

# Онлайн образование

• REC

Проверить, идет ли запись

Меня хорошо видно  
&& слышно?



Тема вебинара

# Углубленный анализ производительности. Профилирование. Оптимизация



**Коробков Виктор**

ООО «ИТ ИКС5 Технологии»

Telegram: @Korobkov\_Viktor



# Правила вебинара



Активно  
участвуем



Off-topic обсуждаем  
в Telegram



Задаем вопрос  
в чат или голосом



Вопросы вижу в чате,  
могу ответить не сразу

## Условные обозначения



Индивидуально



Время, необходимое  
на активность



Пишем в чат



Говорим голосом



Документ



Ответьте себе или  
задайте вопрос

# Маршрут вебинара

Анализ причин медленной работы

Тюнинг настроек

Планы запросов

Расширения



# Цели вебинара

После занятия вы сможете

1. Определять почему все медленно работает
2. Вносить изменения в структуру БД для улучшения производительности
3. Вносить изменения в настройки СУБД для улучшения производительности

# Смысл

## Зачем вам это уметь

1. Понимать узкие места и учитывать их при проектировании изначально. Ну или пытаться 😊
2. Знать когда это вопрос DBA, а когда разработчиков
3. Уметь «творчески» подходить к вопросу производительности



# 1 Анализ работы БД

# Итак оно «тупит», с чего начать?



# Итак оно «тупит», с чего начать?

Запросы?

Настройки?

Индексы?

WAL?

VACUM?

Checkpoint?



Off Topic! Сперва попробуйте очевидное

# Прежде чем заниматься оптимизацией

1. Посмотрите последний Commit в Git-е
2. Если есть ORM – последнюю миграцию
3. Изменение postgresql.conf



Естественно для этого весь код должен быть в git-е.

Postgresql.conf должен быть в git-е и накатываться на production сервера автоматически (если не управляется внешней системой).

Все изменения в хранимые процедуры и структуру БД должны вноситься скриптами которые версионируются.

# Прежде чем заниматься оптимизацией

Убедитесь что Postgresql установлен на Linux системе

- Работа со множеством процессов не принята в Windows
- NTFS не самая оптимальная ФС для работы Postgres, используется другой способ доступа к диску – другие планы запросов
- Антивирусная защита Windows



Но если надо – можно и на Windows.

Только учитывать разницу в планах запросов: dev, test и prod контуры должны быть на одной и той же ОС.

# Прежде чем заниматься оптимизацией

## Проверьте память

**huge pages** – подключить (при работе с большими объемами данных).

```
grep HUGETLB /boot/config -$(uname -r)
          CONFIG_CGROUP_HUGETLB=y CONFIG_HUGETLBFS=y CONFIG_HUGETLB_PAGE=y
grep Huge /proc/meminfo
```

```
# head -1 /var/lib/postgresql/14/main/postmaster.pid      3076
# grep ^VmPeak /proc/3076/status                         VmPeak: 4742563 kB
# echo $((4742563 / 2048 + 1))                          2316
# sysctl -w vm.nr_hugepages=2316
```

в postgresql.conf параметр huge\_page = try

# Прежде чем заниматься оптимизацией

**transparent\_hugepage** (THP прозрачные огромные страницы) - отключить.

```
cat /sys/kernel/mm/transparent_hugepage/enabled  
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

Для ОС CentOS 8 и Ubuntu 22.04:

1. Создать новый файл: sudo vi /etc/systemd/system/disable-thp.service
2. Занести в него информацию:

```
[Unit]  
Description=Disable Transparent Huge Pages
```

```
[Service]  
Type=oneshot  
ExecStart=/bin/sh -c 'echo never > /sys/kernel/mm/transparent_hugepage/enabled'  
ExecStart=/bin/sh -c 'echo never > /sys/kernel/mm/transparent_hugepage/defrag'
```

```
[Install]  
WantedBy=multi-user.target
```

**3. Перезагрузить:**

```
sudo systemctl daemon-reload  
sudo systemctl enable disable-thp.service  
sudo systemctl start disable-thp.service
```

<https://anotherboringtechblog.com/2024/02/disabling-thp-ubuntu-22-04/>



# Прежде чем заниматься оптимизацией

## Проверьте память

**swappiness** – определяет частоту сброса данных из RAM в SWAP (значение от 0 до 200). Для PostgreSQL рекомендуется от 1 до 5.

