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 №4 «****Введение в мониторинг РСУБД. PromQL»**

Авторы:

Кудрявцев А.П., [kudryavtsevap@bmstu.ru](mailto:kudryavtsevap@bmstu.ru)

Фомин М.М.

Москва, 2023

## Лабораторная работа №4. Введение в мониторинг РСУБД. PromQL

1. Работа с Prometheus

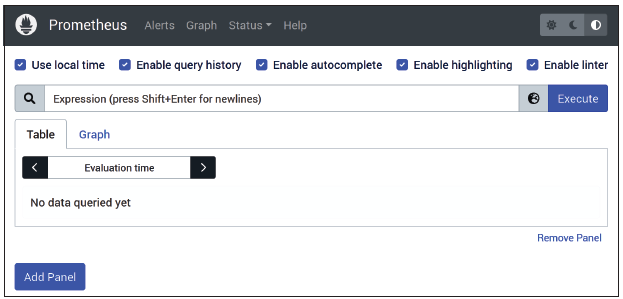
Prometheus – это система мониторинга с открытым исходным кодом, основанная на метриках. Prometheus выполняет только мониторинг. Она имеет простую, но мощную модель данных и язык запросов, позволяющий анализировать производительность приложений и инфраструктуры. Она не пытается решать проблемы, выходящие за рамки метрик, оставляя решение этой задачи другим, более подходящим инструментам. Модель данных идентифицирует каждый временной ряд не только по имени, но и по неупорядоченному набору пар ключ–значение, называемых метками. Язык запросов PromQL допускает агрегирование по любой из этих меток, что позволяет анализировать данные мониторинга не только по отдельным процессам, но также по центрам обработки данных, службам и любым другим меткам, которые вы определите. Эти данные можно отобразить в виде графиков в системах поддержки Dashboard (информационных панелей), таких как Grafana.

1.1 Браузер Prometheus

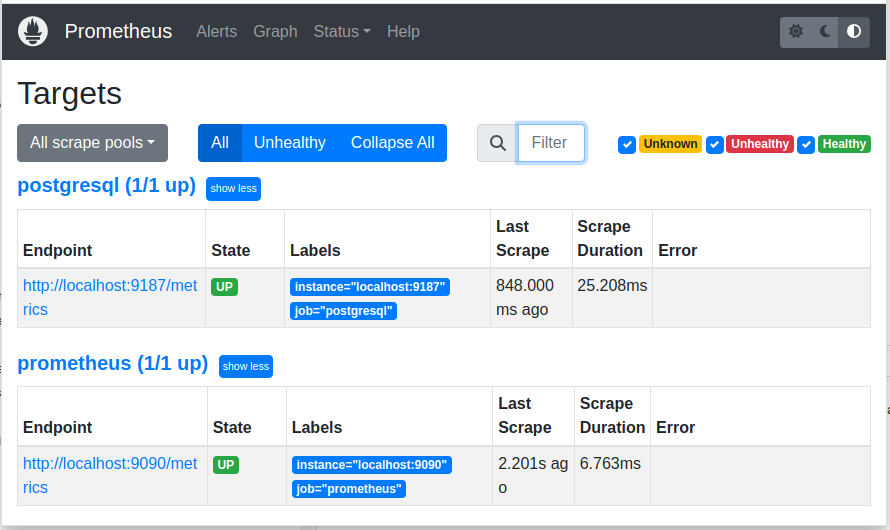
Браузер выражений удобно использовать для выполнения специальных запросов, разработки выражений PromQL и отладки перед тем как использовать их в Grafana.

По умолчанию Prometheus использует TCP-порт 9090, а

данные извлекаются каждые 10 с. Запустим пользовательский интерфейс Prometheus в браузере по ссылке http://localhost:9090/, как это показано на рисунке.

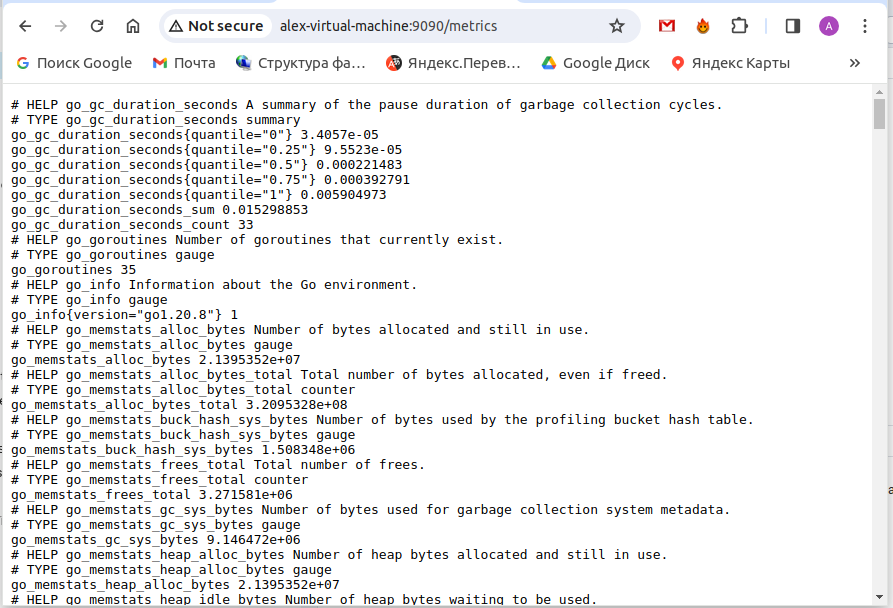


Это браузер выражений, в котором можно запускать запросы PromQL. В пользовательском интерфейсе также есть еще несколько страниц, которые помогут вам понять, что делает Prometheus, например страница Targets (Цели) на вкладке Status (Состояние), которая выглядит, как показано на рисунке.



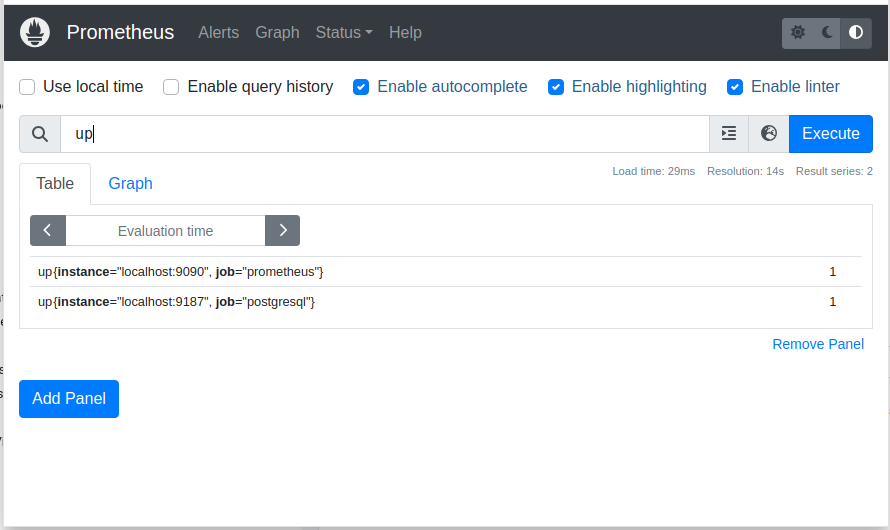
В списке на этой странице присутствует два сервера PostgreSQL и Prometheus, находящийся в состоянии UP (работает), которое означает, что последняя попытка извлечения информации прошла успешно. Если бы возникла какая-то проблема при извлечении метрик, то в поле Error (Ошибка) появилось бы сообщение.

