## Отчет по лабораторной работе №3

Стенд, включая потсгресы,, приложение, приклад для нагрузки, моинторинг cadviser, прометеус и графана запускалось в докерах.

Исходные код и докер композеры:

<https://github.com/filatkinen/socialnet/tree/main/labs/lab03>

1. Создаем сеть, запоминаем адрес

docker network create socialnet

docker network inspect socialnet | grep Subnet

172.28.0.0/16

2.1 Определяем в docker-compose.yaml прометеус, графану и cadviser для мониторинга постгреса

2. Поднимаем мастер postgres

cd ~/dev/highload/socialnet/labs/lab03/postgres/p01

создаем docker-compose.yaml, в котором маппим папку с дампом в папку docker-entrypoint-initdb.d для postgres

Также в нем определяем postgres-exporter

docker-compose up

3. Меняем postgresql.conf на мастере

postgres01/postgresql.conf

ssl = off

wal\_level = replica

max\_wal\_senders = 4 # expected slave num

4.1 Устанавливаем репликацию в режим асинхронный - режим local(локальная запись WAL)

synchronous\_commit = local # synchronization level;

4. Подключаемся к мастеру и создаем пользователя для репликации

docker exec -it postgres01 psql -U postgres

create role replicator with login replication password 'pass';

6. Добавляем запись в pg\_hba.conf с ip с первого шага

host replication replicator 172.28.0.0/16 md5

7. Перезапустим мастера

docker restart postgres01

8. Сделаем бэкап для реплик

docker exec -it postgres01 bash

mkdir /pgslave

pg\_basebackup -h pgmaster -D /pgslave -U replicator -v -P --wal-method=stream

9. Копируем директорию себе

docker cp postgres01:/pgslave postgres02

10. Создадим файл, чтобы реплика узнала, что она реплика

touch postgres02/standby.signal

11. Меняем postgresql.conf на реплике

primary\_conninfo = 'host=postgres01 port=5432 user=replicator password=pass application\_name=postgres02'

12. Запускаем реплику

cd ~/dev/highload/socialnet/labs/lab03/postgres/p02

docker-compose up

12.1 Проводим нагрузочный тест: с мастера и с реплики. Результаты отражены в Приложении.

13. Запустим вторую реплику

docker cp postgres01:/pgslave postgres03

primary\_conninfo = 'host=postgres01 port=5432 user=replicator password=pass application\_name=postgres03'

touch postgres03/standby.signal

13.1 Уточним статус реплик:

docker exec -it postgres01 psql -U postgres

postgres=# select application\_name, sync\_state from pg\_stat\_replication;

application\_name | sync\_state

------------------+------------

postgres02 | async

postgres03 | async

(2 rows)

14. Включаем синхронную репликацию на мастере:

synchronous\_commit = on

synchronous\_standby\_names = 'FIRST 1 (postgres01, postgres02)'

docker exec -it postgres01 psql -U postgres

select pg\_reload\_conf();

postgres=# select application\_name, sync\_state from pg\_stat\_replication;

application\_name | sync\_state

------------------+------------

postgres02 | sync

postgres03 | async

(2 rows)

15. Создадим тестовую таблицу на мастере

docker exec -it postgres01 psql -U postgres

CREATE TABLE IF NOT EXISTS test

(

id SERIAL

CONSTRAINT test\_pk

PRIMARY KEY,

uid CHAR(36)

);

ALTER TABLE test

OWNER TO socialnet;

ALTER TABLE

15.1 запускаем нагрузку на запись

lab03/load : docker-compose up

15.2 Убиваем мастер:

docker exec -it postgres01 /bin/bash

killall -9 postgres

В логах контейнера с прикладом загрузки:

pq: the database system is in recovery mode

Last insert ID=125535, RowsCount=108121

Time to take: 5m43.43817654s

15.3 Проверяем что записалось в синх реплику и асинкреплику

docker exec -it postgres02 psql -U postgres

snet=# select id from test

ORDER BY id DESC

limit 1;

id

--------

125536

(1 row)

docker exec -it postgres03 psql -U postgres

snet=# select id from test

ORDER BY id DESC

limit 1;

id

--------

125536

(1 row)

И синхронная и асинхронная реплики успели записать изменения, хотя приклад вернул последнее значение на 1 меньше.

