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 №1** **«****Сбор статистики в PostgreSQL»**

Авторы:

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

Фомин М.М.

Москва, 2023

Лабораторная работа №1. «Сбор статистики в PostgreSQL»

1. Предварительная подготовка

Все лабораторные работы по курсу «Администрирование РСУБД PostgreSQL» с использованием ОС Linux (UBUNTU 22.04.3). Для тех, у кого установлена ОС Windows, необходимо установить виртуальную машину и Linux.

2. Развертывание РСУБД PostgreSQL

2.1. Краткая информация о СУБД PostgreSQL

**PostgreSQL** – комплекс программ, относящийся к классу систем управления базами данных. Когда эта система выполняется, мы называем ее сервером PostgreSQL или экземпляром сервера.

Данные, которыми управляет PostgreSQL, хранятся в базах данных. Один экземпляр PostgreSQL может одновременно управлять несколькими базами, которые вместе называются кластером баз данных.

Чтобы кластер можно было использовать, его необходимо инициализировать (создать). Папку, в которой размещаются все файлы, относящиеся к кластеру, обычно называют - **PGDATA** по имени переменной окружения, указывающей на эту папку.

2.2. Установка СУБД PostgreSQL

Для выполнения заданий лабораторной работы предлагается использовать СУБД PostgreSQL 14 или более новых версий, дистрибутив которых можно скачать по ссылке: <https://www.postgresql.org/download/>

Там же имеются инструкции по установке PostgreSQL на различные ОС.

Ниже излагается установка PostgreSQL 14 на UBUNTU.

Запустите терминал, например, нажав комбинацию клавиш Ctrl+Alt+T.

Войдите с правами пользователя root, выполнив команду:

sudo -i

После нажатия Enter, будет предложено ввести пароль. А далее появится приглашение для ввода команд:

root@...:~#

Проверим установлена ли утилита **wget:**

which wget

Если установлена, то отобразится путь к утилите:

/usr/bin/wget

В противном случае, установим:

apt update

apt install wget

Создаем конфигурацию файлового репозитория:

sh -c 'echo "deb https://apt.postgresql.org/pub/repos/apt $(lsb\_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'

Импортируем ключ подписи репозитория:

wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -

Обновите списки пакетов:

apt-get update

Устанавливаем 14 версию:

apt-get -y install ‘postgresql-14’

При установке пакета инсталлятор создаст новый PostgreSQL-кластер. Данный кластер представляет из себя коллекцию баз данных, которая управляется одним сервером. Также установщик создаст рабочие директории для PostgreSQL. Данные, необходимые для работы PostgreSQL, будут находиться в папке **/var/lib/postgresql/14/main**, а файлы конфигурации – в папке **/etc/postgresql/14/main**.

После завершения установки вы можете убедиться, что служба PostgreSQL активна. Для этого в командной строке наберите:

systemctl is-active postgresql

Также посмотрите, включена ли служба:

systemctl is-enabled postgresql

И наконец, вы можете увидеть статус службы PostgreSQL:

systemctl status postgresql

● postgresql.service - PostgreSQL RDBMS

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

Active: active (exited) since Tue 2023-12-26 08:51:52 MSK; 1h 24min ago

Process: 1357 ExecStart=/bin/true (code=exited, status=0/SUCCESS)

Main PID: 1357 (code=exited, status=0/SUCCESS)

CPU: 784us

…

В PostgreSQL автоматически создается пользователь ОС без права подключения и одноименный суперпользователь БД без пароля – postgres. И первым шагом начало работы добавим суперпользователя БД с паролем. Подключитесь к системе с помощью учётной записи postgres:

su – postgres

Должно появиться приглашение от имени **postgres**:

postgres@...:~$ exit

Подключившись, выполните команду psql:

psql

Если утилита не запускается, то скорее всего указан порт не 5432 в файле **/etc/postgresql/14/main/postgresql.conf**. Возможны два варианта: первый, для psql указывать порт подключения; второй, это - выяснить, какая программа использует этот порт (скорее всего ранее была неудачная попытка установитьPostgreSQL) и, решившись, измените порт в файле **postgresql.conf.**

Если вы видите приглашение ко вводу команд **postgres=#**, значит вы находитесь в оболочке СУБД PostgreSQL. И значит, можно приступать к добавлению пользователя, созданию базы данных и т.д. Добавим в PostgreSQL пользователя (на своем сервере вы можете использовать свои имена пользователей и баз данных):

CREATE USER имя\_пользователя WITH SUPERUSER LOGIN PASSWORD 'пароль';

Добавив пользователя, завершим работу с PostgreSQL командой:

exit

Все работы будут проводиться на локальном компьютере, это позволит нам избежать ввода пароля при подключении к БД из терминала, изменив файл **/etc/postgresql/14/main/pg\_hba.conf**. Откроем его:

nano /etc/postgresql/14/main/pg\_hba.conf

Найдем строку:

# "local" is for Unix domain socket connections only

local all all peer

И вместо peer ведем **trust**.