```
cat /proc/sys/vm/swappiness  
sysctl vm.swappiness=5  
echo 'vm.swappiness=5' >> /etc/sysctl.conf
```

# Прежде чем заниматься оптимизацией

## Проверьте загрузку оборудования

**top/htop/atop/iotop/pgtop** – нет смысла браться за оптимизацию, если загрузка процессора в среднем составляет 70% и выше при этом ядра загружены более менее равномерно и загрузка процессора не вырастала скачкообразно.

**iostat** – утилита для использования дисковых разделов (входит в пакет sysstat)

Ключи:

**-c** - отобразить только информацию об использовании процессора;

**-d** - отобразить только информацию об использовании устройств;

**-h** - выводить данные в отчёте в удобном для чтения формате;

**-k** - выводить статистику в килобайтах;

**-m** - выводить статистику в мегабайтах;

**-o JSON** - выводить статистику в формате JSON;

...

<https://losst.ru/opisanie-iostat-linux>



# Прежде чем заниматься оптимизацией

## Проверьте оборудование

- CPU**
  - для OLTP больше быстродействующих ядер,
  - для DWH процессоры с большим L3 кэшем
  - (полезно для параллельных запросов).
- RAM**
  - чем больше, тем лучше. ))
- Disk**
  - лучше SSD.

## Рекомендации:

1. Сохранять WAL файлы и данные на отдельных дисках.
2. Использовать отдельные табличные пространства и диски для индексов и данных (особенно для SATA дисков).
3. Отключить atime – время последнего доступа к файлу.

<https://www.enterprisedb.com/postgres-tutorials/introduction-postgresql-performance-tuning-and-optimization>

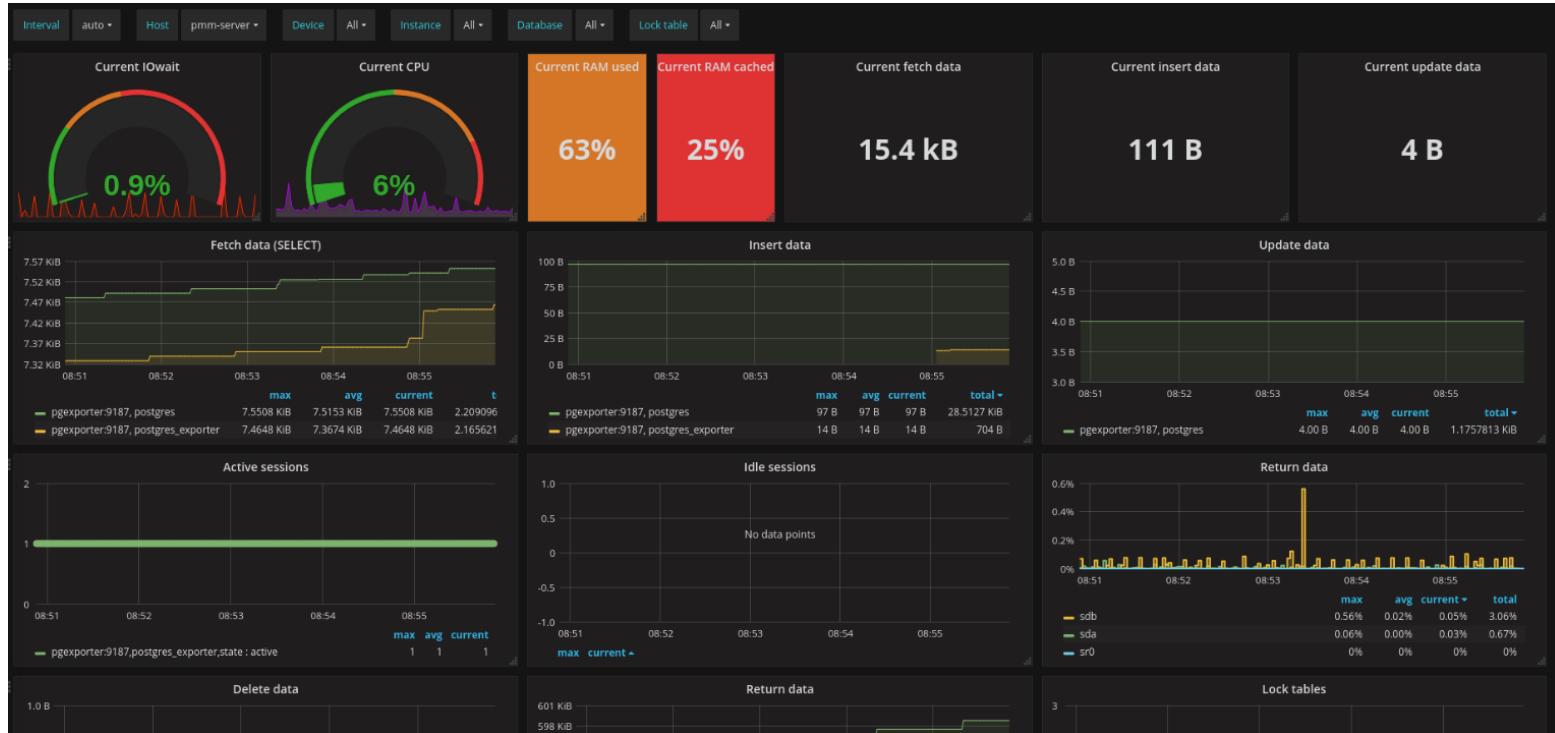
# Прежде чем заниматься оптимизацией

Воспользуйтесь Prometheus + Grafana



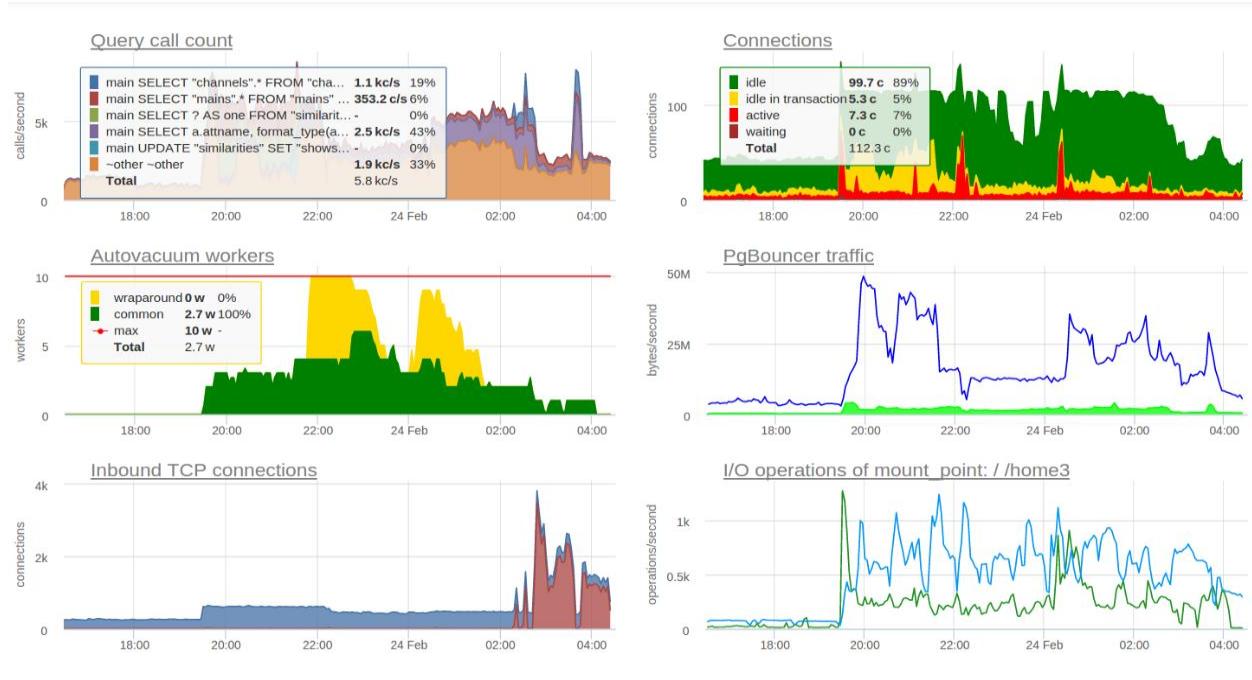
# Прежде чем заниматься оптимизацией

или Percona



# Прежде чем заниматься оптимизацией

## или Okmeter



© 2012–2015 OKmeter

<https://okmeter.io/>



# Прежде чем заниматься оптимизацией

Еще:

- <https://www.zabbix.com/ru> - железо, хотя при желании можно и Постгрес
- pgHero - <https://habr.com/ru/company/domclick/blog/546910/>
- pgWatch2 - <https://github.com/cybertec-postgresql/pgwatch2>



# Сбор статистики

# Представление pg\_stat\_activity

Самое популярное – «найти большого и убить» если кто-то написал что-то вроде