Но наиболее вероятное объяснение следующее: так как запрос, который пишет в базу идет с retuning\_id, то вероятно запрос записал и на полпути в момент возвращения записанного значения постгрес получил kill -9 и не успел вернуть.

16. Запромоутим реплику pgslave

docker stop postgres01

docker exec -it postgres02 psql -U postgres

select \* from pg\_promote();

synchronous\_commit = on

synchronous\_standby\_names = 'ANY 1 (pgmaster, pgasyncslave)'

17. Подключим вторую реплику к новому мастеру

primary\_conninfo = 'host=postgres02 port=5432 user=replicator password=pass application\_name=postgres03'

18. Смотрим статус репликации на новом мастере

select application\_name, sync\_state from pg\_stat\_replication;

application\_name | sync\_state

------------------+------------

postgres03 | quorum

(1 row)

19. Вставляем запись на мастере в таблицу

snet=# INSERT INTO test (uid) VALUES ('10');

INSERT 0 1

select \* from test where uid='10';

id | uid

--------+--------------------------------------

125560 | 10

(1 row)

20. Проверяяем что запись появлась на слейве:

select \* from test where uid='10';

id | uid

--------+--------------------------------------

125560 | 10

(1 row)

21. Что интересно, последнее значение serial, которое было добавлен прикладом 125536, а новый, который появился через запрос уже 125560, т.е. между ними разница в 64 значения. Почему неизвестно.

select id from test

ORDER BY id DESC

limit 10;

id

--------

125560

125536

125535

125534

125533

125532

125531

125530

125529

125528

(10 rows)

22. Если поднять первоначальный мастер(без реплик) и подключиться к нему, то запрос:

select id from test

ORDER BY id DESC

limit 10;

id

--------

125536

125535

125534

125533

125532

125531

125530

125529

125528

125527

(10 rows)

выдает последний id=125536

Вставим запись:INSERT INTO test (uid) VALUES ('10');

Видим, что следующий serial на 64 больше предыдущего перед падением.

snet=# select id from test

ORDER BY id DESC

limit 10;

id

--------

125560

125536

125535

125534

125533

125532

125531

125530

125529

125528

(10 rows)

**Выводы: В принципе все понятно, поведение с репликацией ожидаемо, но ситуация с перепрыгиванием поля serial ставит в тупик.**

