# Лабораторная работа № 10

**Настройка производительности - планы выполнения POSTGRESQL**

Вещи, которые могут помочь улучшить планы выполнения:

* Индексы
  + Исключите последовательное сканирование (Seq Scan), добавив индексы (если размер таблицы не мал)
  + При использовании многоколоночного индекса убедитесь, что вы обращаете внимание на порядок, в котором вы определяете включенные столбцы — [Дополнительная информация](https://infostart.ru/redirect.php?url=aHR0cDovL3VzZS10aGUtaW5kZXgtbHVrZS5jb20vc3FsL3doZXJlLWNsYXVzZS90aGUtZXF1YWxzLW9wZXJhdG9yL2NvbmNhdGVuYXRlZC1rZXlz)
  + Попробуйте использовать индексы, которые очень избирательны к часто используемым данным. Это сделает их использование более эффективным.
* Условие ГДЕ
  + Избегайте LIKE
  + Избегайте вызовов функций в условии WHERE
  + Избегайте больших условий IN()
* JOINы
  + При объединении таблиц попробуйте использовать простое выражение равенства в предложении ON (т.е. a.id = b.person\_id). Это позволяет использовать более эффективные методы объединения (т. Е. Hash Join, а не Nested Loop Join)
  + Преобразуйте подзапросы в операторы JOIN, когда это возможно, поскольку это обычно позволяет оптимизатору понять цель и, возможно, выбрать лучший план.
  + Правильно используйте СОЕДИНЕНИЯ: используете ли вы GROUP BY или DISTINCT только потому, что получаете дублирующиеся результаты? Это обычно указывает на неправильное использование JOIN и может привести к более высоким затратам
  + Если план выполнения использует Hash Join, он может быть очень медленным, если оценки размера таблицы неверны. Поэтому убедитесь, что статистика вашей таблицы точна
  + По возможности избегайте [коррелированных подзапросов](https://infostart.ru/redirect.php?url=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvQ29ycmVsYXRlZF9zdWJxdWVyeQ==); они могут значительно увеличить стоимость запроса
  + Используйте [EXISTS](https://infostart.ru/redirect.php?url=aHR0cDovL3d3dy5wb3N0Z3Jlc3FsLm9yZy9kb2NzL2N1cnJlbnQvc3RhdGljL2Z1bmN0aW9ucy1zdWJxdWVyeS5odG1s) при проверке существования строк на основе критерия, поскольку он подобен короткому замыканию (останавливает обработку, когда находит хотя бы одно совпадение)
* Общие рекомендации
  + Делайте больше с меньшими затратами; Процессор быстрее чем операции ввода/вывода (I/O)
  + Используйте [Common Table Expressions](https://infostart.ru/redirect.php?url=aHR0cDovL3d3dy5wb3N0Z3Jlc3FsLm9yZy9kb2NzL2N1cnJlbnQvc3RhdGljL3F1ZXJpZXMtd2l0aC5odG1s) и временные таблицы, когда вам нужно выполнить цепочечные запросы.
  + Избегайте операторов LOOP и предпочитайте операции SET
  + Избегайте COUNT (\*), поскольку PostgresSQL для этого выполняет сканирование таблиц ([только для версий <= 9.1](https://infostart.ru/redirect.php?url=aHR0cHM6Ly93aWtpLnBvc3RncmVzcWwub3JnL3dpa2kvRkFRI1doeV9pc18uMjJTRUxFQ1RfY291bnQuMjguMkEuMjlfRlJPTV9iaWd0YWJsZS4zQi4yMl9zbG93LjNG))
  + По возможности избегайте ORDER BY, DISTINCT, GROUP BY, UNION, поскольку это приводит к высоким начальным затратам
  + Ищите большую разницу между оценочными и фактическими строками в выражении *EXPLAIN*. Если счетчик сильно отличается, статистика таблицы может быть устаревшей, а PostgreSQL оценивает стоимость с использованием неточной статистики.

PostgreSQL создает план выполнения для каждого запроса. Поскольку результирующий план имеет решающее значение для производительности, в PostgreSQL имеется планировщик, выбирающий оптимальный вариант на основе оценки стоимости каждой из альтернатив. Вы можете использовать команду [EXPLAIN](https://www.postgresql.org/docs/14/sql-explain.html), чтобы увидеть, какой план создает планировщик для запроса.