# "local" is for Unix domain socket connections only

local all all trust

Сохраним (Ctrl+S) и выйдем (Ctrl+X). Перезапусти PostgreSQL:

systemctl restart postgresql

systemctl status postgresql

Проверим возможность подключения к БД postgres без ввода пароля от имени только что созданного пользователя:

psql postgres -Uимя\_пользователя

Должно появиться приглашение:

postgres=#

И выходим:

exit

И если далее не собираетесь устанавливать pgAdmi4, то отключитесь от root командой:

logout

2.4. Установка pgAdmin4

pgAdmin 4 является графическим клиентом для работы с сервером, через который удобно удалять, изменять базы данных и управлять ими, а также создавать и выполнять SQL-запросы. Имеется альтернатива pgAdmin 4 это - DBeaver.

Дистрибутив для установки можно найти на сайте  [https://www.pgadmin.org /download/](%20%20%20https://www.pgadmin.org%20/download/).

Ниже излагается установка PostgreSQL 14 на UBUNTU.

Запустите терминал, например, нажав комбинацию клавиш Ctrl+Alt+T.

Войдите с правами пользователя root, выполнив команду:

sudo -i

Проверьте установлена ли утилита curl, запустив команду:

**which curl**

Если утилита установлена, то будет указан к ней путь:

**/snap/bin/curl**

В противном случае, надо curl установить:

**apt update**

**apt install curl**

Установим открытый ключ репозитория (если это не было сделано ранее):

**curl -fsS https://www.pgadmin.org/static/packages\_pgadmin\_org.pub | sudo gpg --dearmor -o /usr/share/keyrings/packages-pgadmin-org.gpg**

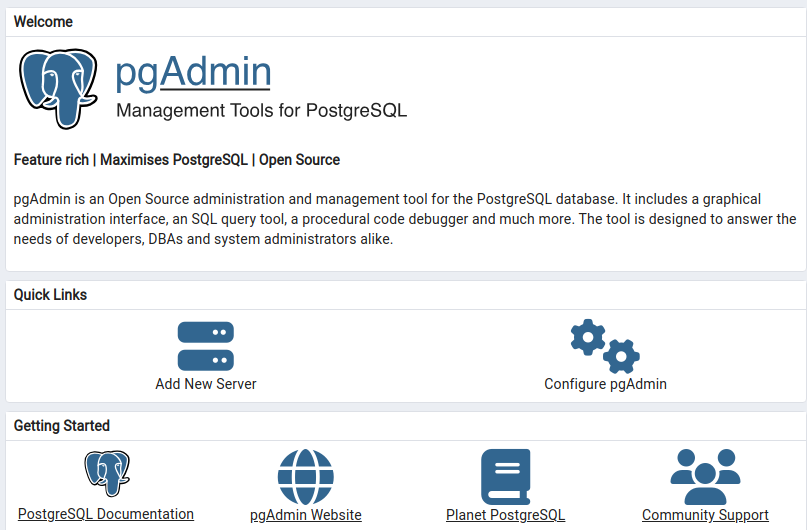
Создаем файл конфигурации репозитория:

**sh -c 'echo "deb [signed-by=/usr/share/keyrings/packages- pgadmin-org.gpg] https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/$(lsb\_release -cs) pgadmin4 main" > /etc/apt/sources.list.d/pgadmin4.list && apt update'**

Установим pgAdmin 4 только для рабочего стола:

**apt install pgadmin4-desktop**

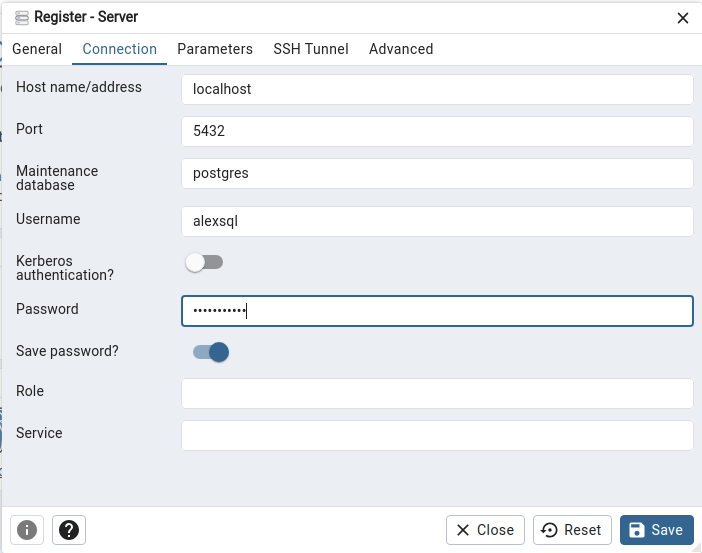
Подключим pgAdmin 4 к PostgreSQL. Запустите pgAdmin 4. На стартовой странице (см. рис.) щелкните по **Add New Server**.