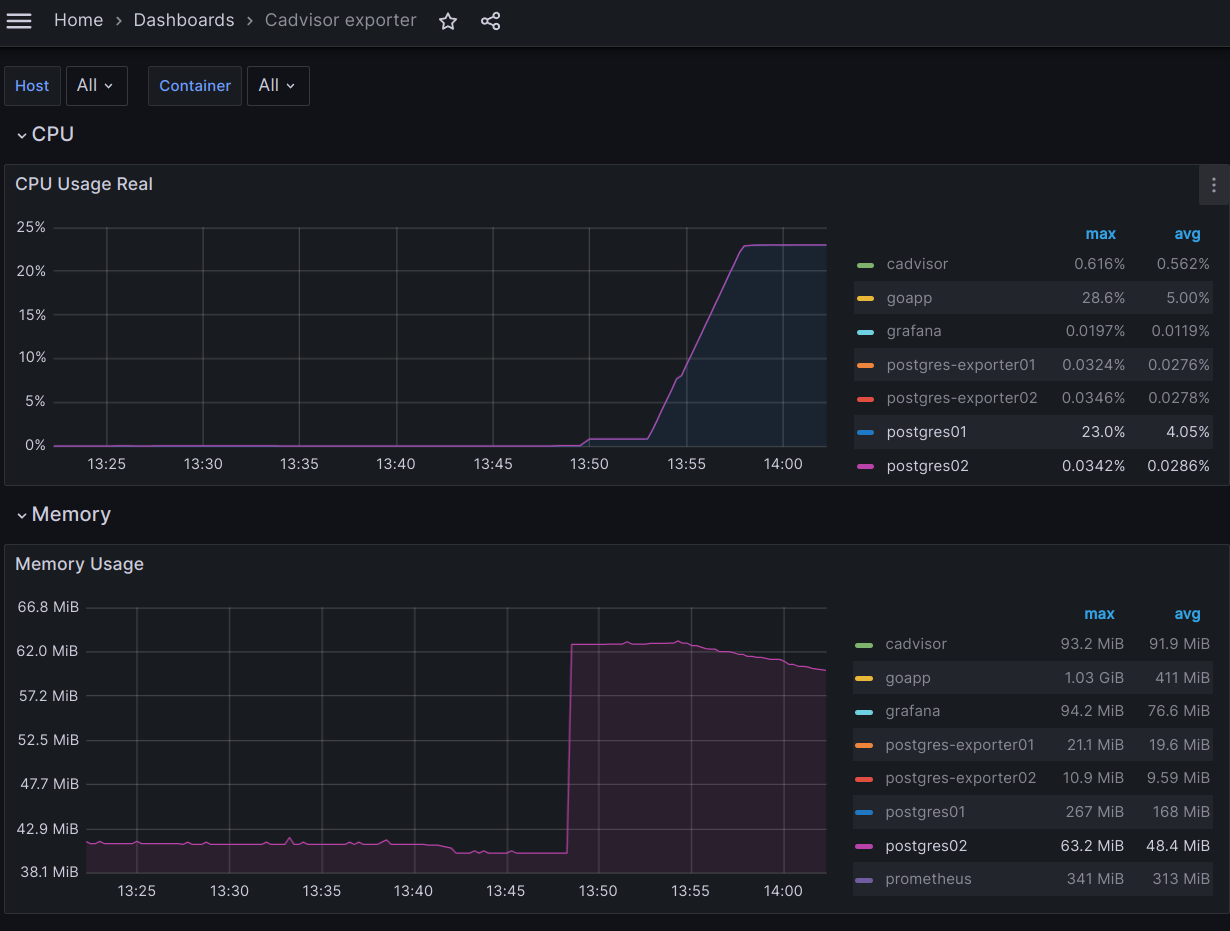
# Приложение 1 – Нагрузочный тест. Мастер и асинхронная реплика. Чтение с мастера

## Нагрузочный тест 2-ух методов с использованием hey:

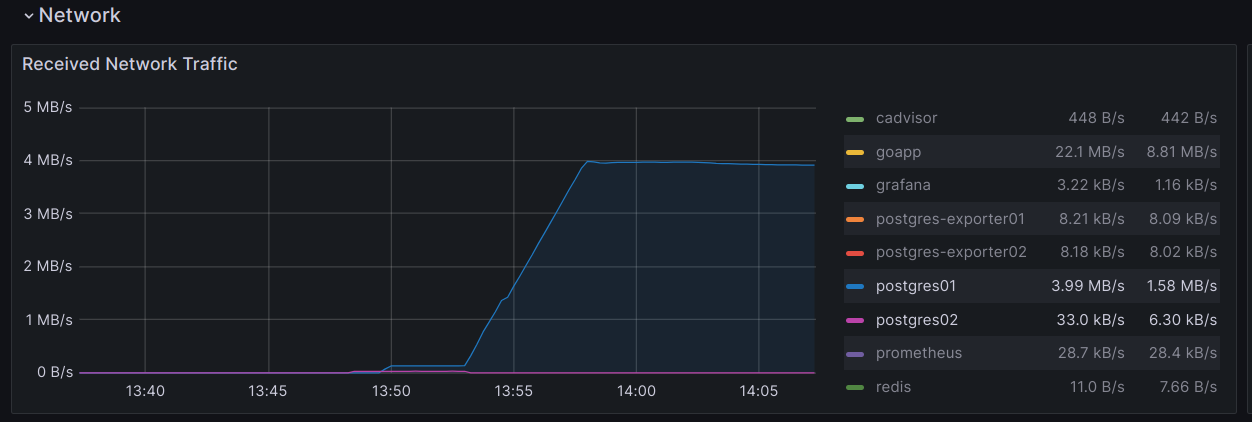
hey -z 20m -c 100 -m GET 'http://localhost:8800/user/search?first\_name=Иван&second\_name=Бес'