Следует обратить внимание на страницы /metrics как у PostgreSQL, так и у Prometheus. Сама система Prometheus инструментирована и подготовлена для сбора метрик. Щелкнув по адресу <http://localhost:9090/metrics>, перейдем на страницу метрик Prometheus.



1.2. Применение браузера выражений

Браузер выражений удобно использовать для выполнения специальных запросов, разработки выражений PromQL и отладки как PromQL, так и данных внутри Prometheus. Для начала убедитесь, что вы находитесь в представлении консоли, введите выражение up и щелкните по кнопке Execute (Выполнить).

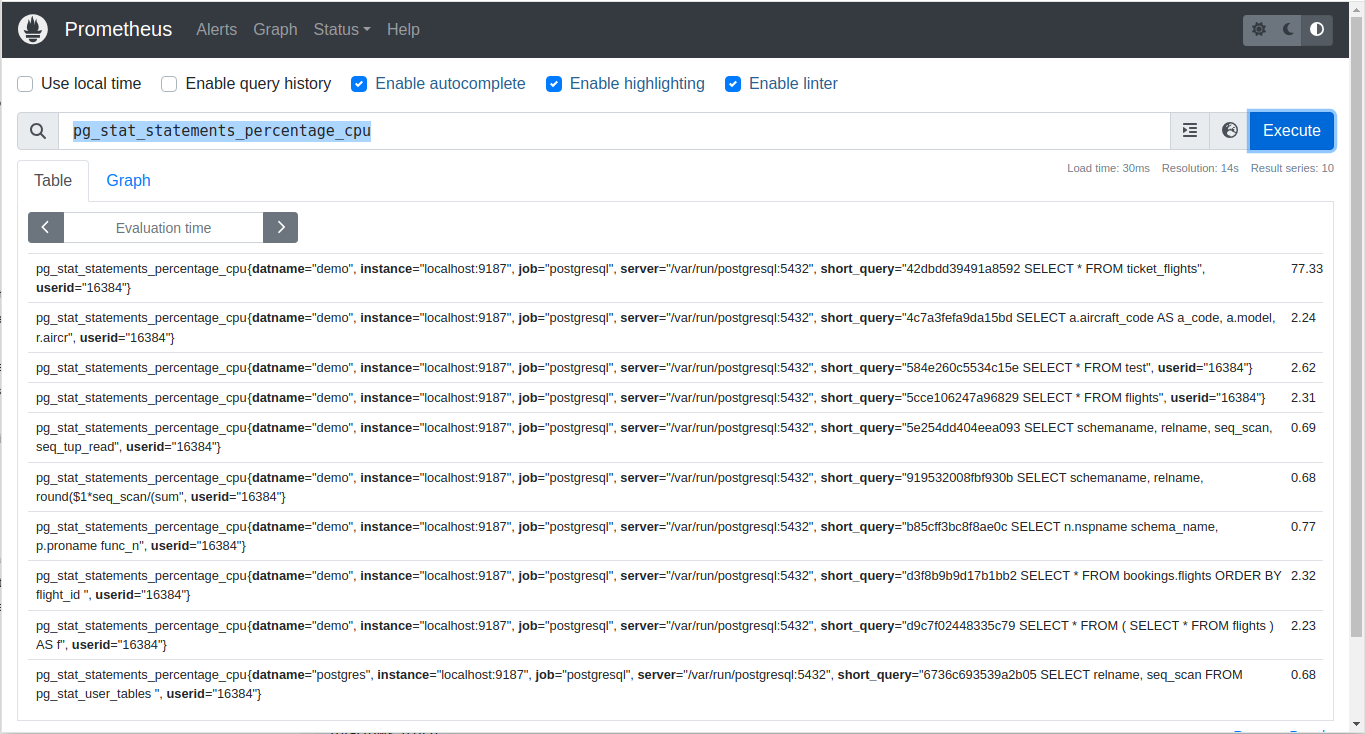


Этот запрос возвращает две строки со значениями 1 и именами:

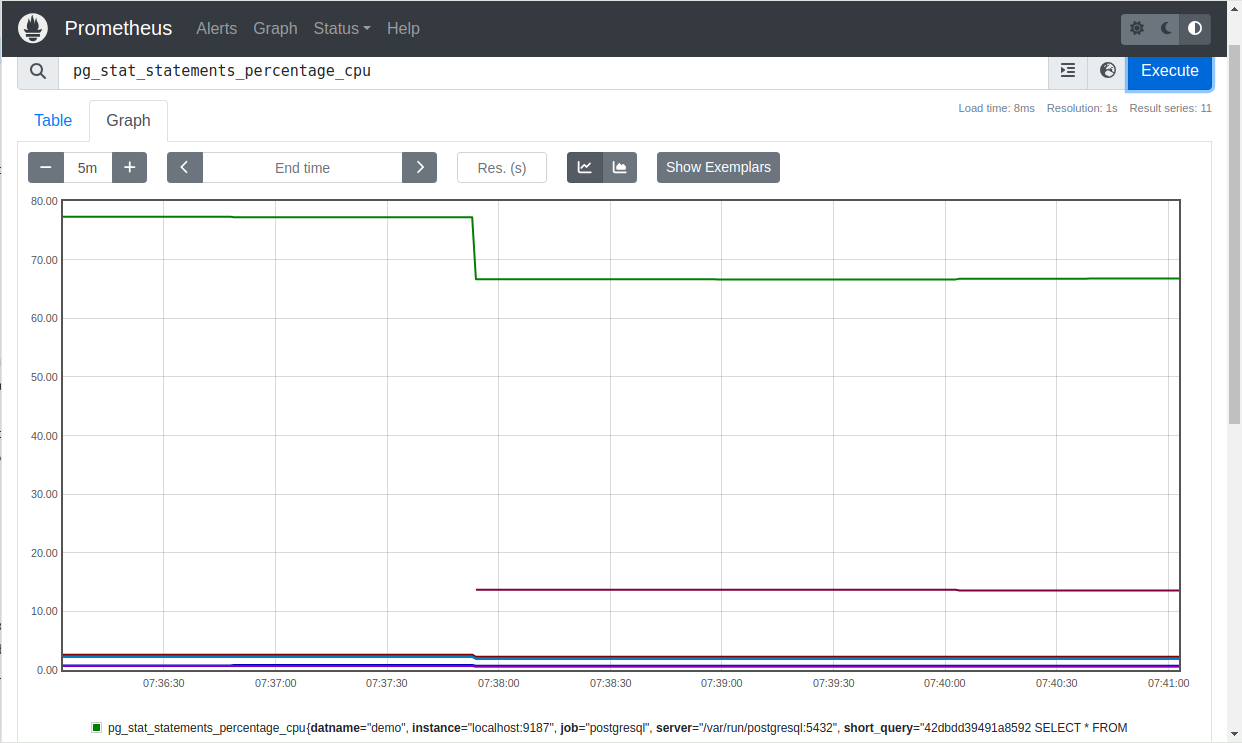
up{instance="localhost:9090",job="prometheus"},  
up{instance="localhost:9090",job="postgresql"}.

up – это специальная метрика, добавляемая Prometheus перед началом сбора данных; 1 означает, что сбор прошел успешно. instance – это метка, идентифицирующая цель, откуда были извлечены данные. В данном случае она сообщает, что целями был сам сервер Prometheus и сервер PostgreSQL.

Далее выполним запрос pg\_stat\_statements\_percentage\_cpu.



В правой колонке отображаются значения метрики – проценты используемого процессорного времени. Можно посмотреть изменения значений с течением времени, щелкнув по вкладке Graph (График).



Такие метрики, как pg\_stat\_statements\_percentage\_cpu, называются датчиками (Gauge), они могут как увеличиваться, так и уменьшаться. Для датчиков важно текущее абсолютное значение.