```
select * from * ☺
```

**Получаем активные запросы длительностью более 5 секунд:**

```
SELECT now() - query_start as "runtime", username, datname, wait_event_type, state,  
query FROM pg_stat_activity WHERE now() - query_start > '5 seconds'::interval and  
state='active' ORDER BY runtime DESC;
```

State = ‘idle’ тоже собственно вызывают подозрения.  
Но хуже всего – idle in transaction!

**Далее убиваем:**

```
SELECT pg_cancel_backend(procpid); для active
```

```
SELECT pg_terminate_backend(procpid); для idle
```

	column_name	data_type
	character varying	character varying
1	datid	oid
2	datname	name
3	pid	integer
4	usesysid	oid
5	username	name
6	application_name	text
7	client_addr	inet
8	client_hostname	text
9	client_port	integer
10	backend_start	timestamp with time...
11	xact_start	timestamp with time...
12	query_start	timestamp with time...
13	state_change	timestamp with time...
14	waiting	boolean
15	state	text
16	query	text



# Представление pg\_stat\_activity

«Повисшие транзакции» - зло

```
SELECT pid, xact_start, now() - xact_start AS duration  
FROM pg_stat_activity  
WHERE state LIKE '%transaction%'  
ORDER BY duration DESC;
```

В общем случае транзакции должны выполняться как можно меньшее время.  
Если что-то висит несколько часов, то это что-то не нормальное.  
Открытые транзакции влияют на работу VACUUM, WAL, репликацию.



# Представление pg\_stat\_user\_tables

Системное представление содержит данные о таблицах.

Большое зло – «последовательное чтение» больших таблиц.

```
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 25;
```

Сверху этого запроса будут таблицы в которых больше всего операций последовательного чтения. Они и являются подозрительными для анализа причин отсутствия индексов.

Можно ещё посмотреть кэширование этих таблиц по представлению pg\_statio\_user\_tables:  
колонки heap\_blks... и idx\_blks...

column_name	data_type
1	relid
2	schemaname
3	relname
4	seq_scan
5	seq_tup_read
6	idx_scan
7	idx_tup_fetch
8	n_tup_ins
9	n_tup_upd
10	n_tup_del
11	n_tup_hot_upd
12	n_live_tup
13	n_dead_tup
14	n_mod_since_analy...
15	last_vacuum
16	last_autovacuum
17	last_analyze
18	last_autoanalyze
19	vacuum_count
20	autovacuum_count
21	analyze_count
22	autoanalyze_count



# Анализ исторической нагрузки

## 1. Представление pg\_stat\_statements

```
create extension pg_stat_statements;
```

```
shared_preload_libraries = 'pg_stat_statements'
```

По умолчанию чаще всего не включено, но очень нужно.

По этому представлению можно увидеть не только выполняемые в данный момент запросы, но и уже выполненные, кроме того – с анализом их воздействия на сервер.

column_name name	data_type character varying
1 userid	oid
2 dbid	oid
3 queryid	bigint
4 query	text
5 calls	bigint
6 total_time	double precision
7 min_time	double precision
8 max_time	double precision
9 mean_time	double precision
10 stddev_time	double precision
11 rows	bigint
12 shared_blk_hit	bigint
13 shared_blk_read	bigint
14 shared_blk_dirtied	bigint
15 shared_blk_written	bigint
16 local_blk_hit	bigint
17 local_blk_read	bigint
18 local_blk_dirtied	bigint
19 local_blk_written	bigint
20 temp_blk_read	bigint
21 temp_blk_written	bigint
22 blk_read_time	double precision
23 blk_write_time	double precision



# Представление pg\_stat\_statements

## ТОП по загрузке CPU

```
SELECT    substring(query, 1, 50) AS short_query,  
         round(total_time::numeric, 2) AS total_time, calls, rows,  
         round(total_time::numeric / calls, 2) AS avg_time,  
         round((100 * total_time / sum(total_time::numeric) OVER ())::numeric, 2) AS percentage_cpu  
FROM pg_stat_statements ORDER BY total_time DESC LIMIT 20;
```

## ТОП по времени выполнения