Откроется диалог для регистрации сервера.



На первой вкладке в поле **Name** введем название сервера. Далее, щелкнем по вкладке **Connection**.



Если при установке PostgreSQL был выбран другой порт (и он не был изменен), то в поле **Port** введите его значение. Сохраните, нажав на кнопку **Save**.

2.5. Установка демонстрационной базы данных

Скачайте последнюю версию демонстрационной БД с сайта:

<https://postgrespro.ru/education/demodb>

Там же находится описание БД, с которой необходимо ознакомиться.

Следующим шагом надо распаковать скаченный файл либо с помощью утилиты tar, предварительно запустив терминал, либо с помощью nautilus (проводник) графической оболочки, щелкнув правой клавишей и выбрав один из вариантов Extract. В итоге в выбранной папке создастся файл   
**demo-big-20170815.sql**.

Возвращаемся в терминал, подключаемся к БД postgres:

psql postgres - Uимя\_пользователя

И подключаем demo к нашей БД:

postgres=# \i 'путь\_к\_файлу/demo-big-20170815.sql'

Запускаем pg\_Admin 4 или, если он запущен, то обновляем узел databases, выполнив команду Refresh из контекстного меню. И должны увидеть БД demo.

Закрываем терминал:

logout

exit

3. Планы выполнения

Результатом работы оптимизатора PostgreSQL является план выполнения. В то время как запрос определяет, что нужно сделать, план выполнения определяет, как выполнять операции SQL.

Оптимизатор использует внутренние показатели, основанные на объеме вычислительных ресурсов, необходимых для выполнения запроса или отдельной физической операции в рамках плана. Наиболее важными ресурсами являются те, которые влияют на время выполнения, а именно циклы процессора и количество операций ввода-вывода (чтение и запись дисковых блоков). Другие ресурсы, такие как память или дисковое пространство, тоже косвенно влияют на время выполнения. Например, количество доступной памяти будет влиять на соотношение циклов процессора и количество операций ввода-вывода.

Для построения плана оптимизатор использует алгоритмы оптимизации на основе стоимости. Стоимостная модель физической операции оценивает ресурсы, необходимые для выполнения операции.

Посмотрим стоимость некоторых операций:

SELECT current\_setting('seq\_page\_cost') AS seq\_page\_cost,

current\_setting('cpu\_tuple\_cost') AS cpu\_tuple\_cost,

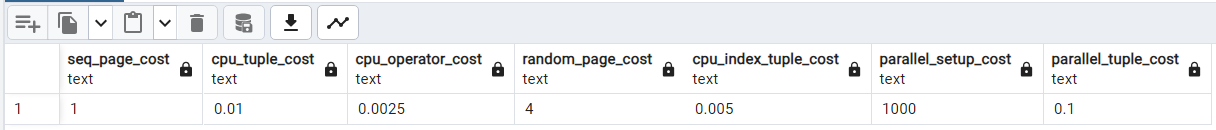
current\_setting('cpu\_operator\_cost') AS cpu\_operator\_cost,

current\_setting('random\_page\_cost') AS random\_page\_cost,

current\_setting('cpu\_index\_tuple\_cost') AS cpu\_index\_tuple\_cost,

current\_setting('parallel\_setup\_cost') AS parallel\_setup\_cost,

current\_setting('parallel\_tuple\_cost') AS parallel\_tuple\_cost;



* seq\_page\_cost – задаёт приблизительную стоимость чтения одной страницы с диска, которое выполняется в серии последовательных чтений.
* cpu\_tuple\_cost – задаёт приблизительную стоимость обработки каждой строки при выполнении запроса.
* cpu\_operator\_cost – задаёт приблизительную стоимость обработки оператора или функции при выполнении запроса.
* random\_page\_cost – задаёт приблизительную стоимость чтения одной произвольной страницы с диска.
* cpu\_index\_tuple\_cost – задаёт приблизительную стоимость обработки каждой записи индекса при сканировании индекса.
* parallel\_setup\_cost – задаёт приблизительную стоимость запуска параллельных рабочих процессов.
* parallel\_tuple\_cost – задаёт приблизительную стоимость передачи одного кортежа от параллельного рабочего процесса другому процессу.

Структура плана запроса представляет собой дерево, состоящее из узлов плана. Узлы на нижних уровнях дерева отвечают за просмотр и выдачу строк таблиц. Если конкретный запрос требует выполнение операций агрегирования, соединения таблиц, сортировки, то над узлами выборки строк будут располагаться дополнительные узлы дерева плана. Самая первая строка содержит общую оценку стоимости общего выполнения запроса.