Но есть еще один основной тип метрик – счетчики (Counter). Счетчики накапливают количество произошедших событий или общий размер данных, связанных с событиями. Счетчик никогда не убывает, но может быть сброшен в ноль специальными функциями.

Рассмотрим пример счетчика, построенного с использования запроса.

SELECT sum(st.calls) AS calls,

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

db.datname

FROM pg\_stat\_statements AS st

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

WHERE 1000 < st.userid

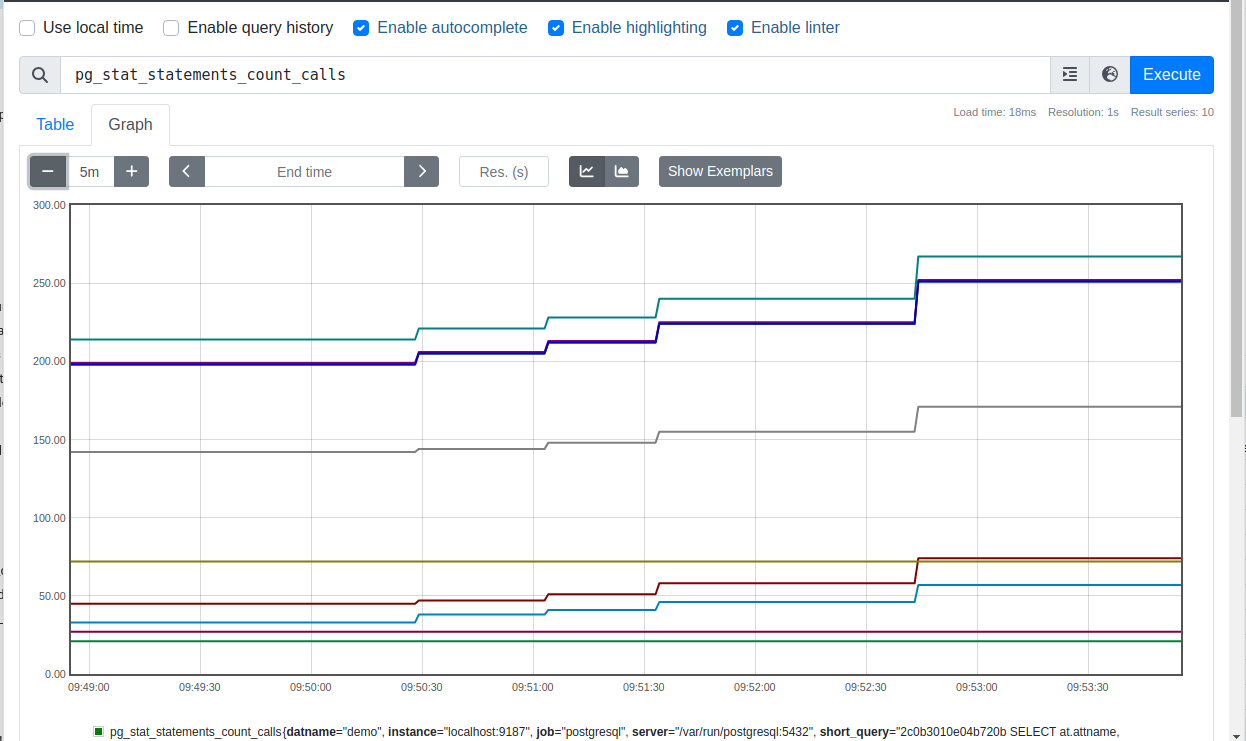
GROUP BY db.datname, st.queryid, st.query

ORDER BY sum(st.calls) DESC

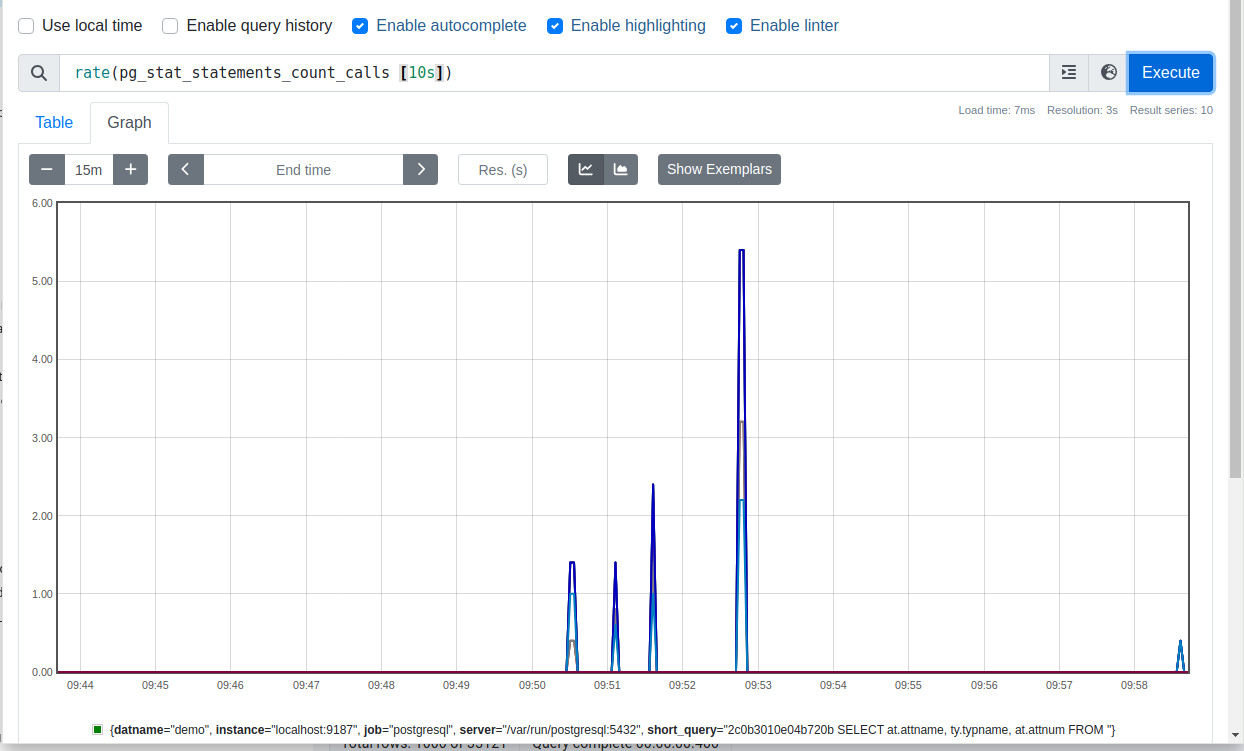
LIMIT 10

В нем подсчитывается количество вызовов запросов всеми пользователями. В этом запросе исключен пользователь postgres. Запросы, которые формируют пользовательские метки, то же учитываются в статистике. В моем случае они запускаются от пользователя postgres, userid которого равно 10. Итак, рассмотрим метрику pg\_stat\_statements\_count\_calls.

Счетчики всегда только увеличиваются, поэтому их графики имеют вид линий, всегда стремящихся вверх и вправо.



Но значения счетчиков сами по себе не очень полезны. Намного полезнее знать, как быстро увеличивается значение счетчика, и для этого можно использовать функцию скорости rate (о ней и других функциях см. ниже). Функция rate вычисляет насколько быстро увеличивается значение счетчика за 1 секунду. Измените выражение на rate(pg\_stat\_statements\_count\_calls [10s]) в браузере. Оно подсчитает количество метрик, обрабатываемых в секунду, усредненное за 10 секунд, и даст результат, подобный изображенному на рисунке.



И что мы видим? В течении короткого времени были вызваны несколько раз запросы, а после активность пользователей закончилась.

