Министерство науки и образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н.Э.

Баумана

(национальный исследовательский университет)»

 (МГТУ им. Н.Э. Баумана)



**МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ**

**ПО КУРСУ «АДМИНИСТРИРОВАНИЕ БАЗЫ ДАННЫХ»**

**Лабораторная работа №3 «****Введение в мониторинг РСУБД.** **Пользовательские метрики»**

Авторы:

Кудрявцев А.П., kudryavtsevap@bmstu.ru

Фомин М.М.

Москва, 2023

## Лабораторная работа №3. Введение в мониторинг РСУБД. Пользовательские метрики

Данная лабораторная состоит из 3 взаимосвязанных частей, в которых излагаются основы формирования пользовательских метрик для визуализации в графической оболочке Grafana. После теоретической части студентам предлагаются задания для самостоятельного выполнения в рамках лабораторной работы.

**Цель:**

Данная лабораторная работа призвана сформировать у студента навыки мониторинга и использования статистики, предоставляемой РСУБД PostgreSQL.

**Задачи:**

* Изучить язык разметок YAML.
* Ознакомиться с алгоритмом формирования пользовательских метрик.
* Продолжить изучение практических основ организации мониторинга с помощью Grafana и Prometheus.

1. Пользовательский набор метрик

Для описания пользовательского набора метрик используется язык разметки yaml. Начнем с краткого ознакомлением с этим языком.

1. 1. YAML

YAML (Yet Another Markup Language) — это ещё один язык разметки. Доминанта сместилась с упора на удобную разметку в сторону работы с данными, при этом удобный читаемый формат, конечно же, сохранился. Благодаря такому решению, YAML со временем стал одним из самых популярных языков разметки.

Как было отмечено выше, YAML имеет простой человекочитаемый синтаксис — это его основной плюс и одновременно минус, так как такой свободный синтаксис способствует большому количеству ошибок. Валидаторы YAML-файлов помогают избежать ошибок.

Для Linux можно воспользоваться VS Code, установив расширение для yaml файлов. Он наглядно выводит отчет об ошибках. Либо для валидации использовать встроенную библиотеку Python — yaml.

Синтаксис YAML очень прост, достаточно запомнить несколько простых правил.

* + 1. Структура YAML

Каждый документ в потоке yaml должен быть разделен с помощью трех тире (---) и трех точек (…). Три тире (---) указывают на начало документа, а три точки (…) – на конец документа в потоке. В случае отсутствия нескольких документов нет необходимости использовать тире и точки.

В синтаксисе YAML значительным является интервал. Каждый отступ с двумя пробелами представляет новый уровень:

level1:

 level2:

 level3:

Каждый новый уровень — это объект. В этом примере объект level1 содержит объект level2, который содержит объект level3.

В YAML обычно не используется клавиша Tab (так как интервал между вкладками нестандартен). Вместо этого дважды нажимается пробел. В вашем любимом редакторе посмотрите в настройках возможность заменить Tab на пробелы. Например, в VS Code при нажатии Tab в текстовом редакторе при работе с yaml, он оставляет только два пробела.

1.1.2. Комментарии YAML

Комментарии дают лучшее понимание тому, кто читает yaml-файл. Чтобы добавить комментарии в yaml-файл, используйте символ #. Yaml не поддерживает многострочные комментарии. Поэтому, если вы хотите добавить многострочные комментарии, вы должны начинать каждую строку символом #.

1.1.3. Тип строк YAML

Строки в yaml могут быть созданы как с кавычками, так и без кавычек. Yaml достаточно умен, чтобы интерпретировать тип данных внутри. Здесь ключ имеет строковый тип, и значение также задано строковым типом. Важные моменты, который следует запомнить.

* Создавайте строки без использования кавычек, если в этом нет необходимости.
* Используйте двойные кавычки, если строка содержит специальные символы.
* Используйте одинарные кавычки, если специальные символы должны интерпретироваться как литералы.