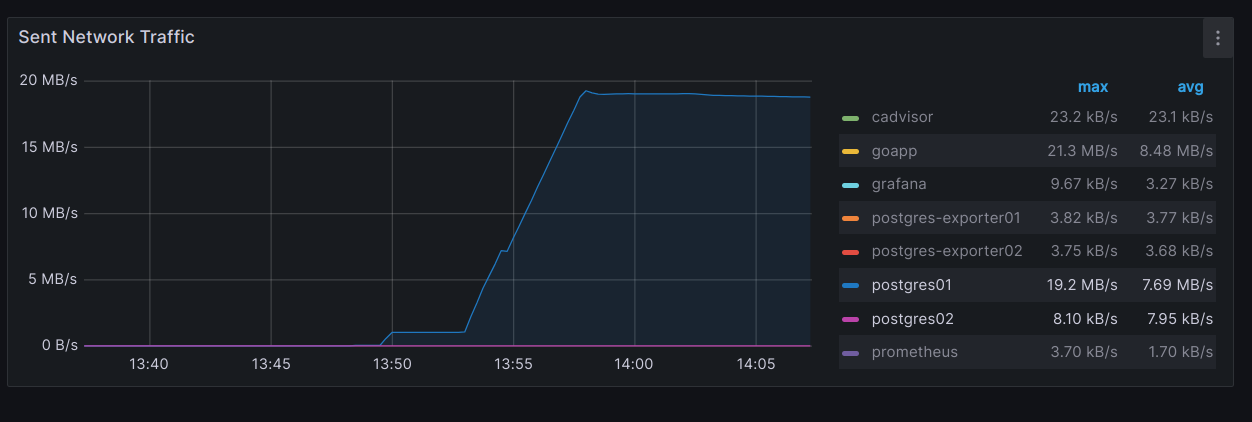
hey -z 20m -c 100 -m GET 'http://localhost:8800/user/get/5342fd18-e8fb-3bc1-39ab-b7132e963348'