Выполним запрос:

EXPLAIN SELECT \* FROM aircrafts\_data;

Получим план запроса:

QUERY PLAN

---------------------------------------------------------------

Seq Scan on aircrafts\_data (cost=0.00..1.09 rows=9 width=52)

(1 row)

Это простейший запрос, выводящий все строки таблицы. Поэтому планировщик выбирает последовательный просмотр **Seq Scan**. В скобках приведена стоимость плана. Первое число (0.00) это оценка ресурсов, требуемых для того, чтобы приступить к выводу данных. Никаких дополнительных операций не требуется и можно приступить к выводу данных сразу. Второе число (1.09) – это оценка общей стоимости выполнения запроса.

Усложним запрос:

EXPLAIN SELECT \*

FROM aircrafts\_data

ORDER BY aircraft\_code;

QUERY PLAN

---------------------------------------------------------------------

Sort (cost=1.23..1.26 rows=9 width=52)

Sort Key: aircraft\_code

-> Seq Scan on aircrafts\_data (cost=0.00..1.09 rows=9 width=52)

(3 rows)

Дополнительный узел обозначен на плане символом «->». Хотя по столбцу aircraft\_code создан индекс, планировщик предпочел не использовать этот индекс, а прибегнуть к последовательному сканированию (Seq Scan) таблицы. На верхнем узле выполняется сортировка выбранных строк. Предполагаемая оценка затрат – 1.23 отражена в первой строке. Это время, которое потребуется для того, чтобы приступить к выводу отсортированных строк. Время на саму сортировку равно разнице между 1.23 и временем последовательного сканирования 1.09. Когда таблица маленькая, то обращения к индексу не дает выигрыша в скорости.

Рассмотрим пример плана с параллельным последовательным сканированием:

EXPLAIN SELECT count(\*)

FROM tickets;

QUERY PLAN

----------------------------------------------------------------------------------------

Finalize Aggregate (cost=65778.30..65778.31 rows=1 width=8)

-> Gather (cost=65778.09..65778.30 rows=2 width=8)

Workers Planned: 2

-> Partial Aggregate (cost=64778.09..64778.10 rows=1 width=8)

-> Parallel Seq Scan on tickets (cost=0.00..61705.47 rows=1229047 width=0)

(5 rows)

Все, что находится ниже узла Gather — параллельная часть плана. Она выполняется в каждом из рабочих процессов (которых запланировано два) и, возможно, в ведущем процессе.

Узел Gather и все предшествующие выполняются только в ведущем процессе. Это последовательная часть плана. Узел Parallel Seq Scan представляет сканирование таблицы в параллельном режиме. В поле rows показана оценка числа строк, которые обработает один рабочий процесс. Всего их запланировано 2, и еще часть работы выполнит ведущий.

4. Сбор статистики о работе PostgreSQL

PostgreSQL предлагает обширный набор системных представлений, позволяющих администраторам и разработчикам глубже изучить, что происходит внутри системы.

4.1. Динамическая проверка выполняемых запросов

Приступать к инспекции системы желательно с понимания того, что сейчас происходит. То есть с системного представления **pg\_stat\_activity**. Его идея – показать, что происходит прямо сейчас.

demo=# \d pg\_stat\_activity

View "pg\_catalog.pg\_stat\_activity"

Column | Type | Collation | Nullable | Default

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

datid | oid | | |

datname | name | | |

pid | integer | | |

leader\_pid | integer | | |

usesysid | oid | | |

usename | name | | |

application\_name | text | | |

client\_addr | inet | | |

client\_hostname | text | | |

client\_port | integer | | |

backend\_start | timestamp with time zone | | |

xact\_start | timestamp with time zone | | |

query\_start | timestamp with time zone | | |

state\_change | timestamp with time zone | | |

wait\_event\_type | text | | |

wait\_event | text | | |

state | text | | |

backend\_xid | xid | | |

backend\_xmin | xid | | |

query\_id | bigint | | |

query | text | | |

backend\_type | text | | |

Представление pg\_stat\_activity содержит одну строку на каждое активное соединение. Вы видите идентификатор объекта базы данных (datid), имя базы данных, с которой установлено соединение, и идентификатор процесса, обслуживающего это соединение (pid). PostgreSQL также сообщает, кто именно подключился к базе (usename) и идентификатор внутреннего объекта этого пользователя (usesysid). Далее следует поле application\_name.

Следующие три столбца (client\_) позволяют узнать, откуда было открыто соединение. PostgreSQL показывает IP-адреса и (если это задано в конфигурации) даже имена хостов.

Поле backend\_start сообщает, когда было открыто соединение, а xact\_start – когда начата транзакция. Имеются также поля query\_start и state\_change. Поле state\_change позволяет определить длительность запросов.