User1\_review: Linux отлично подходит для работы

User2\_review: "Linux \t отлично подходит для работы"

User3\_review: 'Linux \t отлично подходит для работы'

Yaml по умолчанию знает тип данных. Существует также явный способ указать этот тип. Используйте символ !!, за которым следует тип данных и строковое значение.

User1\_review: !!str C Linux очень удобно работать

Если у вас есть длинная строка, но вы хотите записать ее в несколько строк внутри вашего yaml-файла, вы можете сделать это с помощью символа >. Это называется **folded style**. Ваш анализатор будет интерпретировать текст как одну строку, хотя вы написали его в несколько строк.

User4\_review: >

 Среди всех дистрибутивов

 я использовал

 Linux. Он лучший.

Для того, чтобы удалить пробелы в конце строки можно использовать символ «-» после «>» или «**|**».

User4\_review: >-

 Среди всех дистрибутивов

 я использовал

 Linux. Он лучший.

User5\_review: |-

 Среди всех дистрибутивов

 я использовал

 Linux. Он лучший.

Если вы хотите сохранить «белое» пространство, вы можете добавить символ «+» после символов «|» или «>».

User4\_review: >+

 Среди всех дистрибутивов

 я использовал

 Linux. Он лучший.

User5\_review: |+

 Среди всех дистрибутивов

 я использовал

 Linux. Он лучший.

1.1.4. Числовой тип YAML

Yaml поддерживает числовые типы Integer, Float, Decimal или Hexadecimal. По умолчанию парсер yaml определяет тип данных, но есть и явный способ определения типов данных int и float, как показано в примере ниже.

int1: 98765

int2: !!int 56789 # Явное определение Integer

float1: 20.0481

float2: !!float 20.0482 # Явное определение Float

Шестнадцатеричные и восьмеричные значения будут преобразованы интерпретатором в десятичные.

hex1: 0x14d3

oct1: 014442

1.1.5. Логический тип YAML

YAML поддерживает булевы значения «True» и «False». Вы также можете использовать «Yes» или «on», что указывает на «True», и «no» или «off», что указывает на «false». Вы также можете явно определить тип данных, используя !!bool. Для булевых значений не существует ограничений по регистру.

В приведенном ниже примере видно, что во многих случаях написано «True», и синтаксический анализатор интерпретирует все как значение «True».

upgrade: True

Reboot\_After\_Upgrade: TRUE

Enable\_Firewall: on

Set\_Power\_Profile: yes

Аналогичное условие применимо и к значению «False».

upgrade: False

Reboot\_After\_Upgrade: FALSE

Enable\_Firewall: off

Set\_Power\_Profile: no

1.1.6. YAML Null

Чтобы сделать значение нулевым, можно использовать символ “~” или ключевое слово “null”. Вы также можете определить ключ и оставить значение пустым, которое будет рассматриваться как null. Вы также можете сделать явные определения, используя !!null. Как и для boolean, для ключевого слова “null” не существует ограничений по регистру.

В приведенном ниже примере ключевое слово null записано в трех разных случаях.

upgrade: !!null null

Reboot\_After\_Upgrade: NULL

Enable\_Firewall: Null

Set\_Power\_Profile: ~

Set\_Network\_Interface: # значения не переданы

1.1.7. YAML Последовательность

Последовательность YAML – это список значений, хранящихся по порядку. Существует несколько способов определения последовательности в yaml.

Первый – это стиль in-flow, когда вы задаете имя ключа, за которым следует список значений в квадратных скобках.

# IN-FLOW СТИЛЬ

app\_to\_be\_updated: [ "firefox", "timeshift"]

Второй способ – создание последовательности с помощью блочного стиля. Каждый элемент в последовательности будет иметь префикс тире, за которым следует пробел и значение элемента. Каждый элемент должен быть написан в отдельной строке. Вы можете либо сделать отступ в два пробела, либо создать последовательность без отступа. Но в качестве лучшей практики придерживайтесь отступа.

# БЛОКОВЫЙ СТИЛЬ

app\_to\_be\_installed:

 - vscode

 - virtualbox

 - tilix

app\_to\_be\_removed:

- pycharm

- stacer

- ufw

Также можно создать вложенную последовательность, как показано ниже.

# ВЛОЖЕННАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ

applications:

 - Productivity:

 - vscode

 - vagrant

 - docker

 - python3

 - Browser:

 - firefox

 - chrome

 - brave

1.1.8. Тип словарь YAML

Словарь – это пара ключ-значение, которые группируются под одним ключом. Они позволяют структурировать данные по логическим категориям. Ключи словаря всегда строкового типа, а значения могут быть любого скалярного типа. Подобно последовательности, словарь также может быть записан несколькими способами.

Первый – это стиль in-flow.

application: { Install: "Vscode", Remove: Stacer, Update: Firefox}

Вы также можете создать список вложенных словарей.

application2:

 Install:

 python: 3.9

 Vscode: 1.58.2

 Remove:

 - Stacer

 - pycharm

 Update: Firefox

1.1.9. Анкоры и алиасы YAML

Вы избежать повторений в вашем yaml-файле с помощью анкоров и алиасов. Анкор обозначается с помощью символа “&”, а алиас – с помощью символа “\*”. Анкор похож на переменную в программировании. Вы определяете анкор с помощью символа &, за которым следует ключевое слово, а затем используете алиас (ключевое слово \*) для расширения значения анкор.

Посмотрите на следующий пример. &x определяется как анкор, а затем расширяется с помощью алиаса \*x.

User4\_review: &x Среди всех дистрибутивов я использовал Linux.
 Он лучший.

User5\_review: \*x

User6\_review: \*x

При использовании анкоров и алиасов можно переопределять конкретные значения с помощью символа <<. Здесь я переопределяю значение версии с 21.10 на 21.04.

Linux: &pos

 version: 21.10

 code\_name: cosmic desktop

 d\_flavour: gnome

rewrite:

 <<: \*pos

 version: 21.04

1.2. Пользовательские метрик

В дополнение или взамен метрик по умолчанию можно использовать собственные. Для этого достаточно задать путь к файлу с пользовательскими запросами **queries.yaml**, через аргумент --extend.query-path=queries.yaml или через переменную окружения PG\_EXPORTER\_EXTEND\_QUERY\_PATH. Это - вкратце. А в действительности, путь состоит из несколько шагов.

1.2.1. Подготовительная работа

Для начала определимся какие метрики хотим обрабатывать. Остановимся на выявлении запросов, на которые было использовано больше всего процессорного времени. Эти данные поставляет представление **pg\_stat\_statements**. Это представление содержит отдельные строки для каждой комбинации идентификатора базы данных, идентификатора пользователя, идентификатора запроса. Описание полей можно посмотреть в документации <https://postgrespro.ru/docs/postgrespro/14/pgstatstatements>.

Ниже приведен запрос, который позволяет выявить наиболее затратные пользовательские запросы, без учета количества вызовов.

SELECT substring(st.query, 1, 100) AS short\_query,

 to\_hex(st.queryid),

 round(st.mean\_exec\_time::numeric, 2) AS mean\_time,

 st.calls,

 round((100\*st.mean\_exec\_time/sum(st.mean\_exec\_time::numeric) OVER ())::numeric, 2) AS percentage\_cpu,

 db.datname AS datname

FROM pg\_stat\_statements AS st

 JOIN pg\_database AS db ON db.oid = st.dbid

ORDER BY st.mean\_exec\_time DESC

LIMIT 10;

Этот запрос более подходит для детального анализа, а так как для отображения метрик потребуются не все поля, поэтому его сократим и оставим то, что необходимо.

SELECT to\_hex(st.queryid) || ' ' || substring(st.query, 1, 50) AS short\_query,

 round((100\*st.mean\_exec\_time/sum(st.mean\_exec\_time::numeric) OVER ())::numeric, 2) AS percentage\_cpu,

 db.datname AS datname

FROM pg\_stat\_statements AS st

 JOIN pg\_database AS db ON db.oid = st.dbid

ORDER BY st.mean\_exec\_time DESC

LIMIT 10

В метриках необходима однозначность, основанная на значениях, которые принимают метки (Label), в нашем случае пара **short\_query** и **datname** должны обеспечить уникальность строки. Поэтому перед текстом пользовательского запроса вставлен ID этого запроса. Можно в качестве метки использовать ID, но в этом случае понижается информативность (ниже мы увидим, как это выглядит).

