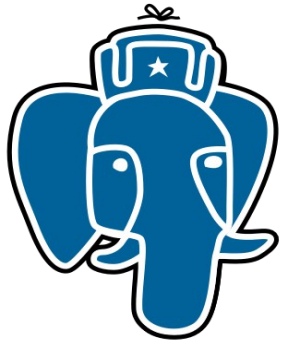


Вся правда об индексах в PostgreSQL (v4)

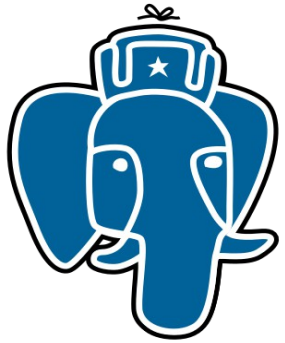
Олег Бартунов, Александр Коротков, Федор Сигаев

PostgreSQL Development Team

Олег Бартунов::Teodor Sigaev

- Locale support
- Extendability (indexing)
  - GiST, GIN, SP-GiST
- Extensions:
  - intarray
  - pg_trgm
  - ltree
  - hstore, hstore v2.0 → jsonb
  - plantuner
- Full Text Search (FTS)

https://www.facebook.com/oleg.bartunov
obartunov@gmail.com



Alexander Korotkov

- Indexed regexp search
- GIN compression & fast scan
- Fast GiST build
- Range types indexing
- Split for GiST

aekorotkov@gmail.com

PostgreSQL

Самая продвинутая из открытых СУБД в мире

PostgreSQL - это свободно распространяемая объектно-реляционная система управления базами данных (ORDBMS), наиболее развитая из открытых СУБД в мире и являющаяся реальной альтернативой коммерческим базам данных.

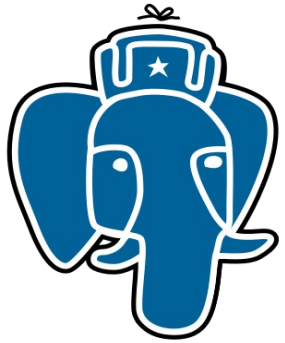
Важнейшие свойства PostgreSQL

- **Надежность и устойчивость PostgreSQL**
  Надежность PostgreSQL является известным и доказанным фактом на примере многих проектов, в которых PostgreSQL работает без единого сбоя и при больших загрузках на протяжении нескольких лет.
- **Превосходная поддержка**
  Сообщество PostgreSQL предоставляет очень квалифицированную и быструю помощь. Также, коммерческие компании предлагают свои услуги по всему миру.
- **Конкурентная работа при большой нагрузке**
  PostgreSQL использует многоверсионность (MVCC) для обеспечения надежной и быстрой работы в конкурентных условиях под большой нагрузкой.
- **Кроссплатформенность**
  PostgreSQL работает под всеми видами Unix, включая Linux, FreeBSD, Solaris, HPUX, Mac OS X, а также под MS Windows.
- **Расширяемость**
  PostgreSQL доступен с исходными кодами, что означает возможность добавления новой функциональности для вашего проекта без дополнительных проблем. Богатые возможности расширяемости PostgreSQL позволяет разрабатывать новые типы данных
- **Доступность**
  PostgreSQL распространяется под лицензией BSD, которая не накладывает никаких ограничений на коммерческое использование и не требует лицензионных выплат. Вы можете даже продавать PostgreSQL под своим именем !

Технические детали PostgreSQL

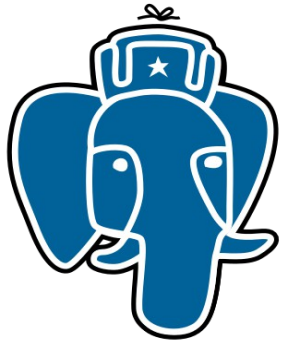
- Высокий уровень соответствия ANSI SQL 92, ANSI SQL 99 и ANSI SQL 2003, 2011
- Интерфейсы для Tcl, Perl, C, C++, PHP, ODBC, JDBC, Embedded SQL in C, Python, Ruby, Java, ...
- Развитые административные утилиты: PgAdmin, phpPgAdmin, Navicat
- View (materialized), sequences,inheritance,outer joins, subselects, referential integrity, window functions, CTE (WITH queries)
- Rules, triggers
- User defined functions, stored procedures
- Процедурные языки: PI/PgSQL, PI/Perl, PI/Python, PI/Tcl и другие.
- Расширяемый набор типов данных с поддержкой индексов (GiST, GIN, SP-GiST)
- Встроенная поддержка слабо-структурированных данных (xml, json,jsonb) с поддержкой индексов
- Горячее резервирование и репликация (синхронная, асинхронная), PITR
- Полная поддержка ACID, сериализация транзакций
- Функциональные и частичные индексы
- Интернационализация, поддержка Unicode и locale
- Загружаемые расширения, например, POSTGIS, hstore
- Поддержка SSL и Kerberos аутентификации
- Утилиты, облегчающие миграцию с других DBMS
- Foreign Data Wrappers (writable), поддержка всех основных баз данных

Имея более чем 20-летнюю историю развития, одно из самых больших и квалифицированных сообществ в мире, и отличную репутацию надежного и высококачественного продукта, PostgreSQL поможет вам выполнить ваш проект без проблем !



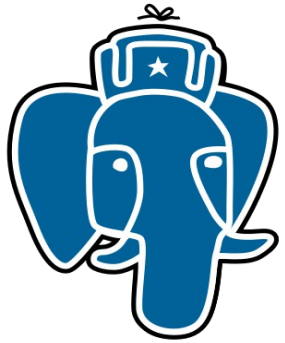
Индекс как «серебряная пуля»

the only weapon that is effective against a werewolf, witch, or other monsters.



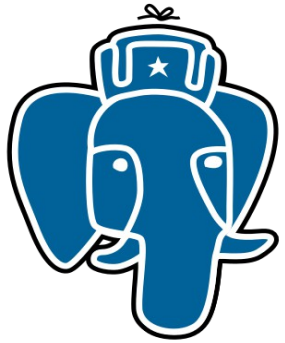
Индекс как «серебряная пуля»

- Индекс — это дополнительная структура данных (не SQL) для **ускорения** работы запросов.
- Результат запроса с индексом и без должны быть одинаковы !
- Индексы важны для
  - Поиска - обычное использование (KNN !)
  - Ограничений целостности (exclusion contraint!)
  - Сортировки,группировки, соединения таблиц
- Индексы не всегда полезны
  - Малая селективность, затраты на поддержание



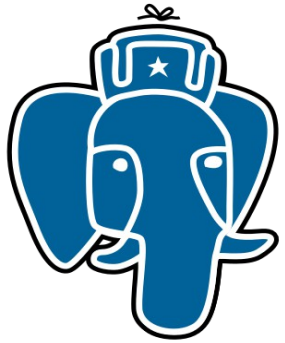
Индекс как «серебряная пуля»

- Разработчики приложения
  - SQL как язык — таблицы, представления, транзакции, ограничения, запросы, работа с данными
- Администраторы СУБД
  - Хранилище, бэкапы и восстановление, индексы, настройки, высокая доступность
  - Не знают запросов, а если узнают, то не могут их поменять.
- **Индексы должны быть заботой разрабочиков приложений !**
- **Мониторинг индексов остается администраторам**



Индекс в PostgreSQL

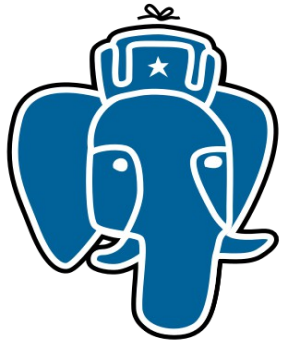
- Все индексы — вторичные, они отделены от таблицы. Вся информация о них содержится в системном каталоге
- Индексы связывают ключи и TID (tuple id - #page: offset)
- Индексы могут быть многоколончатые, порядок важен (не для GIN)
- MVCC: записи таблицы имеют версии (туплы), из которых только одна видна конкретной транзакции.
- Индексы не содержат информации о видимости
- Любое обновление записи в таблице приводит к появлению новой записи в индексе, index bloat.



Использование индекса при поиске

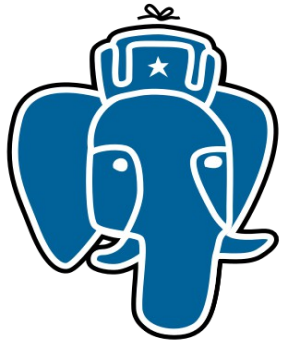
Индекс может использоваться для:

- фильтрации - WHERE expr opr value
- сортировки - ORDER BY expr [ASC|DESC]
- knn - ORDER BY expr opr value [ASC]



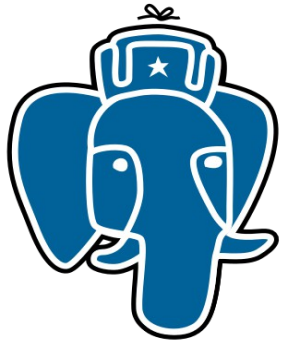
Условия для использования индекса

- Совпадают оператор и типы аргументов (порядок важен)
- Индекс валиден (например, concurrent index может быть не валиден)
- В многоколончатом индексе важен порядок (для GIN не важен)
- План с его использованием — оптимален (минимальная стоимость)
- Всю информацию постгрес берет из системного каталога



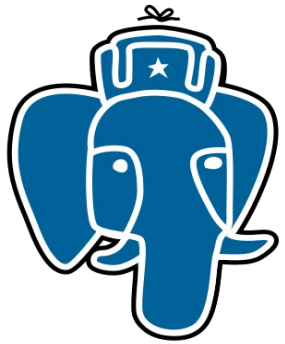
Выбор нужного индекса: пример

```
CREATE TABLE test3 AS (SELECT id, random() AS v, point(random(), random()) AS p
FROM generate_series(1,1000000) id);
ALTER TABLE test3 ADD PRIMARY KEY (id);
CREATE INDEX test3_p_idx ON test3 USING gist (p);
CREATE INDEX test3_v_id_idx ON test3 (v, id);

SELECT * FROM test3 WHERE p <@ box(point(0.5, 0.5), point(0.51, 0.51));
Bitmap Heap Scan on test3  (cost=48.03..2805.08 rows=1000 width=28)
                           (actual time=0.123..0.226 rows=98 loops=1)
   Recheck Cond: (p <@ '(0.51,0.51),(0.5,0.5)'::box)
   Heap Blocks: exact=95
   ->  Bitmap Index Scan on test3_p_idx  (cost=0.00..47.78 rows=1000 width=0)
                                         (actual time=0.102..0.102 rows=98 loops=1)
         Index Cond: (p <@ '(0.51,0.51),(0.5,0.5)'::box)
 Planning time: 0.111 ms
 Execution time: 0.532 ms
(7 rows)
```



Выбор нужного индекса: алгоритм

1.Поиск номера и типа данных столбца в pg_attribute по имени столбца и таблицы
2.Поиск oid оператора в pg_operator по имени оператора и типам операндов
3.Поиск подходящих индексов по таблице и номеру столбца
4.Поиск oid семейства операторов в pg_opclass по классам операторов подходящих индексов
5.Поиск поддерживаемых индексами операторов в pg_amop по oid семейства операторов и oid оператора
6.Индексы, которые поддерживают нужные операторы (найдены в pg_amop) могут быть использованы.



Выбор нужного индекса: схема

```
SELECT * FROM test3 WHERE p <@ box(point(0.5, 0.5), point(0.51, 0.51));
```

pg_operator

oid	oprname	oprleft	oprright
511	<@	600	603 (box)
512	<@	600	602
756	<@	600	604
...			

pg_attribute

attrelid	attname	attnum	atttypid
377315	id	1	23
377315	v	2	701
377315	p	3	600
...			

pg_amop

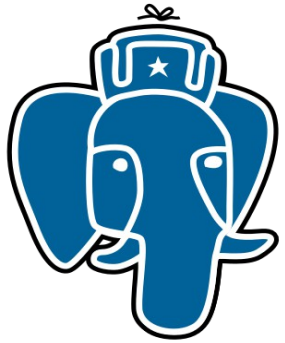
amopfamily	amopopr	amopstrategy
1029	511	28
4015	511	8
4016	511	8
...		

pg_opclass

opcfamily	opcname	oid
1970	float8_ops	3123
1976	int4_ops	1978
1029	point_ops	10068
...		

pg_index

indrelid	indkey	indclass
377315	1	1978
377315	3	10068
377315	2 1	3123 1978
...		



Что все эти «сканы» означают ?

Seq Scan on tbl (cost=0.00..17.50 rows=700 width=8)
  Filter: (a > 30)
  Rows Removed by Filter: 300

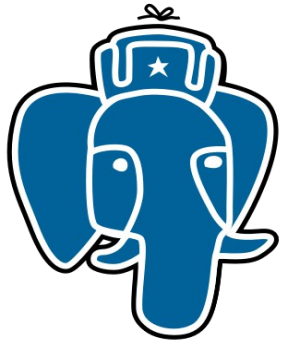
Index Only Scan using ab_idx on tbl (cost=0.28..6.03 rows=100 width=8)
  Index Cond: (a > 90)
  Heap Fetches: 0

update tbl set a=18;

Bitmap Heap Scan on tbl (cost=9.67..20.92 rows=180 width=8)
  Recheck Cond: (a > 90)
  -> Bitmap Index Scan on ab_idx (cost=0.00..9.63 rows=180 width=0)
      Index Cond: (a > 90)

set enable_indexonlyscan to off; set enable_bitmapscan to off;

Index Scan using ab_idx on tbl (cost=0.28..6.04 rows=1 width=8)
  Index Cond: (a > 90)

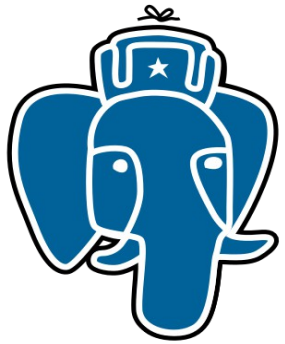


Sequential Scan

- Читаем последовательно таблицу и фильтруем записи по предикату

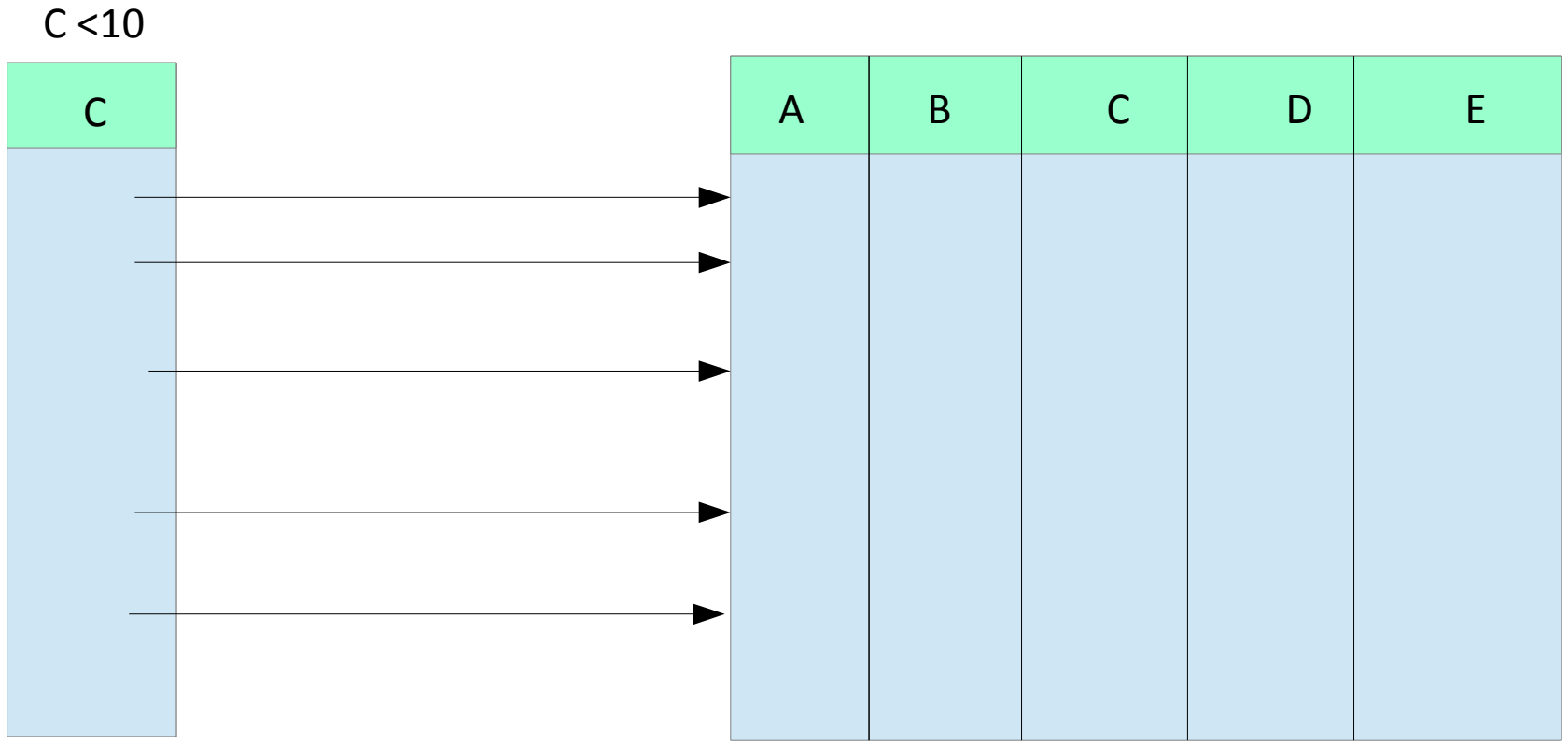
C <10

A	B	C	D	E



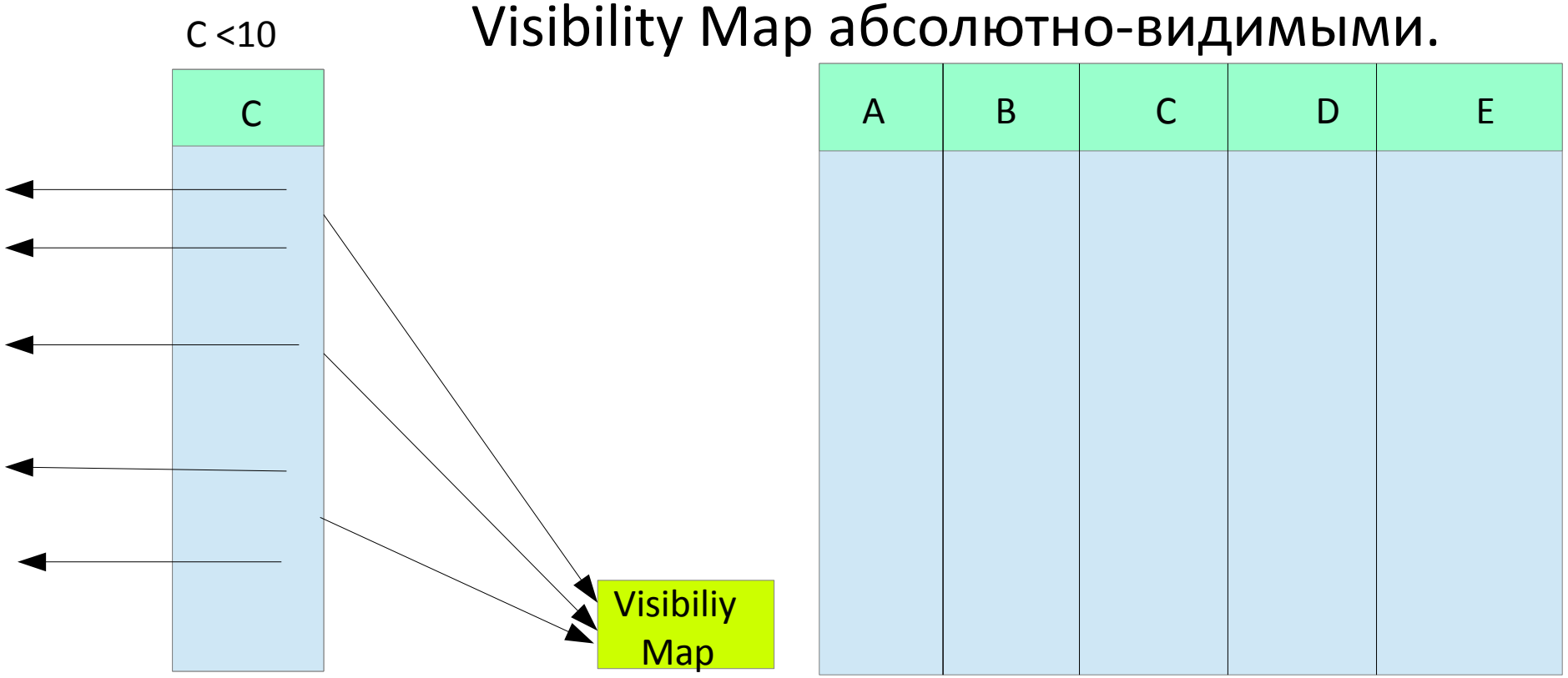
Простейший индекс (Index Scan)