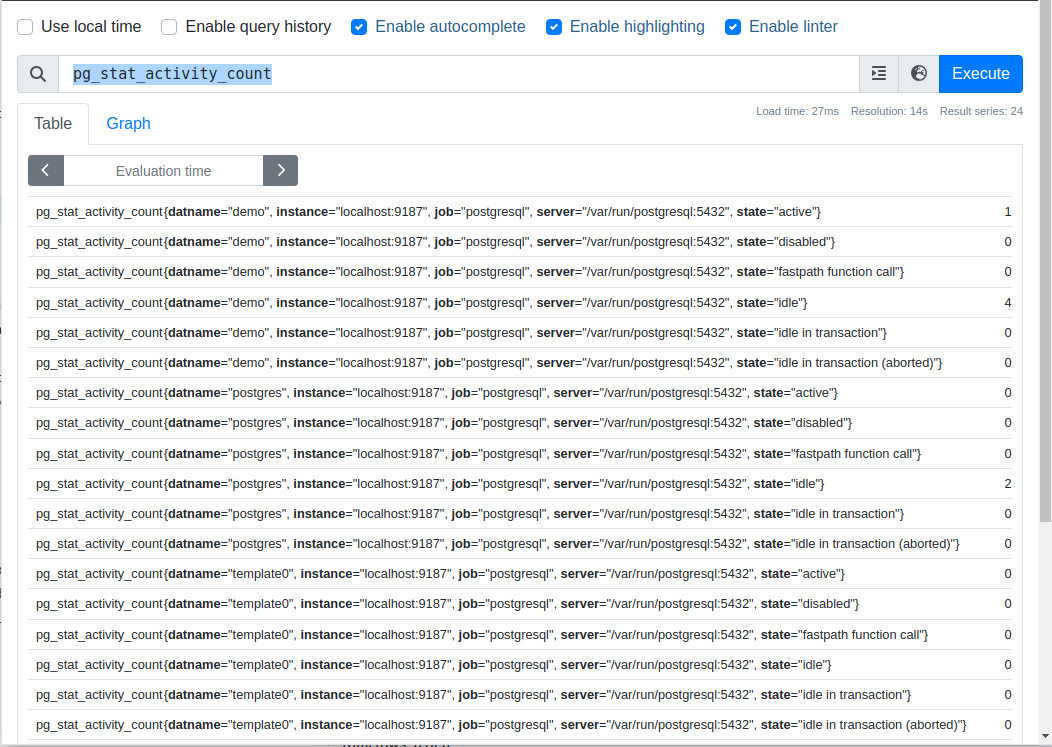
2. Введение в PromQL

PromQL – это язык запросов Prometheus (Prometheus Query Language). Хотя его название заканчивается на QL, вы увидите, что он совсем не похож на язык SQL.

2.1. Значения метрик

2.1.1. Датчики (Gauge)

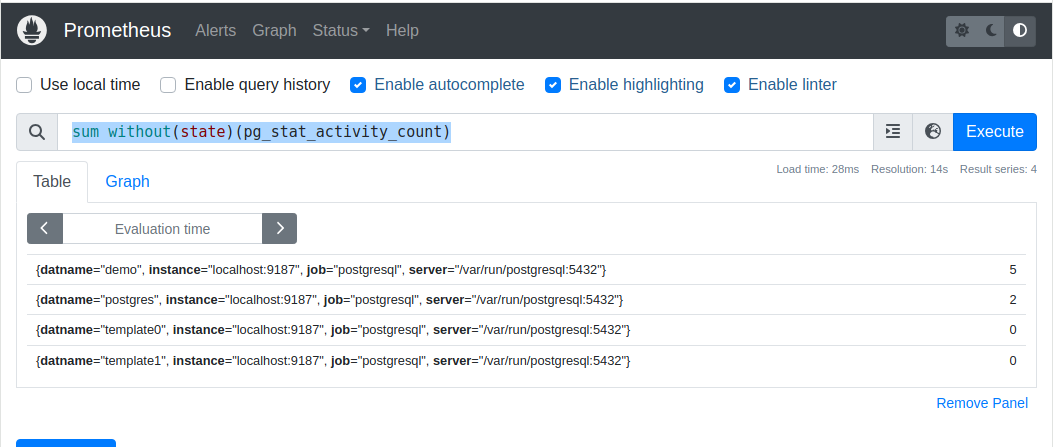
Датчики – это моментальные снимки состояния, и обычно под их агрегированием понимается получение суммы, среднего значения, минимума или максимума. Например, метрика pg\_stat\_activity\_count, получающая количество подключенных пользователей к БД. Значение может увеличиваться, так же и уменьшаться.



Рассмотрим метрику pg\_stat\_activity\_count из postgres\_exporter, которая сообщает количество подключений к базам данных, их состояние, на каких серверах расположены БД и от кого получены метрики. Определим сколько подключений выполнено независимо от статуса, это можно сделать следующим образом:

sum without(state)(pg\_stat\_activity\_count).

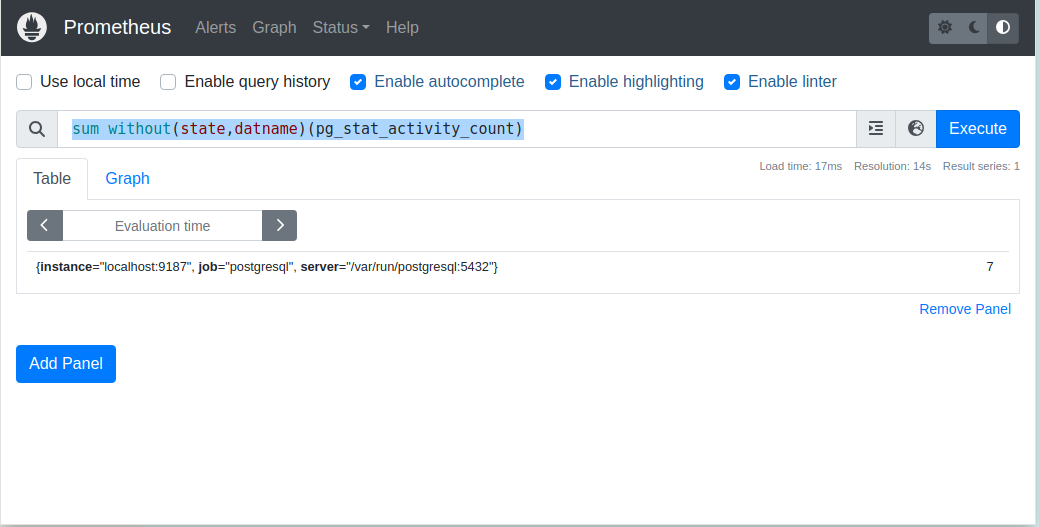
Здесь without сообщает функции агрегирования sum, что она должна суммировать все с одинаковыми метками, игнорируя только status. Таким образом, получаем временные ряды:



Мы видим, что метка state исчезла. Продолжим, исключим метку datname.

sum without(state,datname)(pg\_stat\_activity\_count).

Получаем:



Аналогичный подход можно использовать с другими функциями агрегирования.