```
SELECT    substring(query, 1, 100) AS short_query,  
         round(total_time::numeric, 2) AS total_time, calls, rows,  
         round(total_time::numeric / calls, 2) AS avg_time,  
         round((100 * total_time / sum(total_time::numeric) OVER ())::numeric, 2) AS percentage_cpu  
FROM pg_stat_statements ORDER BY avg_time DESC LIMIT 20;
```

# Анализ исторической нагрузки

## 2. Расширение pg\_profile

Позволяет создавать отчеты за указанный период по установленным базам данных.

Основано на представлениях статистики PostgreSQL и расширениях pg\_stat\_statements и pg\_stat\_kcache.

[https://github.com/zubkov-andrei/pg\\_profile](https://github.com/zubkov-andrei/pg_profile)

Создать базу для отчетов: create database reports;

Создать схему: CREATE SCHEMA profile;

Подключить расширения: CREATE EXTENSION dblink;

CREATE EXTENSION pg\_stat\_statements;

CREATE EXTENSION pg\_profile SCHEMA profile;



# Расширение pg\_profile

## Report sections

- [Server statistics](#)
  - [Database statistics](#)
  - [Session statistics by database](#)
  - [Statement statistics by database](#)
  - [JIT statistics by database](#)
  - [Cluster statistics](#)
  - [WAL statistics](#)
  - [Tablespace statistics](#)
  - [Wait sampling](#)
    - [Wait events types](#)
    - [Top wait events \(statements\)](#)
    - [Top wait events \(All\)](#)
- [SQL query statistics](#)
  - [Top SQL by elapsed time](#)
  - [Top SQL by planning time](#)
  - [Top SQL by execution time](#)
  - [Top SQL by executions](#)
  - [Top SQL by I/O wait time](#)
  - [Top SQL by shared blocks fetched](#)
  - [Top SQL by shared blocks read](#)
  - [Top SQL by shared blocks dirtied](#)
  - [Top SQL by shared blocks written](#)
  - [Top SQL by WAL size](#)
  - [Top SQL by temp usage](#)
  - [Top SQL by temp I/O time](#)
  - [Top SQL by JIT elapsed time](#)
  - [rusage statistics](#)
    - [Top SQL by system and user time](#)
    - [Top SQL by reads/writes done by filesystem layer](#)
- [Schema object statistics](#)
  - [Top tables by estimated sequentially scanned volume](#)
  - [Top tables by blocks fetched](#)
  - [Top tables by blocks read](#)
  - [Top DML tables](#)
  - [Top tables by updated/deleted tuples](#)
  - [Top growing tables](#)
  - [Top indexes by blocks fetched](#)
  - [Top indexes by blocks read](#)
  - [Top growing indexes](#)
- [User function statistics](#)
  - [Top functions by total time](#)

Database	Transactions			Block statistics			Block I/O times		Tuples					Temp files		Size	Growth
	Commits	Rollbacks	Deadlocks	Hit(%)	Read	Hit	Read	Write	Ret	Fet	Ins	Upd	Del	Size	Files		
csa	18			96.32	222	5804			17126	2204						8817 kB	
db10	18			96.37	223	5927	0.35		17468	2239						1970 MB	
db10old	18			96.37	223	5927			17468	2239						3428 MB	
db3	48	1		15.30	58662	10598	7.15		8075471	58159						9448 MB	

Database	I	Transactions			Block statistics			Tuples					Temp files		Size	Growth
		Commits	Rollbacks	Deadlocks	Hit(%)	Read	Hit	Ret	Fet	Ins	Upd	Del	Size	Files		
csa	1	30			100.00		6524	24650	2288						8817 kB	
	2	32			100.00		11886	31744	4380						8817 kB	
db10	1	30			100.00		6648	25190	2323						1970 MB	
	2	34			100.00		12217	33649	4464						1970 MB	
db10old	1	30			100.00		6648	25190	2323						3428 MB	
	2	32			100.00		12134	32362	4450						3428 MB	
db3	1	28			100.00		6931	25149	2415						9448 MB	
	2	11295136			100.00		16480	36931	6190						9448 MB	



# Анализ исторической нагрузки

3. pgBadger <https://github.com/darold/pgbadger>
4. PoWA <https://powa.readthedocs.io/en/latest/>
5. PoWA like <https://habr.com/ru/post/345370/>
6. pgFouine <https://highload.today/profilirovaniye-v-postgresql/>