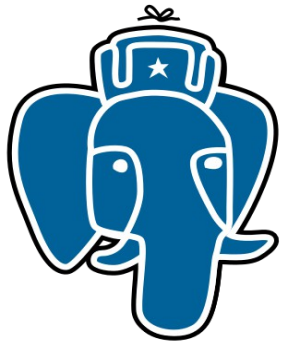
- Читаем последовательно колонку, читаем таблицу (только нужные записи), выигрыш за счет меньшего размера колонки



Простейший индекс (Index-only Scan)

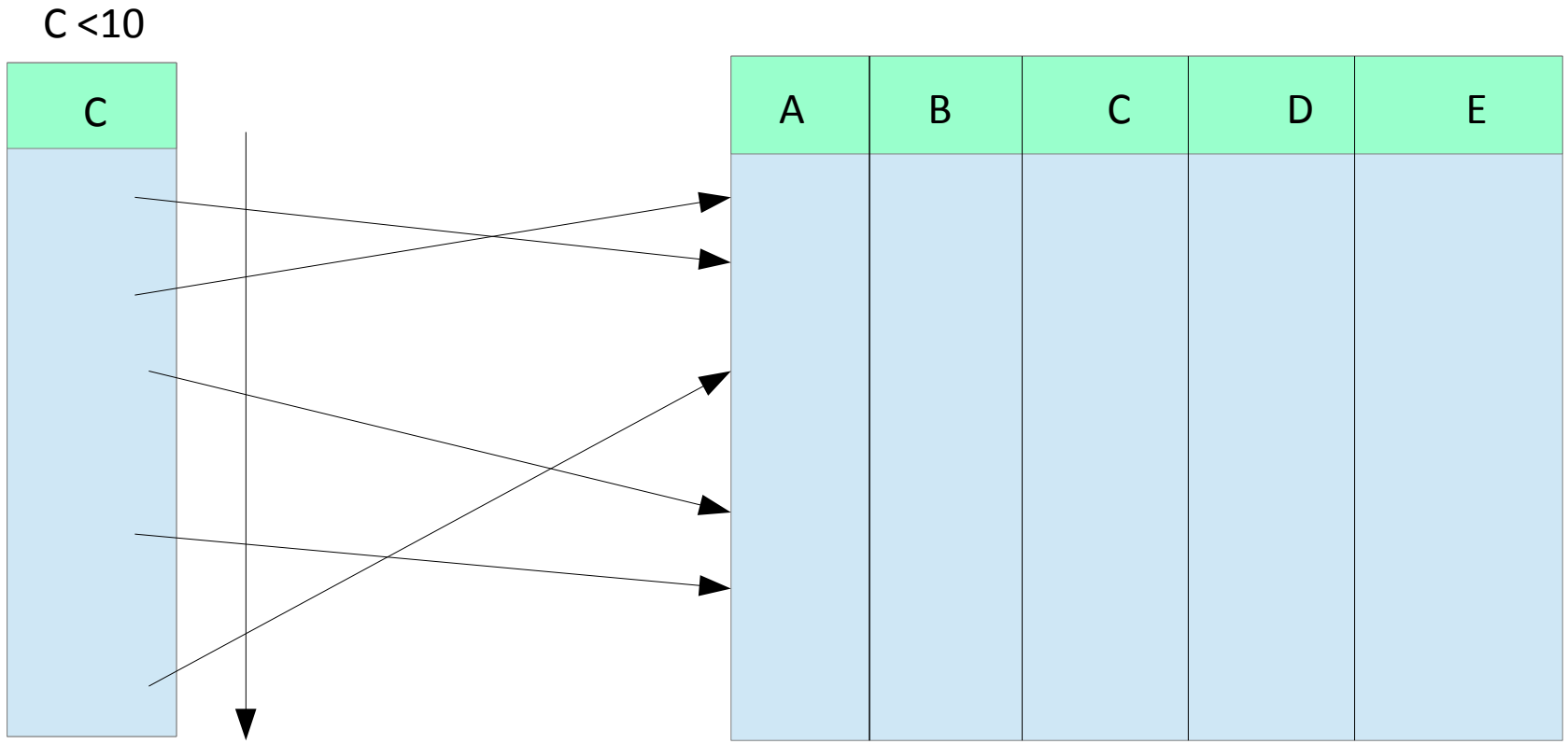
- Читаем последовательно колонку, находим нужные значения и напрямик выдаем наружу, если страницы таблицы помечены в Visibility Map абсолютно-видимыми.





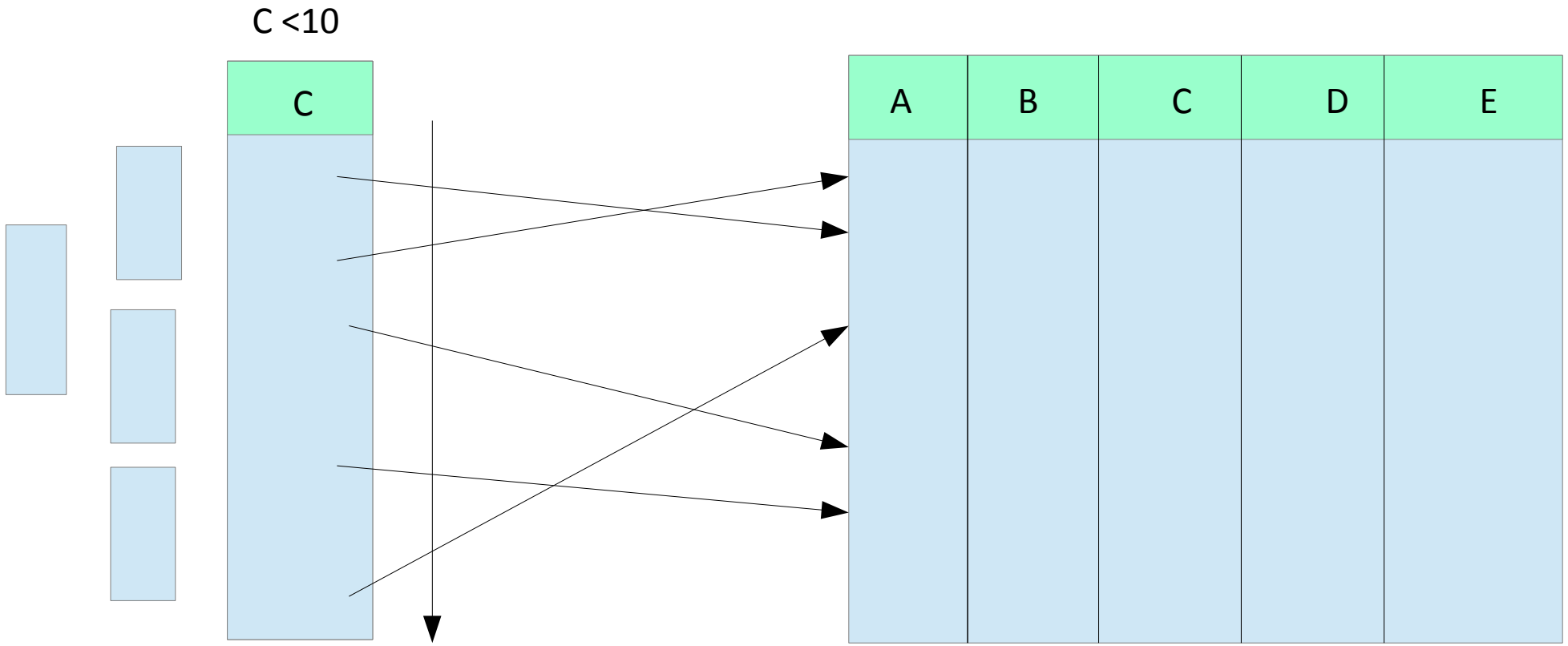
Простейший индекс++

- Упорядочиваем — получаем быстрый поиск, ускоряем ORDER BY, но случайное чтение таблицы.



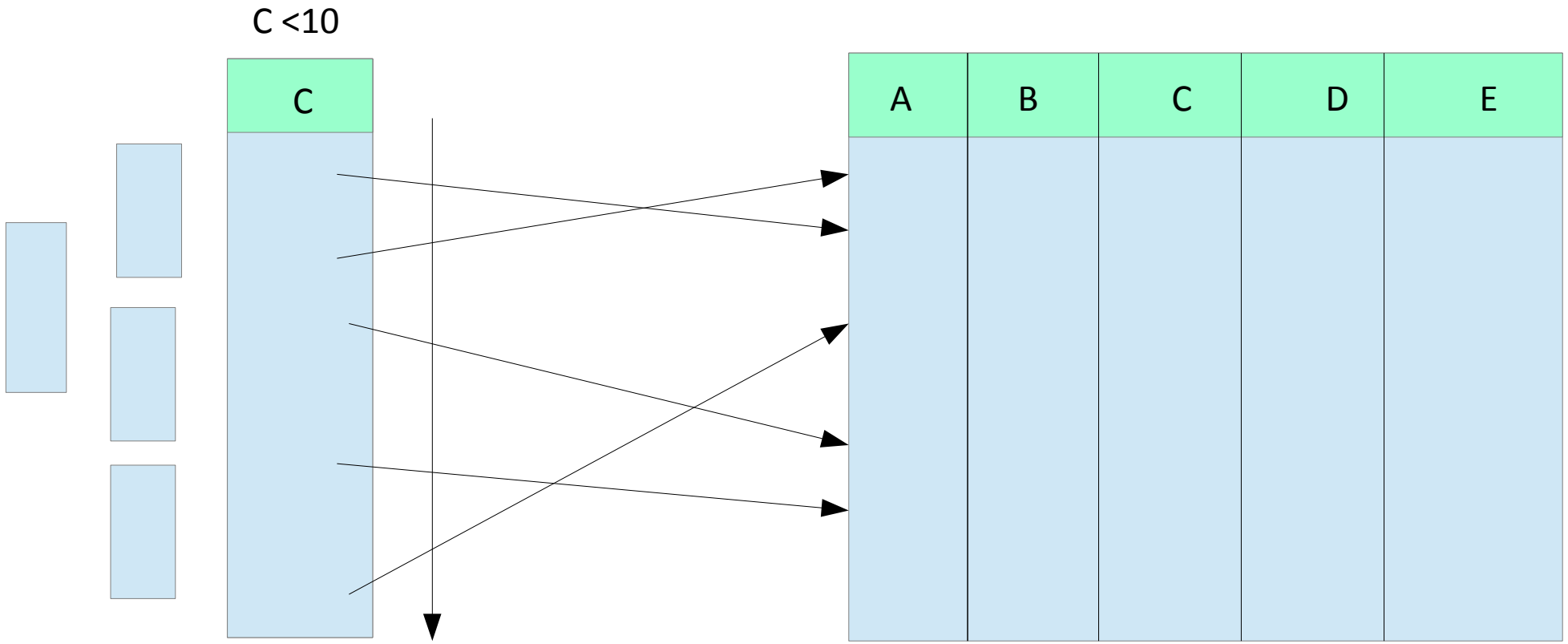
B-tree индекс

- Строим дерево — уменьшаем чтение индекса при быстром поиске



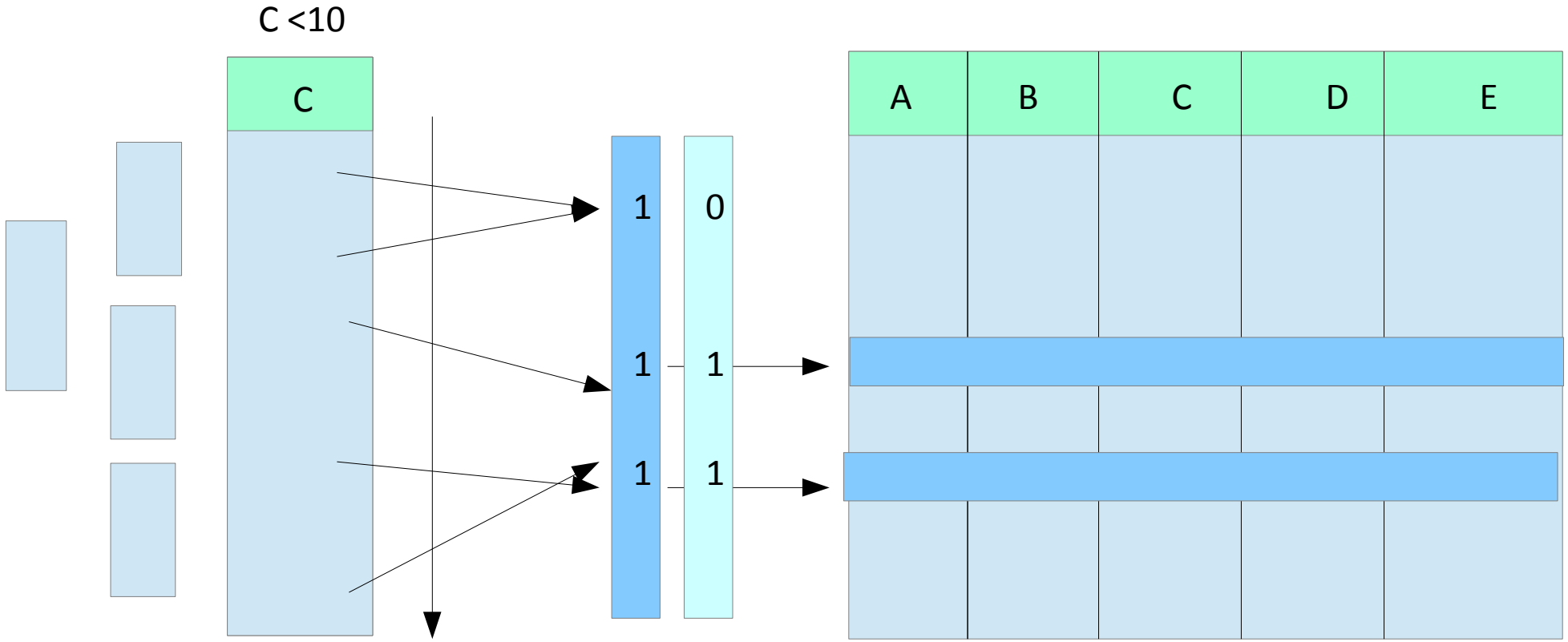
GiST,GIN,SP-GiST индексы

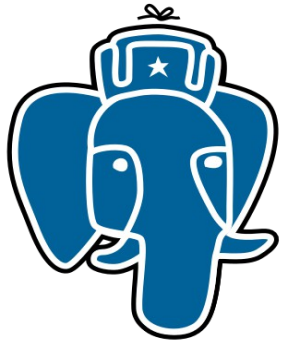
- Дерево — шаблон с API, поддержка произвольных типов данных



Bitmap index scan

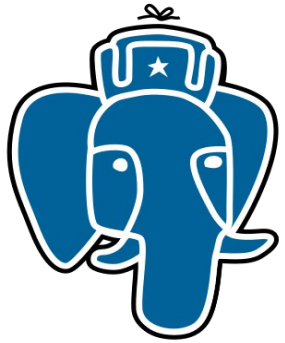
- Результат Index scan сортируем, строим в памяти bitmap и читаем таблицу последовательно. Можно комбинировать индексы.





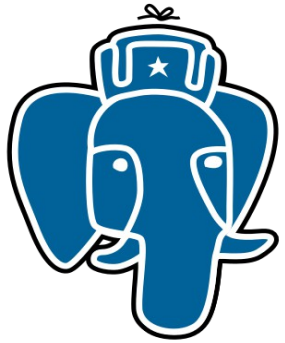
Bitmap index scan

- Результат Index scan сортируем, строим в памяти bitmap и читаем таблицу последовательно. Можно комбинировать индексы.
- bitmap строится в памяти (work_mem) используя (#page,#offset)
- если work_mem не хватает, то используется просто #page
  - Требуется RECHECK — проверка туплов на соответствие условию, что приводит к замедлению !



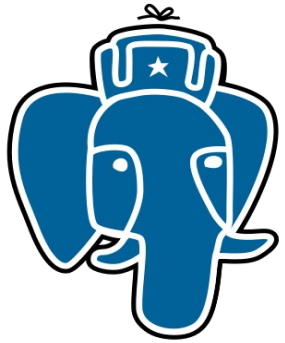
Управление использованием индексов

- Cost-based планер обычно умнее, но:
  - Иногда он ошибается и хочется разобраться в причинах
  - Хочется «пощупать» разные индексы
    - индексы долго строить
    - индексы реально используются
- Инструментарий
  - EXPLAIN, explain.depesz.com
  - Генераторы данных (generate_series())
  - contrib/pgbench



Управление использованием индексов

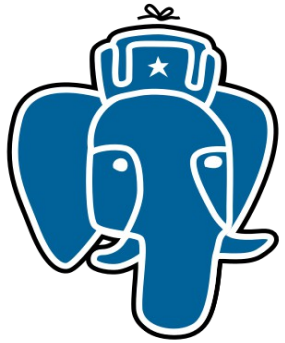
- Переменные планера (postgresql.conf, SET)
  - enable_seqscan
  - enable_indexscan
  - enable_bitmapscan
  - enable_indexonlyscan
- plantuner — скрывать индексы от планера
  - http://www.sigaev.ru/git/gitweb.cgi?p=plantuner.git;a=summary
- pg_hint_plan — хинты в комментариях
  - http://en.sourceforge.jp/projects/pghintplan/
  - http://habrahabr.ru/post/169751/



Управление использованием индексов

- Переменные планера (postgresql.conf, SET)
  - effective_cache_size (1/2, ¾ RAM) - planner использует для оценки кэша ОС и СУБД
  - random_page_cost (4.0) — стоимость в единицах последовательного чтения.

    Чем меньше random_page_cost, тем больше предпочтение планер отдаст использованию индексов. Для SSD может быть 1.0.
  - maintenance_work_mem — используется для CREATE INDEX
  - work_mem — используется для BITMAP HEAP SCAN. Малый размер приводит к RECHECK (существенное замедление).
- Наличие статистики (analyze) влияет на использование индексов.