## CPU и память

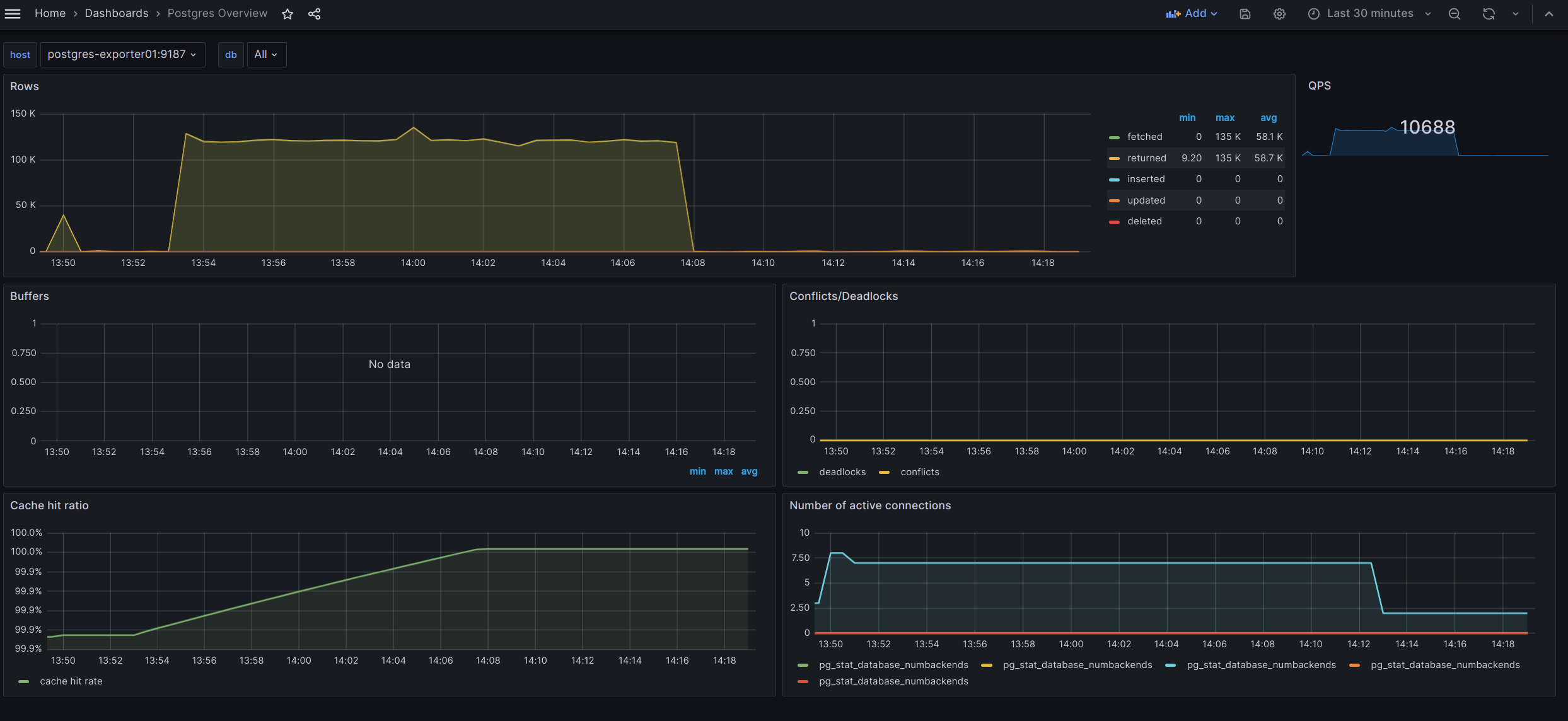


## Сеть:



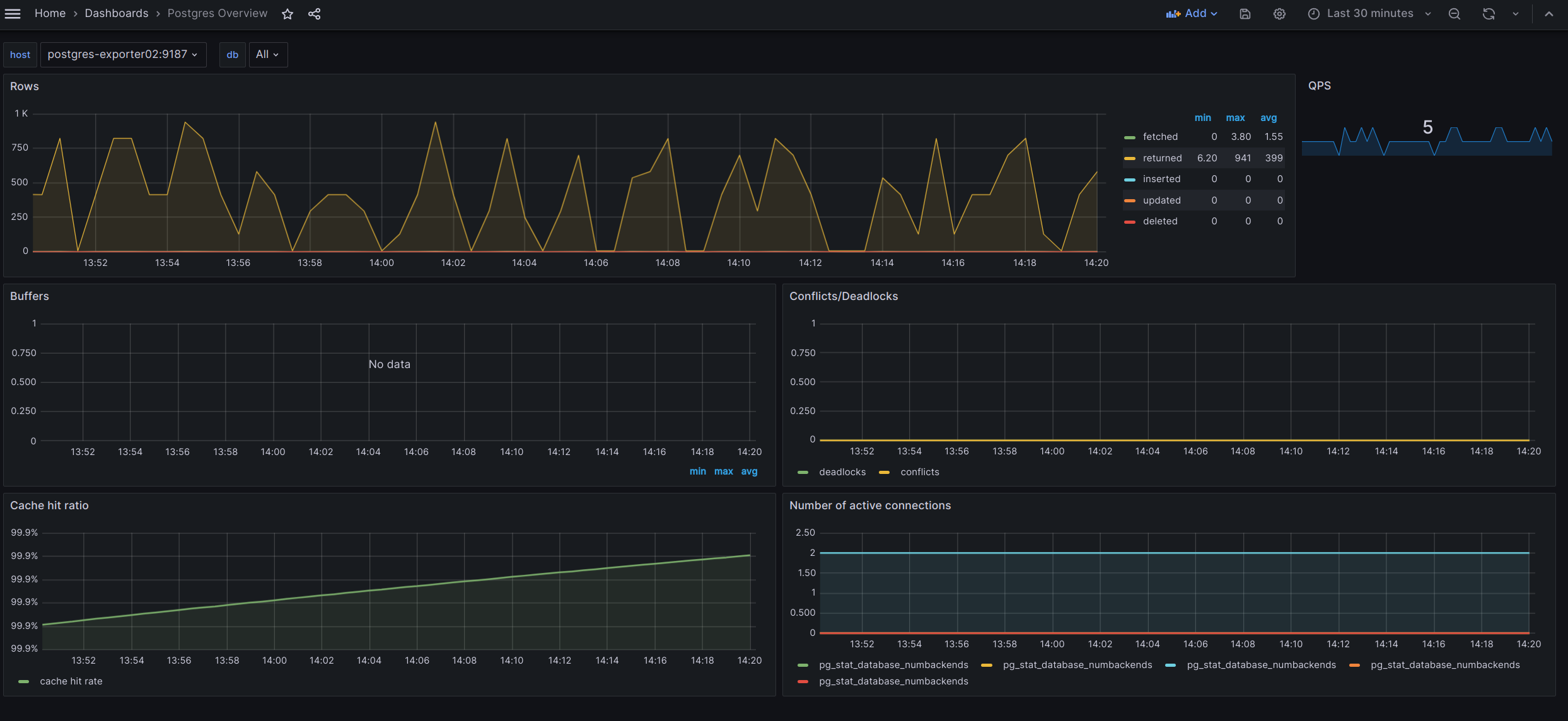


## Postgres статистика master



## Postgres статистика slave

небольшое количество запросов – это postgres\_exporter, фильтровать его через прометеус не нашел способа, так как лейбла в прометеус экспортер не передает