4.2. Получение информации о базах данных

Изучив активные соединения с сервером, можно «копнуть глубже» и поинтересоваться статистикой на уровне отдельных баз данных. Представление **pg\_stat\_database** возвращает по одной строке на каждую базу данных, обслуживаемую экземпляром PostgreSQL.

demo=# \d pg\_stat\_database

View "pg\_catalog.pg\_stat\_database"

Column | Type | Collation | Nullable | Default

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

datid | oid | | |

datname | name | | |

numbackends | integer | | |

xact\_commit | bigint | | |

xact\_rollback | bigint | | |

blks\_read | bigint | | |

blks\_hit | bigint | | |

tup\_returned | bigint | | |

tup\_fetched | bigint | | |

tup\_inserted | bigint | | |

tup\_updated | bigint | | |

tup\_deleted | bigint | | |

conflicts | bigint | | |

temp\_files | bigint | | |

temp\_bytes | bigint | | |

deadlocks | bigint | | |

checksum\_failures | bigint | | |

checksum\_last\_failure | timestamp with time zone | | |

blk\_read\_time | double precision | | |

blk\_write\_time | double precision | | |

session\_time | double precision | | |

active\_time | double precision | | |

idle\_in\_transaction\_time | double precision | | |

sessions | bigint | | |

sessions\_abandoned | bigint | | |

sessions\_fatal | bigint | | |

sessions\_killed | bigint | | |

stats\_reset | timestamp with time zone | | |

После идентификатора и имени базы данных следует столбец numbackends, который показывает, сколько соединений с базой данных открыто в данный момент.

Далее идут столбцы xact\_commit и xact\_rollback. Они показывают, что приложение делает чаще: фиксацию или откат. Столбцы blks\_hit и blks\_read сообщают о количестве попаданий и промахов кэша. Имейте в виду, что речь, обычно, идет о кэше разделяемых буферов. Далее следуют столбцы temp\_files и temp\_bytes. Они весьма важны, потому что показывают, приходится ли системе создавать временные файлы на диске, что неизбежно замедляет работу. Причины использования временных файлов:

* неправильная настройка: если параметр work\_mem слишком мал, то нет возможности выполнять операции в оперативной памяти, так что PostgreSQL обращается к диску;
* «дурацкие» операции: довольно часто пользователи мучат систему дорогостоящими, но бессмысленными запросами. Если в OLTP-системе создается много временных файлов, поищите «тяжелые» запросы;
* индексирование и другие административные задачи: время от времени приходится строить индексы или выполнять DDL-команды. При этом могут создаваться временные файлы, но это необязательно признак проблемы (в большинстве случаев).

4.3. Получение информации о таблицах

В этом нам поможет системное представление **pg\_stat\_user\_tables**.

demo=# \d pg\_stat\_user\_tables

View "pg\_catalog.pg\_stat\_user\_tables"

Column | Type | Collation | Nullable | Default

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

relid | oid | | |

schemaname | name | | |

relname | name | | |

seq\_scan | bigint | | |

seq\_tup\_read | bigint | | |

idx\_scan | bigint | | |

idx\_tup\_fetch | bigint | | |

n\_tup\_ins | bigint | | |

n\_tup\_upd | bigint | | |

n\_tup\_del | bigint | | |

n\_tup\_hot\_upd | bigint | | |

n\_live\_tup | bigint | | |

n\_dead\_tup | bigint | | |

n\_mod\_since\_analyze | bigint | | |

n\_ins\_since\_vacuum | bigint | | |

last\_vacuum | timestamp with time zone | | |

last\_autovacuum | timestamp with time zone | | |

last\_analyze | timestamp with time zone | | |

last\_autoanalyze | timestamp with time zone | | |

vacuum\_count | bigint | | |

autovacuum\_count | bigint | | |

analyze\_count | bigint | | |

autoanalyze\_count | bigint | | |

**pg\_stat\_user\_tables** – одно из важнейших системных представлений. Для каждой таблицы имеется одна запись, в которой, в частности, присутствует количество последовательных просмотров таблицы (seq\_scan). А поле seq\_tup\_read говорит, сколько кортежей система прочитала информации во время этих просмотров. Это может помочь в решении проблем с производительностью.

Следующим в списке является поле idx\_scan, которое показывает, сколько раз для доступа к этой таблице использовался индекс. Кроме того, PostgreSQL сообщает, сколько строк было прочитано в процессе просмотра по индексу. Далее мы видим несколько полей с префиксом n\_tup\_. Они говорят, сколько было выполнено операций вставки, обновления и удаления. Самое важное из них – n\_tup\_hot\_upd. При выполнении команды UPDATE PostgreSQL должна скопировать строку, чтобы гарантировать правильную работу ROLLBACK. Горячее обновление (HOT UPDATE) хорошо тем, что новая строка остается в том же блоке. Большое количество горячих обновлений означает, что вы на правильном пути в ситуации, когда рабочая нагрузка включает много команд UPDATE. Невозможно сказать, каким должно быть идеальное соотношение между обычными и горячими обновлениями, все зависит от ситуации.