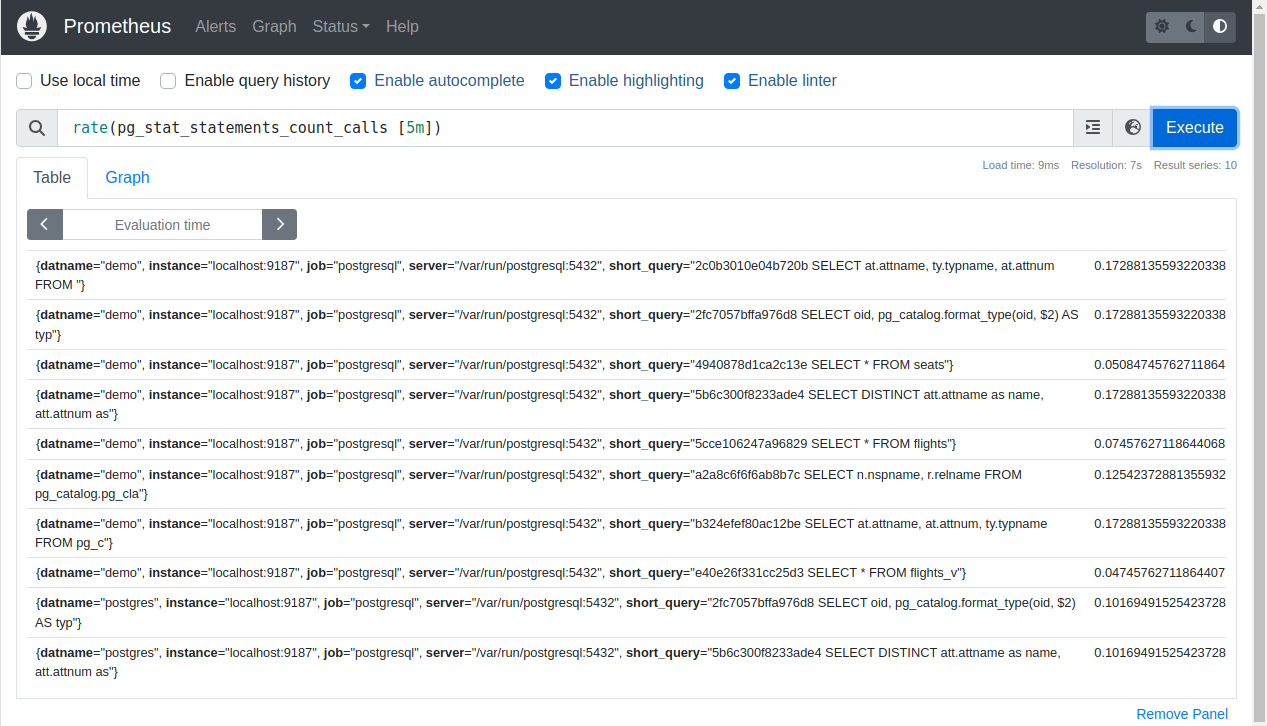
2.1.2. Счетчики (Counter)

Счетчики отслеживают количество или размеры событий, а значения, экспортируемые приложениями в /metrics, являются общими, накопленными с момента запуска. Но значения счетчиков сами по себе мало полезны. Чаще бывает желательно знать, как быстро счетчик увеличивается с течением времени. Обычно для этого используется функция rate, но к счетчикам можно также применять функции increase и irate.

Например, вот как можно рассчитать «производную» (на 1с) от количества вызовов запросов:

rate(pg\_stat\_statements\_count\_calls [5m])

rate вычисляет среднюю скорость в секунду по всем точкам данных в каждом интервале [5m].



Результатом rate является датчик, поэтому к нему применимы те же агрегирующие функции, что и к обычным датчикам. Метрика pg\_stat\_statements\_count\_calls имеет метки datname и short\_query, поэтому, если агрегировать по ним, то можно получить общее количество вызовов запросов, выполняемых базами данных в секунду:

sum without(datname, short\_query) (rate(pg\_stat\_statements\_count\_calls [5m]))

2.2. Селекторы

Работа с разными временными рядами метрики, имеющими разные значения меток, может вызывать путаницу, если метрика поступает с нескольких серверов разных типов. Обычно требуется сузить временные ряды, участвующие в расчетах. В большинстве случаев желательно ограничиться меткой job, и в зависимости от конкретных требований вы можете захотеть, например, исследовать только один экземпляр (instance) или одну БД (datname).

Такое ограничение по меткам осуществляется с помощью селекторов.

Например,

pg\_stat\_statements\_count\_calls {datname="demo"}

это - селектор, возвращающий все временные ряды с именем pg\_stat\_statements\_count\_calls и значением demo в метке datname, соответствующие данному моменту времени. Знак равенства в условии datname="demo" называется сопоставителем (matcher), и в одном селекторе может быть указано несколько сопоставителей, которые объединяются по AND.

2.2.1. Сопоставители

Есть четыре сопоставителя.

=

Это - сопоставитель равенства; например, datname="demo". Он позволяет указать, что возвращаемый временной ряд должен иметь заданную метку с заданным значением. Если задать пустое значение для метки, то это будет

равносильно отсутствию метки, поэтому можно использовать выражение

foo="", чтобы указать, что метка foo отсутствует.

!=

Это - сопоставитель неравенства; например, datname!="demo". Он позволяет указать, что возвращаемый временной ряд должен иметь заданную метку со значением, отличным от заданного.

=~

Это - сопоставитель соответствия регулярному выражению; например, datname=~"d.\*". Он позволяет указать, что возвращаемый временной ряд должен иметь заданную метку, значение которой соответствует заданному регулярному выражению.

!~

Это - сопоставитель несоответствия регулярному выражению.

2.2.2. Мгновенный вектор

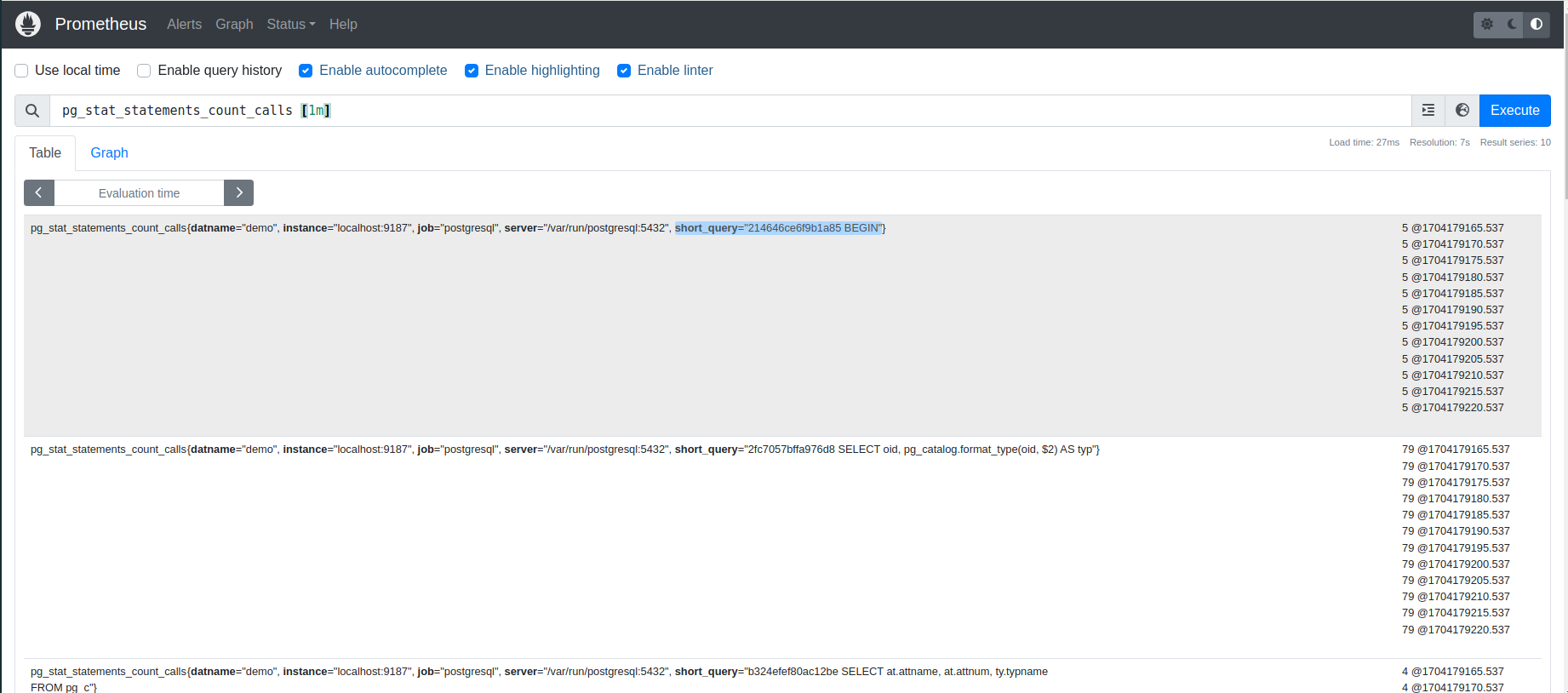
Селектор мгновенного вектора возвращает мгновенный вектор c самыми последними значениями, полученными к моменту выполнения запроса, т. е. список из нуля или более временных рядов. Каждый из этих временных рядов будет иметь один образец, а образец будет содержать значение и отметку времени. Мгновенный вектор, возвращаемый селектором мгновенного вектора, будет содержать отметки времени из исходных данных, но любые мгновенные векторы, возвращаемые другими операциями или функциями, будут содержать отметки времени, соответствующие моменту выполнения запроса.