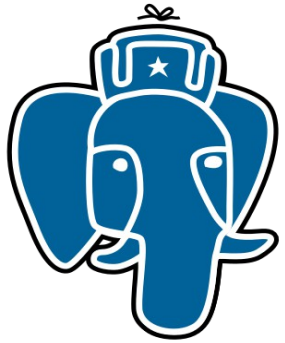
EXPLAIN

- Показать выбраный план выполнения запроса
- ANALYZE - Выполнить запрос и показать результирующий план
- COSTS ON|OFF, стоимости операций
- BUFFERS ON|OFF - прочитанные страницы
  - SHARED READ,HIT — с диска, из разделяемой памяти
- TIMING ON|OFF — времена выполнения

Syntax:
EXPLAIN [(option [, ...])] statement
EXPLAIN [ANALYZE] [VERBOSE] statement

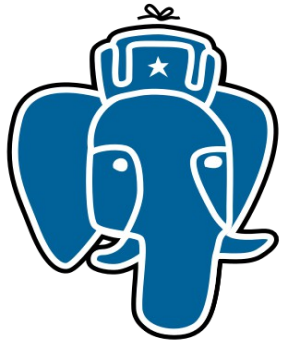
where option can be one of:

ANALYZE [boolean]
VERBOSE [boolean]
COSTS [boolean]
BUFFERS [boolean]
TIMING [boolean]
FORMAT { TEXT | XML | JSON | YAML }



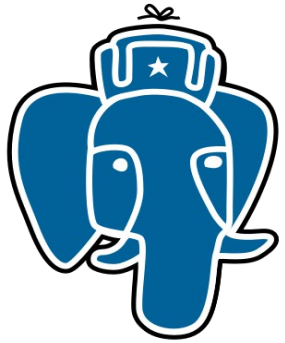
EXPLAIN

- Читать снизу вверх
- Надо обращать внимание на
  - Rows estimated и actual rows — большое несоответствие плохо (analyze table, ALTER TABLE … ALTER COLUMN … SET STATISTICS ...)
  - количество прочитанных страниц с диска — идеально только Shared hit
  - Recheck — идеально Heap Blocks: только exact, нет lossy (work_mem)
  - costs
  - Planning time должно быть небольшим, иначе, задуматься над prepared statements



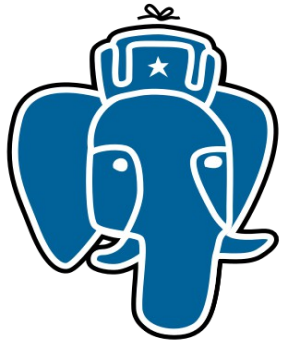
EXPLAIN

```
explain(analyze,buffers, costs on) SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on test1  (cost=2149.53..9073.25 rows=101181 width=12)
                           (actual time=12.384..51.759 rows=99831 loops=1)
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
   Heap Blocks: exact=5406
   Buffers: shared hit=2155 read=3527
   ->  Bitmap Index Scan on test1_v_idx  (cost=0.00..2124.24 rows=101181 width=0)
                                         (actual time=11.617..11.617 rows=99831 loops=1)
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
         Buffers: shared read=276
 Planning time: 0.097 ms
 Execution time: 57.792 ms
(9 rows)
```



EXPLAIN

```
explain(analyze,buffers, costs on) SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
                                                                 QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on test1  (cost=2149.53..9073.25 rows=101181 width=12) (actual time=10.894
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
   Heap Blocks: exact=5406
   Buffers: shared hit=5682 – было    Buffers: shared hit=2155 read=3527
   ->  Bitmap Index Scan on test1_v_idx  (cost=0.00..2124.24 rows=101181 width=0) (actual ti
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
         Buffers: shared hit=276
 Planning time: 0.092 ms
 Execution time: 38.000 ms
(9 rows)
```



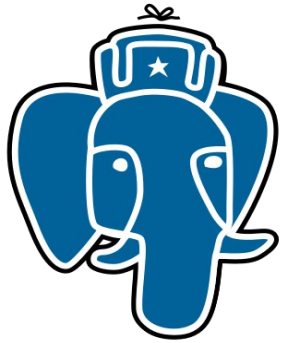
CREATE INDEX

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ]
    [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

Syntax:

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name
ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter = value [, ... ] )
ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
```

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```



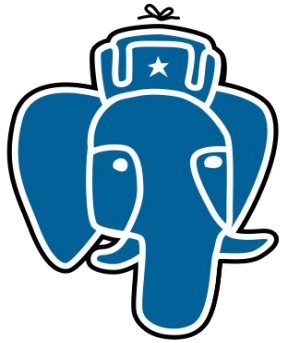
CREATE INDEX

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ]
    [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

```
=# \d pg_am
        Table "pg_catalog.pg_am"
     Column      |   Type    | Modifiers
-----------------+-----------+-----------
 amname          | name      | not null
 amstrategies    | smallint  | not null
 amsupport       | smallint  | not null
 amcanorder      | boolean   | not null
 ……..............................
 amcanunique     | boolean   | not null
 amoptions       | regproc   | not null
```

```
=# select amname from pg_am;
 amname
--------
 btree
 hash
 gist
 gin
 spgist
(5 rows)
```

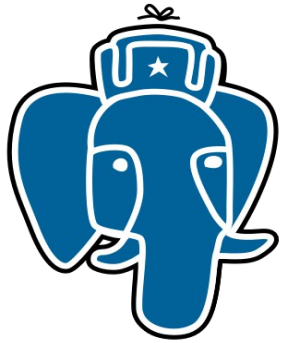
ACCESS METHODS



CREATE INDEX

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ]
    [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

- storage_parameter
  - BTREE, GiST - FILLFACTOR
  - GIN — FASTUPDATE
- TABLESPACE — размещение индекса на альтернативном хранилище для улучшения ввода/вывода
- opclass — оператор для колонки (если доступны несколько)
- ASC|DESC — матчить ORDER BY для использования индекса

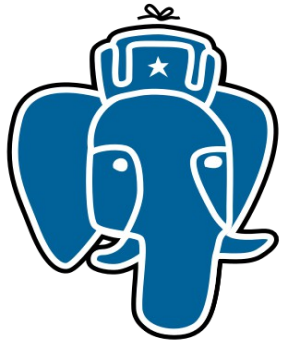


CREATE INDEX

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ]
    [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

- CONCURRENTLY — конкурентное создание индекса
  - Не блокирует таблицу на изменения
  - Спасает в «боевых» условиях
  - Требует два прохода и окончания всех текущих транзакций
  - Нельзя создавать в транзакции
  - DROP INDEX CONCURRENTLY

```
postgres=# \d tt
      Table "public.tt"
 Column |  Type   | Modifiers
--------+---------+-----------
 i      | integer |
Indexes:
    "tt_idx" btree (i) INVALID
```

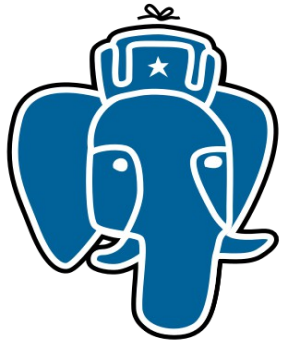


CREATE INDEX

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ]
    [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

- Функциональный индекс
  - Функция должна быть IMMUTABLE
  - «Дорогое» изменение индекса
  - Условие при поиске должно «матчить» CREATE INDEX

```
create index sin_idx on foo(sin(id));
=# explain select 1 from foo where sin(id)=0;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using sin_idx on foo  (cost=0.14..8.16 rows=1 width=0)
   Index Cond: (sin((id)::double precision) = 0::double precision)
(2 rows)
```



CREATE INDEX

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ]
    [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

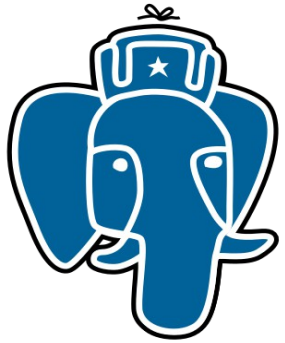
- COLLATE collation может сильно влиять !

  tags text[]

- CREATE INDEX tagidx ON t USING gin (tags collate "C")

  В 11 раз быстрее на 10,000,000 записях

  CREATE INDEX tagidx ON t USING gin (tags collate «ru_RU.utf8»);

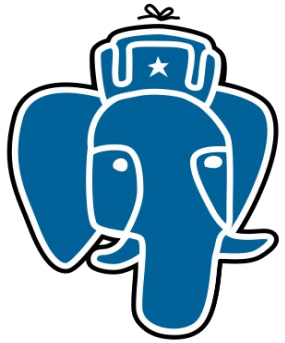


Выбор способа сканирования таблицы (нет статистики)

```
CREATE TABLE test1 WITH (autovacuum_enabled = off) AS (SELECT id, random() v
FROM generate_series(1,1000000) id);
CREATE INDEX test1_v_idx ON test1 (v);
```

Нет статистики!

```
SELECT * FROM test1 WHERE v BETWEEN 0.1 AND 0.9

  Bitmap Heap Scan on test1  (cost=107.67..5701.65 rows=5000 width=12)
                      (actual time=82.343..155.903 rows=799793 loops=1)
    Recheck Cond: ((v >= 0.1::double precision) AND (v <= 0.9::double precision))
    Heap Blocks: exact=5406
    ->  Bitmap Index Scan on test1_v_idx  (cost=0.00..106.42 rows=5000 width=0)
                                (actual time=81.576..81.576 rows=799793 loops=1)
          Index Cond: ((v >= 0.1::double precision) AND (v <= 0.9::double precision))
 Planning time: 0.069 ms
 Execution time: 192.032 ms
```



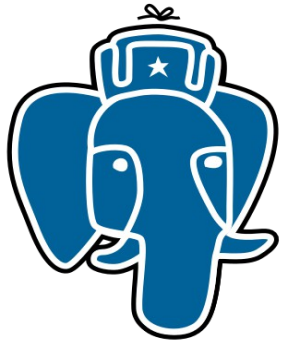
Выбор способа сканирования таблицы (нет статистики)

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.500001;
 Bitmap Heap Scan on test1  (cost=107.67..5701.65 rows=5000 width=12)
                          (actual time=0.022..0.022 rows=0 loops=1)
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.500001::double precision))
   Heap Blocks:
   ->  Bitmap Index Scan on test1_v_idx  (cost=0.00..106.42 rows=5000 width=0)
                                       (actual time=0.021..0.021 rows=0 loops=1)
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.500001::double precision))
 Planning time: 0.097 ms
 Execution time: 0.037 ms
```

Нет статистики!

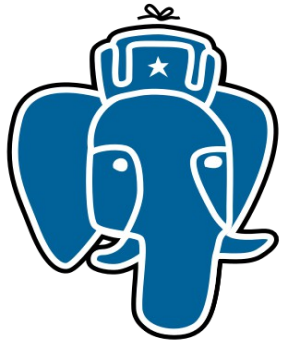
Нет статистики!

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
 Bitmap Heap Scan on test1  (cost=107.67..5701.65 rows=5000 width=12)
                       (actual time=11.256..31.522 rows=99831 loops=1)
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
   Heap Blocks: exact=5406
   ->  Bitmap Index Scan on test1_v_idx  (cost=0.00..106.42 rows=5000 width=0) (actual time=10.55
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
 Planning time: 0.086 ms
 Execution time: 36.370 ms
```



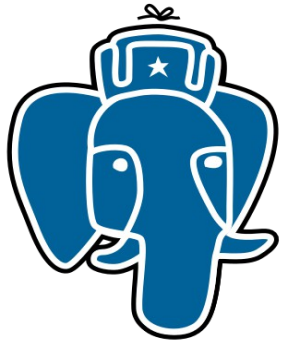
Выбор способа сканирования таблицы (есть статистика)

```
VACUUM ANALYZE test1;

SELECT * FROM test1 WHERE v BETWEEN 0.1 AND 0.9;
 Seq Scan on test1  (cost=0.00..20406.00 rows=800835 width=12)
                    (actual time=0.008..144.527 rows=799793 loops=1)
   Filter: ((v >= 0.1::double precision) AND (v <= 0.9::double precision))
   Rows Removed by Filter: 200207
 Planning time: 0.095 ms
 Execution time: 180.556 ms

SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.500001;
 Index Scan using test1_v_idx on test1  (cost=0.42..8.45 rows=1 width=12)
                                        (actual time=0.011..0.011 rows=0 loops=1)
   Index Cond: ((v >= 0.5::double precision) AND (v <= 0.500001::double precision))
 Planning time: 0.104 ms
 Execution time: 0.025 ms
```



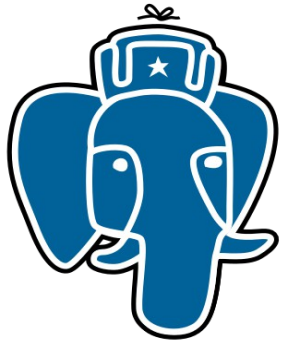
Влияние random_page_cost

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
Bitmap Heap Scan on test1  (cost=2149.53..9073.25 rows=101181 width=12)
                          (actual time=10.302..30.350 rows=99831 loops=1)
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
   Heap Blocks: exact=5406
   ->  Bitmap Index Scan on test1_v_idx  (cost=0.00..2124.24 rows=101181 width=0)
(actual time=9.556..9.556 rows=99831 loops=1)
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
 Planning time: 0.104 ms
 Execution time: 35.187 ms

SET random_page_cost = 1;

 Index Scan using test1_v_idx on test1  (cost=0.42..7708.04 rows=101181 width=12)
                                       (actual time=0.016..54.971 rows=99831 loops=1)
   Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
 Planning time: 0.096 ms
 Execution time: 59.795 ms
```



Влияние work_mem

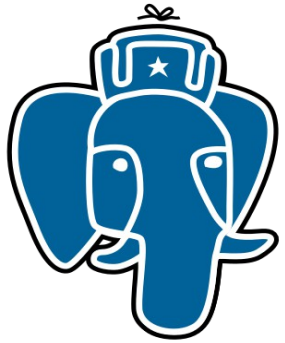
```
show work_mem;
 work_mem
----------
 4MB
(1 row)
SET work_mem = '128 kB';

 Bitmap Heap Scan on test1  (cost=2149.53..9073.25 rows=101181 width=12)
                          (actual time=8.527..104.225 rows=99831 loops=1)
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
   Rows Removed by Index Recheck: 687690
   Heap Blocks: exact=1269 lossy=4137
   ->  Bitmap Index Scan on test1_v_idx  (cost=0.00..2124.24 rows=101181 width=0)
                                       (actual time=8.386..8.386 rows=99831 loops=1)
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
 Planning time: 0.109 ms
 Execution time: 108.821 ms
```



maintenance_work_mem

- Влияет на создание индекса (BULK INDEX CREATION)
  Чем больше памяти, тем быстрее создастся индекс (если она действительно есть)

```
SET maintenance_work_mem = '1 MB';
CREATE INDEX test1_v_idx ON test1 (v);
Time: 1194,299 ms

SET maintenance_work_mem = '128 MB';
CREATE INDEX test1_v_idx ON test1 (v);
Time: 708,644 ms
```

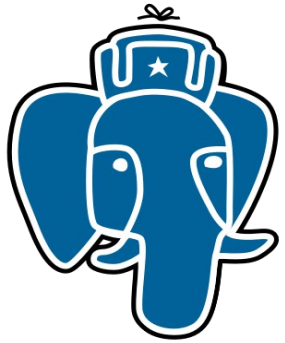


CLUSTER

- Физическая реорганизация записей таблицы по индексу
  - Значительное ускорение запросов на диапазоны значений за счет уменьшения количества прочитанных блоков
  - ACCESS EXCLUSIVE lock блокирует чтение-запись таблицы :(
  - Требуется рекластеринг после обновлении таблицы
  - FILLFACTOR < 100% помогает

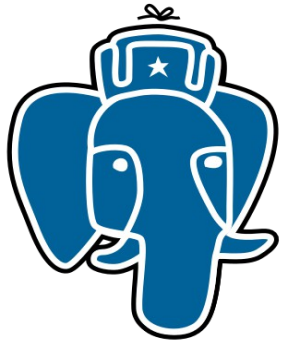
  CLUSTER [VERBOSE] table_name [USING index_name]
  CLUSTER [VERBOSE]

  ALTER TABLE ... CLUSTER ON index_name
  SET WITHOUT CLUSTER



CLUSTER

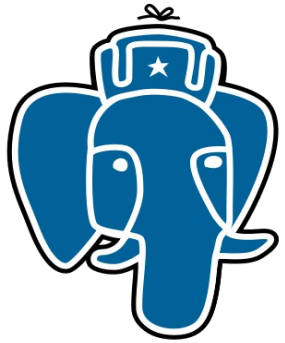
```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
  Bitmap Heap Scan on test1 (actual time=20.269..51.084 rows=200321 loops=1)
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.7::double precision))
   Heap Blocks: exact=5406
   Buffers: shared hit=5956
   ->  Bitmap Index Scan on test1_v_idx (actual time=19.490..19.490 rows=200321 loops=1)
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.7::double precision))
         Buffers: shared hit=550
 Execution time: 62.382 ms

CLUSTER test1 USING test1_v_idx;
=# \d test1
         Table "public.test1"
 Column |       Type       | Modifiers
--------+------------------+-----------
 id     | integer          |
 v      | double precision |
Indexes:
    "test1_v_idx" btree (v) CLUSTER
```



CLUSTER

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
Bitmap Heap Scan on test1 (actual time=15.824..34.043 rows=200321 loops=1)
   Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.7::double precision))
   Heap Blocks: exact=1084
   Buffers: shared hit=1634
   ->  Bitmap Index Scan on test1_v_idx (actual time=15.704..15.704 rows=200321 loops=1)
         Index Cond: ((v >= 0.5::double precision) AND (v <= 0.7::double precision))
         Buffers: shared hit=550
 Execution time: 44.517 ms
```

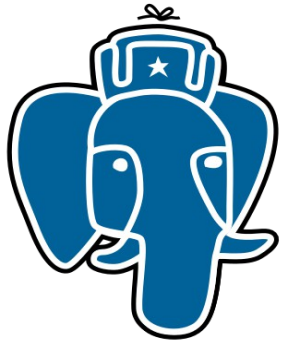
- Количество прочитанных блоков уменьшилось ~ в 5 раз, 5046 vs 1084 !



Tablespace

- Размещение индексов на отдельном tablespace (шпинделе) позволяет уменьшить IO

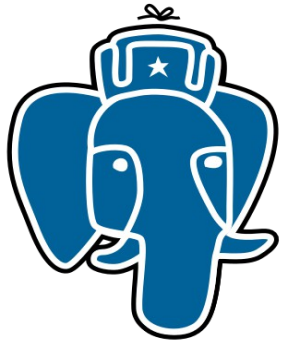
```
Задать tablespace при создании:
CREATE INDEX test1_v1_idx ON test1(v1) TABLESPACE idx_tablespace;
Изменить tablespace:
ALTER INDEX test1_v1_idx SET TABLESPACE idx_tablespace; - долгая эксклюзивная блокировка!

CREATE INDEX CONCURRENTLY test1_v1_idx2 ON test1(v1) TABLESPACE idx_tablespace;
BEGIN;
DROP INDEX test1_v1_idx; - короткая эксклюзивная блокировка!
ALTER INDEX test1_v1_idx2 RENAME TO test1_v1_idx;
COMMIT;
```



Unlogged tables

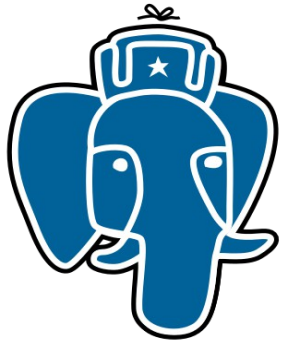
- Нет оверхеда генерации и записи WAL
- Не реплицируются
- Не выживают при краше кластера
- Используются для хранения пользовательских сессий, логов
- Индексы тоже UNLOGGED

```
CREATE TABLE test (id integer, r float);
INSERT INTO test SELECT id, random() FROM generate_series(1,1000000) id;
INSERT 0 1000000
Time: 3055.022 ms
CREATE UNLOGGED  TABLE testU (id integer, r float);
INSERT INTO testU SELECT id, random() FROM generate_series(1,1000000) id;
INSERT 0 1000000
Time: 572.509 ms
```



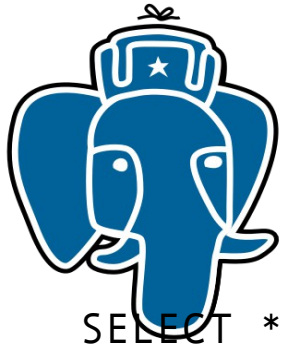
Индексы и LIKE

```
# CREATE INDEX dblp_titles_idx ON dblp_titles(s);
# EXPLAIN ANALYZE SELECT * FROM dblp_titles WHERE s LIKE 'Transaction%';
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on dblp_titles  (cost=0.00..57433.89 rows=12634 width=36) (actual time=0.030..321.498 r
   Filter: (s ~~ 'Transaction%'::text)
   Rows Removed by Filter: 2526351
 Total runtime: 321.652 ms
(4 rows)

# CREATE INDEX dblp_titles_pattern_idx ON dblp_titles(s text_pattern_ops);
# EXPLAIN ANALYZE SELECT * FROM dblp_titles WHERE s LIKE 'Transaction%';
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Scan using dblp_titles_pattern_idx on dblp_titles  (cost=0.00..9.18 rows=159 width=52) (ac
   Index Cond: ((s ~>=~ 'Transaction'::text) AND (s ~<~ 'Transactioo'::text))
   Filter: (s ~~ 'Transaction%'::text)
 Total runtime: 0.706 ms
(4 rows)
```



Индекс и ORDER BY

```
SELECT * FROM test1 ORDER BY v;
 Index Scan using test1_v_idx on test1  (cost=0.00..47604.02 rows=1000000 width=12)
                                        (actual time=0.018..145.569 rows=1000000 loops=1)
   Buffers: shared hit=8141
 Total runtime: 188.832 ms

SET enable_indexscan = off;

SELECT * FROM test1 ORDER BY v;
 Sort  (cost=132154.34..134654.34 rows=1000000 width=12)
    (actual time=636.567..752.384 rows=1000000 loops=1)
   Sort Key: v
   Sort Method: external sort  Disk: 25424kB
   Buffers: shared hit=5406, temp read=3178 written=3178
   ->  Seq Scan on test1  (cost=0.00..15406.00 rows=1000000 width=12)
                          (actual time=0.010..86.157 rows=1000000 loops=1)
         Buffers: shared hit=5406
 Total runtime: 819.682 ms
```



Индекс, ORDER BY и LIMIT

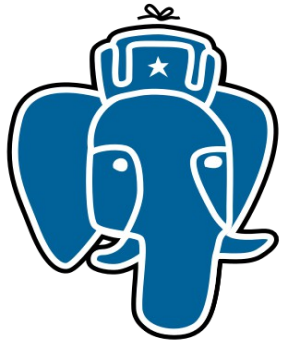
```
SELECT * FROM test1 ORDER BY v LIMIT 20;
 Limit  (cost=0.00..0.95 rows=20 width=12) (actual time=0.014..0.021 rows=20 loops=1)
   Buffers: shared hit=4
   ->  Index Scan using test1_v_idx on test1  (cost=0.00..47604.02 rows=1000000 width=12) (actua
         Buffers: shared hit=4
 Total runtime: 0.033 ms
```

А если нужно «листание»?

