

Вся правда об индексах в PostgreSQL

Олег Бартунов, Александр Коротков, Федор Сигаев
PostgreSQL Development Team

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
 - Ограничений целостности
 - 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 разработчиков 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 многоколоночные, только GIN-индекс порядок колонок не важен.
- 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:

- WHERE expr opr value
- ORDER BY expr [ASC|DESC]
- 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)
- Вся информацию постгрес берет из системного каталога

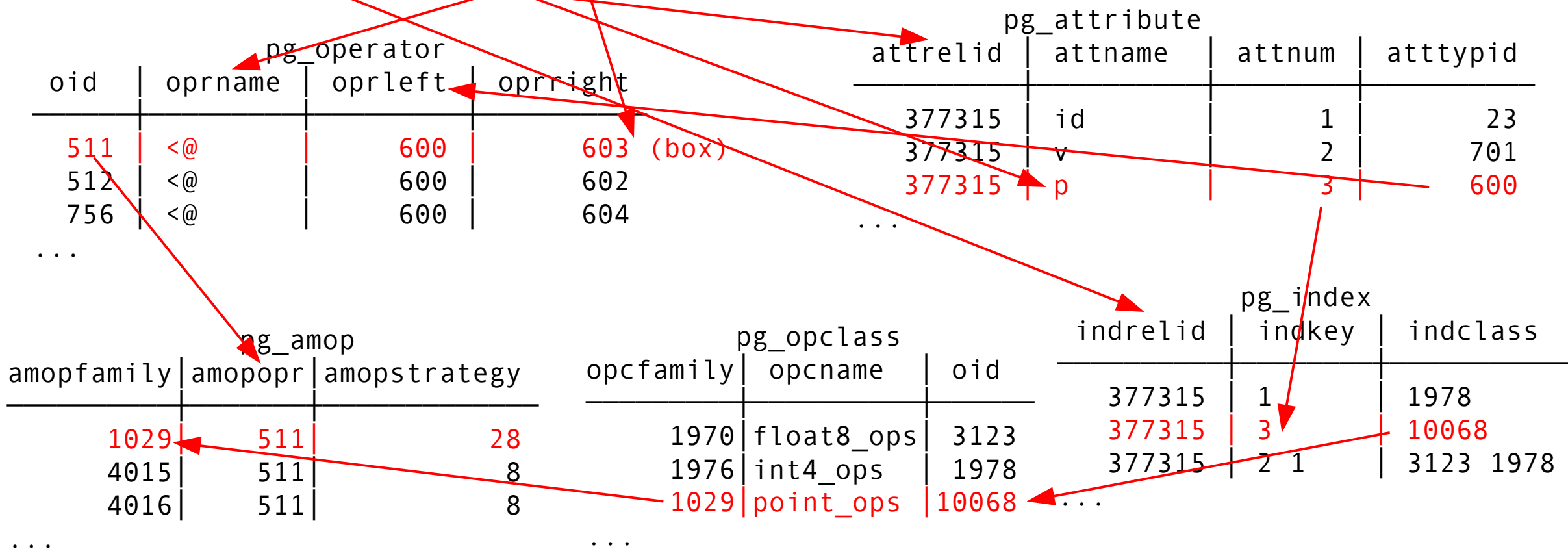
Выбор нужного индекса: пример

```
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=48.36..2805.41 rows=1000 width=28) (actual time=0.069..0.150)
  Recheck Cond: (p <@ '(0.51,0.51),(0.5,0.5)'::box)
  -> Bitmap Index Scan on test3_p_idx (cost=0.00..48.11 rows=1000 width=0) (actual time=0.000..0.000)
    Index Cond: (p <@ '(0.51,0.51),(0.5,0.5)'::box)
Total runtime: 0.172 ms
```


Выбор нужного индекса: схема

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



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 (найжены в `pg_amop`) могут быть использованы.