2.2.3. Вектор диапазона

Существуют также селекторы второго типа, называемые селекторами векторов диапазона. Их вы тоже уже видели. В отличие от селектора мгновенного вектора, возвращающего единственный образец из каждого соответствующего временного ряда, селектор вектора диапазона может вернуть множество образцов. Векторы диапазонов всегда используются с функцией rate, например:

rate(pg\_stat\_statements\_count\_calls [1m])

[1m] превращает селектор мгновенного вектора в селектор вектора диапазона и сообщает движку PromQL, что он должен вернуть все образцы из всех временных рядов, соответствующих селектору, полученные за минуту до времени выполнения запроса. Если выполнить только pg\_stat\_statements\_count\_calls [1m] на вкладке Table браузера выражений, то результат будет выглядеть примерно так, как показано на рисунке.



В данном случае в каждом временном ряду оказалось по двенадцать образцов, полученных за последнюю минуту. Обратите внимание, что образцы в каждом временном ряду располагаются с шагом 5 с, согласно настроенному интервалу опроса (см. prometheus.yml).

Интервалы времени в запросах PromQL и в конфигурационных файлах Prometheus можно задавать с использованием разных единиц измерения.

|  |  |
| --- | --- |
| Суффикс | Значение |
| ms | Миллисекунды |
| s | Секунды |
| m | Минуты |
| h | Часы |
| d | День (сутки) |
| w | Неделя |
| y | Год |
|  |  |

Допускается комбинировать несколько единиц измерения с целыми числами. Единственное условие: они должны следовать в порядке увеличения точности. Например, 90m, 1h30m и 1.5h – допустимые значения, а   
30m1h – недопустимое. Високосные годы не поддерживаются; 1y всегда содержит 60\*60\*24\*365 секунд.

2.2.4. Подзапросы

Несмотря на то, что векторы диапазонов применяются к временным рядам, их нельзя использовать в сочетании с функциями. Если потребуется объединить max\_over\_time с rate, то в таком случае можно использовать правила записи, которые будут записывать результат функции rate и передавать его векторной функции, или подзапрос.

Подзапрос – это часть запроса, позволяющая выполнять запрос диапазона внутри запроса. Подзапросы должны заключаться в квадратные скобки подобно селекторам диапазонов. Но для этого нужно указать два разных интервала: диапазон и разрешение.

Диапазон – это интервал, возвращаемый подзапросом, а разрешение действует как шаг:

max\_over\_time(rate(pg\_stat\_statements\_count\_calls[5m])[30m:1m])

Предыдущий запрос будет вызывать rate(pg\_stat\_statements\_count\_calls[5m]) каждую минуту (1m) в течение последних 30 мин (30m), а затем передаст результат в функцию max\_over\_time() (максимальное значение в диапазоне). Разрешение можно не указывать, например [30m:]. В этом случае в качестве разрешения используется глобальный интервал оценки.

2.2.5. Смещение

При работе с любыми векторными селекторами можно также использовать дополнительный модификатор смещения offset. Он может сдвинуть время вычисления запроса назад во времени для каждого селектора. Например:

pg\_stat\_statements\_count\_calls{datname="demo"} offset 1h

даст количество вызовов за час до момента выполнения запроса. Модификатор offset обычно не используется в таких простых запросах, как этот, так как гораздо проще изменить начальное время для всего запроса. Но он может пригодиться, когда требуется настроить только один селектор в выражении запроса.

2.3. Операторы агрегирования

Агрегирование играет важную роль. Для приложений с тысячами или даже десятками экземпляров нецелесообразно просеивать метрики каждого экземпляра по отдельности. Агрегирование позволяет суммировать метрики не только в рамках одного приложения, но и по всем приложениям.

2.3.1. Группировка

Операторы агрегирования в PromQL подобно SQL применяются к группам. Прежде чем рассматривать функции агрегирования, выясним, как группируются временные ряды.

**without**

Мы уже использовали модификатор without, он позволяет исключить метки и сгруппировать временные ряды по оставшимся меткам и для каждой группы применит функцию sum.

**by**

В дополнение к without есть также модификатор by. Но, в отличие от without, который указывает, какие метки удалить, модификатор by указывает, какие метки оставить. Соответственно, при использовании by требуется некоторая осторожность, чтобы не удалить целевые метки, которые вы собирались рассылать в предупреждениях или использовать в Dashboard. Имейте в виду, что в одном выражении агрегирования нельзя использовать сразу оба модификатора, by и without.

2.3.2. Операторы

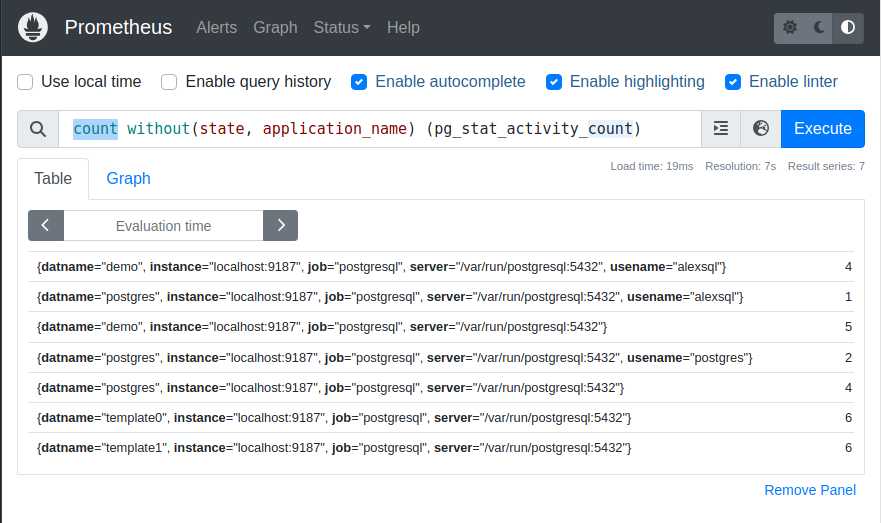
Все 11 операторов агрегирования используют одинаковую логику группировки, работой которой можно управлять с помощью модификаторов without и by. Но они по-разному обрабатывают сгруппированные данные.