Создадим таблицу для последующих тестовых примеров:

CREATE TABLE test\_table (field1 integer, field2 text);

Добавим данные в таблицу и выполним команду [ANALYZE](https://www.postgresql.org/docs/14/sql-analyze.html), чтобы создать статистику таблицы после добавления данных:

INSERT INTO test\_table

SELECT i, md5(random()::text)

FROM generate\_series(1, 1000000) AS i;

ANALYZE test\_table;

select \* from test\_table limit 20;

Вызовем команду EXPLAIN, чтобы посмотреть план запроса:

EXPLAIN SELECT \* FROM test\_table;

Результат:

QUERY PLAN

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

Seq Scan on test\_table (cost=0.00..18334.00 rows=1000000 width=37)

PostgreSQL может считывать данные из таблицы несколькими способами. Seq Scan означает, что PostgreSQL считывает данные последовательно, блок за блоком.

Параметр cost содержит два значения: предполагаемые начальные и общие затраты на исполнение запроса. Начальные затраты имеют значение 0.00. Это затраты до начала вывода, например, затраты на сортировку в операции сортировки. Предполагаемые общие затраты (18334.00) показывают затраты, необходимые для извлечения всех строк. Параметр cost не учитывает затраты на отправку результатов клиенту, поскольку планировщик не может на них повлиять. По умолчанию, за единицу измерения cost принимается стоимость последовательного доступа к странице seq\_page\_cost. Величина cost чаще всего говорит о потреблении ресурсов, а не о затраченном времени, хотя эти величины коррелируют между собой. Обратитесь к статье [Planner Cost Constants](https://www.postgresql.org/docs/14/runtime-config-query.html" \l "RUNTIME-CONFIG-QUERY-CONSTANTS" \t "_blank) за дополнительными сведениями.

Параметр rows показывает предполагаемое количество строк, которое возвращает Seq Scan.

Параметр width представляет собой предполагаемый средний размер выходной строки в байтах.

Когда вы вызываете EXPLAIN, PostgreSQL не выполняет запрос, вместо этого он создает предполагаемый план выполнения на основе доступной статистики. Фактический план может отличаться. Используйте параметр ANALYZE, чтобы выполнить запрос и получить актуальную статистику исполнения.

EXPLAIN (ANALYZE) SELECT \* FROM test\_table;

Результат:

QUERY PLAN

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

Seq Scan on test\_table

(cost=0.00..18334.00 rows=1000000 width=37)

(actual time=0.005..62.035 rows=1000000 loops=1)

Planning Time: 0.056 ms

Execution Time: 96.757 ms

Параметр actual time — время, затраченное на получение первой строки и всех строк, в миллисекундах.

Параметр rows — фактическое количество строк, которое возвращает Seq Scan.

Параметр loops показывает, сколько раз PostgreSQL полностью отсканировал таблицу, т.е. выполнил Seq Scan над всеми данными таблицы.

Execution Time — общее время выполнения запроса.

Команда EXPLAIN ANALYZE выполняет запросы. Если вам нужно получить статистику по модифицирующим запросам INSERT, DELETE или UPDATE и не изменять данные, используйте команды [BEGIN](https://www.postgresql.org/docs/14/sql-begin.html) и [ROLLBACK](https://www.postgresql.org/docs/14/sql-rollback.html), чтобы выполнить EXPLAIN ANALYZE внутри блока транзакции с последующим откатом:

BEGIN;

EXPLAIN (ANALYZE) INSERT INTO test\_table

SELECT i, md5(random()::text)

FROM generate\_series(1, 100) AS i;

ROLLBACK;

Использование кеша

Команда EXPLAIN позволяет получить информацию об использовании кеша. Для этой цели нужно указать в запросе параметр BUFFERS.

Чтобы получить корректные результаты, перезапустите ADPG для очистки буферного кеша перед тестированием BUFFERS. Для этого можно использовать действие [Reconfigure & Restart](https://docs.arenadata.io/ru/ADPG/current/how-to/cluster-management/reconfigure-restart.html).