Наконец, имеется статистика работы VACUUM.

4.4. Изучение индексов

Представление **pg\_stat\_user\_tables** важно для поиска недостающих индексов, но иногда необходимо выявить индексы, которых на самом деле быть не должно. Для выявления бесполезных индексов есть представление **pg\_stat\_user\_indexes**:

demo=# \d pg\_stat\_user\_indexes

View "pg\_catalog.pg\_stat\_user\_indexes"

Column | Type | Collation | Nullable | Default

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

relid | oid | | |

indexrelid | oid | | |

schemaname | name | | |

relname | name | | |

indexrelname | name | | |

idx\_scan | bigint | | |

idx\_tup\_read | bigint | | |

idx\_tup\_fetch | bigint | | |

Это представление говорит, сколько раз использовался каждый индекс (idx\_scan). При этом указывается, над какой таблицей из какой схемы этот индекс построен.

4.5. Мониторинг фонового рабочего процесса

В этом разделе мы познакомимся со статистикой фонового процесса записи. Сервер обычно не записывает блоки данных на прямую. Это делает фоновый процесс записи или процесс контрольной точки. Понять, как записываются данные, поможет представление **pg\_stat\_bgwriter**:

demo=# \d pg\_stat\_bgwriter

View "pg\_catalog.pg\_stat\_bgwriter"

Column | Type | Collation | Nullable | Default

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

checkpoints\_timed | bigint | | |

checkpoints\_req | bigint | | |

checkpoint\_write\_time | double precision | | |

checkpoint\_sync\_time | double precision | | |

buffers\_checkpoint | bigint | | |

buffers\_clean | bigint | | |

maxwritten\_clean | bigint | | |

buffers\_backend | bigint | | |

buffers\_backend\_fsync | bigint | | |

buffers\_alloc | bigint | | |

stats\_reset | timestamp with time zone | | |

Прежде всего нужно обратить внимание на первые два столбца. PostgreSQL регулярно записывает контрольные точки. Это необходимо, чтобы гарантировать попадание данных на физический диск. О том, что интервал между контрольными точками слишком мал, расскажет поле checkpoint\_req (количество запрошенных контрольных точек). Слишком большая величина в нем может означать, что записывается очень много данных, что свидетельствует о высоком трафике. Кроме того, PostgreSQL расскажет, сколько времени потребовалось на запись данных во время контрольной точки и сколько времени ушло на синхронизацию. Поле buffers\_checkpoint говорит, сколько буферов было записано во время контрольной точки, а поле buffers\_clean – сколько их было записано фоновым процессом записи.

Но и это еще не все: поле maxwritten\_clean говорит, сколько раз фоновый процесс записи останавливал сброс грязных страниц на диск, потому что записал слишком много буферов.

Наконец, имеются поля buffers\_backend (сколько буферов записано самими обслуживающими процессами), buffers\_backend\_fsync (сколько раз обслуживающим процессам пришлось выполнять fsync самостоятельно) и buffers\_alloc (количество выделенных буферов). Вообще говоря, нехорошо, когда обслуживающий процесс начинает сам записывать буферы на диск.

4.6. Мониторинг, архивация и потоковая репликация

Первым делом нужно обратиться к представлению **pg\_stat\_archiver**, которое расскажет о процессе архивации, занятом перемещением журнала транзакций (WAL) с главного сервера на устройство резервного хранения:

demo=# \d pg\_stat\_archiver

View "pg\_catalog.pg\_stat\_archiver"

Column | Type | Collation | Nullable | Default

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

archived\_count | bigint | | |

last\_archived\_wal | text | | |

last\_archived\_time | timestamp with time zone | | |

failed\_count | bigint | | |

last\_failed\_wal | text | | |

last\_failed\_time | timestamp with time zone | | |

stats\_reset | timestamp with time zone | | |

Прежде всего мы видим, сколько файлов журнала транзакций было архивировано (archived\_count), а также какой файл был архивирован последним и когда (last\_archived\_wal и last\_achived\_time).

Наиболее востребованные поля это failed\_count, last\_failed\_wal и last\_failed\_time. Если при архивации журнала транзакций произошла ошибка, last\_failed\_wal сообщит имя последнего не архивированного файла, а поле last\_failed\_time – время ошибки. Рекомендуется следить за этими полями, потому что иначе архивация может остановиться, а вы об этом даже не узнаете.

Если в вашей системе работает потоковая репликация, то очень пригодится следующее представление. **pg\_stat\_replication**, содержит информацию о процессе потоковой репликации с главного сервера на резервный. Показывается по одной записи для каждого процесса отправки WAL. Если нет ни одной записи, то значит, что потоковая репликация журнала транзакций не производится.