# 3

# Тюнинг настроек



# Тюнинг настроек

1. Настройки памяти
  2. Настройки дискового пространства
  3. Настройки оптимизатора
- 
-

# Настройки памяти Postgres

С чего начать  
настройки памяти  
для Postgres?



# Настройки памяти Postgres

**Настройки памяти в зависимости от памяти сервера должны быть примерно такими:**

**effective\_cache\_size** = 2/3 RAM – определяет для планировщика ориентировочную оценку эффективного размера дискового кеша, доступного для одного запроса

**shared\_buffers** = RAM/4 – задаёт объём памяти, который будет использовать сервер баз данных для буферов в разделяемой памяти

`shared_buffers + effective_cache_size <= ОЗУ`



# Настройки памяти Postgres

**temp\_buffers** = 256МВ – задаёт максимальный объём памяти, выделяемой для временных буферов в каждом сеансе (по умолчанию 8 Мб).

**work\_mem** = RAM/32 или 128 – 256 Мб – задаёт максимальный объём памяти, который будет использоваться во внутренних операциях при обработке запросов, например, для сортировки или хеш-таблиц (по умолчанию 4 Мб).

**maintenance\_work\_mem** = RAM/16 или 256МБ .. 4Гб – задаёт максимальный объём памяти для операций обслуживания БД, в частности VACUUM, CREATE INDEX и ALTER TABLE ADD FOREIGN KEY (по умолчанию 64Мб).

`work_mem <= maintenance_work_mem`

# Настройки дисковой подсистемы

**fsync** – данные журнала принудительно сбрасываются на диск с кэша ОС.

**synchronous\_commit** – транзакция завершается только когда данные фактически сброшены на диск

**checkpoint\_completion\_target** – чем ближе к единице тем менее резкими будут скачки I/O при операциях checkpoint

**effective\_io\_concurrency** – число параллельных операций ввода/вывода (по количеству дисков, для SSD – несколько сотен)

**random\_page\_cost** – отношение рандомного чтения к последовательному.  
Рекомендуется для SSD дисков 1.1 – 1.25, для SAS дисков 1.25 - 2.

# Настройки оптимизатора

**join\_collapse\_limit** – сколько перестановок имеет смысл делать для поиска оптимального плана запроса

**default\_statistics\_target** – число записей просматриваемых при сборе статистики по таблицам. Чем больше тем тяжелее собрать статистику.

**track\_activity\_query\_size** – по умолчанию 1024, в современных системах этого в большинстве случаев недостаточно.

**geqo** – включает генетическую оптимизацию запросов

enable\_bitmapscan = on

enable\_hashjoin = on

enable\_indexscan = on

enable\_indexonlyscan = on

enable\_mergejoin = on

enable\_nestloop = on

enable\_seqscan = on

enable\_sort = on



# Настройки памяти Postgres

Но проще воспользоваться конфигураторами, которые посчитают и порекомендуют ещё ряд полезных настроек:

pgtune

– <https://pgtune.fariton.ru/>

pgconfig

– <https://www.pgconfig.org/>

pgtune.sainth

– <https://pgtune.sainth.de/>

pgconfigurator.cybertec

– <https://pgconfigurator.cybertec.at/>



# 4

# Оптимизация запросов

# **Планы запросов**

**У кого есть все нужные контуры  
окружения: DEV/TEST/STAGE/PROD?**

**И на каком контуре нужно отлаживать  
запросы?**

# Порядок выполнения оператора select

Структура:

- |             |   |
|-------------|---|
| 1. SELECT   | поля выборки  |
| 2. FROM     | источник выборки: таблица, представление, подзапрос |
| 3. WHERE    | условие выборки                                     |
| 4. GROUP BY | группировка данных                                  |
| 5. HAVING   | фильтры по группам                                  |
| 6. ORDER BY | сортировка  |

В какой последовательности выполняется ?



# Порядок выполнения оператора select

Структура:

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

