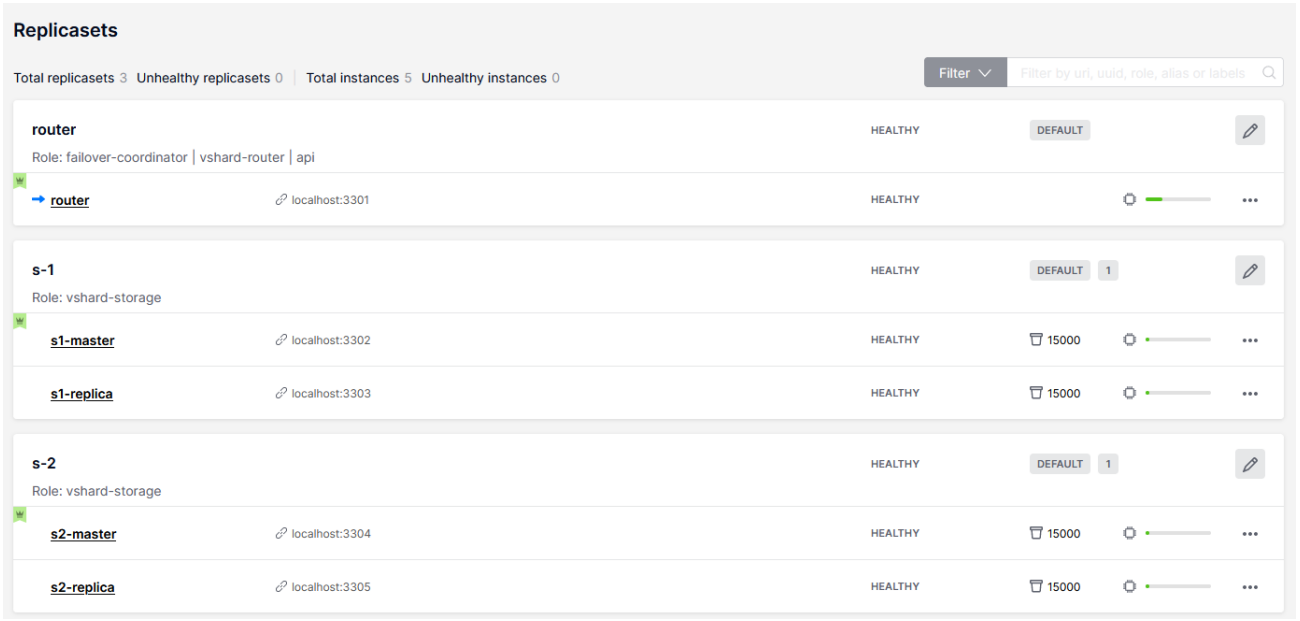
```
    for index in range(args.threads):

        if index > args.threads / 2:
            t = threading.Thread(target=thread_insert_3,
args=(args.store_connection,))
        else:
            t = threading.Thread(target=thread_select_3,
args=(args.store_connection,))
        threads.append(t)
        t.start()

    try:
        while True:
            time.sleep(100)
    except (KeyboardInterrupt, SystemExit):
        global is_running
        is_running = False
        print('Received keyboard interrupt, quitting threads.')

    for index, thread in enumerate(threads):
        thread.join()
        print(f"Main    : thread {index} done")
```

Tarantool поддерживает сбор метрик в формате Prometheus, поэтому был настроен мониторинг и дашборды на основе Grafana + Prometheus.

Нагрузочное тестирование

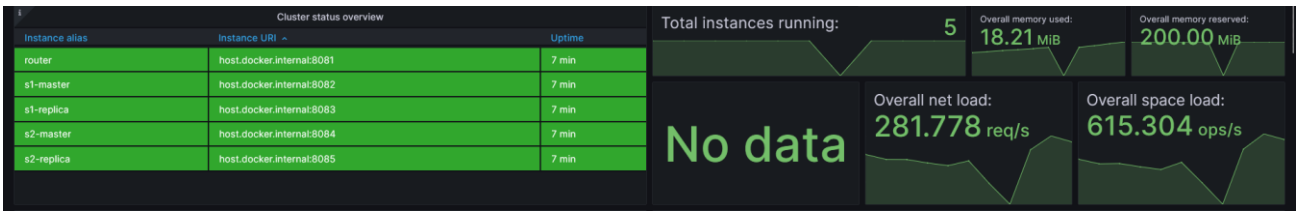
Tarantool оказался крайне живучей базой данных, которая спокойно справляется с 13 000 запросов в секунду. Поэтому для попыток убить базу было решено ограничить параметр `memtx_memory` на 40МБ на один экземпляр. И ограничили оперативную память, доступную WSL до 800Мб. Суммарно было запущено около 1000 потоков, получающих доступ к БД и выполняющих операции чтения-записи.