Выполним команду EXPLAIN с параметрами ANALYZE и BUFFERS:

EXPLAIN (ANALYZE,BUFFERS) SELECT \* FROM test\_table;

Результат:

QUERY PLAN

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

Seq Scan on test\_table

(cost=0.00..18334.00 rows=1000000 width=37)

(actual time=1.400..626.291 rows=1000000 loops=1)

Buffers: shared read=8334

Planning:

Buffers: shared hit=47 read=14

Planning Time: 35.535 ms

Execution Time: 663.332 ms

PostgreSQL читает таблицу частями, называемыми блоками. Параметр Buffers: shared read показывает количество блоков, считанных с диска. Таблица полностью считана с диска, считано 8334 блока, так как кеш пуст.

Выполним этот же запрос:

EXPLAIN (ANALYZE,BUFFERS) SELECT \* FROM test\_table;

Результат:

QUERY PLAN

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

Seq Scan on test\_table

(cost=0.00..18334.00 rows=1000000 width=37)

(actual time=0.037..73.519 rows=1000000 loops=1)

Buffers: shared hit=32 read=8302

Planning Time: 0.033 ms

Execution Time: 108.673 ms

PostgreSQL добавил к результату параметр Buffers: shared hit. Buffers: shared hit — это количество блоков, считанных из кеша PostgreSQL. Если вы повторите этот запрос несколько раз, то увидите, что PostgreSQL с каждым новым выполнением будет брать все больше и больше данных из кеша. PostgreSQL также добавляет данные в кеш при каждом запросе. Так как чтение из кеша быстрее, чем с диска, значение Buffers: shared hit увеличивается, а Execution Time сокращается при каждом выполнении запроса.

Конфигурационный параметр shared\_buffers определяет размер кеша. Вы можете изменить его через ADCM. За дополнительной информацией обратитесь к статье [Конфигурационные параметры](https://docs.arenadata.io/ru/ADPG/current/references/adcm-config.html).

Использование индексов

Добавим в запрос выражение WHERE:

EXPLAIN SELECT \* FROM test\_table WHERE field1 > 750;

Результат:

QUERY PLAN

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

Seq Scan on test\_table (cost=0.00..20834.00 rows=999288 width=37)

Filter: (field1 > 750)

Таблица не имеет индексов, и PostgreSQL считывает каждую запись последовательно (Seq Scan).

Filter: (field1 > 750) означает, что к каждой записи применяется условие фильтрации field1 > 750. Если значение field1 отвечает условию, PostgreSQL добавляет соответствующую строку в результат.

Значение cost увеличилось. Предполагаемое количество строк в результате запроса (rows) уменьшилось из-за выражения WHERE.

Создадим индекс для field1 и выполним тот же запрос:

CREATE INDEX ON test\_table(field1);

EXPLAIN (ANALYZE) SELECT \* FROM test\_table WHERE field1 > 750;

Результат:

QUERY PLAN

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

Seq Scan on test\_table

(cost=0.00..20834.00 rows=999279 width=37)

(actual time=0.132..89.711 rows=999250 loops=1)

Filter: (field1 > 750)

Rows Removed by Filter: 750

Planning Time: 0.083 ms

Execution Time: 121.256 ms

Было отфильтровано только 750 строк, запрос вернул более 99.9% таблицы. Несмотря на созданный индекс, планировщик выбрал Seq Scan.

С помощью опции [enable\_seqscan](https://www.postgresql.org/docs/14/runtime-config-query.html" \t "_blank) запретим Seq Scan. Планировщик запросов будет вынужден использовать индекс.

SET enable\_seqscan TO off;

EXPLAIN (ANALYZE) SELECT \* FROM test\_table WHERE field1 > 750;

Результат:

QUERY PLAN

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

Index Scan using test\_table\_field1\_idx on test\_table

(cost=0.42..36794.81 rows=999279 width=37)

(actual time=0.054..162.518 rows=999250 loops=1)

Index Cond: (field1 > 750)

Planning Time: 0.087 ms

Execution Time: 195.522 ms

Index Scan показывает, что PostgreSQL использует индекс. Index Cond содержит условие для поиска строк.

PostgreSQL использовал индекс, но это увеличило время выполнения. В предыдущем примере планировщик выбрал лучший план без индекса.

Включим опцию enable\_seqscan и изменим запрос следующим образом:

SET enable\_seqscan TO on;

EXPLAIN (ANALYZE) SELECT \* FROM test\_table WHERE field1 < 750;

Результат:

QUERY PLAN

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

Index Scan using test\_table\_field1\_idx on test\_table

(cost=0.42..31.02 rows=720 width=37)

(actual time=0.031..0.148 rows=749 loops=1)

Index Cond: (field1 < 750)

Planning Time: 0.074 ms

Execution Time: 0.182 ms

Наилучший план использует индекс test\_table\_field1\_idx.