```
SELECT * FROM test1 ORDER BY v LIMIT 20 OFFSET 900000;
 Limit  (cost=42843.62..42844.57 rows=20 width=12) (actual time=178.863..178.868 rows=20 loops=1
   Buffers: shared hit=7327
   ->  Index Scan using test1_v_idx on test1  (cost=0.00..47604.02 rows=1000000 width=12) (actua
         Buffers: shared hit=7327
 Total runtime: 178.887 ms
```

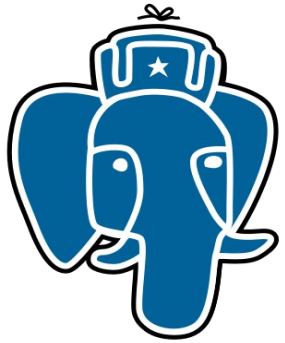
Лучше постараться сделать вот так

```
SELECT * FROM test1 WHERE v >= 0.9 ORDER BY v LIMIT 20;
 Limit  (cost=0.00..4.85 rows=20 width=12) (actual time=0.050..0.058 rows=20 loops=1)
   Buffers: shared hit=3 read=1
   ->  Index Scan using test1_v_idx on test1  (cost=0.00..24503.79 rows=101021 width=12) (actual
         Index Cond: (v >= 0.9::double precision)
         Buffers: shared hit=3 read=1
 Total runtime: 0.075 ms
```



Составной индекс, ORDER BY и LIMIT

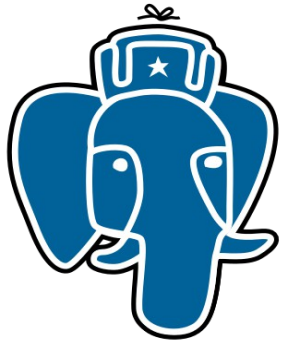
```
CREATE TABLE test4 AS (SELECT id, (random()*20::int) AS v1, random() AS v2
FROM generate_series(1,1000000) id);

CREATE INDEX test4_v1_v2_idx ON test4 (v1, v2);

SELECT * FROM test4 ORDER BY v1, v2 LIMIT 20;
 Limit  (cost=0.00..1.12 rows=20 width=20) (actual time=0.012..0.097 rows=20 loops=1)
   ->  Index Scan using test4_v1_v2_idx on test4  (cost=0.00..55892.40 rows=1000000 width=20) (ac
 Total runtime: 0.110 ms

SELECT * FROM test4 WHERE (v1, v2) > (9,0.5) ORDER BY v1, v2 LIMIT 20;
 Limit  (cost=0.00..1.58 rows=20 width=20) (actual time=0.025..0.088 rows=20 loops=1)
   ->  Index Scan using test4_v1_v2_idx on test4  (cost=0.00..43628.52 rows=551512 width=20) (act
         Index Cond: (ROW(v1, v2) > ROW(9::double precision, 0.5::double precision))
 Total runtime: 0.111 ms
```



Составной индекс, ORDER BY и LIMIT

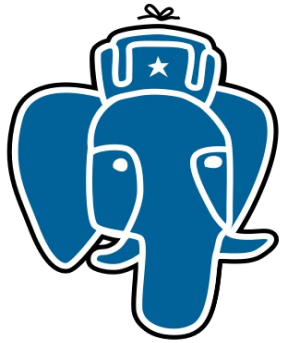
- Порядок важен !

```
SELECT * FROM test4 ORDER BY v1 LIMIT 20;
 Limit  (cost=0.00..1.12 rows=20 width=20) (actual time=0.016..0.042 rows=20 loops=1)
   ->  Index Scan using test4_v1_v2_idx on test4  (cost=0.00..55889.49 rows=1000000 width=20)
                                                  (actual time=0.016..0.042 rows=20 loops=1)
 Total runtime: 0.060 ms

SELECT * FROM test4 ORDER BY v2 LIMIT 20;
 Limit  (cost=42979.64..42979.69 rows=20 width=20) (actual time=161.618..161.620 rows=20 loops=1)
   ->  Sort  (cost=42979.64..45479.64 rows=1000000 width=20)
             (actual time=161.616..161.618 rows=20 loops=1)
         Sort Key: v2
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Seq Scan on test4  (cost=0.00..16370.00 rows=1000000 width=20)
                                (actual time=0.036..83.817 rows=1000000 loops=1)
 Total runtime: 161.645 ms
```



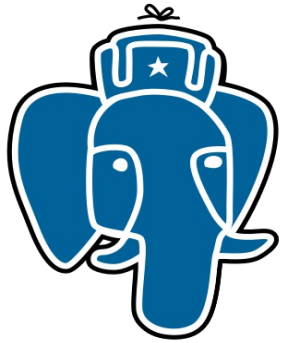
Index-only scan (9.2+)

- Index Scan возвращает указатели на записи-кандидаты, которые надо проверять видимость текущей транзакции.
  - надо прочитать страницы таблицы и проверить требуемые записи на видимость (xmin, xmax). Медленно.
- Crash-safe Visibility Map (создается при вакууме) содержит список страниц, все записи которых «видимы» **всем транзакциям.**
  - Index-Only Scan не читает страницы таблицы из этой Visibility Map и возвращает данные из индекса
  - Данные должны быть реконструированы из индекса



Index-only scan

```
\d test4
         Table "public.test4"
 Column |       Type       | Modifiers
--------+------------------+-----------
 id     | integer          |
 v1     | double precision |
 v2     | double precision |
Indexes:
    "test4_v1_v2_idx" btree (v1)

ANALYZE test4;
SELECT v1, v2 FROM test4 WHERE v1 < 0.5;
 Bitmap Heap Scan on test4 (actual time=5.618..16.428 rows=24617 loops=1)
   Recheck Cond: (v1 < 0.5::double precision)
   Heap Blocks: exact=6238
   ->  Bitmap Index Scan on test4_v1_v2_idx (actual time=4.030..4.030 rows=24617 loops=1)
         Index Cond: (v1 < 0.5::double precision)
 Execution time: 17.835 ms
```



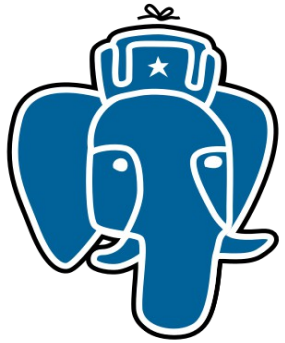
Index-only scan

- Index-only Scan возможен, только если все данные могут быть восстановлены из индекса !

```
SET enable_bitmapscan = OFF;
SET enable_seqscan = OFF;
v1,v2 не могут быть восстановлены из индекса (v1)
SELECT v1,v2 FROM test4 WHERE v1 < 0.5;
 Index Scan using test4_v1_v2_idx on test4 (actual time=0.028..16.846 rows=24617 loops=1)
   Index Cond: (v1 < 0.5::double precision)
 Execution time: 18.381 ms
```

- v1 может быть восстановлен из индекса, но почему Heap Fetches ?

```
SELECT v1 FROM test4 WHERE v1 < 0.5;
 Index Only Scan using test4_v1_v2_idx on test4 (actual time=0.038..13.890 rows=24617 loops=1)
   Index Cond: (v1 < 0.5::double precision)
   Heap Fetches: 24617
 Execution time: 15.172 ms
```

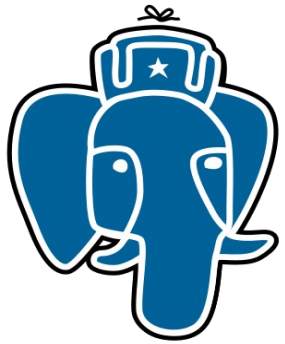


Index-only scan

- v1 может быть восстановлен из индекса, но почему Heap Fetches ?
  - Visibility Map создается при вакууме !

```
select relname,100*relallvisible/(relpages+1) as allvisible from pg_class where relname='test4';
 relname | allvisible
---------+------------
 test4   |          0
vacuum test4;
select relname,100*relallvisible/(relpages+1) as allvisible from pg_class where relname='test4';
 relname | allvisible
---------+------------
 test4   |         99
 Index Only Scan using test4_v1_v2_idx on test4 (actual time=0.021..2.762 rows=24617 loops=1)
   Index Cond: (v1 < 0.5::double precision)
   Heap Fetches: 0
 Execution time: 3.878 ms
```

- Index-only scan дал существенный выигрыш: 3.9 ms vs 18 ms

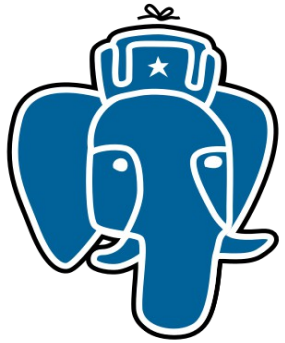


Loose Indexscan — используем CTE

- MySQL — эффективный поиск distinct values. Loose indexscan скачет по distinct values.

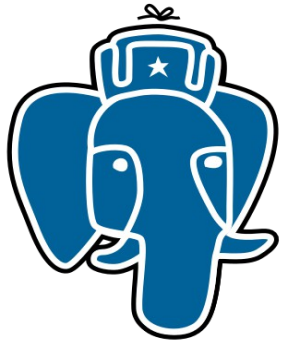
```
\d tt
      Table "public.tt"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
Indexes:
    "tt_idx" btree (id)
select distinct(id) from tt;
 id
----
  0
  1
(2 rows)

Time: 279.461 ms
```

```
=# with recursive t as
(select min(id) as id from tt
 union all
 select (select min(id) from tt where id > t.id) from t where t.id is not null
)
select id from t where id is not null
 union all
select null where exists (select * from tt where id is null);
 id
----
  0
  1
(2 rows)

Time: 0.895 ms
```



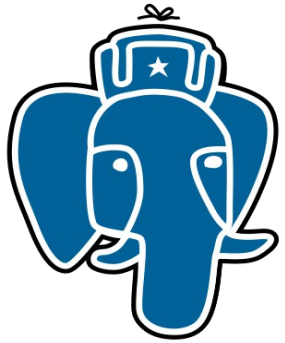
Когда индекс используется неправильно

```
CREATE TABLE test2 AS (SELECT id, polygon(20,circle(point(random(), random()),0.01)) AS p
FROM generate_series(1,50000) id);
ALTER TABLE test2 ADD PRIMARY KEY (id);
CREATE INDEX test2_box_idx ON test2 USING gist (p);
SELECT * FROM test2 t1 JOIN test2 t2 ON t2.id > t1.id AND t1.p && t2.p;
 Nested Loop  (cost=196.89..15740565.92 rows=4166667 width=72)
              (actual time=9.464..93463.065 rows=15742 loops=1)
   ->  Seq Scan on test2 t1  (cost=0.00..13000.00 rows=50000 width=36)
                             (actual time=0.027..54.015 rows=50000 loops=1)
   ->  Bitmap Heap Scan on test2 t2  (cost=196.89..313.72 rows=83 width=36)
                                     (actual time=1.845..1.866 rows=0 loops=50000)
         Recheck Cond: ((t1.p && p) AND (id > t1.id))
         Rows Removed by Index Recheck: 0
         ->  BitmapAnd  (cost=196.89..196.89 rows=83 width=0) (actual time=1.751..1.751 rows=0 lo
               ->  Bitmap Index Scan on test2_box_idx  (cost=0.00..6.41 rows=250 width=0)
                     (actual time=0.015..0.015 rows=2 loops=50000)
                     Index Cond: (t1.p && p)
               ->  Bitmap Index Scan on test2_pkey  (cost=0.00..190.18 rows=16667 width=0)
                     (actual time=1.733..1.733 rows=25000 loops=50000)
                     Index Cond: (id > t1.id)
 Total runtime: 93465.872 ms
```



Когда индекс используется неправильно

```
 Nested Loop  (cost=0.28..320195.00 rows=4166667 width=72)
(actual time=0.117..50501.640 rows=1523159 loops=1)
   ->  Seq Scan on test2 t1  (cost=0.00..3000.00 rows=50000 width=36)
(actual time=0.005..8.904 rows=50000 loops=1)
   ->  Index Scan using test2_box_idx on test2 t2  (cost=0.28..5.51 rows=83 width=36)
(actual time=0.272..1.005 rows=30 loops=50000)
         Index Cond: (t1.p && p)
         Rows Removed by Index Recheck: 18
         Filter: (id > t1.id)
         Rows Removed by Filter: 31
 Planning time: 0.530 ms
 Execution time: 50574.670 ms
(9 rows)
```



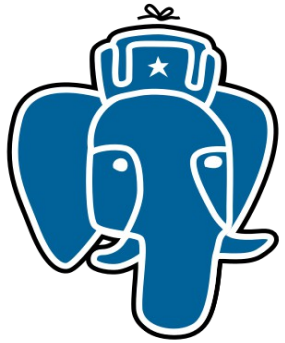
Почему так?

pg_operator

oprname	oprrest	oprjoin	oprleft	oprright
&&	areasel	areajoinsel	604	604 (polygon)

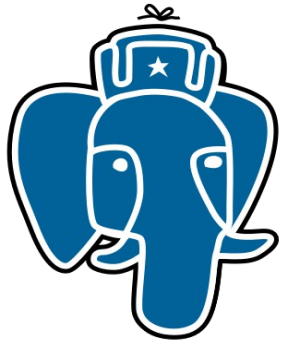
src/backend/utils/adt/geo_selfuncs.c

```
Datum
areasel(PG_FUNCTION_ARGS)
{
        PG_RETURN_FLOAT8(0.005);
}

Datum
areajoinsel(PG_FUNCTION_ARGS)
{
        PG_RETURN_FLOAT8(0.005);
}
```



Переменные планировщика

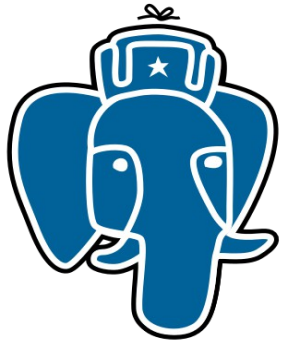
```
SET enable_bitmapscan = OFF;

SELECT * FROM test2 t1 JOIN test2 t2 ON t2.id > t1.id AND t1.p && t2.p;
 Nested Loop  (cost=0.00..18358598.00 rows=4166667 width=72)
              (actual time=2.458..11824.565 rows=15742 loops=1)
   ->  Seq Scan on test2 t1  (cost=0.00..13000.00 rows=50000 width=36)
                             (actual time=0.005..20.801 rows=50000 loops=1)
   ->  Index Scan using test2_box_idx on test2 t2  (cost=0.00..366.08 rows=83 width=36)
                                                   (actual time=0.215..0.236 rows=0 loops=50000)
         Index Cond: (t1.p && p)
         Rows Removed by Index Recheck: 0
         Filter: (id > t1.id)
         Rows Removed by Filter: 1
 Total runtime: 11826.029 ms
```



Использование plantuner

```
LOAD 'plantuner';
SET plantuner.forbid_index='test2_pkey';

SELECT * FROM test2 t1 JOIN test2 t2 ON t2.id > t1.id AND t1.p && t2.p;
 Nested Loop  (cost=6.43..17418092.51 rows=4166667 width=72) (actual time=1.547..12355.722 rows=1
   ->  Seq Scan on test2 t1  (cost=0.00..13000.00 rows=50000 width=36) (actual time=0.004..22.919
   ->  Bitmap Heap Scan on test2 t2  (cost=6.43..347.27 rows=83 width=36) (actual time=0.224..0.2
         Recheck Cond: (t1.p && p)
         Rows Removed by Index Recheck: 0
         Filter: (id > t1.id)
         Rows Removed by Filter: 1
         ->  Bitmap Index Scan on test2_box_idx  (cost=0.00..6.41 rows=250 width=0)
             (actual time=0.014..0.014 rows=2 loops=50000)
               Index Cond: (t1.p && p)
 Total runtime: 12357.238 ms
```

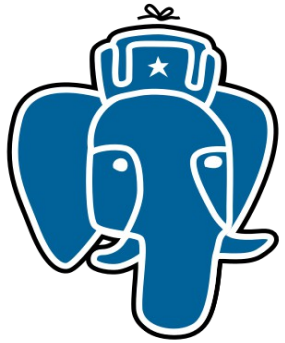
Больше про plantuner: http://www.sai.msu.su/~megera/wiki/plantuner



Использование pg_hint_plan

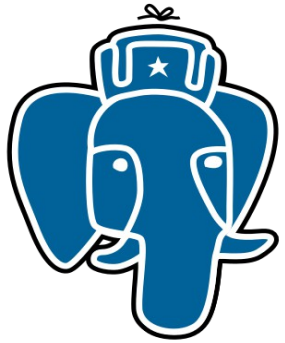
```
LOAD 'pg_hint_plan';

/*+ IndexScan(test2 test2_box_idx) */
SELECT * FROM test2 t1 JOIN test2 t2 ON t2.id > t1.id AND t1.p && t2.p;
 Nested Loop  (cost=0.00..18358598.00 rows=4166667 width=72) (actual time=2.457..11912.368 rows=1
   ->  Seq Scan on test2 t1  (cost=0.00..13000.00 rows=50000 width=36) (actual time=0.005..21.094
   ->  Index Scan using test2_box_idx on test2 t2  (cost=0.00..366.08 rows=83 width=36) (actual t
         Index Cond: (t1.p && p)
         Rows Removed by Index Recheck: 0
         Filter: (id > t1.id)
         Rows Removed by Filter: 1
 Total runtime: 11913.821 ms
```

Больше про pg_hint_plan: http://habrahabr.ru/post/169751/



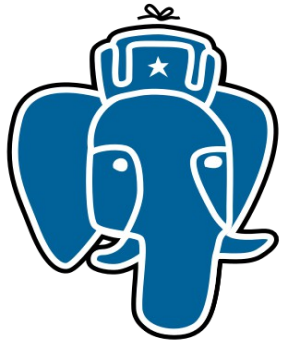
Не всё ли равно когда сделать индекс?