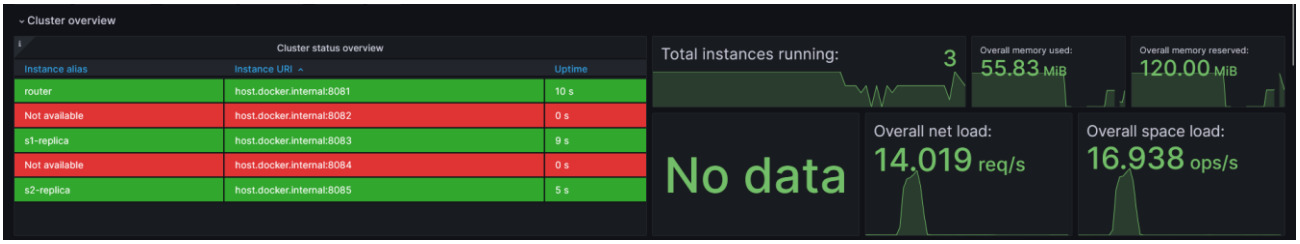
Падение роутера

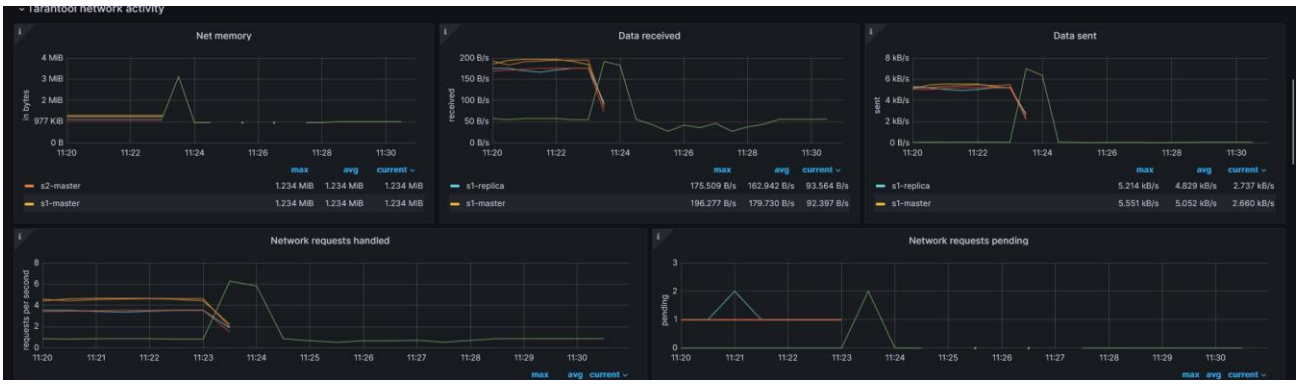
В определенный момент тестирования удалось добиться состояния, когда произошло падения экземляра роутера и все запросы к бд повисли в режиме ожидания. После этого было решено немного переконфигурировать кластер, чтобы роутер не выпонял никаких действий по записи\чтению, а только вызывал функции экземпляров хранилища.