Добавим индекс для field2. Поскольку наша база данных хранит текстовые значения в кодировке UTF8, создадим индекс с помощью класса операторов [text\_pattern\_ops](https://www.postgresql.org/docs/14/indexes-opclass.html" \t "_blank).

CREATE INDEX ON test\_table(field2 text\_pattern\_ops);

Добавим в запрос условие для текстового поля:

EXPLAIN (ANALYZE) SELECT \* FROM test\_table WHERE field2 LIKE 'a%';

Результат:

QUERY PLAN

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

Bitmap Heap Scan on test\_table

(cost=2711.24..11912.67 rows=70714 width=37)

(actual time=7.672..25.626 rows=62382 loops=1)

Filter: (field2 ~~ 'a%'::text)

Heap Blocks: exact=8332

-> Bitmap Index Scan on test\_table\_field2\_idx1

(cost=0.00..2693.57 rows=69314 width=0)

(actual time=6.343..6.344 rows=62382 loops=1)

Index Cond: ((field2 ~>=~ 'a'::text) AND (field2 ~<~ 'b'::text))

Planning Time: 0.097 ms

Execution Time: 28.520 ms

В выводе команды EXPLAIN знак -> и отступы отражают иерархическую структуру дерева плана.

Bitmap Index Scan собирает битовую карту с позициями записей, соответствующих условию. Далее PostgreSQL в рамках оператора Bitmap Heap Scan последовательно сканирует целевую таблицу по этим позициям.

Bitmap Heap Scan эффективно работает с большим количеством строк, попадающих под условие.

PostgreSQL также поддерживает операцию Index Only Scan, которая используется, если запрос ссылается только на индексированные столбцы. Выполним запрос, который возвращает одно индексированное поле:

EXPLAIN SELECT field1 FROM test\_table WHERE field1 < 750;

Результат:

QUERY PLAN

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

Index Only Scan using test\_table\_field1\_idx on test\_table

(cost=0.42..29.69 rows=758 width=4)

Index Cond: (field1 < 750)

Index Only Scan работает быстрее, чем Index Scan, так как Index Only Scan не требует чтения всей строки (width=4), за исключением случая проверки видимости индексной записи.

Индексы повышают производительность различных операций, включая сортировку. Рассмотрим выражение ORDER BY.

Удалим индекс для field1:

DROP INDEX test\_table\_field1\_idx;

Отсортируем таблицу по field1:

EXPLAIN (ANALYZE) SELECT \* FROM test\_table ORDER BY field1;

Результат:

QUERY PLAN

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

Gather Merge (cost=63795.04..161033.70 rows=833416 width=37)

(actual time=94.311..273.651 rows=1000000 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Sort (cost=62795.02..63836.79 rows=416708 width=37)

(actual time=91103..127.215 rows=333367 loops=3)

Sort Key: field1

Sort Method: external merge Disk: 13568kB

Worker 0: Sort Method: external merge Disk: 12040kB

Worker 1: Sort Method: external merge Disk: 20504kB

-> Parallel Seq Scan on test\_table

(cost=0.00..12502.08 rows=416708 width=37)

(actual time=0.008..29.574 rows=333367 loops=3)

Planning Time: 0.099 ms

Execution Time: 316.971 ms

Когда планировщик определяет, что [параллельный запрос](https://www.postgresql.org/docs/14/how-parallel-query-works.html) является самой быстрой стратегией выполнения, он создает план, который включает ноду Gather или Gather Merge (в нашем случае Gather Merge). Эта нода содержит поддерево плана выполнения, который обрабатывается параллельно несколькими процессами.

Первая операция — Parallel Seq Scan с таблицей test\_table. Разные последовательные блоки таблицы параллельно сканируются процессами Worker 0 и Worker 1.

После этого PostgreSQL сортирует результаты. Sort Key содержит условие сортировки field1. Метод сортировки external merge Disk означает, что система создает на диске временные файлы.

План выполнения запроса, приведенный выше, содержит нескольких операций и выглядит ресурсозатратным.