4.7. Мониторинг очистки

Для того, чтобы следить за ходом очистки, имеется представление **pg\_stat\_progress\_vacuum**:

demo=# \d pg\_stat\_progress\_vacuum

View "pg\_catalog.pg\_stat\_progress\_vacuum"

Column | Type | Collation | Nullable | Default

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

pid | integer | | |

datid | oid | | |

datname | name | | |

relid | oid | | |

phase | text | | |

heap\_blks\_total | bigint | | |

heap\_blks\_scanned | bigint | | |

heap\_blks\_vacuumed | bigint | | |

index\_vacuum\_count | bigint | | |

max\_dead\_tuples | bigint | | |

num\_dead\_tuples | bigint | | |

Есть лишь две вещи, о которых следует помнить. Во-первых, процесс очистки нелинейный – он то ускоряется, то замедляется. Во-вторых, очистка обычно происходит быстро, поэтому проследить за ее ходом не всегда удается.

4.8. Использование pg\_stat\_statements

Одно из самых важных представлений является **pg\_stat\_statements**. Оно служит для получения информации о запросах в системе и помогает выделить медленные запросы и узнать, как часто запросы выполняются.

demo=# \d pg\_stat\_statements

View "bookings.pg\_stat\_statements"

Column | Type | Collation | Nullable | Default

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

userid | oid | | |

dbid | oid | | |

toplevel | boolean | | |

queryid | bigint | | |

query | text | | |

plans | bigint | | |

total\_plan\_time | double precision | | |

min\_plan\_time | double precision | | |

max\_plan\_time | double precision | | |

mean\_plan\_time | double precision | | |

stddev\_plan\_time | double precision | | |

calls | bigint | | |

total\_exec\_time | double precision | | |

min\_exec\_time | double precision | | |

max\_exec\_time | double precision | | |

mean\_exec\_time | double precision | | |

stddev\_exec\_time | double precision | | |

rows | bigint | | |

shared\_blks\_hit | bigint | | |

shared\_blks\_read | bigint | | |

shared\_blks\_dirtied | bigint | | |

shared\_blks\_written | bigint | | |

local\_blks\_hit | bigint | | |

local\_blks\_read | bigint | | |

local\_blks\_dirtied | bigint | | |

local\_blks\_written | bigint | | |

temp\_blks\_read | bigint | | |

temp\_blks\_written | bigint | | |

blk\_read\_time | double precision | | |

blk\_write\_time | double precision | | |

wal\_records | bigint | | |

wal\_fpi | bigint | | |

wal\_bytes | numeric | | |

Для каждого пользователя в каждой базе данных оно содержит по одной строке на запрос. По умолчанию отслеживается 5000 запросов (это можно изменить, задав параметр pg\_stat\_statements.max). Запросы отделены от параметров. Для каждого запроса PostgreSQL сообщает полное время выполнения, а также число вызовов. PostgreSQL также сообщает о кешировании в процессе выполнения запроса. Поля shared\_blks\_ говорят, сколько блоков прочитано из кеша (\_hit), а сколько из операционной системы (\_read).

Следующая группа полей относится к буферам. Локальные буферы – это блоки в памяти, выделенные конкретному подключению к базе данных.

Помимо всего прочего, PostgreSQL предоставляет информацию о вводе-выводе во временные файлы. Отметим, что временные файлы создаются при построении большого индекса или выполнении другой DDL-команды, требующей создания большого объекта. Но в OLTP-системе временные файлы – обычно зло, так как замедляют работу всей системы вследствие потенциальной конкуренции за диск. Большой объем ввода-вывода во временные файлы может указывать на нежелательные вещи:

1. неправильное задание параметра work\_mem (OLTP);
2. неоптимальная настройка maintenance\_work\_mem (DDL-команды);
3. запросы, которые вообще не следовало бы выполнять.

Заключение

Полный обзор можно найти в официальной документации по адресу

<https://www.postgresql.org/docs/14/monitoring-stats.html>

5. Практическое задание

5.1. Подготовительная работа

Для того, чтобы использовать представление **pg\_stat\_statements,** необходимо установить расширение, желательно, 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:

~$ sudo -i

[sudo] password for …:

root@…:~#

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

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

etc/postgresql/14/main/postgresql.conf

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

root@…:~# 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)

Заодно, в файле postgresql.conf включим мониторинг времени чтения/записи страниц данных (по умолчанию эта опция отключена):

track\_io\_timing = on

Сохраняем изменения и выходим из редактора. Перезапустим PostgreSQL:

root@…:~# 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

…

Служба активна. Для завершения работа с терминалом выполните команды:

root@alex:~# logout

alex@alex:~$ exit

5.2. Задание

1. Разобрать план выполнения следующих запросов:

* SELECT count(\*) FROM bookings;
* SELECT total\_amount FROM bookings   
  ORDER BY total\_amount DESC   
  LIMIT 1;

1. Необходимо собрать данные статистики для представлений, описанных в п. 4, и продемонстрировать соответствующие запросы.  
   **pg\_stat\_activity**:

* количество запросов к базе данных, которые выполняются в данный момент;
* количество активных запросов к базе данных, которые выполняются в данный момент;
* количество неактивных запросов внутри транзакции;
* количество клиентов, подключенных к БД. Группировать по IP адресам, пользователям и БД (datname).