```
CREATE TABLE test5 (id integer PRIMARY KEY, v float8);
Time: 1,991 ms
CREATE INDEX test5_v_idx ON test5(v);
Time: 0,506 ms
INSERT INTO test5 (SELECT id, random() FROM generate_series(1,1000000) id);
Time: 4909,127 ms
Total: 4911 ms

CREATE TABLE test5 (id integer, v float8);
Time: 0,763 ms
INSERT INTO test5 (SELECT id, random() FROM generate_series(1,1000000) id);
Time: 938,852 ms
ALTER TABLE test5 ADD PRIMARY KEY (id);
Time: 779,618 ms
CREATE INDEX test5_v_idx ON test5(v);
Time: 1195,492 ms
Total: 2915 ms
```



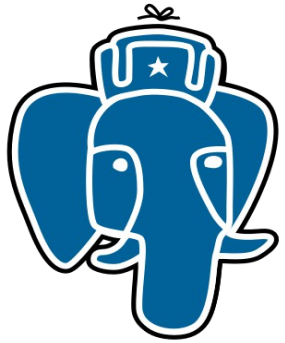
Частичный индекс

```
CREATE TABLE test6 AS (
SELECT id, (random()*20::int) AS v1, random() AS v2 FROM generate_series(1,1000000) id);
CREATE INDEX test6_v1_0_v2_idx ON test6 (v2) WHERE v1 = 0; – index size 8Kb vs 21 Mb (full)

SELECT * FROM test6 WHERE v1 = 0 AND v2 BETWEEN 0.1 AND 0.4;
 Index Scan using test6_v1_0_v2_idx on test6  (cost=0.00..8.27 rows=1 width=20) (actual time=0.00
   Index Cond: ((v2 >= 0.1::double precision) AND (v2 <= 0.4::double precision))
 Total runtime: 0.037 ms

SELECT * FROM test6 WHERE v1 = 0;
 Index Scan using test6_v1_0_v2_idx on test6  (cost=0.00..8.27 rows=1 width=20) (actual time=0.00
 Total runtime: 0.021 ms

SELECT * FROM test6 WHERE v2 BETWEEN 0.1 AND 0.2;
 Seq Scan on test6  (cost=0.00..21370.00 rows=99962 width=20) (actual time=0.044..157.861 rows=10
   Filter: ((v2 >= 0.1::double precision) AND (v2 <= 0.2::double precision))
   Rows Removed by Filter: 899919
 Total runtime: 162.312 ms
```



Функциональный индекс

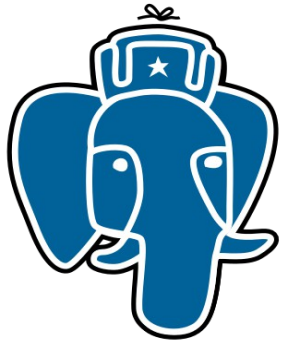
```
CREATE TABLE test7 AS (
SELECT id, random() AS v1, random() AS v2 FROM generate_series(1,1000000) id);

CREATE INDEX test7_v1_plus_v2_idx ON test7((v1 + v2));

SELECT * FROM test7 WHERE v1 + v2 > 1.9;
  Bitmap Heap Scan on test7 (actual time=1.229..4.534 rows=4897 loops=1)
   Recheck Cond: ((v1 + v2) > 1.9::double precision)
   Heap Blocks: exact=3439
   Buffers: shared hit=3456
   ->  Bitmap Index Scan on test7_v1_plus_v2_idx (actual time=0.656..0.656 rows=4897 loops=1)
         Index Cond: ((v1 + v2) > 1.9::double precision)
         Buffers: shared hit=17
 Execution time: 4.921 ms
```



Функциональный индекс

- Выражение в запросе должно точно матчить выражение в CREATE INDEX !

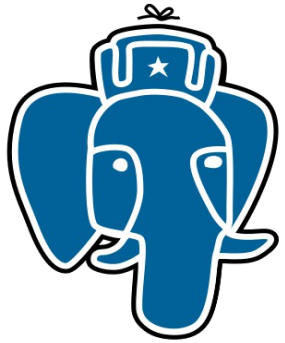
```
SELECT * FROM test7 WHERE v2 + v1 > 1.9;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on test7 (actual time=0.020..118.885 rows=4897 loops=1)
   Filter: ((v2 + v1) > 1.9::double precision)
   Rows Removed by Filter: 995103
   Buffers: shared hit=4388 read=1982
 Execution time: 119.202 ms
(5 rows)
```



Heap-Only Tuples (HOT)

- Вставка в индекс — это delete, insert (MVCC)
- 8.2: Обновление таблицы приводит к обновлению **всех** индексов
- 8.3+: Обновляются только те индексы, которые построены по изменившшися полям (Heap-Only Tuples, HOT)
- Пример:

```
CREATE TABLE test8 AS (SELECT id, random() AS v1, random() AS v2, random() AS v3
FROM generate_series(1,1000000) id);
CREATE INDEX test8_v1_idx ON test8(v1);
CREATE INDEX test8_v2_idx ON test8(v2);
CREATE INDEX test8_v3_idx ON test8(v3);
```

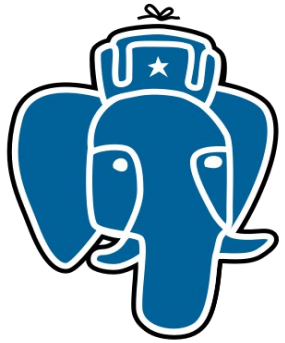


Head Only Tuples (HOT)

```
test8_v1_idx - 21 MB
test8_v2_idx - 21 MB
test8_v3_idx - 21 MB

UPDATE test8 SET v1 = v1 + 1 WHERE id % 20 = 0;

test8_v1_idx - 23 MB
test8_v2_idx - 21 MB
test8_v3_idx - 21 MB

VACUUM test8;
UPDATE test8 SET v2 = v2 + 1 WHERE id % 20 = 0;

test8_v1_idx - 23 MB
test8_v2_idx - 23 MB
test8_v3_idx - 21 MB
```

```
=# select n_tup_upd, n_tup_hot_upd from pg_stat_user_tables;
 n_tup_upd | n_tup_hot_upd
-----------+---------------
         0 |             0
         0 |             0
         0 |             0
     50000 |             0
```

```
=# select n_tup_upd, n_tup_hot_upd from pg_stat_user_tables;
 n_tup_upd | n_tup_hot_upd
-----------+---------------
         0 |             0
         0 |             0
         0 |             0
    100000 |             0
```

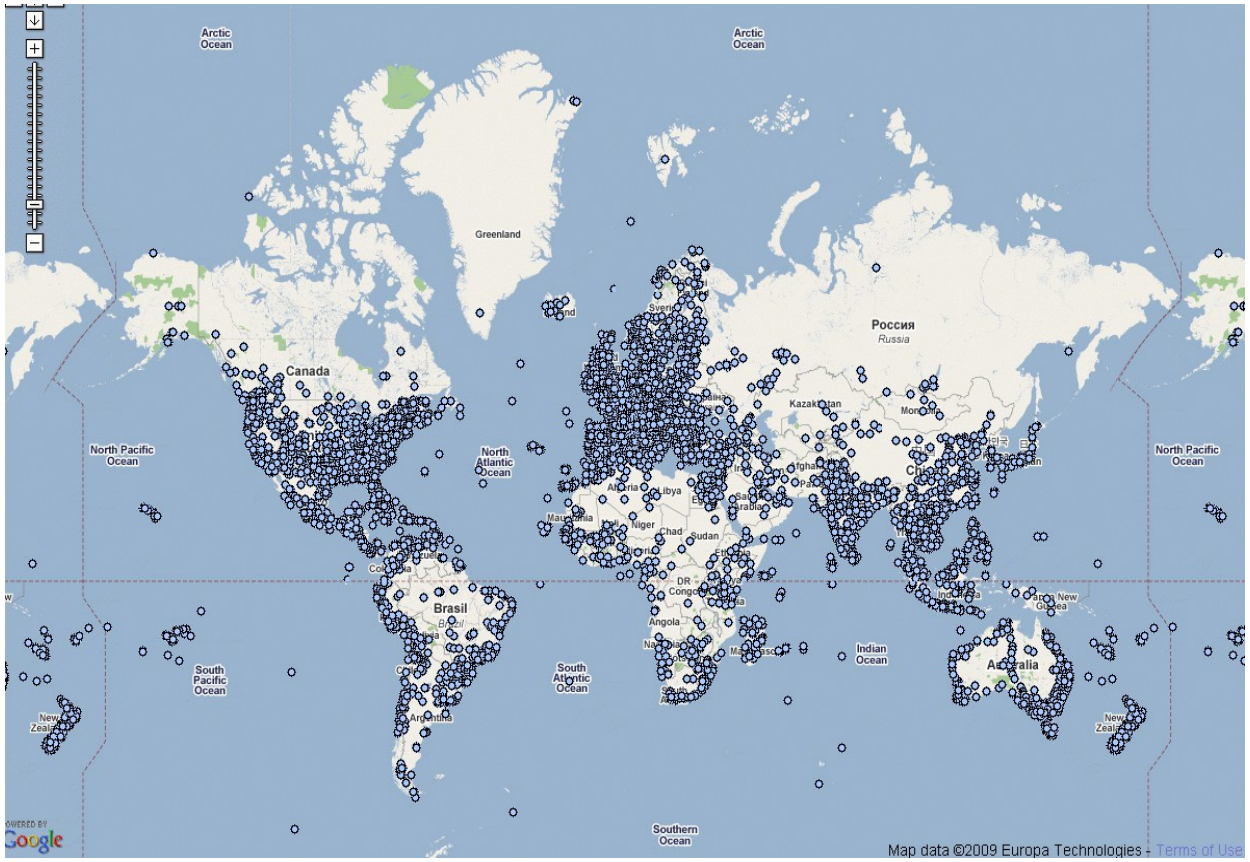


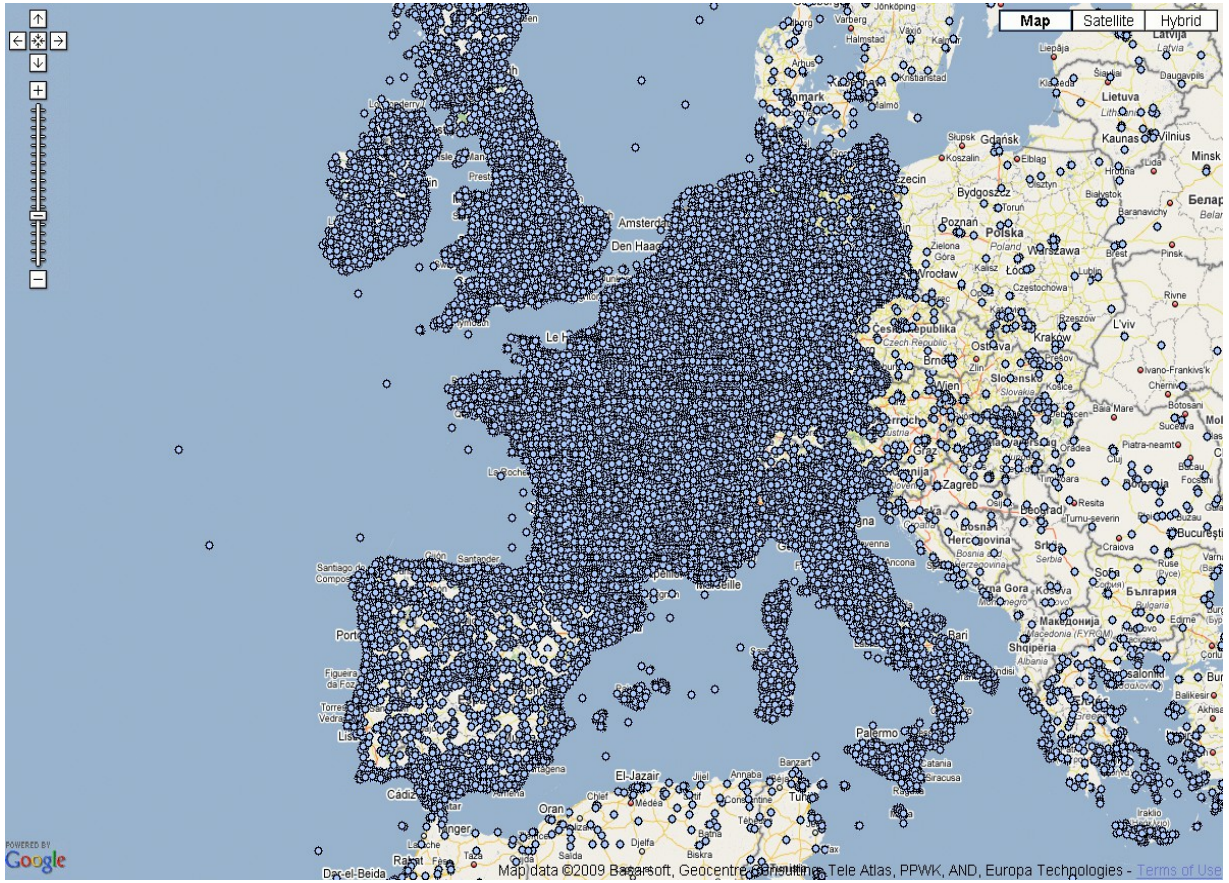
KNN-GiST

- KNN — найти K ближайших соседей (K- nearest neighbourhood)
- Очень трудная для традиционных алгоритмов поиска
  - Индекс не помогает, так как нет предиката
  - Требуется full table scan, сортировка
  - Используются разные «костыли» - range search, как задать нужный радиус ?
- Идея KNN-GiST
  - Использовать новую стратегию обхода поискового дерева (GiST) - Priority Queue вместо DFS (BFS)
  - Сразу получаем необходимые K-записей в нужном порядке
  - Чем больше таблица, тем больше выигрыш KNN-GiST

KNN-GiST: Как найти нужный радиус ?

Рестораны





KNN-search: Examples

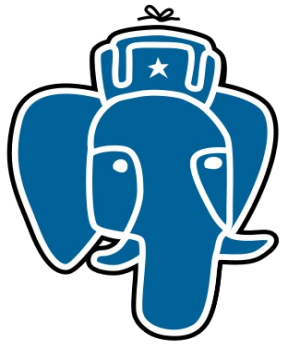
- Synthetic data – 1,000,000 randomly distributed points

```
create table qq ( id serial, p point, s int4);

insert into qq (p,s)  select point( p.lat, p.long), (random()*1000)::int
from ( select  (0.5-random())*180 as lat, random()*360 as long
       from ( select generate_series(1,1000000) ) as t
) as p;
create index qq_p_s_idx on qq using gist(p);
analyze qq;
```

- Query – find k-closest points to (0,0)

```
set enable_indexscan=on|off;
explain (analyze on, buffers on)

  SELECT * FROM qq ORDER BY (p <-> '(0,0)') ASC LIMIT 10;
```

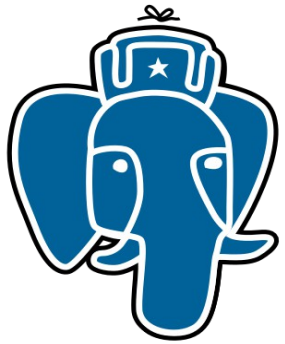
- <-> - distance operator, provided by data type



KNN-search: Examples

- Выигрыш 1700 раз для K=10 !

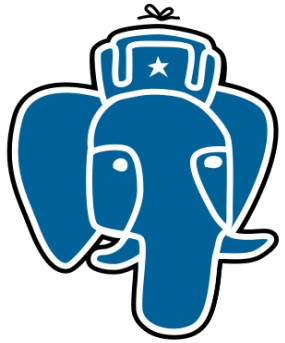
```
 Limit (actual time=327.674..327.675 rows=10 loops=1)
   Buffers: shared hit=7353
   ->  Sort (actual time=327.673..327.674 rows=10 loops=1)
         Sort Key: ((p <-> '(0,0)'::point))
         Sort Method: top-N heapsort  Memory: 25kB
         Buffers: shared hit=7353
         ->  Seq Scan on qq (actual time=0.009..180.513 rows=1000000 loops=1)
               Buffers: shared hit=7353
 Execution time: 327.698 ms
--------------------------
 Limit (actual time=0.116..0.160 rows=10 loops=1)
   Buffers: shared hit=14
   ->  Index Scan using qq_p_s_idx on qq (actual time=0.116..0.156 rows=10 loops=1)
         Order By: (p <-> '(0,0)'::point)
         Buffers: shared hit=14
 Execution time: 0.183 ms
```



KNN-search: Examples

- Выигрыш 220 раз для K=1000 !

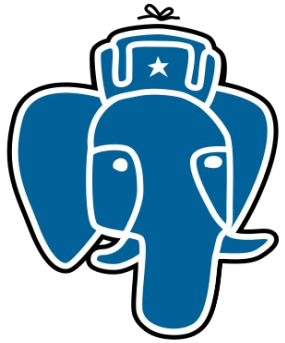
```
 Limit (actual time=314.262..314.471 rows=1000 loops=1)
   Buffers: shared hit=7353
   ->  Sort (actual time=314.261..314.365 rows=1000 loops=1)
         Sort Key: ((p <-> '(0,0)'::point))
         Sort Method: top-N heapsort  Memory: 127kB
         Buffers: shared hit=7353
         ->  Seq Scan on qq (actual time=0.008..172.222 rows=1000000 loops=1)
               Buffers: shared hit=7353
 Execution time: 314.599 ms
--------------------------
 Limit (actual time=0.073..1.334 rows=1000 loops=1)
   Buffers: shared hit=1016
   ->  Index Scan using qq_p_s_idx on qq (actual time=0.072..1.225 rows=1000 loops=1)
         Order By: (p <-> '(0,0)'::point)
         Buffers: shared hit=1016
 Execution time: 1.429 ms1
```



KNN-search: Examples

- Выигрыш 15600 раз для K=10 и N=10,000,000 !

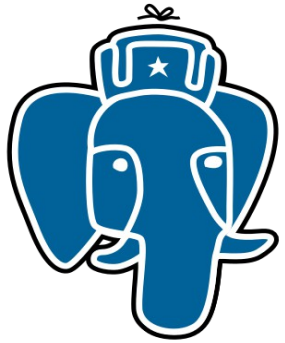
```
 Limit (actual time=3212.470..3212.473 rows=10 loops=1)
   Buffers: shared hit=96 read=73434
   ->  Sort (actual time=3212.469..3212.470 rows=10 loops=1)
         Sort Key: ((p <-> '(0,0)'::point))
         Sort Method: top-N heapsort  Memory: 25kB
         Buffers: shared hit=96 read=73434
         ->  Seq Scan on qq (actual time=0.009..1827.449 rows=10000000 loops=1)
               Buffers: shared hit=96 read=73434
 Execution time: 3212.492 ms
--------------------------
 Limit (actual time=0.143..0.184 rows=10 loops=1)
   Buffers: shared read=14
   ->  Index Scan using qq_p_s_idx on qq (actual time=0.143..0.182 rows=10 loops=1)
         Order By: (p <-> '(0,0)'::point)
         Buffers: shared read=14
 Execution time: 0.205 ms
```



KNN-search: Очепятки

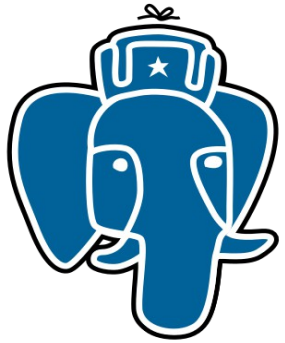
- Триграммы (лопата: '__л', '_ло', 'лоп', 'опа', 'пат', 'ата')
- Метрика (похожесть) S (W1, W2)
  - $N_{union}/max(N_1, N_2)$
  - $2 * N_{union}/(N_1 + N_2)$
  - $N_{union}/(N_1 + N_2 - N_{union})$
- Расстояние D (W1,W2):
  - 1 – S
  - 1/S

```
CREATE EXTENSION pg_trgm;

=# \d words
   Table "public.words"
 Column | Type | Modifiers
--------+------+-----------
 w      | text |
Indexes:
    "words_w_idx" gist (w gist_trgm_ops)
```



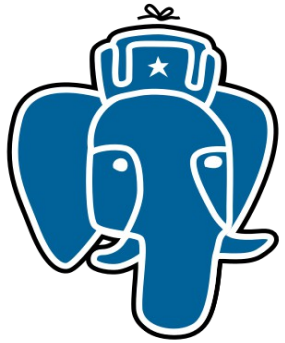
KNN-search: Очепятки

```
select w from words order by w <-> 'risource' limit 5;
    w
----------
 resource
 rice
 rivets
 ridden
 rivalled
(5 rows)
Limit (actual time=0.333..0.345 rows=5 loops=1)
   ->  Index Scan using words_w_idx on words (actual time=0.333..0.344 rows=5 loops=1)
         Order By: (w <-> 'risource'::text)
 Execution time: 0.366 ms
```



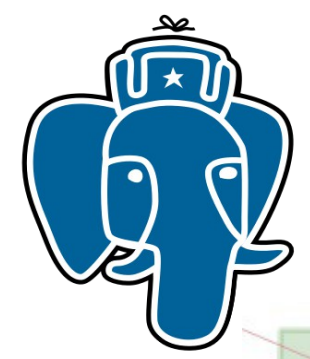
KNN-search: Очепятки (Lateral Query)