Создадим индекс для field1 и выполним тот же запрос:

CREATE INDEX ON test\_table(field1);

EXPLAIN (ANALYZE) SELECT \* FROM test\_table ORDER BY field1;

Результат:

QUERY PLAN

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

Index Scan using test\_table\_field1\_idx on test\_table

(cost=0.42..34319.93 rows=1000100 width=37)

(actual time=0.018..106.251 rows=1000100 loops=1)

Planning Time: 0.140 ms

Execution Time: 148.662 ms

С индексом план выполнения включает только операцию Index Scan, и, как следствие, время выполнения запроса сокращается.

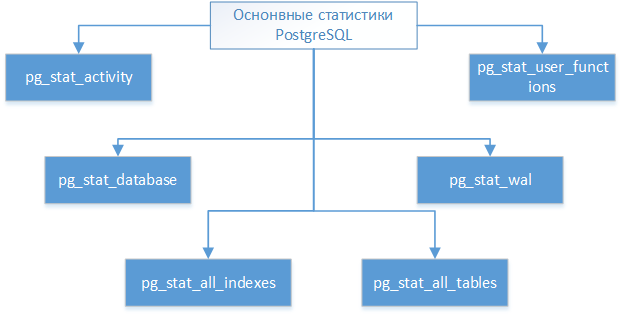
Анализ статистики для мониторинга PostgreSQL

Введение

Для эффективной работы системы важно отслеживать основные показатели производительности её базы данных. В PostgreSQL существует готовая подсистема, которая осуществляет сбор информации о работе сервера БД, а также предоставляет к ней доступ в виде готовых представлений и функций. Регулярный анализ собранной статистики может помочь администратору и разработчику найти наиболее уязвимые места при ежедневном эксплуатировании базы данных. В этой статье я расскажу об основных статистиках, которые накапливаются в PostgreSQL, в качестве примеров я приведу SQL-запросы на выборку основных показателей работы базы данных, а также затрону встроенные инструменты визуализации статистики в современных GUI для PostgreSQL.

Базовые статистики для анализа работы БД

В PostgreSQL сборщик статистики предоставляет доступ к накопленным данным через предопределённые представления. Основные из них показаны ниже на схеме:



* pg\_stat\_activity – предоставление информации о текущей активности серверного процесса, включая его состояние и текущий запрос;
* pg\_stat\_database – представление статистических данных по каждой базе данных на сервере;
* pg\_stat\_all\_indexes – представление статистических данных по каждому индексу;
* pg\_stat\_all\_tables – представление статистических данных по каждой таблице;
* pg\_stat\_wal – представление статистических данных о работе журнала предзаписи (Write-Ahead Logging) в кластере;
* pg\_stat\_user\_functions – представление статистических данных о выполнении функций.

На основе данных из представлений пользователь может анализировать полезные показатели, характеризующие работу базы данных.

Примеры запросов для анализа основных показателей БД

**Нагрузка на базу данных**

Для понимания нагрузки на базу данных хорошо знать общий объём транзакций за определенный период времени. Для получения этих данных можно использовать следующий запрос:

**SELECT**

datname,

xact\_commit + xact\_rollback ,

stats\_reset

**FROM** pg\_stat\_database;

Здесь сумма xact\_commit и количество xact\_rollback – суммарное количество транзакций за период с момента сброса статистических данных stats\_reset.

**Распределение серверных процессов по состояниям**

Для мониторинга клиентских подключений можно использовать представление pg\_stat\_activity, которое отображает информацию по работе серверных процессов. Каждый серверный процесс может находиться в следующих состояниях:

* active - выполнение запроса;
* idle - ожидание новой команды от клиента;
* idle in transaction - серверный процесс находится внутри транзакции, но в настоящее время не выполняет никакой запрос;
* idle in transaction (aborted) - серверный процесс находится внутри транзакции, но один из операторов в транзакции вызывал ошибку;
* fastpath function call - выполнение fast-path функции;
* disabled – у серверного процесса отключён параметр track\_activities.