1. **pg\_stat\_database**:
   * количество созданных временных файлов и общий объём данных, записанных во временные файлы (temp\_files, temp\_bytes);
   * время, затраченное на ввод-вывод (blk\_read\_time, blk\_write\_time);
   * количество зафиксированных и откатанных транзакций (xact\_commit, xact\_rollback);
   * количество строк, считанных из БД и возвращенных клиенту (tup\_returned, tup\_fetched).
2. **pg\_stat\_user\_tables**:

* поиск больших таблицы, которые часто просматриваются последовательно (seq\_scan, seq\_tup\_read, seq\_tup\_read/seq\_scan, scan);
* количество строк прочитанных, добавленных и т.д. (seq\_tup\_read, idx\_tup\_fetch, n\_tup\_ins, n\_tup\_upd, n\_tup\_del).

Для этого необходимо подключить не менее 20 клиентов к БД demo, либо войдя в psql в терминале, либо в pgAdmin или использовать клиентов, написанных на любом языке программирования. Для каждого соединения с БД организуйте цикл на 30 мин (используя временные задержки ~ 1 с), в котором будут выполняться «тяжёлые» запросы к разным таблицам, добавление, удаление записей (для добавления, изменения и удаления записей создайте таблицу в demo). Параллельно следует собирать статистику в файл, вызывая перечисленные выше представления 1 раз в минуту. Собрав статистику, необходимо ее проанализировать, отметить изменения и их прокомментировать.

Сброс статистики

Для сброса статистики используйте функции:

pg\_stat\_reset() – для текущей БД;

pg\_stat\_reset\_shared() – для кластера.

Примечание.

Рассмотрим, как сделать цикл запросов в psql.

1. Подключимся к БД demo:  
   psql -U <имя пользователя> -d demo
2. Ввести запрос и выполнить:  
   demo=# SELECT … \; SELECT 1;  
   Первый SELECT – это тестовый, который загружает БД, а второй – вспомогательный, предназначенный для сокращения вывода строк на экран. PostgreSQL выполнит эти два запроса, но выведет результат только последнего запроса. Чтобы передать две SQL – команды для выполнения в одной транзакции используется комбинация «\;».
3. Задать цикл с временем ожидания 1 сек после завершения предыдущей   
   SQL – команды:  
   demo=# \watch 1
4. Чтобы завершить цикл нажмите Ctrl+C.

Для INSERT, UPDATE и DELETE также, только необходимость использовать SELECT 1 отпадает.

Перейдем к сбору статистики.

1. Откроем еще один терминал и подключимся к БД demo. Либо pgAdmin откроем окно терминала (при выделенной БД demo).
2. Задаем файл, куда направлять вывод (например, для pg\_stat\_database):  
   demo=# \o ~/database.txt;
3. И вызовем представление:  
   demo=# SELECT <список полей> FROM pg\_stat\_database;
4. Задать цикл со временем ожидания 1 мин:   
   demo=# \watch 60
5. После сбора данных завершим цикл, нажав Ctrl+C.
6. Для возвращения к выводу на экран в текущем соединении выполните команду:  
   demo=# \o;

Данные будут накапливаться в файле. Далее файл, открыв программой типа Excel, преобразуйте в удобочитаемую таблицу.

Запросы можно подобрать в книге Е.П. Моргунова «PostgreSQL. Основы языка SQL». Вот некоторые запросы из книги:

1.

SELECT t.ticket\_no, t.passenger\_name, tf.flight\_id, tf.amount

FROM tickets t

JOIN ticket\_flights tf ON t.ticket\_no = tf.ticket\_no

ORDER BY t.ticket\_no;

2.

SELECT a.aircraft\_code,

a.model,

s.seat\_no,

s.fare\_conditions

FROM seats s

JOIN aircrafts a ON s.aircraft\_code = a.aircraft\_code

ORDER BY s.seat\_no;

6. Список использованных источников

1. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. - СПб.: БХВ-Петербург, 2018.
2. PostgreSQL 14 изнутри. / Е.В. Рогов; ДМК Пресс, 2022.
3. PostgreSQL 11. Мастерство разработки. Ганс-Юрген Шениг / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2019.