**sum –** суммирует все значения в группе и возвращает полученную сумму. Например,

sum without(datname, short\_query)(rate(pg\_stat\_statements\_count\_calls [5m]))

**count –** подсчитывает количество временных рядов в группе и возвращает полученное число. Например,

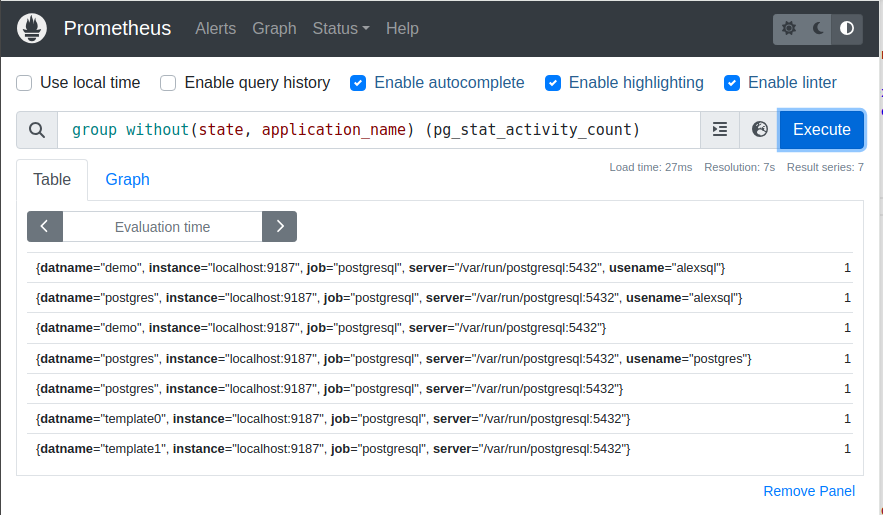
count without(state, application\_name) (pg\_stat\_activity\_count)



**avg –** возвращает среднее значение временных рядов в группе.

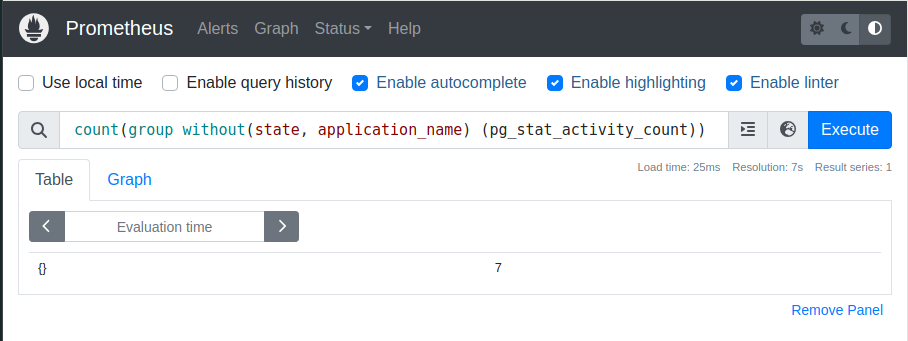
**group –** возвращает 1 для каждой группы. Например,

group without(state, application\_name) (pg\_stat\_activity\_count)



Подсчитаем количество групп:

count(group without(state, application\_name) (pg\_stat\_activity\_count))



**stddev и stdvar** – стандартное отклонение – это статистическая мера разброса набора чисел. Основное применение стандартного отклонения в мониторинге – обнаружение выбросов. В случае нормально распределенных данных можно ожидать, что около 68 % образцов будут находиться в пределах одного стандартного отклонения от среднего значения, а 95 % – в пределах двух стандартных отклонений. Если один экземпляр в задании имеет метрику, отличающуюся на несколько стандартных отклонений от среднего, то это верный признак того, что с ним что-то не так.

Стандартная дисперсия – это квадрат стандартного отклонения и широко

используется в статистике.

**min и max –** агрегаторы возвращают минимальное и максимальное значение в группе соответственно. При работе с ними применяются те же правила группировки, что и везде.

**topk и bottomk** – имеют три важных отличия от других агрегаторов, обсуждав-

шихся до сих пор. Во-первых, метки временных рядов, которые они воз-

вращают для группы, не являются метками группы; во-вторых, для каждой

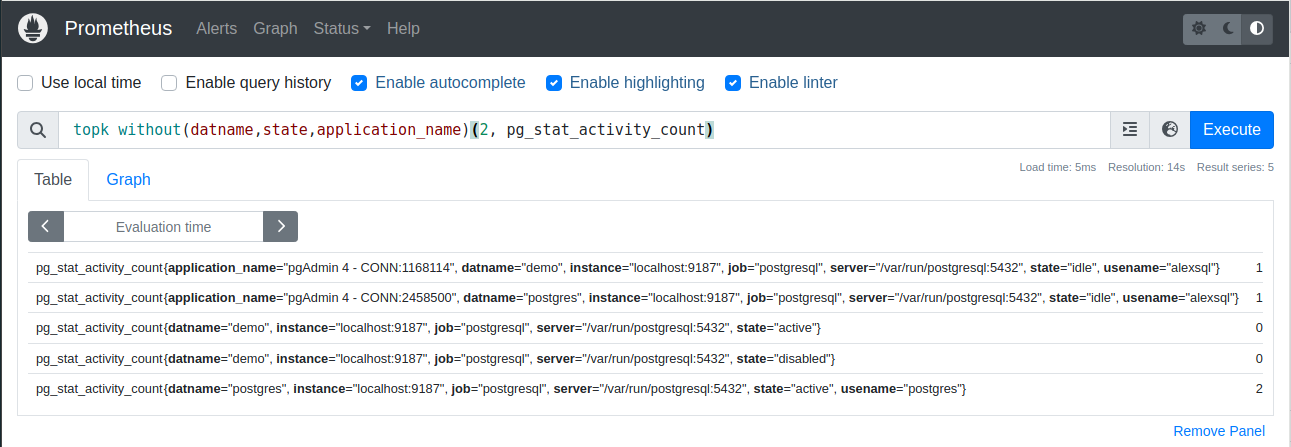
группы они могут возвращать несколько временных рядов; и в-третьих, они

принимают дополнительный параметр.

topk возвращает k временных рядов с наибольшими значениями, например:

topk without(datname,state,application\_name)(2, pg\_stat\_activity\_count)

вернет до двух временных рядов для каждой группы (последняя группа состоит из одного временного ряда):



Как видите, topk возвращает входные временные ряды со всеми их метками, включая имя метрики. Результаты дополнительно сортируются.

bottomk действует так же, как topk, но возвращает k временных рядов с наименьшими значениями.

**quantile** – возвращает указанный квантиль значений группы. Подобно topk, оператор quantile принимает дополнительный параметр.

Например, с помощью следующего запроса можно узнать 0.9 квантиль

времени снятия статистических данных для БД в секундах:

quantile(0.9, pg\_scrape\_collector\_duration\_seconds {collector="database"} )

В моем случае оно равно 0.018297127 сек.

**count\_values** – подобно topk он принимает дополнительный параметр и может вернуть несколько временных рядов из группы. Он строит гистограмму частот значений временного ряда в группе со счетчиком каждого значения в качестве значения выходного временного ряда и исходным значением в качестве новой метки.

Рассмотрим пример. Допустим, у вас есть временной ряд software\_version со следующими значениями:

software\_version{instance="a",job="j"} 7

software\_version{instance="b",job="j"} 4

software\_version{instance="c",job="j"} 8

software\_version{instance="d",job="j"} 4

software\_version{instance="e",job="j"} 7

software\_version{instance="f",job="j"} 4

Применив к этому временному ряду запрос

count\_values without(instance)("version", software\_version)

получим такой результат:

{job="j",version="7"} 2

{job="j",version="8"} 1

{job="j",version="4"} 3

В группе есть два временных ряда со значением 7, поэтому для временного ряда с version="7" и метками группы было возвращено значение 2. Аналогичный результат возвращается для других временных рядов.