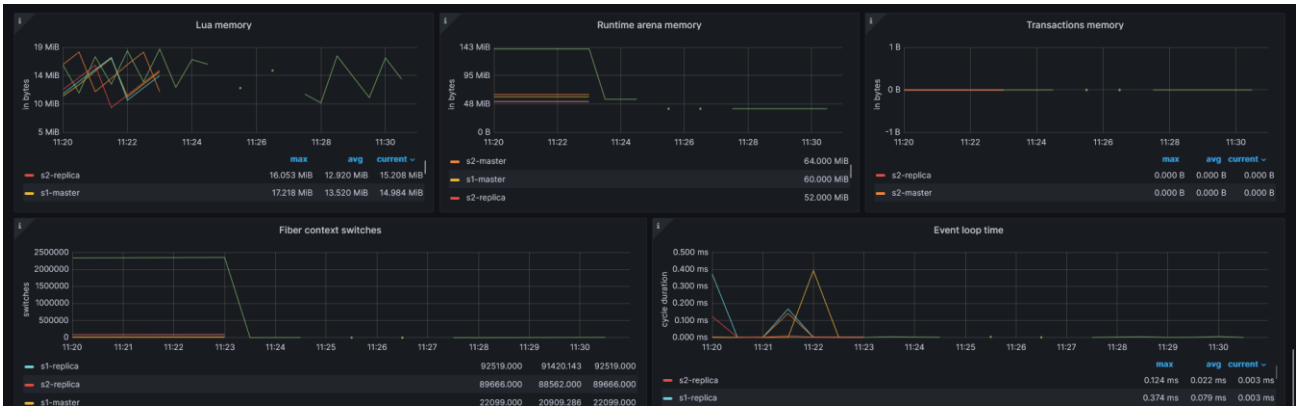


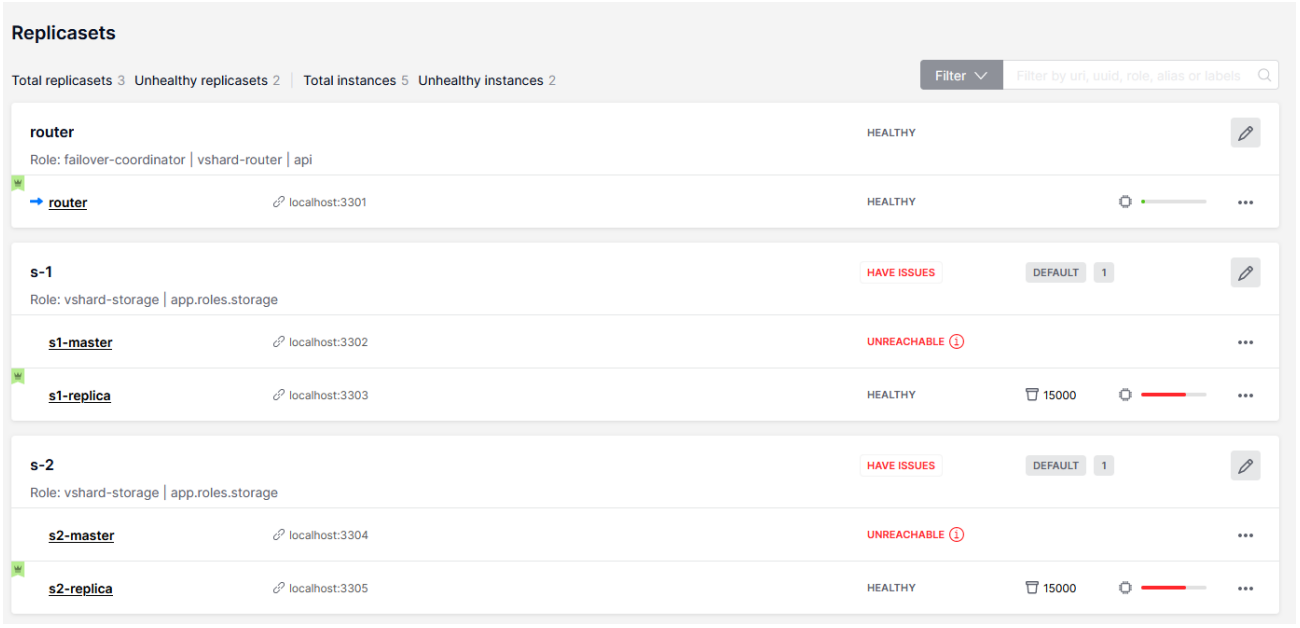
Падение мастер узлов сетов реплик

При достижении близкого к 40 МБ значения заполнения памяти на экземлярах начали наблюдаться проблемы с запуском мастер узлов и упало количество запросов, которые кластер может обработать за единицу времени.









Вывод

В ходе выполнения лабораторной работы были изучены подходы к разработке кластера на базе Tarantool. Был сконфигурирован кластер с использованием 5 экземпляров и роутером оберткой над двумя шардами. Было проведено нагрузочное, в результате которого мы пришли к мнению, что Tarantool крайне высокопроизводительная и отказоустойчивая БД, способная обрабатывать десятки тысяч запросов в секунду даже на серьёзных ограничениях по памяти.