Получить общее количество соединений по состояниям позволяет следующая группировка:

**Длительность текущих активных транзакций и запросов**

Длительность текущих активных транзакций и запросов можно проанализировать, выполнив запрос:

**SELECT**

datname,

usename,

**now**() - xact\_start **AS** TransactionDuration,

**now**() - query\_start **as** QueryDuration

**FROM** pg\_stat\_activity

**WHERE** state = 'active';

**Определение наиболее нагруженных таблиц**

Статистику обращений к таблицам базы данных предоставляет pg\_stat\_all\_tables. Представление позволяет оценить, например, общий объём insert, update, delate операций к таблице. Определить наиболее часто используемые таблицы в БД можно с помощью запроса:

**SELECT**

relname,

n\_tup\_upd+n\_tup\_ins+n\_tup\_del **AS** operationsAmount

**FROM** pg\_stat\_all\_tables

**ORDER** **BY** operationsAmount **DESC**;

**Отношение сканирований по индексам к последовательным сканированиям**

Для анализа эффективности чтения данных в конкретной таблице можно получить соотношение запросов, выполненных с использованием индексов к количеству запросов, читающих данные путём последовательного сканирования таблиц. Отсортированный список таблиц по данному соотношению вернёт следующий запрос:

**SELECT**

relname,

seq\_scan,

idx\_scan,

idx\_scan/seq\_scan **as** IndexStat

**FROM** pg\_stat\_all\_tables

**WHERE** seq\_scan <> 0

**ORDER** **BY** IndexStat **DESC**;

**Отслеживание устаревших индексов**

Полную информацию по созданным в базе данных индексам содержит представление pg\_stat\_all\_indexes. Устаревшие индексы можно обнаружить с помощью запроса:

**SELECT**

indexrelname,

relname,

idx\_tup\_read/idx\_tup\_fetch **as** stats

**FROM** pg\_stat\_all\_indexes

**WHERE** idx\_tup\_fetch <> 0

**ORDER** **BY** stats **DESC**;

Здесь idx\_tup\_read/idx\_tup\_fetch – это отношение записей из индекса, возвращённых в запросах по этому индексу, к общему числу записей, для которых пришлось обращаться к родительским таблицам. Если этот коэффициент меньше единицы, значит много данных читается в обход индекса, поэтому его необходимо обновить.