При создании частотной гистограммы группировка не используется, используются точные значения временного ряда. Поэтому этот прием по-настоящему полезен только с целочисленными значениями и там, где не слишком много уникальных значений.

2.4. Регулярные выражения

Prometheus использует механизм регулярных выражений RE2, заимствованный из Go. Он обеспечивает линейное время оценки выражений, но не поддерживает обратных ссылок, опережающих проверок и некоторых других продвинутых возможностей. Для тех, кто не знаком с регулярными выражениями, отметим, что они определяют правила (называемые шаблонами), которые затем сопоставляются с текстом. Ниже приводится краткий справочник по регулярным выражениям.

|  |  |
| --- | --- |
| a | Символу **a** |
| . | Любому одному символу |
| \. | Одному символу точки |
| .\* | Любому количеству любых символов |
| .+ | По меньшей мере одному символу |
| a+ | Одному или нескольким символам **a** |
| [0-9] | Любой одной цифре от 0 до 9 |
| \d | Любой одной цифре от 0 до 9 |
| \d\* | Любому количеству цифр |
| [^0-9] | Одному символу, не являющемуся цифрой |
| ab | Символу **a**, за которым следует символ **b** |
| a(b/c\*) | Символу **a**, за которым следует один символ **b** или любое количество символов **c** |

Круглые скобки создают сохраняющую группу. То есть при применении шаблона (.)(\d+) к тексту a123 будут сохранены две группы: с символом a и с цифрами 123. Сохраняющие группы используются для извлечения фрагментов строки с целью их использования в дальнейшем.

3. Пример использования

Создадим Dashboard, в котором будем следить за нагрузкой на БД в части выборки данных. Для этой цели нам подойдет метрика по умолчанию pg\_stat\_database\_tup\_fetched, являющаяся счетчиком, которая возвращает количество извлеченных строк. Под нагрузкой будем понимать количество извлеченных строк в секунду, умноженное на диапазон равный, скажем 30с. Будем использовать функцию increase:

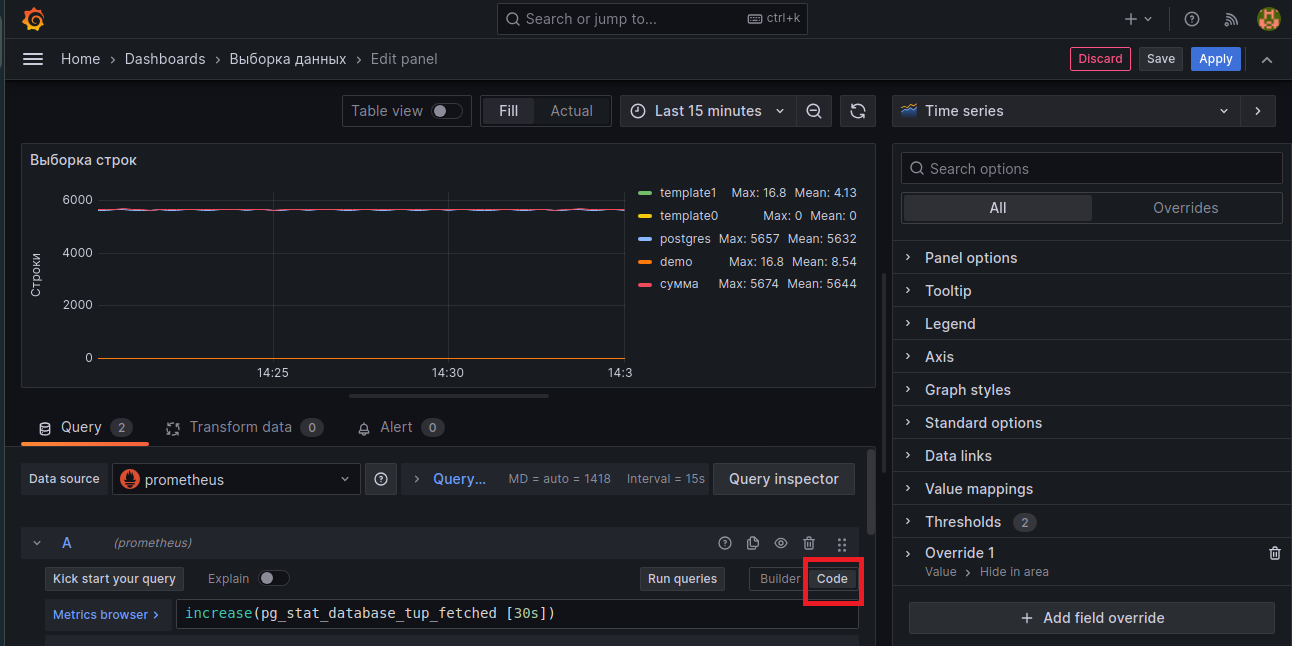
increase(pg\_stat\_database\_tup\_fetched [30s])

Функция increase(…[30s]) эквивалентна функции rate(…[30s])\*30. Данное значение подсчитывается для каждой БД. Неплохо бы получить итоговую нагрузку на сервер PostgreSQL. Нам необходимо просуммировать значения для по всем БД:

sum by(pg\_stat\_database\_tup\_fetched)(increase(pg\_stat\_database\_tup\_fetched[30s]))

Перед тем как, создавать Dashboard, откроем страницу Prometheus <http://localhost:9090/> и проверим выражения.

Приступаем к созданию Dashboard. Запускаем Grafana <http://localhost:3000/>. Создаем новый Dashboard, в нем мы не будем набирать формулы используя combobox’ы, а нажмем на кнопку Code и



в поле Metrics browser > скопируем increase(pg\_stat\_database\_tup\_fetched [30s]). Нажимаем на Run queries и получаем графики.

Добавим к нашим графикам итоговый временной ряд. В низу страницы нажимаем на +Add query и копируем второе выражение sum…. Приводим в порядок легенду, добавим максимальные и средние значения. Вводим название оси и т.д. Сохраняем и применяем. Запустим pgAdmin, в нем сделаем выборку из БД demo.



4. Практическое задание

1. Использование временных файлов при выполнении запросов замедляет работу сервера, поэтому необходимо отслеживать такие запросы. Представление pg\_stat\_statements содержит статистику по количеству блоков, записанных и прочтенных во временный файл в полях temp\_blks\_written и temp\_blks\_read, соответственно. Задание состоит в следующем:

* создать Dashboard с двумя панелями, в которых бы отслеживались изменения этих полей (использовать increase);
* выполнить суммирование временных рядов по пользователям и БД;
* выбрать только пять наиболее «плохих» запросов;
* рассмотреть различные диапазоны для функции increase и определиться с наиболее подходящим;
* оформить Dashboard надлежащим образом.

**Примечание**.

Для того, чтобы вынудить PostgreSQL создавать временные файлы, используйте случайную сортировку, например:

SELECT \* FROM ticket\_flights ORDER BY random();

2. В представлении pg\_stat\_database поля xact\_commit и xact\_rollback содержат количество успешных и отклоненных транзакций. Откат транзакций может быть вызван backup’ом БД, изменениями в таблицах и т.д. Но, если это происходит в иных случаях, то необходимо обратить на это внимание.

Задание:

* необходимо создать Dashboard с двумя панелями для xact\_commit, xact\_rollback и отслеживать изменение полей только для БД demo;
* использовать функции increase, rate и irate с одинаковыми диапазонами для каждой панели (три графика в каждой панели), объяснить различие;
* рассмотреть различные диапазоны для функций;
* тестировать с использованием pgbench (см. ЛР №2).

5. Список использованных источников

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