```
(5) SELECT (5-2) DISTINCT (5-3) TOP(<top_specification>) (5-1) <select_list>
(1) FROM (1-J) <left_table> <join_type> JOIN <right_table> ON <on_predicate>
      | (1-A) <left_table> <apply_type> APPLY <right_table_expression> AS <alias>
      | (1-P) <left_table> PIVOT(<pivot_specification>) AS <alias>
      | (1-U) <left_table> UNPIVOT(<unpivot_specification>) AS <alias>
(2) WHERE <where_predicate>
(3) GROUP BY <group_by_specification>
(4) HAVING <having_predicate>
(6) ORDER BY <order_by_list>;
```

Порядок: 2, 3, 4, 5, 1, 6



# Планы запросов



## ОСНОВНЫЕ ОПЕРАТОРЫ ПЛАНА PG SQL

Оператор	Пояснение
<b>Seq Scan</b>	Последовательный перебор строк таблицы (возможно с отбором по условию)
<b>Index Only Scan</b>	Поиск по покрывающему индексу без захода в основную таблицу
<b>Index Scan</b>	Поиск по индексу, с заходом в основную таблицу за доп. колонками
<b>Nested Loops</b>	Соединение вложенными циклами
<b>Hash Join</b>	Соединение с помощью хеш-таблицы (Соответствия)
<b>Merge Join</b>	Соединение заранее отсортированных наборов с помощью алгоритма слияния
<b>Sort</b>	Сортировка УПОРЯДОЧИТЬ ПО
<b>Прочие</b>	Агрегаты СГРУППИРОВАТЬ ПО, ПЕРВЫЕ и пр.



# Работа с планом запросов в PostgreSQL

`EXPLAIN [ ( параметр [, ...] ) ] оператор`

- показывает план выполнения оператора

Параметры:

`ANALYZE`

- фактически выполняет оператор

`COSTS`

- стоимость выполнения оператора каждого элемента плана запроса

`TIMING`

- включает фактическое время запуска и время, затраченное на каждый узел

`SUMMARY`

- общая информация после выполнения запроса

`FORMAT { TEXT | XML | JSON | YAML }` - формат вывода плана запросов



# Планы запросов

## Самые частые ошибки оптимизатора:

- 1) Большое различие стоимости оценочного и фактического планов – пересобрать статистику.
- 2) Используется оператор Nested Loops с большой стоимостью и большим количеством строк – этот оператор должен соединять только небольшие выборки строк (примерно до 10 000).
- 3) Используется оператор Seq Scan и в запросе есть условие поиска WHERE – если по полю поиска нет индекса, рассмотреть возможность его добавления.

## Логика чтения плана запроса:

- 1) Смотрим на самый большой cost оператора
- 2) Это Seq Scan или nested loops?
- 3) Смотрим следующий по стоимости оператор

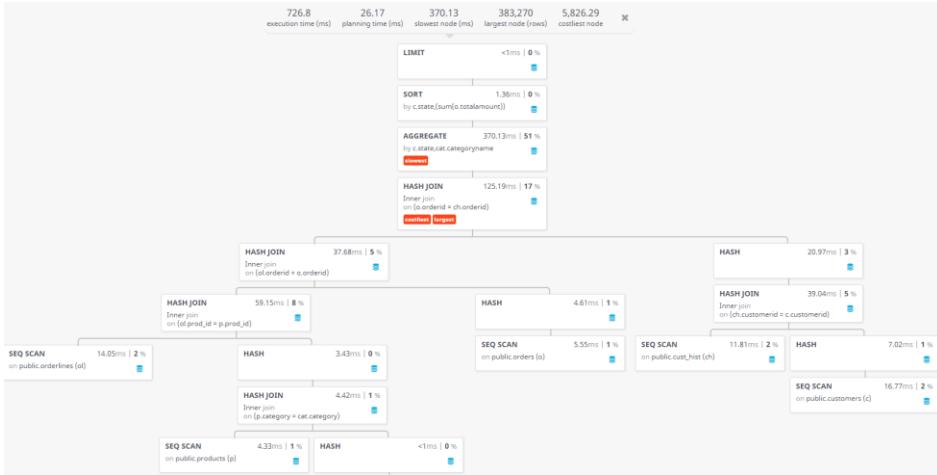
## Оптимизация чаще всего заканчивается

- 1) Добавлением индекса;
- 2) Упрощением запроса (разбиением на части, использованием вложенных таблиц и т.п.);
- 3) Обновлением статистики.



# Средства визуализации планов запросов

<https://tatiyants.com/pev>



<https://explain.tensor.ru/about/>

#	node, ms	tree, ms	rows	ratio	RRbF	node	Q	sh.ht
4	197.760		1		6 876 628	итоговые результаты		66 528
0	0.002	4 197.760	1	----		Limit		
1	0.003	4 197.758	1	1 237.0†		-> Nested Loop		
2	1 889.728	4 197.691	1	100 228.0†		-> Hash Join		
3	2 387.914	2 387.914	2 683 198	----	6 876 517	-> Seq Scan on public.pay_accounts p	Q	66 518
4	0.006	0.049	1	----		-> Hash		
5	0.043	0.843	1	----	28	-> Seq Scan on public.pay_gateways p1	Q	1
6	0.005	0.064	1	----		-> Materialize		
7	0.059	0.059	1	----	75	-> Seq Scan on public.pay_groups p0	Q	
	0.347	0.347				Planning Time		
	0.051	4 197.811				Execution Time		



# Явно повлиять на план запроса

**enable\_nestloop = off** – большие таблицы нельзя соединять вложенными циклами

**enable\_seqscan = off** – большие таблицы нельзя последовательно сканировать

В PostgresSQL нет HINT-ов, но если в это время «встал прод», то данные параметры могут на какое то время помочь – снизив общую производительность системы, но при этом исправив «больные» запросы.

В коммерческих версиях (например, Postgres Pro Enterprise 14) появился pg\_hint\_plan, который позволяет управлять планом выполнения запроса.



# Запросы LIKE

Каждый когда-нибудь писал запрос вида:

SELECT \* FROM Products WHERE name LIKE '%spoon%'

Ну или за вас это делала ORM ;)

The screenshot shows a software interface with a navigation bar at the top containing 'Главная', 'Печатные формы', 'История', 'Система', and 'Связанные документы'. Below the navigation bar is a search bar with the placeholder 'Наименование' and a dropdown arrow. To the right of the search bar are two columns: 'Кол-во' (Quantity) and 'Остаток' (Stock). A 'Справочник' (Dictionary) button is located in the top right corner of the search bar area.

Наименование	Кол-во	Остаток
238415 – 238415 Зебра полосатая	26	
238415 – 238415 Зебра полосатая (Большой, Белый)	0	
238415 – 238415 Зебра полосатая (Большой, Черный)	0	
238415 – 238415 Зебра полосатая (Малый, Белый)	0	
238415 – 238415 Зебра полосатая (Малый, Черный)	0	

At the bottom of the search results, there is a link 'Создать новый товар «Зебра полосатая»'.

Обычные индексы для подобных запросов крайне малоэффективны.

Что поделать если вы не можете позволить себе ElasticSearch :

Gin и GiST индексы – для полнотекстового поиска.

Create index on table using  
gin(column)

# Рекомендации по оптимизации запросов

- делать меньше запросов
- читать меньше данных
- обновлять меньше данных (несколько update insert в одну транзакцию)
- проанализировать передачу данных:
  - сколько данных было просканировано - сколько отослано
  - сколько было отослано - сколько использовано приложением
- не использовать UNION, когда можно UNION ALL
- использовать в выборке только нужные столбцы **select \***
- избегать distinct
- при использовании сложных индексов учитывать наличие в запросе первого столбца индекса
- избегать декартова произведения в джойнах
- используйте читабельный синтаксис SQL
- не используем русский язык в именах полей
- пишите комментарии



# 5 Расширения

# PgMemcache - <https://github.com/ohmu/pgmemcache>

## inMemory таблицы

Если нужно кэширование внутри СУБД, или временную таблицу в памяти, или нужно оперативно заменить таблицу на inmemory KV хранилище.