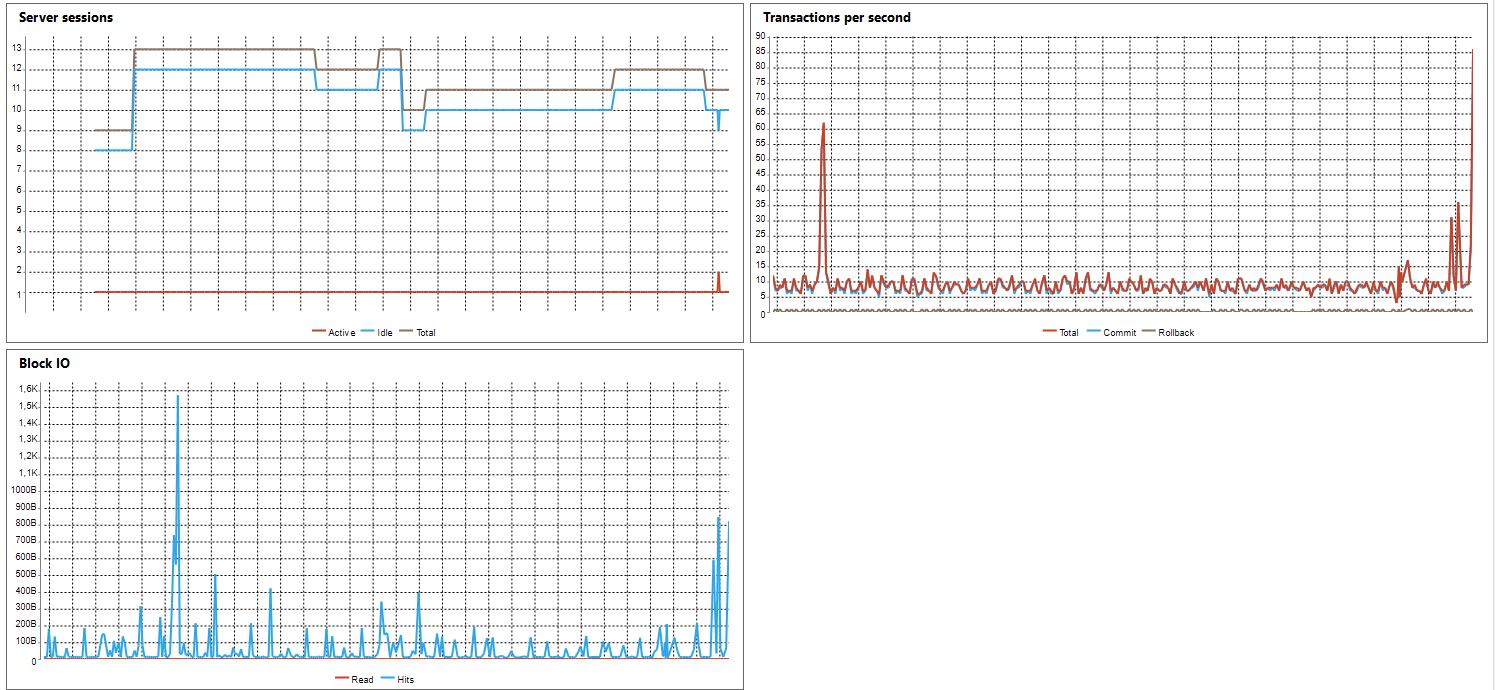
Мониторинг в GUI-клиентах

Для администраторов баз данных и программистам важно быстро определять производительность СУБД, анализировать проблемы с дисковым пространством, контролировать количество подключений и т.д. Поэтому современные GUI-клиенты предлагают собственные панели мониторинга для СУБД. Наиболее развитые системы мониторинга существуют в следующих графических приложениях:

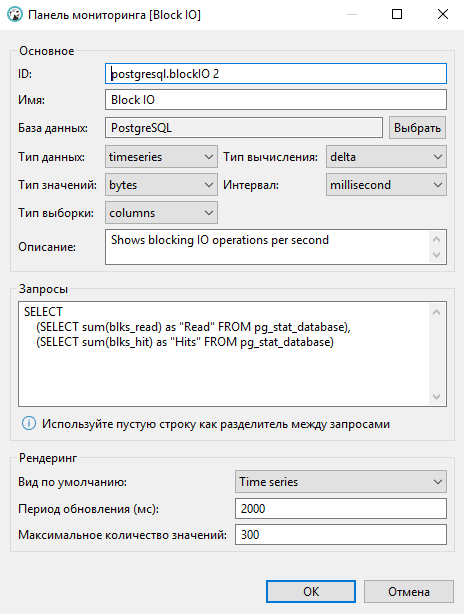
| Название | Описание | OC | Фишки мониторинга |
| --- | --- | --- | --- |
| pgAdmin | Бесплатный GUI-клиент с открытым исходным кодом | Linux, Windows, macOS | Предопределенный дашборд (серверные сеансы, блокировки, транзакции в секунду; операции с записями), отображение дополнительной статистики |
| DBeaver | Бесплатный GUI-клиент с дополнительными платными версиями. | Linux, Windows, macOS | Предопределенный дашборд (серверные сеансы, блокировки, транзакции в секунду), возможность доработки дашбордов на базе предопределённых дашбордов. |

В DBeaver используется функция Dashboard, с помощью которой можно настроить пользовательские панели из соответствующих виджетов для мониторинга. По умолчанию в DBeaver поставляются виджеты для нескольких предопределенных информационных панелей мониторинга. Создать информационную панель можно из набора готовых шаблонов виджетов:

* Server sessions - показывает активные/неактивные сеансы сервера;
* Transactions per second – отображает количество транзакций в секунду;
* Block IO – отображает количество блокировок при операциях ввода/вывода в секунду.