```
SELECT w.w, s.w as similar FROM words w CROSS JOIN LATERAL (SELECT w FROM words
ORDER BY w.w <-> w limit 2) s WHERE w.w LIKE 'ar%' AND w.w <> s.w;
      w       |   similar
--------------+-------------
 archiver     | anniversary
 armadillos   | armchairs
 artillery    | distillery
 arithmetized | authorized
 armchairs    | airspeed
(5 rows)
 Nested Loop (actual time=0.727..2.592 rows=5 loops=1)
   ->  Index Scan using words_w_idx on words w (actual time=0.127..0.165 rows=5 loops=1)
         Index Cond: (w ~~ 'ar%'::text)
   ->  Subquery Scan on s (actual time=0.483..0.483 rows=1 loops=5)
         Filter: (w.w <> s.w)
         Rows Removed by Filter: 1
         ->  Limit (actual time=0.321..0.481 rows=2 loops=5)
               ->  Index Scan using words_w_idx on words (actual time=0.318..0.477 rows=2 loops=5)
                     Order By: (w <-> w.w)
 Execution time: 2.633 ms
```

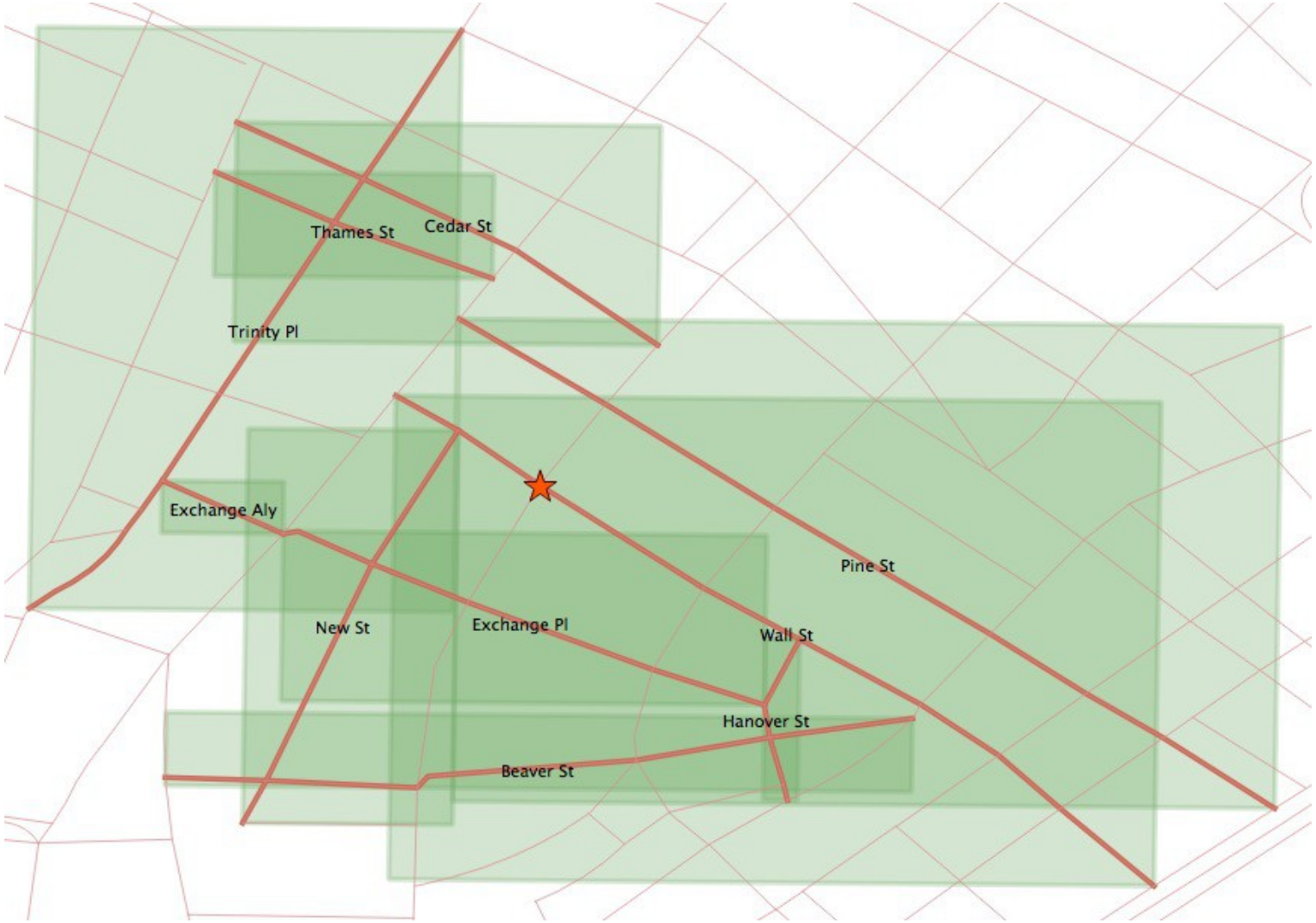


KNN-search: Очепятки