```
CREATE EXTENSION pgmemcache;
shared_preload_libraries = 'pgmemcache'
memcache_server_add('hostname:port'::TEXT)
```

```
memcache_add(key::TEXT, value::TEXT)
newval = memcache_decr(key::TEXT)
memcache_delete(key::TEXT)
```



# **cstore\_fdw** - [https://github.com/citusdata/cstore\\_fdw](https://github.com/citusdata/cstore_fdw)

## **Колоночные хранилища**

### **Достоинства:**

Выборки из огромных таблиц

Нет необходимости в нормализации

Сжатие данных

### **Недостатки:**

Нет Update и Delete

Insert только группами «insert into  
table select \* from source»

```
shared_preload_libraries = 'cstore_fdw'
```

```
CREATE EXTENSION cstore_fdw;
```

```
CREATE SERVER cstore_server FOREIGN DATA WRAPPER cstore_fdw;
```

```
CREATE FOREIGN TABLE table( ) SERVER cstore_server
```

```
OPTIONS(compression 'pglz')
```

```
INSERT INTO table SELECT * FROM sourcetable
```

# TimescaleDB - <https://github.com/timescale/timescaledb>

## Поддержка временных рядов

Создаёт таблицу секционированную по времени.

Применяется если нужно очень много писать данных во времени: системы мониторинга, биржевые системы...

```
CREATE EXTENSION timescaledb;  
shared_preload_libraries = 'timescaledb'
```

```
CREATE TABLE table (time TIMESTAMPTZ, value TEXT);  
SELECT create_hypertable('table', 'time');
```



# Рефлексия

# Рефлексия



Какие варианты оптимизации запомнили ?

А какие будете применять ???

**Заполните, пожалуйста,  
опрос о занятии  
по ссылке в чате**

Спасибо за внимание!

# Приходите на следующие вебинары



Коробков Виктор