 Для того, чтобы запрос работал, необходимо установить расширение для каждой БД.

CREATE EXTENSION pg\_stat\_statements;

А в файле **postgresql.conf** добавить **pg\_stat\_statements** в **shared\_preload\_libraries**. Для этого откроем терминал и войдем от имени пользователя root:

alex@alex:~$ sudo -i

[sudo] password for alex:

root@alex:~#

Найдем файл конфигурации. Если вы знаете, где он находится, то не спешите набирать путь, легче скопировать:

root@alex:~# find / -name "postgresql.conf"

etc/postgresql/14/main/postgresql.conf

Отроем этот файл с помощью редактора, в моем случае это - nano.

root@alex:~# nano /etc/postgresql/14/main/postgresql.conf

Находим **shared\_preload\_libraries** и отредактируем строку следующим образом:

shared\_preload\_libraries = 'pg\_stat\_statements' # (change requires restart)

Сохраняем изменения и выходим из редактора. Перезапустим PostgreSQL:

root@alex:~# systemctl restart postgresql

Проверим

root@alex:~# systemctl status postgresql

● postgresql.service - PostgreSQL RDBMS

 Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor preset: enabled)

 Active: **active (exited)** since Fri 2023-12-15 09:06:21 MSK; 7h ago

 Process: 1581 ExecStart=/bin/true (code=exited, status=0/SUCCESS)

 Main PID: 1581 (code=exited, status=0/SUCCESS)

 CPU: 826us

дек 15 09:06:21 alex systemd[1]: Starting PostgreSQL RDBMS...

дек 15 09:06:21 alex systemd[1]: Finished PostgreSQL RDBMS.

root@alex:~#

Служба активна. Остается убедиться, что подготовленный запрос работает.

Для завершения работа с терминалом выполните команды:

root@alex:~# logout

alex@alex:~$ exit

1.2.2. Создание файла пользовательских метрик

Не будем оригинальными и создадим файл с названием **queries.yaml** в каталоге **/opt/postgres\_exporter-0.10.1.linux-amd64/**. Это можно сделать, запустив терминал. Создаем файл, используя встроенный редактор, например nano, и вводим скрипт. Но, к сожалению, для файлов YAML выдвинуты жесткие требования в части форматирования и, если они не будит выполнены, то postgres-exporter отклонит файл **queries.yaml** и на странице <http://localhost:9187/metrics> можно будет найти следующее сообщение об ошибке:

# HELP pg\_exporter\_user\_queries\_load\_error Whether the user queries file was loaded and parsed successfully (**1 for error**, 0 for success).

# TYPE pg\_exporter\_user\_queries\_load\_error gauge

pg\_exporter\_user\_queries\_load\_error{filename="/opt/postgres\_exporter-0.10.1.linux-amd64/queries.yaml",hashsum="6764328d17cc475803de721eb2cf0f9d8c0c8c863fa5b0797e920f5f5cc2e78c"} **1**

Есть другой способ выполнить задуманное. Откроем VSCode и в нижней части окна по умолчанию располагается вкладка «терминал». Если ее нет, то надо вызвать команду главного меню «Терминал/Создать терминал». В терминале войдем от имени пользователя root и выполним команду создания файла:

* получаем права пользователя root:
alex@alex:~$ sudo -i
* вводим пароль:
[sudo] password for alex:
* создаем файл queries.yaml:
root@alex:~# touch /opt/postgres\_exporter-0.10.1.linux-amd64/queries.yaml

Откроем созданный файл. Установим указатель мышки на файл в команде touch, появится подсказка – «открыть файл в редакторе (CTRL+щелчок)», и следуя подсказке открываем файл. И чтобы не возвращаться к особенностям работы с VSCode, рассмотрим, как сохранить файл. Выполним команду «Файл/Сохранить». Появится окно с предупреждением:



Нажимаем на кнопку «Повторит попытку …», после нам предложат ввести пароль и файл сохраниться.