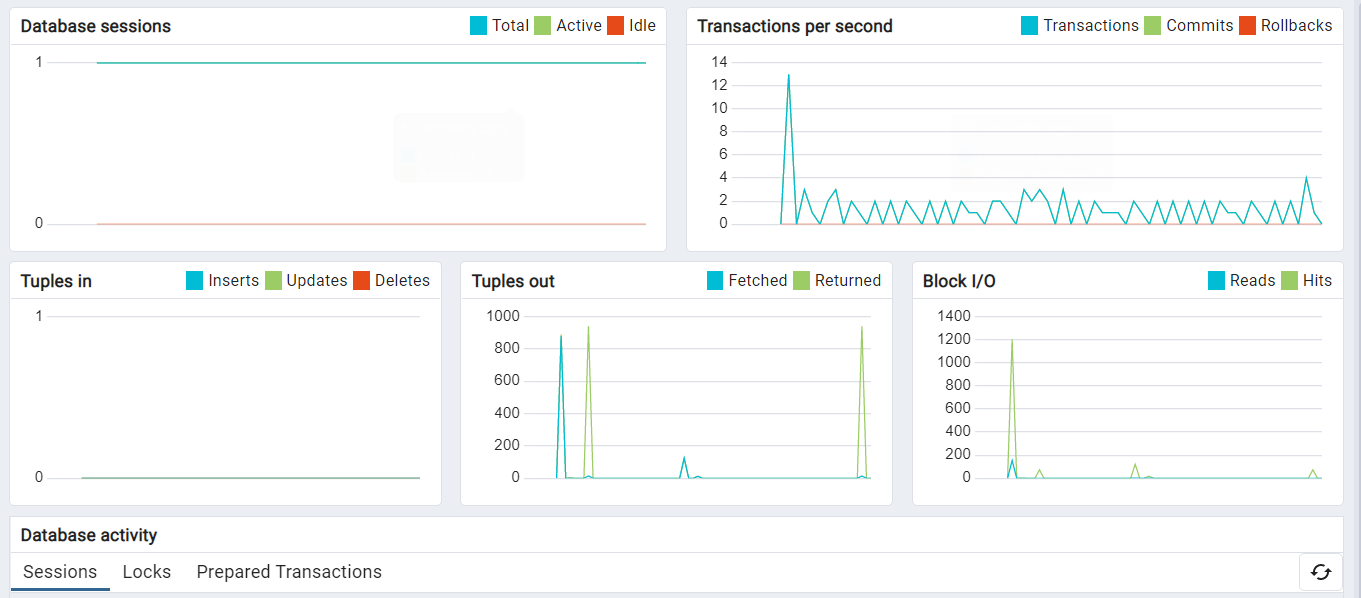


Предопределённые виджеты изменять в DBeaver нельзя, однако можно создать новый виджет путём копирования шаблона. В диалоговом окне можно настроить новые параметры шаблона и реализовать свой запрос для получения данных виджета:



Аналогичные шаблоны также реализованы и в клиенте pgAdmin, однако дополнительно в нём присутствуют следующие виджеты:

* Tuples in – отображает количество записей, вставленных, обновленных и удаленных на сервере или в базе данных;
* Tupels out – отображает количество записей, полученных и возвращенных с сервера или базы данных.



Также в pgAdmin на вкладке "Статистика" отображается готовая сводная статистика для выбранного элемента из дерева БД.

* PID – ИД серверного процесса;
* User – имя пользователя;
* Database – название базы данных;
* Backends – число текущих клиентских подключений к БД;
* Backend start – время старта серверного процесса;
* Xact Committe

Для пользователя доступны для отслеживания следующие показатели:d – число зафиксированных транзакций за последнюю неделю;

* Xact Rolled Back – число отменённых транзакций за последнюю неделю;
* Blocks Read – число блоков, считанных с диска за последнюю неделю;
* Blocks Hit – число блоков, считанных из оперативной памяти за последнюю неделю;
* Tuples Returned – количество, возвращённых записей за последнюю неделю;
* Tuples Fetched – количество записей, выбранных за последнюю неделю;
* Tuples Inserted – количество вставленных записей в базе данных за последнюю неделю;
* Tuples Updated – количество обновлённых записей в базе данных за последнюю неделю;
* Tuples Deleted – количество удалённых записей в базе данных за последнюю неделю;
* Last statistics – дата и время последнего сброса накопленных статистик в базе данных.

Заключение

В PostgreSQL сборщик статистики предоставляет полноценную информацию о работе сервера, которую можно регулярно анализировать. Также дополнительно включив модуль pg\_stat\_statements, СУБД будет отслеживать статистику планирования и выполнения сервером всех операторов SQL. Отдельно стоит выделить базовые возможности GUI-клиентов по мониторингу сервера базы данных. Используя базовые представления статистики PostgreSQL, они предоставляют пользователю готовую информацию в виде дашбордов, что позволяет динамически отслеживать работу сервера и выявлять ошибки.

[Обучение](https://club.directum.ru/tag/320)[SQL](https://club.directum.ru/tag/894)[Анализ](https://club.directum.ru/tag/1399)[Администрирование](https://club.directum.ru/tag/1431)[Быстродействие](https://club.directum.ru/tag/1546)[PostgreSQL](https://club.directum.ru/tag/22139)[PostgresPRO](https://club.directum.ru/tag/72948)