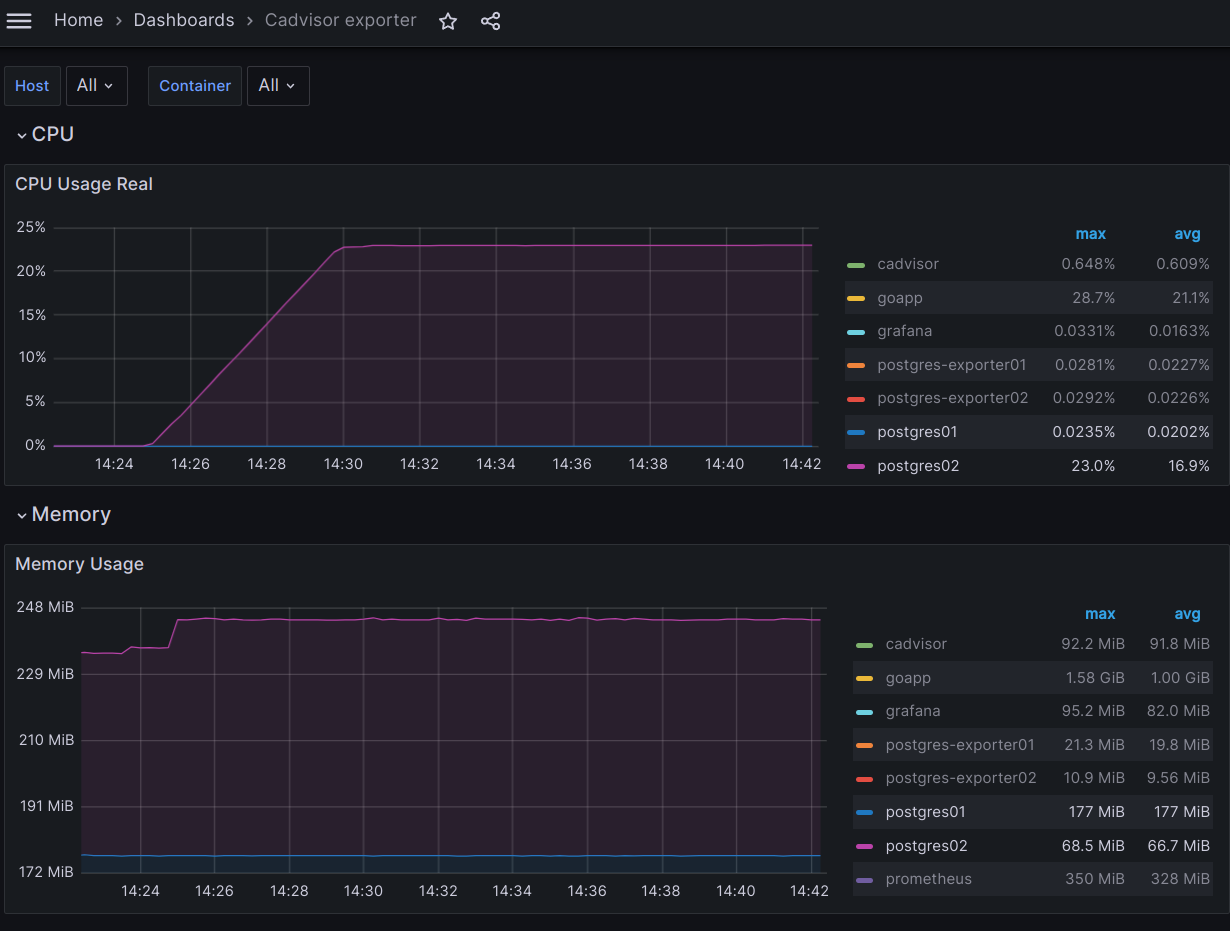
# Приложение 2 – Нагрузочный тест. Мастер и асинхронная реплика. Чтение с реплики

Нагрузочный тест 2-ух методов с использованием hey:

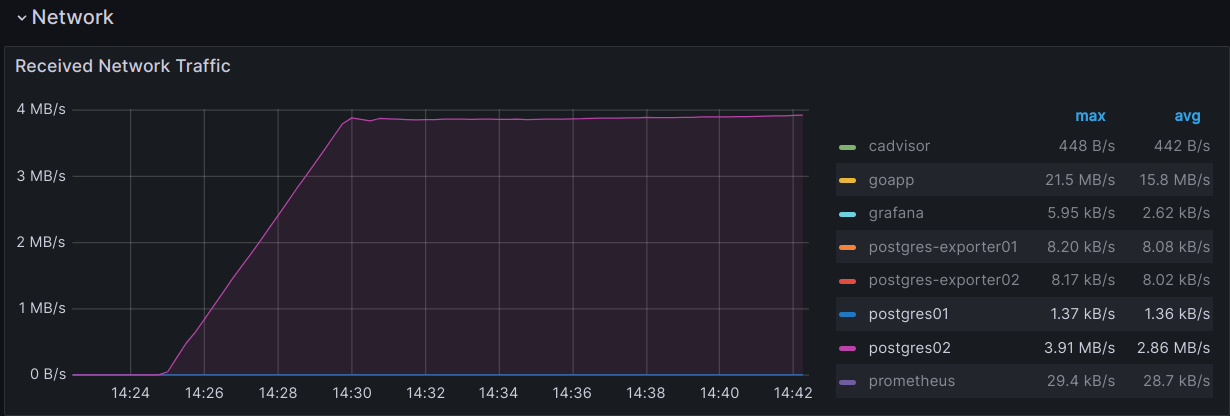
hey -z 20m -c 100 -m GET 'http://localhost:8800/user/search?first\_name=Иван&second\_name=Бес'

hey -z 20m -c 100 -m GET 'http://localhost:8800/user/get/5342fd18-e8fb-3bc1-39ab-b7132e963348'