 Рассмотрим содержимое файла.

**pg\_stat\_statements**:

 **query**: SELECT round((100 \* st.mean\_exec\_time/sum(st.mean\_exec\_time::numeric) OVER ())::numeric, 2) AS **percentage\_cpu**,

 to\_hex(st.queryid) || ' ' || substring(st.query, 1, 50) AS **short\_query**,

 db.datname AS **datname**,

 st.userid AS **userid**

 FROM pg\_stat\_statements AS st

 JOIN pg\_database AS db ON db.oid = st.dbid

 ORDER BY st.mean\_exec\_time DESC

 LIMIT 10

 **master**: true

 **cache\_seconds**: 30

 **metrics**:

 - **datname**:

 **usage**: "LABEL"

 **description**: "Название базы данных"

 - **short\_query**:

 **usage**: "LABEL"

 **description**: "SQL запрос"

 - **userid**:

 usage: "LABEL"

 description: "ID пользователя"

 - **percentage**\_cpu:

 **usage**: "GAUGE"

 **description**: "Показывает проценты среднего времени, потраченного на один запрос"

Метки и значения из примера выше:

* pg\_stat\_statements — префикс метрики, возвращаемый запросом, желательно, чтобы он совпадал с названием представления;
* query — содержит SQL-запрос;
* master — выполнять запрос только в базе данных, указанной при подключении в URI (мастер-БД). Необходим при запуске экспортера с флагом --auto-discover-databases. Принимает значения true или false. По умолчанию false. (не обязательное);
* cache\_seconds — время, в течении которого будут возвращаться данные кэша. Задаётся в секундах;
* metrics — содержит список меток и метрик;
* datname, short\_query, userid, percentage\_cpu — элементы списка. Имена элементов списка должны совпадать с именами колонки в запросе;
* usage — тип значения. Принимает COUNTER, GAUGE, LABLE;
* description — описание метрики.

Отдельно отметим, что в файл **queries.yaml** можно внести несколько метрик с со своими запросами.

После сохранения файла перезапустим службу postgres\_exporter. Сделаем это в терминале VSCode.

root@alex:~# systemctl restart postgres\_exporter

Проверим.

root@alex:~# systemctl status postgres\_exporter

Если служба запустилась, откроем в браузере метрики — <http://localhost:9187/metrics>. Находим метрики.

Не забудьте отказаться от прав пользователя root:

root@alex:~# logout



1.2.3. Причины отсутствия метрик в браузере

Если наши метрики не появились, возможно причина в следующем:

* браузер с метриками был уже открыт, обновите страницу;
* проверить, что нет ошибок в запросе, скопируйте SELECT в клиент PostgreSQL, например pgAdmin, и запустите;
* проверить форматирование файла **queries.yaml**;
* проверить установку расширения **pg\_stat\_statements** для всех баз данных, запустив наш запрос в каждой из них. По умолчанию, выполняется подключение к БД Postgres (если эта БД не удалена) и, если там не установлено расширение, то метрики не появятся;
* проверьте статус службы **postgres\_exporter.**

## 2. Визуализация метрик

Создадим новый Dashboard. Напомним.

* Откройте страницу в браузере <http://localhost:3000/>.
* Выберите New dashboard. + Add visualization и т.д. В итоге попадем на страницу редактирования нового Dashboard.
* Назовем его «Проценты общего времени, затраченного на SQL запрос». Для этого в поле «Title» панели «Panel options» введем заголовок.
* В поле «Metric» выберем нашу метрику **pg\_stat\_statements\_percentage\_cpu**.
* Нажмем на кнопку Run queries.

При анализе графиков неплохо было бы выбирать те метрики, которые относятся к конкретной БД. Такая возможность имеется. Будем заполнять поля «Label filters».