```
=# select w from words order by w <-> '%isourc%' limit 5;
     w
----------
 resource
 iron
 into
 idles
 injure
(5 rows)
 Limit (actual time=0.251..0.261 rows=5 loops=1)
   ->  Index Scan using words_w_idx on words (actual time=0.251..0.260 rows=5 loops=1)
         Order By: (w <-> '%isourc%'::text)
 Execution time: 0.282 ms
```

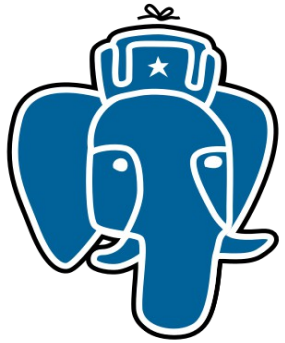


KNN-Search: Ограничения



Как найти ближайшие улицы, зная только их MBR?

Расстояние до границы MBR или до центра MBR даёт лишь приближение.

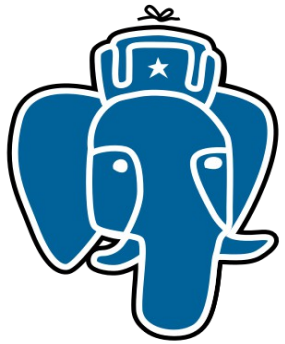


KNN-Search: Ограничения

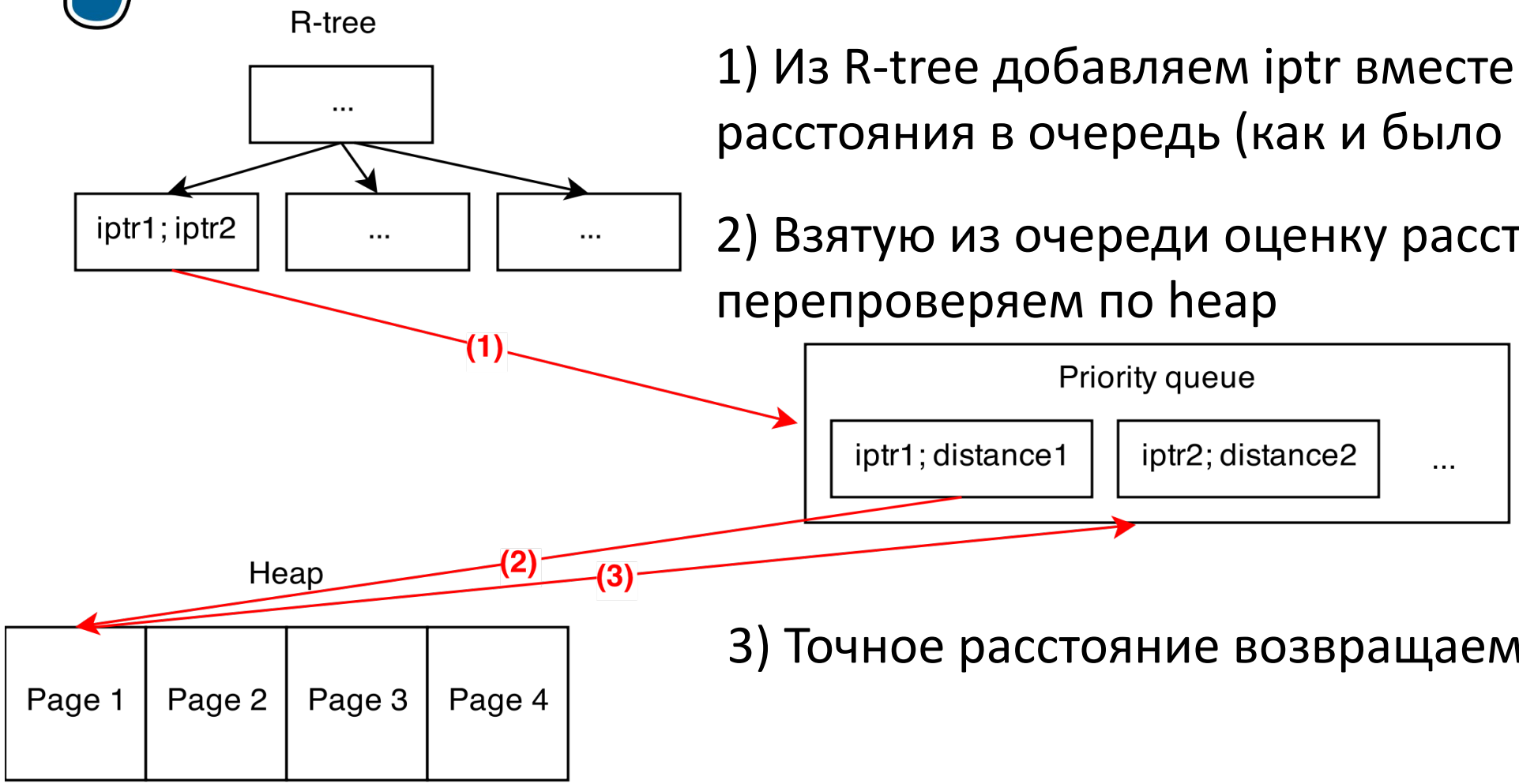
```
WITH closest_candidates AS (
  SELECT streets.gid, streets.name, streets.geom
  FROM nyc_streets streets
  ORDER BY streets.geom <->
    'SRID=26918;POINT(583571.905921312 4506714.34119218)'::geometry
  LIMIT 100
)
SELECT gid, name
FROM closest_candidates
ORDER BY ST_Distance(geom,
    'SRID=26918;POINT(583571.905921312 4506714.34119218)'::geometry)
LIMIT 1;
```

User-level hack.

Откуда известно, что 100 оптимальное значение?



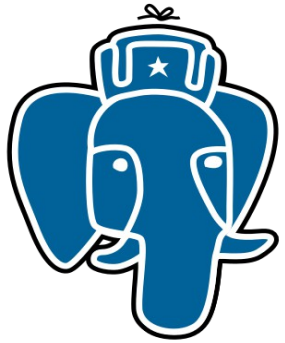
Решение: Exact KNN



1) Из R-tree добавляем iptr вместе с оценкой расстояния в очередь (как и было раньше)

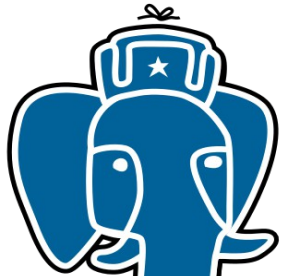
2) Взятую из очереди оценку расстояния перепроверяем по heap

3) Точное расстояние возвращаем в heap



Exact KNN: статус

- Даёт правильный результат за минимально возможное время.
- Выложен патч на commitfest.
- Есть много архитектурных вопросов (доступ к heap из index scan).

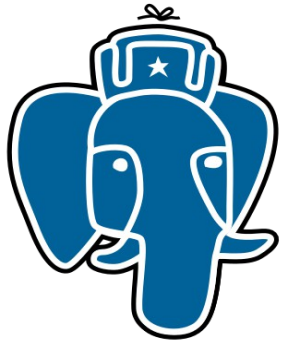


KNN-GiST: Похожие картинки

Query:

Similar:

Работает!



KNN-GiST: Похожие картинки

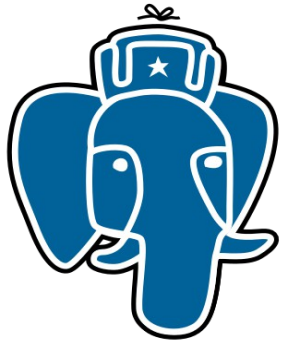
Query:

Similar:

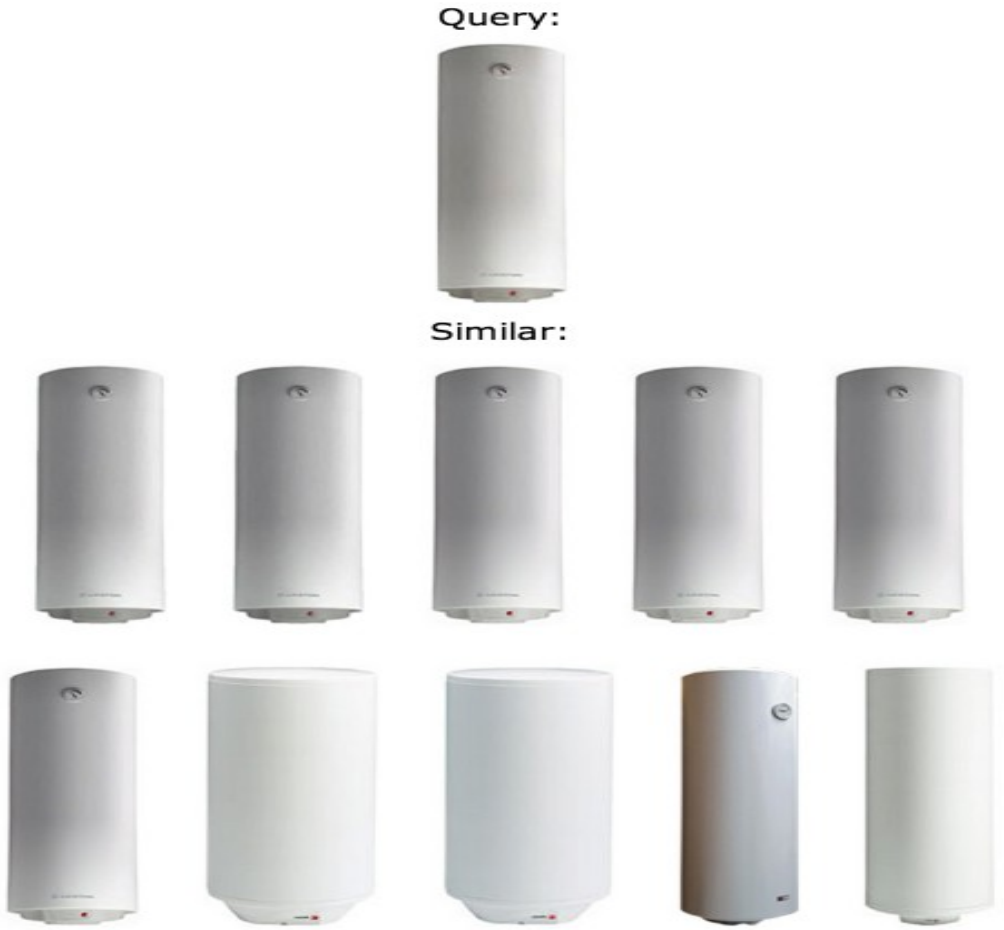
Работает!

KNN-GiST: Похожие картинки

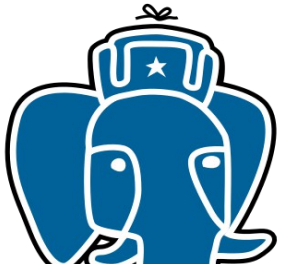
Работает!



KNN-GiST: Похожие картинки



Работает!



KNN-GiST: Похожие картинки

Query:

Similar:

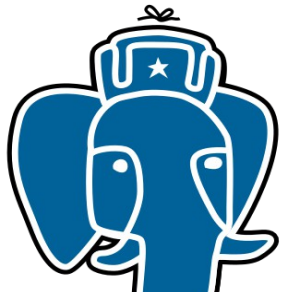









Работает!

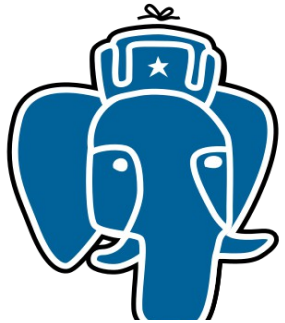


KNN-GiST: Похожие картинки

Query:

Similar:

Работает!

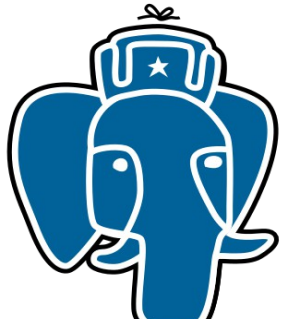


KNN-GiST: Похожие картинки

Query:

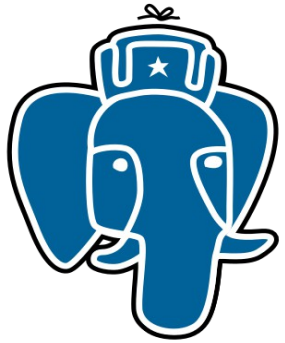
Similar:

Работает!



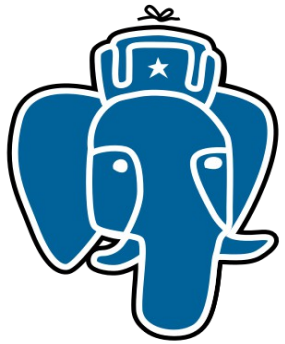
KNN-GiST: Похожие картинки

«Не идеально»



Как это устроено: предобработка

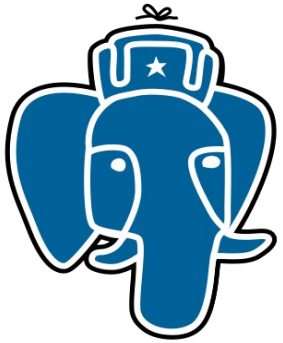
```
CREATE TABLE pat AS (
        SELECT
                id,
                shuffle_pattern(pattern) AS pattern,
                pattern2signature(pattern) AS signature
        FROM (
                SELECT
                        id,
                        jpeg2pattern(data) AS pattern
                FROM
                        image
        ) x
);
CREATE INDEX pat_signature_idx ON pat USING gist (signature);
CREATE INDEX pat_id_idx ON pat(id);
```



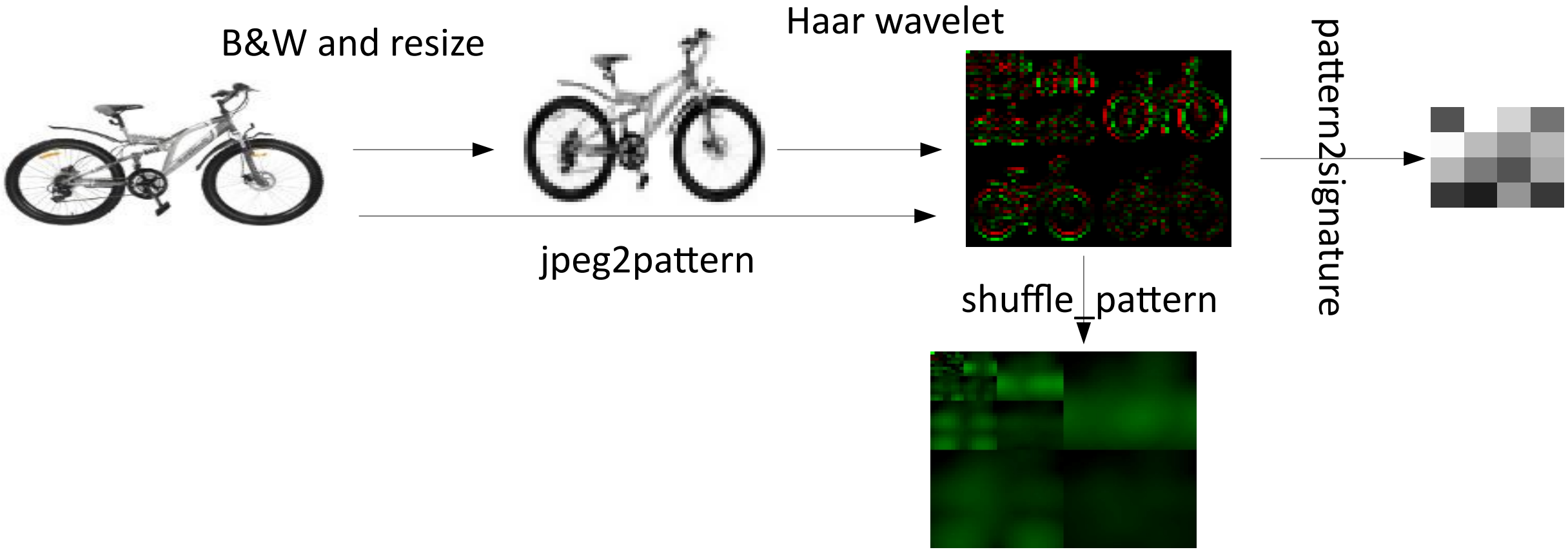
Как это устроено: поисковый запрос

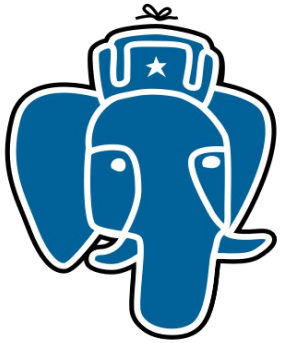
```
SELECT
        id,
        smlr
FROM
(
        SELECT
                id,
                pattern <-> (SELECT pattern FROM pat WHERE id = :id) AS smlr
        FROM pat
        WHERE id <> :id
        ORDER BY
                signature <-> (SELECT signature FROM pat WHERE id = :id)
        LIMIT 100
) x
ORDER BY x.smlr ASC
LIMIT 10
```

Можно будет применить Exact-KNN!

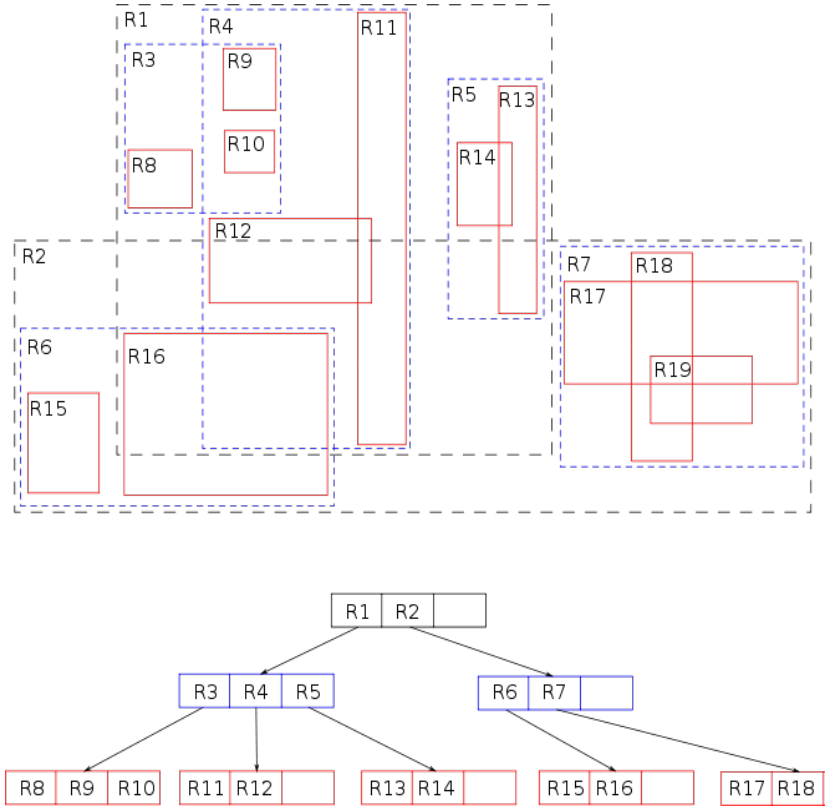


Как это устроено внутри?

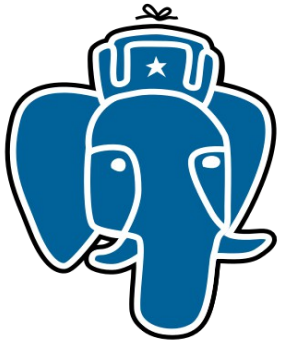




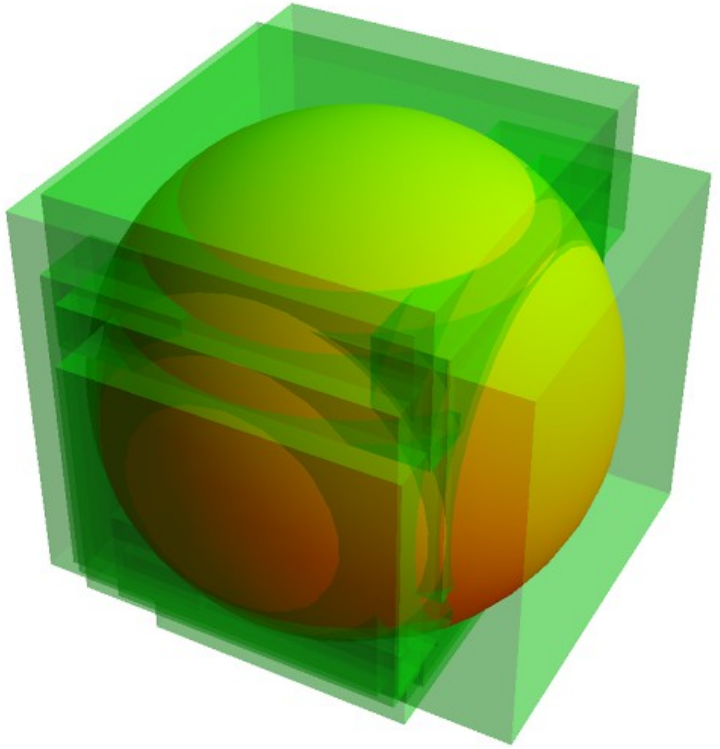
Spaghetti indexing …



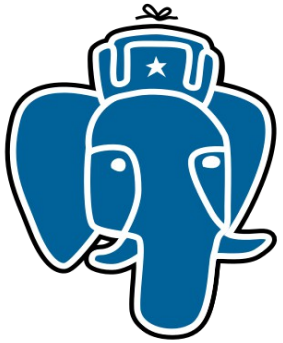
R-tree fails here – bounding box of each separate spaghetti is the same



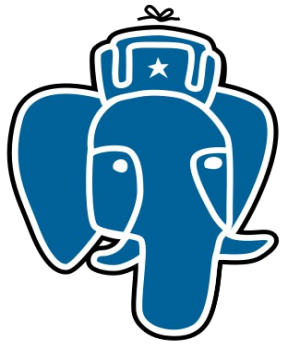
Spaghetti indexing …



R-tree fails here – bounding box of each separate spaghetti is the same

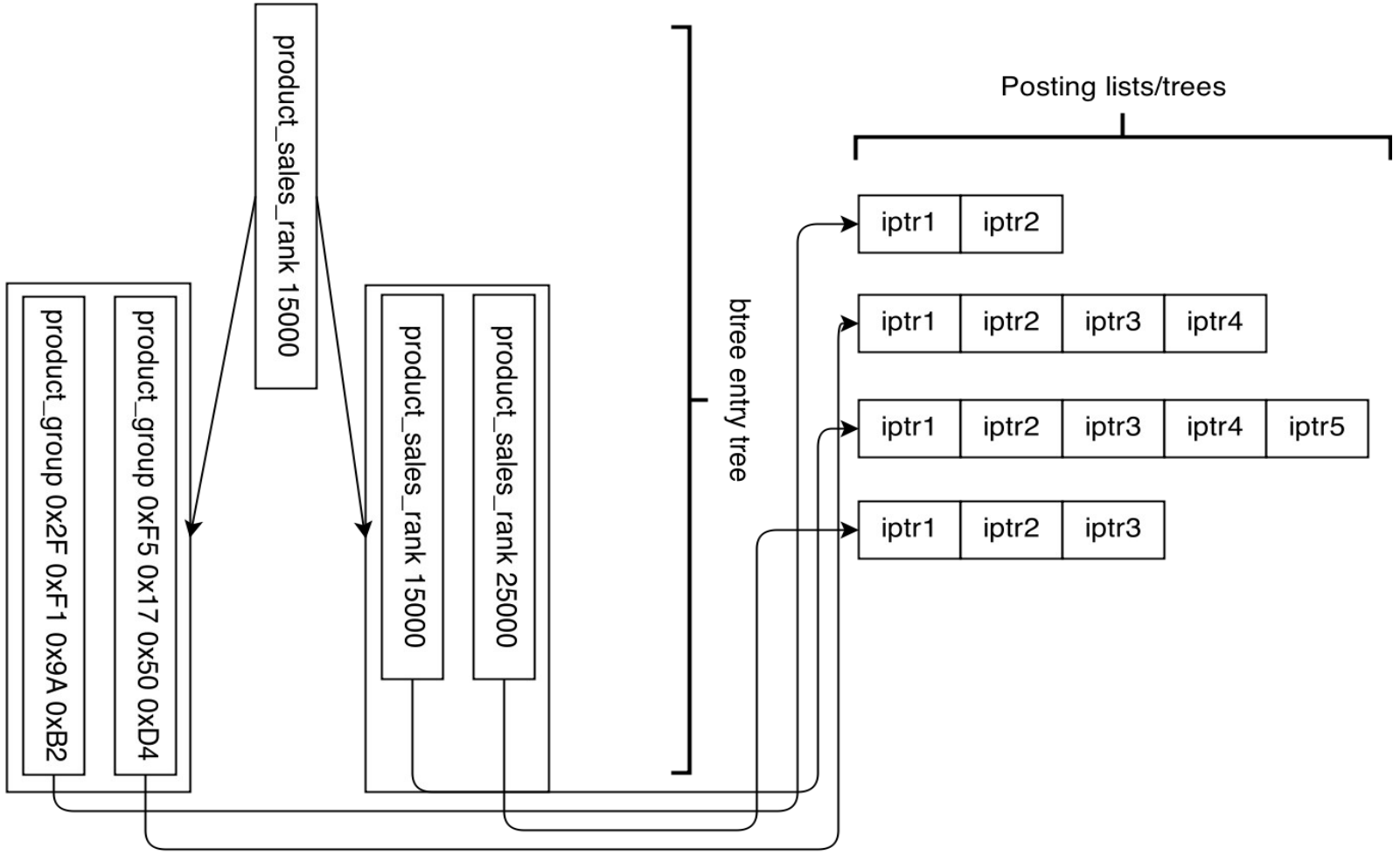


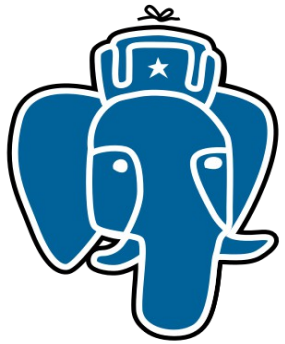
Idea: Use multiple boxes



GIN index structure for jsonb

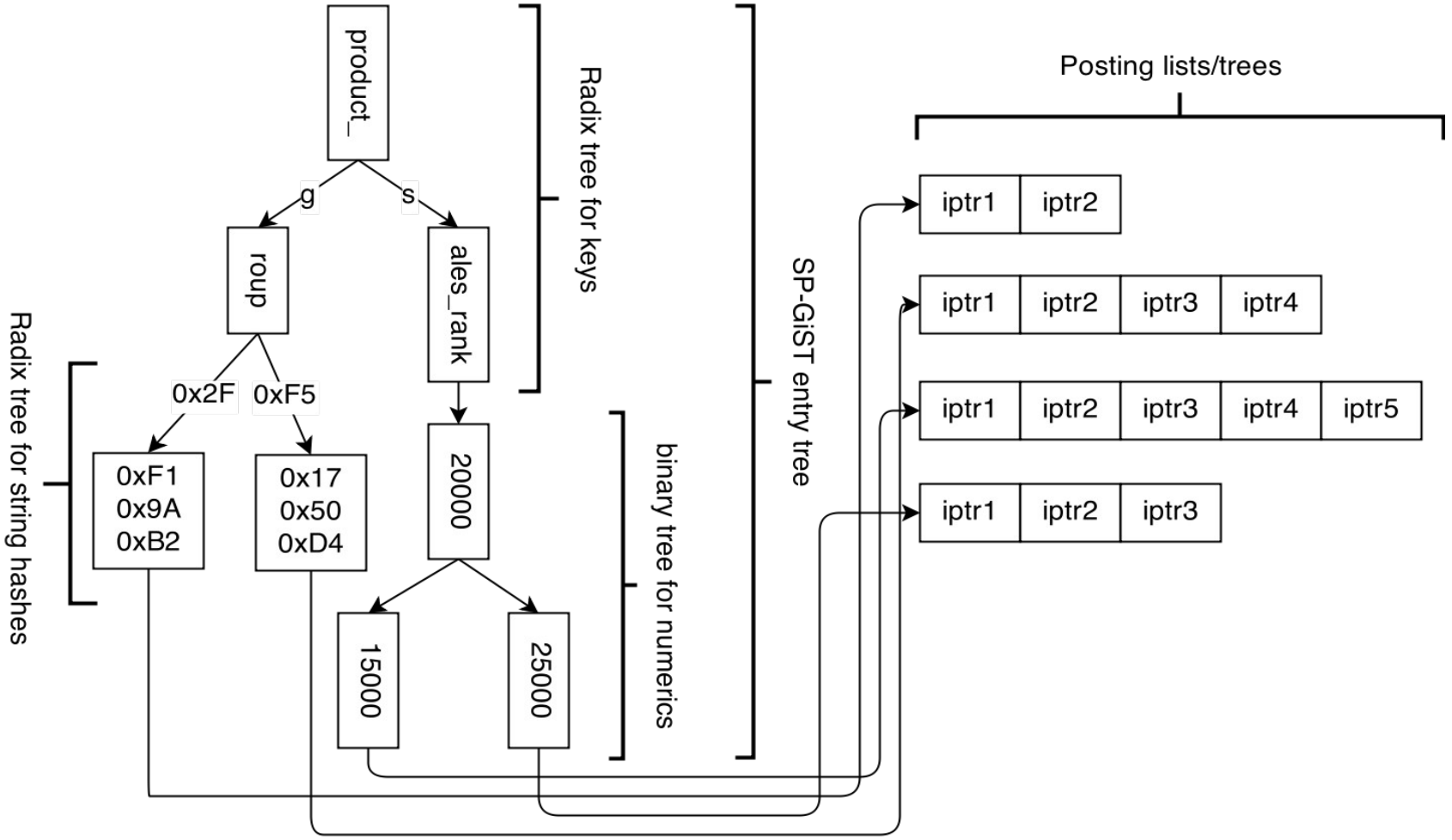
```
{
   "product_group": "Book",
   "product_sales_rank": 15000
},
{
   "product_group": "Music",
   "product_sales_rank": 25000
}
```

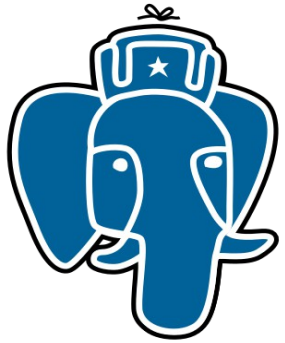




Vodka index structure for jsonb

```
{
   "product_group": "Book",
   "product_sales_rank": 15000
},
{
   "product_group": "Music",
   "product_sales_rank": 25000
}
```

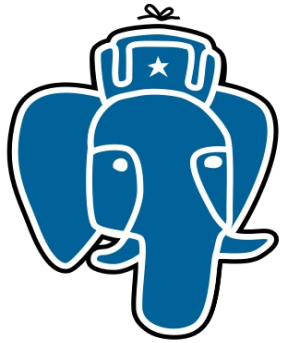




Indexing regular expressions (9.3+)

- contrib/pg_trgm supports indexing of regular-expression searches

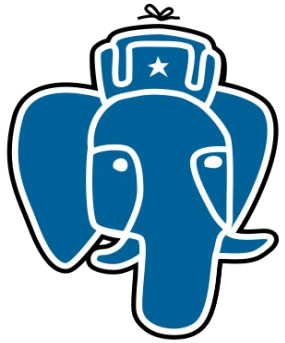
```
select w from words where  w ~ '.*sourc.*';
 Index Scan using words_w_idx on words  (cost=0.14..8.16 rows=1 width=9)
                                (actual time=0.165..0.263 rows=1 loops=1)
   Index Cond: (w ~ '.*sourc.*'::text)
 Planning time: 0.150 ms
 Execution time: 0.279 ms
(4 rows)

explain analyze select w from words where  w ~ '^r.*sour.*e$';
 Index Scan using words_w_idx on words  (cost=0.14..8.16 rows=1 width=9)
                                (actual time=0.209..0.375 rows=1 loops=1)
   Index Cond: (w ~ '^r.*sour.*e$'::text)
 Planning time: 0.168 ms
 Execution time: 0.399 ms
(4 rows)
```



Indexing regular expressions (9.3+)

- contrib/pg_trgm supports indexing of regular-expression searches

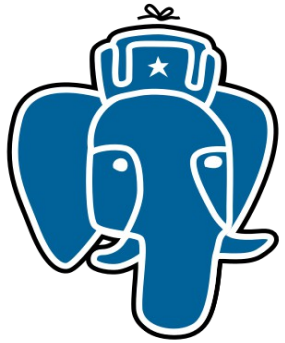
```
select * from man_lines where man_line ~*
'(?:(?:p(?:ostgres(?:ql)?|g?sql)|sql)) (?:(?:(?:mak|us)e|do|is))';