## CPU и память

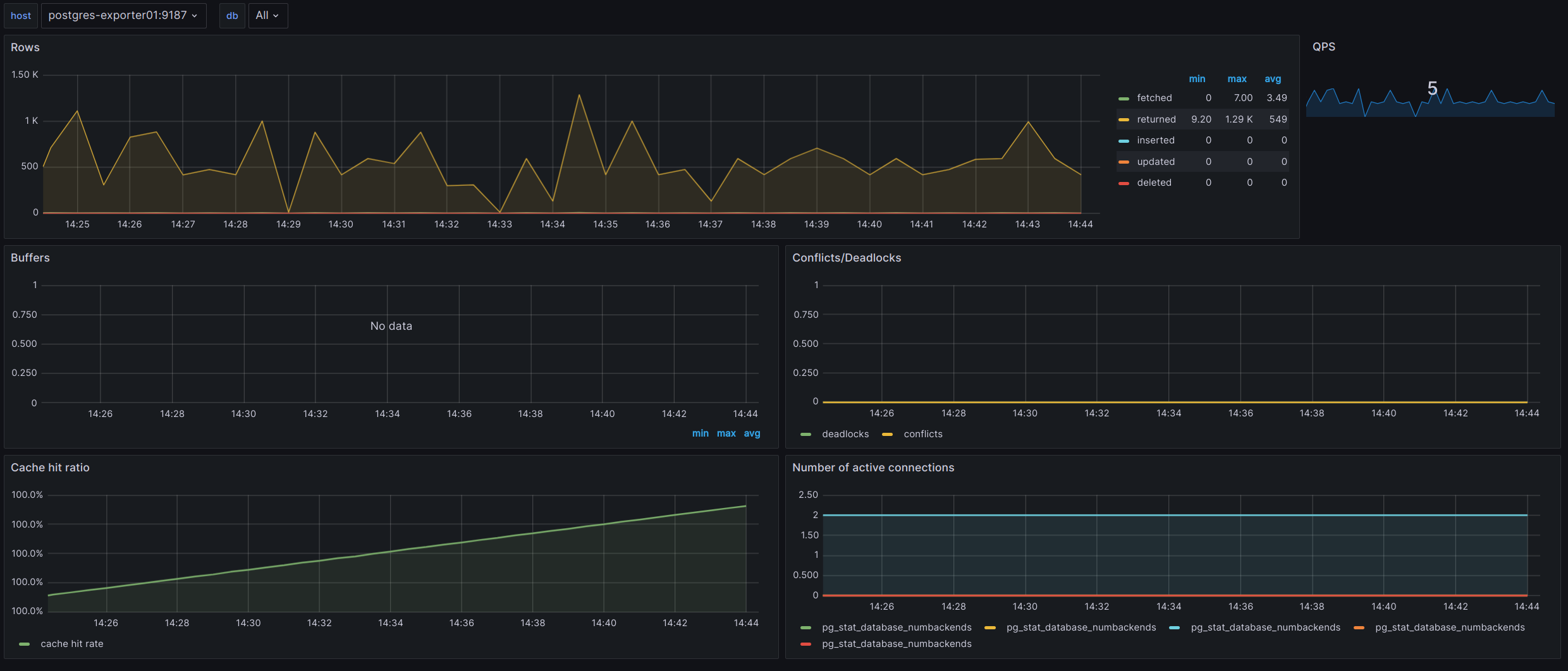


## Сеть



## Postgres статистика master

Здесь аналогично предыдущему тесту, запросы – это экпортер



## Postgres статистика slave

А вот здесь интересно: rps с реплики в 2 раза больше. Загадка.