Нажав на «Select label», выберем метку **datname.** Напомним, что этой метке присваивается название БД. Далее в место строго равенства выберем регулярное выражение «=~». И наконец, нажимая на «Select value», выбираем «**$datname**». Поясним, вместо **$datname** будет подставляться конкретное значение – название БД.



Сохраним Dashboard и применим (Apply). Найдем в правом верхнем углу шестеренку (Dashboard settings) и нажмем. Откроется панель настроек



Щелкнем по «Variables», а потом на кнопку «+ New variable». После заполнения необходимых полей (см. рис.) у нас в Dashboard появится в левом верхнем углу Combobox с меткой «База данных», позволяющий выбрать конкретную БД при просмотре графиков.







Сохраним, нажав на Apply. Перейдем на страницу Dashboard.

Нагрузим БД demo запросами и посмотрим на наши метрики.



Как видим все работает. Но есть замечания – «легенды» слишком много занимают места. Нельзя ли их укоротить, ведь начало текста запросов видны в tooltip’е? Эту возможность оставим для самостоятельного изучения.

И еще, после изучения и исправления «плохих» запросов надо все начинать с чистого листа. Для этого есть функция

pg\_stat\_statements\_reset(userid Oid, dbid Oid, queryid bigint) returns void,

которая, сбрасывает статистику для заданного пользователя, базы данных и запроса. В случае отсутствия одного из параметров для него подразумевается нулевое значение (неприменимое ограничение) и очищается статистика, соответствующая другим параметрам. Если никакой параметр не задан или все параметры имеют нулевое значение (неприменимое), очищается вся статистика. Вместе с очисткой всей статистики в **pg\_stat\_statements** также сбрасывается статистика в **pg\_stat\_statements\_info**. По умолчанию эту функцию могут выполнять только суперпользователи. Другим пользователям можно дать доступ к ней, используя GRANT.

## 3. Практическое задание

1. Выяснить, как кастомизировать «легенду» и минимизировать текст, и применить к последующим заданиям.
2. Почему была введена в приведенном примере метка **userid**, хотя явно она не используется?
3. В рассмотренном примере для одного и того же запроса, но для разных пользователей и БД строятся разные графики. Рассматривается среднее время выполнения операторов (что тоже полезно рассмотреть), но не учитывается количество вызовов. Задание состоит в следующем. Сделать и оформить Dashboard, отображающий десять наиболее затратных в процентах операторов, исходя из общего времени их выполнения, для всех пользователей (т.е. выполнить группирование по ID пользователя и БД). Добавить метку количества вызовов. Выполнить 20 подключений к БД и загрузить сервер тестовыми операторами. Между вызовами включать задержку, длительность которой определяется датчиком случайных чисел. Можно использовать клиента, написанного на любом языке программирования. Запрос позволит выяснить наиболее часто вызываемые быстрые операторы, и, наоборот, долгие операторы, но редко используемые. Те и другие полезно рассмотреть детально с целью оптимизации.
4. Рассмотрим запрос:

SELECT schemaname, relname, seq\_scan, seq\_tup\_read,
seq\_tup\_read/seq\_scan AS avg, idx\_scan
FROM pg\_stat\_user\_tables
WHERE seq\_scan > 0
ORDER BY seq\_tup\_read DESC
LIMIT 10;

Его идея – найти большие таблицы, которые часто просматриваются последовательно. Эти таблицы окажутся в начале списка и будут содержать большие значения в поле seq\_tup\_read. Он помогает найти таблицы с отсутствующими индексами.
Задание состоит в следующем. Изменить этот запрос, чтобы отображались проценты последовательного просмотра в среднем **avg** к их общему числу для всех таблиц (100\***avg/sum(avg))**. Сделать и оформить Dashboard, отображающий десять наиболее «плохих» таблиц. Выполнить тестовые запросы, аналогичные предыдущему заданию.

## 4. Список использованных источников

1. Grafana Labs.
<https://grafana.com/>
2. Prometheus
<https://prometheus.io/>
3. Postgres\_exporter
<https://github.com/prometheus-community/postgres_exporter>