Bitmap Heap Scan on man_lines  (cost=285.32..930.42 rows=170 width=79)
(actual time=579.765..592.435 rows=123 loops=1)
   Recheck Cond: (man_line ~* '(?:(?:p(?:ostgres(?:ql)?|g?sql)|sql)) (?:(?:(?:mak|us
   Rows Removed by Index Recheck: 1909
   ->  Bitmap Index Scan on trgm_idx  (cost=0.00..285.28 rows=170 width=0)
(actual time=578.780..578.780 rows=2032 loops=1)
         Index Cond: (man_line ~* '(?:(?:p(?:ostgres(?:ql)?|g?sql)|sql)) (?:(?:(?:mal
 Total runtime: 592.474 ms
(6 rows)
```



Целостность данных

- Индексы обеспечивают CONSTRAINTS: PRIMARY KEY, UNIQUE (DEFERRABLE)
- EXCLUSION CONSTRAINTS (9.0, Jeff Davis)
  - UNIQUE использует оператор '=' BTREE
  - Как реализовать уникальность более сложных типов, например, многоугольников ? Триггеры — безумно медленно.
  - EXCLUSION CONSTRAINTS позволяют использовать произвольные коммутативные операторы (x*y=y*x) с индексной поддержкой

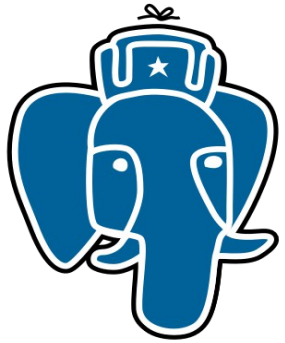
«Add exclusion constraints, which generalize the concept of uniqueness to support any indexable commutative operator, not just equality. Two rows violate the exclusion constraint if "row1.col OP row2.col" is TRUE for each of the columns in the constraint. »



EXCLUSION CONSTRAINTS: Example

- Семинары не должны пересекаться

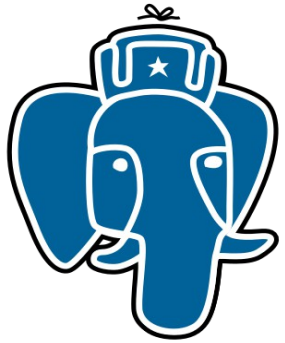
```
CREATE EXTENSION btree_gist;
CREATE TABLE reservation (
      room TEXT,
      professor TEXT,
      during tstzrange,
      EXCLUDE USING gist (room   WITH =,
                          during WITH &&)
);
```



EXCLUSION CONSTRAINTS: Example

- Обычный UNIQUE — оператор BTREE '='

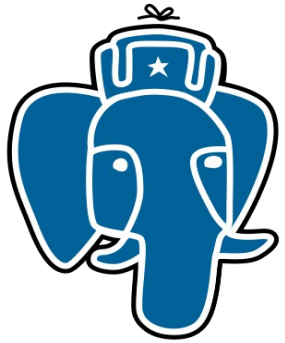
```
=# CREATE TABLE test (i INT4, EXCLUDE (i WITH =));
CREATE TABLE
=# \d test
     Table "public.test"
 Column |  Type   | Modifiers
--------+---------+-----------
 i      | integer |
Indexes:
    "test_i_excl" EXCLUDE USING btree (i WITH =)
```



EXCLUSION CONSTRAINTS: Example

- Все комнаты должны быть одинаковые (unUnique)

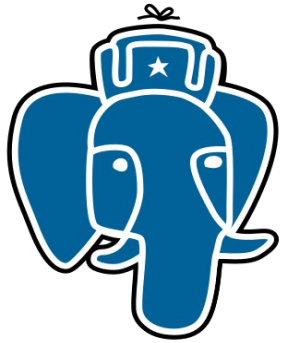
```
CREATE TABLE reservation (
      room TEXT,
      professor TEXT,
      during tstzrange,
      EXCLUDE USING gist (room   WITH <>
);
```



EXCLUSION CONSTRAINTS: Example

- Можно использовать разные типы индексов

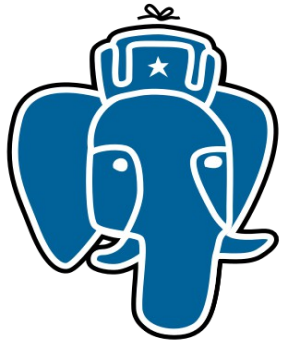
```
CREATE TABLE boxes_unique (
        id integer,
        box box,
       EXCLUDE USING btree (id  WITH = ),
       EXCLUDE USING gist  (box WITH &&)
);
=# \d boxes_unique
 Table "public.boxes_unique"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
 box    | box     |
Indexes:
    "boxes_unique_box_excl" EXCLUDE USING gist (box WITH &&)
    "boxes_unique_id_excl" EXCLUDE USING btree (id WITH =)
```



EXCLUSION CONSTRAINTS: Example

- Можно использовать разные типы индексов. Вставляем вложенный квадрат

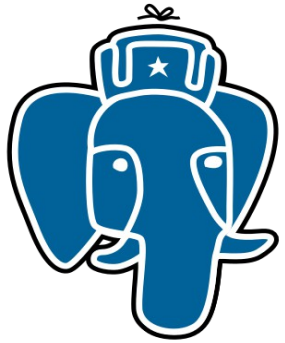
```
insert into boxes_unique values(1, '((0,0),(2,2))');
INSERT 0 1
insert into boxes_unique values(1, '((0,0),(1,1))');
ERROR:  conflicting key value violates exclusion constraint
        "boxes_unique_id_excl"
DETAIL:  Key (id)=(1) conflicts with existing key (id)=(1).
insert into boxes_unique values(2, '((0,0),(1,1))');
ERROR:  conflicting key value violates exclusion constraint
         "boxes_unique_box_excl"
DETAIL:  Key (box)=((1,1),(0,0)) conflicts with existing
         key (box)=((2,2),(0,0)).
```



EXCLUSION CONSTRAINTS: Example

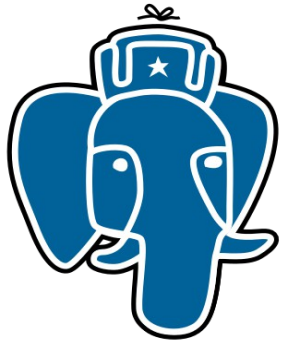
- Можно использовать частичный (partial) индекс.
  - Нельзя вставить одинаковые квадратики в избранную область.

```
create table boxes_unique (
       id integer,
       box box,
      exclude using btree (id  with =),
      exclude using gist (box with &&)
             WHERE      (box <@ '((0,0),(5,5))')
);
insert into boxes_unique values(1, '((0,0),(1,1))');
INSERT 0 1
insert into boxes_unique values(2, '((0,0),(10,10))');
INSERT 0 1
insert into boxes_unique values(3, '((0,0),(4,4))');
ERROR:  conflicting key value violates exclusion constraint "boxes_unique_box_excl"
DETAIL:  Key (box)=((4,4),(0,0)) conflicts with existing key (box)=((1,1),(0,0)).
```



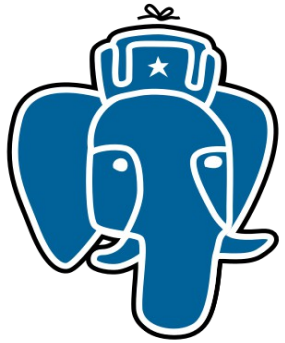
Мониторинг индексов

- Неиспользуемые индексы
  Индексы-дубликаты
  Пересекающиеся индексы
  - Результат эволюционного развития и/или бардака
  - Занимают место
  - Замедляют обновление
  - Замедляют репликацию
- Дело DBA мониторить индексы
- http://wiki.postgresql.org/wiki/Index_Maintenance



Мониторинг индексов::неиспользуемые индексы

```
SELECT
    schemaname || '.' || relname AS table,
    indexrelname AS index,
    pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size,
    idx_scan as index_scans
  FROM pg_stat_user_indexes ui
  JOIN pg_index i ON ui.indexrelid = i.indexrelid
  WHERE NOT indisunique AND idx_scan < 50 AND pg_relation_size(relid) > 5 * 819
  ORDER BY pg_relation_size(i.indexrelid) / nullif(idx_scan, 0) DESC NULLS FIRS
  pg_relation_size(i.indexrelid) DESC;
     table      |      index       | index_size | index_scans
----------------+------------------+------------+-------------
 public.apod    | gin_apod_fts_idx | 1616 kB    |           0
 public.tbl     | ab1_idx          | 40 kB      |           0
 public.tbl     | a_idx            | 40 kB      |           0
 public.tbl     | ba_idx           | 40 kB      |           0
 public.reviews | reviews_gin_idx  | 160 MB     |           9
 public.events  | e_date_id        | 3352 kB    |           4
 public.tbl     | ab_idx           | 88 kB      |          28
(7 rows)
```

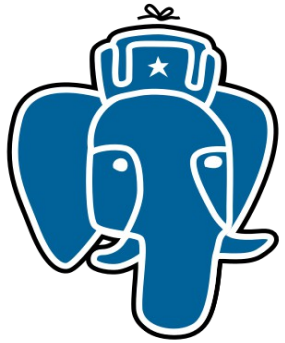


Мониторинг индексов

- Индексы-дубликаты
  - Учитываем все пересечения

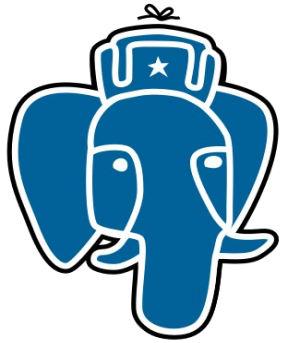
```
=# \d tbl
      Table "public.tbl"
 Column |  Type   | Modifiers
--------+---------+-----------
 a      | integer |
 b      | integer |
Indexes:
    "a_idx" btree (a)
    "ab1_idx" btree (a, b)
    "ab_idx" btree (a, b)
    "ba_idx" btree (a, b)
```

```
SELECT a.indrelid::regclass, a.indexrelid::regclass, b.indexrelid::regclass
FROM (SELECT *,array_to_string(indkey,' ') AS cols FROM pg_index) a JOIN
           (SELECT *,array_to_string(indkey,' ') AS cols FROM pg_index) b  ON
( a.indrelid=b.indrelid AND a.indexrelid > b.indexrelid AND
   (    (a.cols LIKE b.cols||'%' AND coalesce(substr(a.cols,length(b.cols)+1,1),' ')=' ' ) OR
        (b.cols LIKE a.cols||'%' AND coalesce(substr(b.cols,length(a.cols)+1,1),' ')=' ')
   )
) ORDER BY indrelid;
 indrelid | indexrelid | indexrelid
----------+------------+------------
 tbl      | a_idx      | ab1_idx
 tbl      | a_idx      | ba_idx
 tbl      | a_idx      | ab_idx
(3 rows)
```



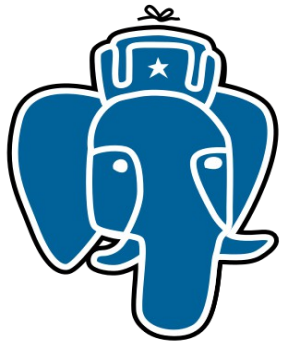
Мониторинг индексов::дубликаты V2

```
SELECT idstat.relname AS tname,indexrelname AS iname, idstat.idx_scan AS used,
       pg_size_pretty(pg_relation_size(idstat.relid)) AS tsize,
       pg_size_pretty(pg_relation_size(indexrelid)) AS isize,
       n_tup_upd + n_tup_ins + n_tup_del as writes, indexdef AS create
FROM pg_stat_user_indexes AS idstat
   JOIN pg_indexes  ON (indexrelname = indexname AND idstat.schemaname = pg_indexes.schemaname)
   JOIN pg_stat_user_tables AS tabstat ON idstat.relid = tabstat.relid
WHERE idstat.idx_scan  < 200 AND indexdef !~* 'unique'
ORDER BY idstat.relname, indexrelname;
  tname  |      iname       | used |  tsize  |  isize  | writes |                                creat
---------+------------------+------+---------+---------+--------+-------------------------------
 apod    | gin_apod_fts_idx |    0 | 2712 kB | 1616 kB |   1754 | CREATE INDEX gin_apod_fts_idx (
 events  | e_date_id        |    4 | 14 MB   | 3352 kB | 151643 | CREATE INDEX e_date_id ON event
 reviews | reviews_gin_idx  |    9 | 270 MB  | 160 MB  | 589859 | CREATE INDEX reviews_gin_idx ON
 tbl     | a_idx            |    0 | 72 kB   | 40 kB   |   2000 | CREATE INDEX a_idx ON tbl USING
 tbl     | ab1_idx          |    0 | 72 kB   | 40 kB   |   2000 | CREATE INDEX ab1_idx ON tbl USI
 tbl     | ab_idx           |   28 | 72 kB   | 88 kB   |   2000 | CREATE INDEX ab_idx ON tbl USIN
 tbl     | ba_idx           |    0 | 72 kB   | 40 kB   |   2000 | CREATE INDEX ba_idx ON tbl USIN
(7 rows)
```



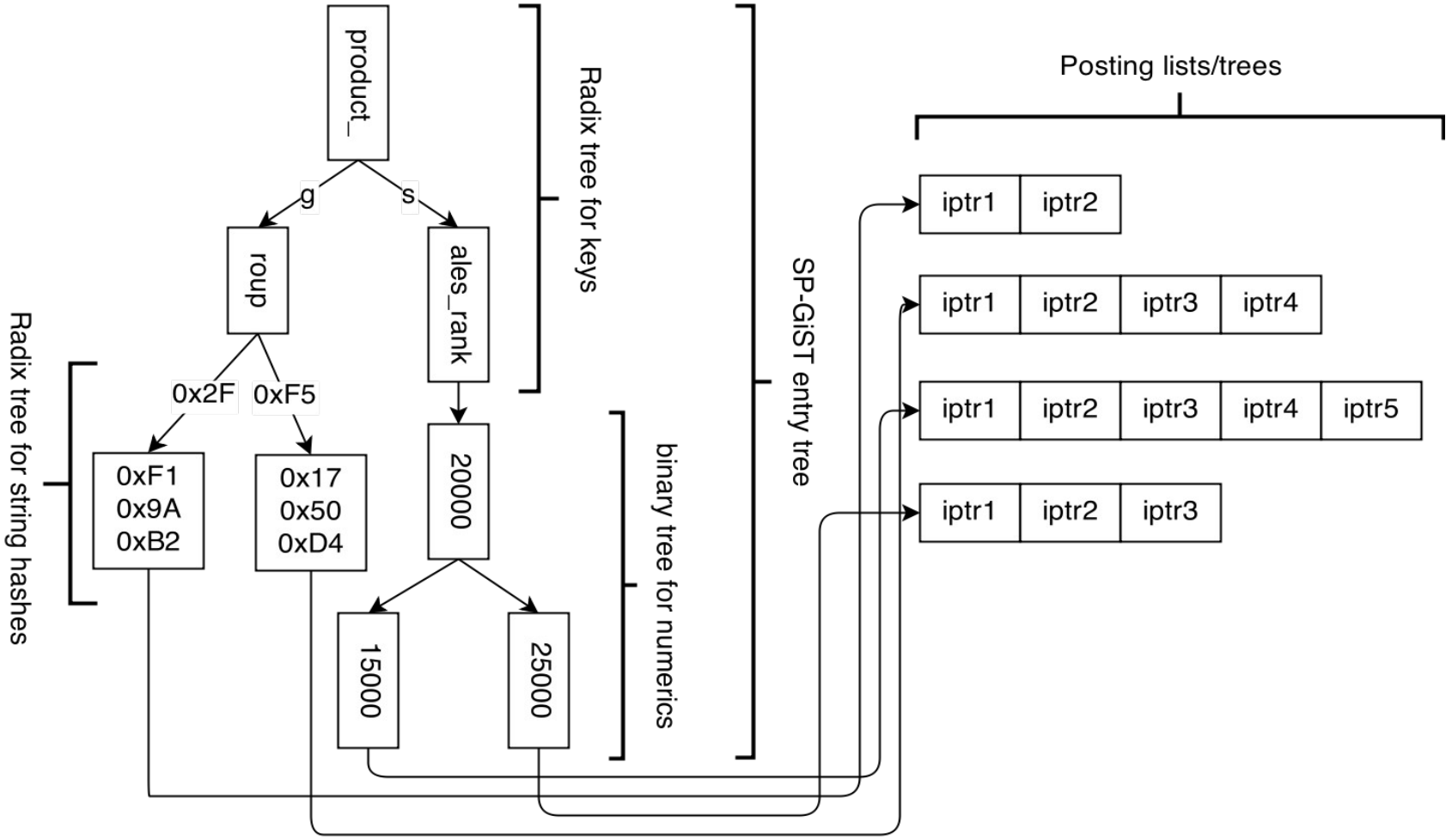
Много индексов - медленная вставка

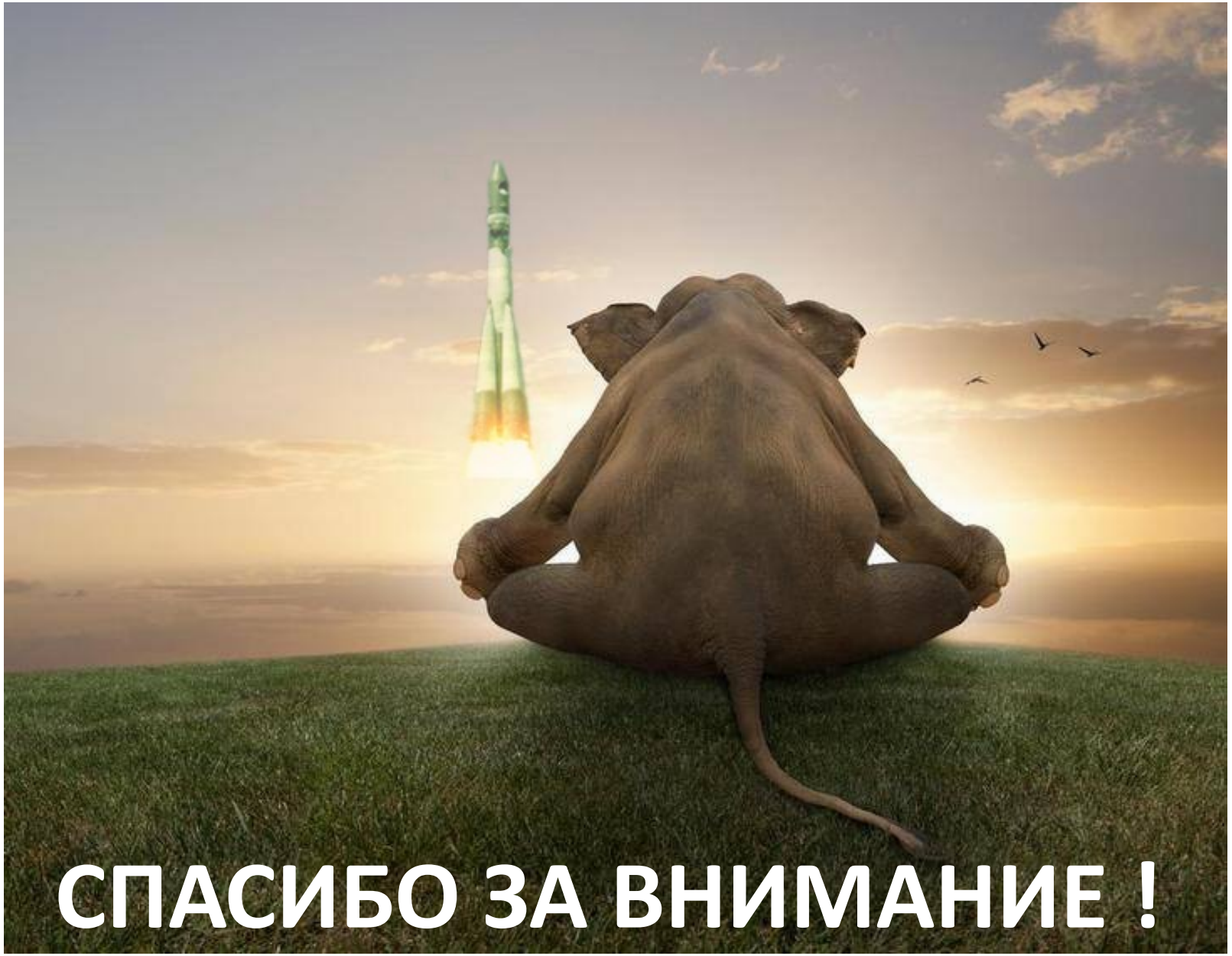
```
\d+ document;
                                  Table "public.document"
     Column      |            Type             | Modifiers | Storage  | Stats target | Description
-----------------+-----------------------------+-----------+----------+--------------+-------------
 id              | integer                     | not null  | plain    |              |
 kind            | character varying(255)      | not null  | extended |              |
 uid             | character varying(255)      | not null  | extended |              |
 sentenceId      | character varying(255)      | not null  | extended |              |
 text            | text                        | not null  | extended |              |
 hepval          | double precision            |           | plain    |              |
 created         | timestamp(0) without time zone | not null  | plain    |              |
 updated         | timestamp(0) without time zone |           | plain    |              |
 cardval         | double precision            |           | plain    |              |
 nephval         | double precision            |           | plain    |              |
 phosval         | double precision            |           | plain    |              |
 patternCount    | double precision            |           | plain    |              |
 ruleScore       | double precision            |           | plain    |              |
 hepTermNormScore | double precision           |           | plain    |              |
 hepTermVarScore  | double precision           |           | plain    |              |
 svm             | double precision            |           | plain    |              |
 svmConfidence   | double precision            |           | plain    |              |
Indexes:
    "DocumentOLD_pkey" PRIMARY KEY, btree (id)
    "document_cardval_index" btree (cardval)
    "document_heptermnorm_index" btree ("hepTermNormScore" DESC NULLS LAST)
    "document_heptermvar_index" btree ("hepTermVarScore" DESC NULLS LAST)
    "document_hepval_index" btree (hepval DESC NULLS LAST)
    "document_kind_index" btree (kind)
    "document_nephval_index" btree (nephval DESC NULLS LAST)
    "document_patterncount_index" btree ("patternCount" DESC NULLS LAST)
    "document_phosval_index" btree (phosval DESC NULLS LAST)
    "document_rulescore_index" btree ("ruleScore" DESC NULLS LAST)
    "document_sentenceid_index" btree ("sentenceId")
    "document_svm_index" btree (svm)
    "document_uid_index" btree (uid)
```



Vodka index structure for jsonb

```
{
   "product_group": "Book",
   "product_sales_rank": 15000
},
{
   "product_group": "Music",
   "product_sales_rank": 25000
}
```




СПАСИБО ЗА ВНИМАНИЕ !