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

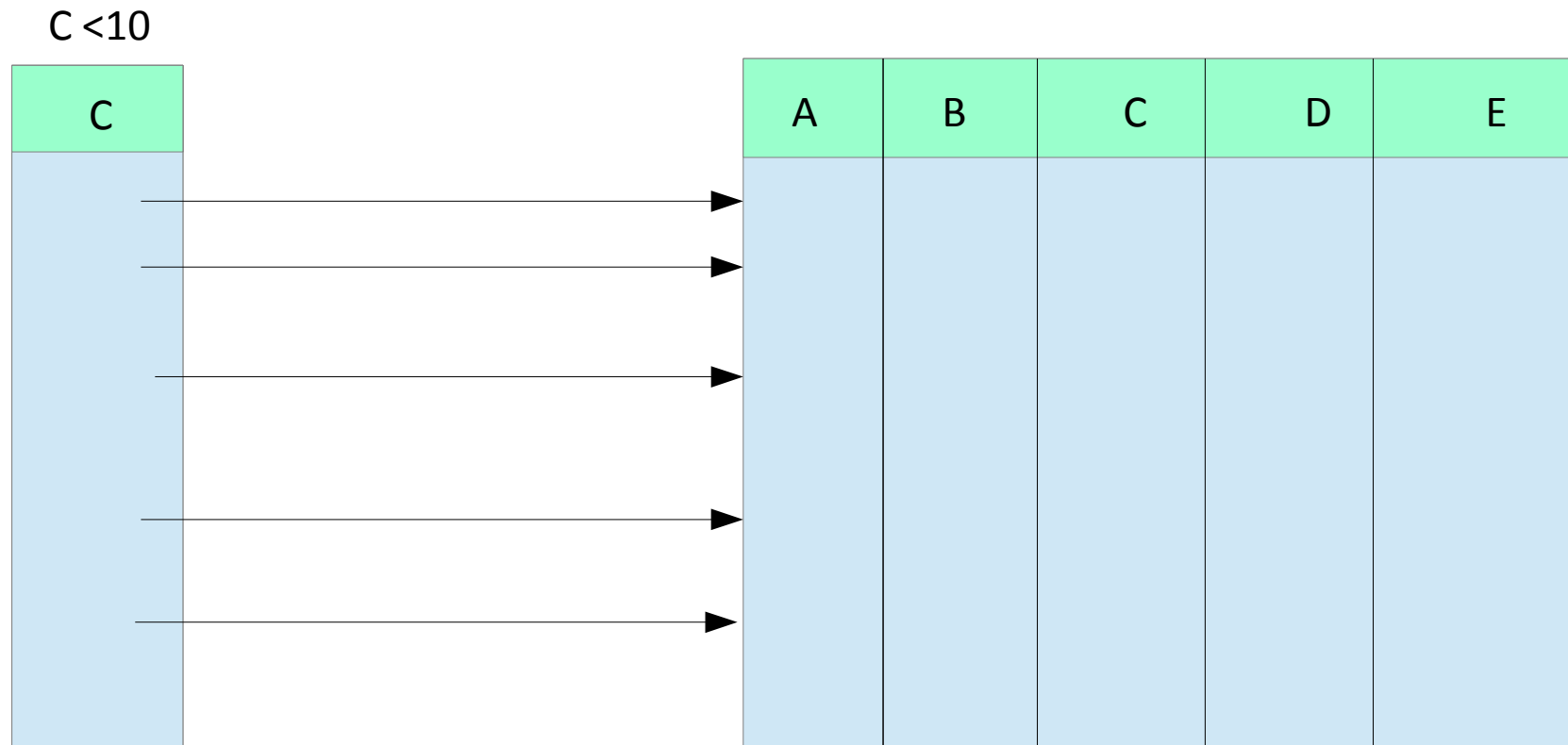
- Читаем последовательно таблицу и фильтруем записи по предикату

$C < 10$

A	B	C	D	E

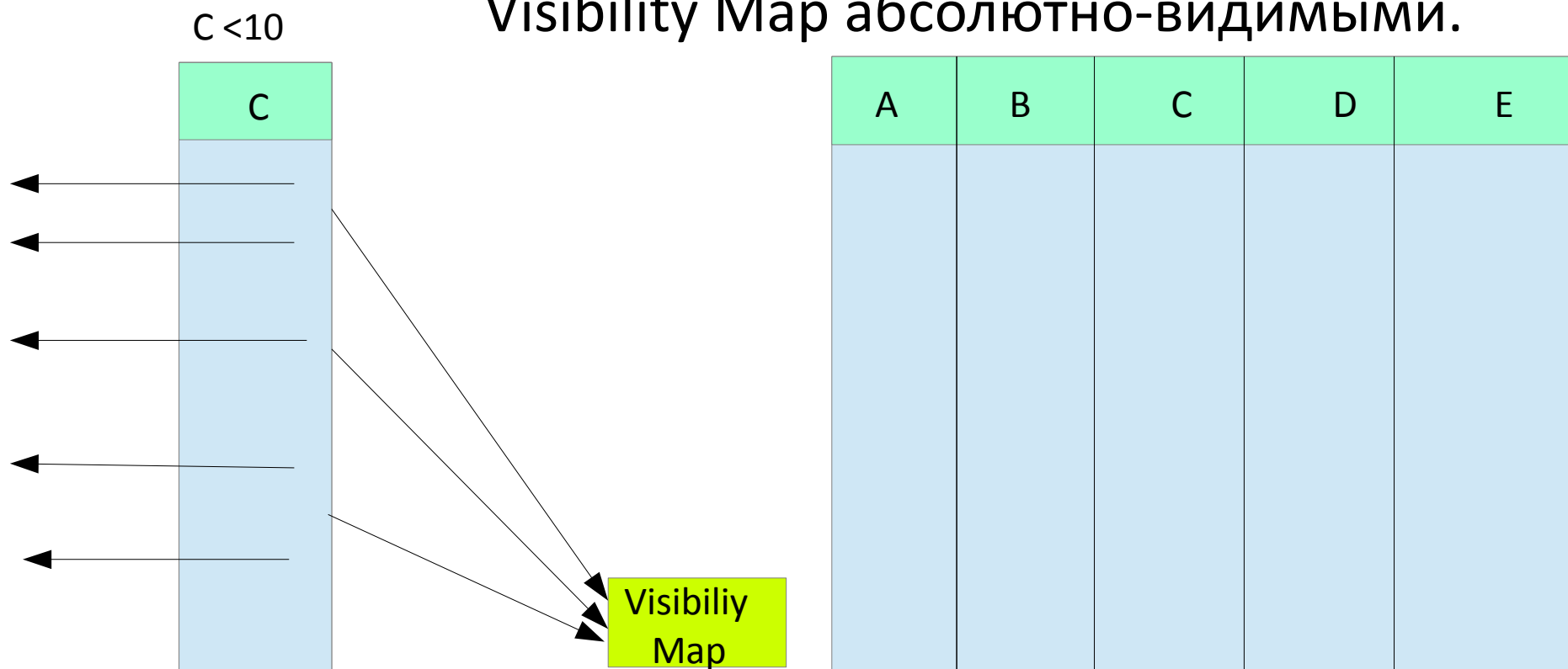
Простейший индекс (Index Scan)

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



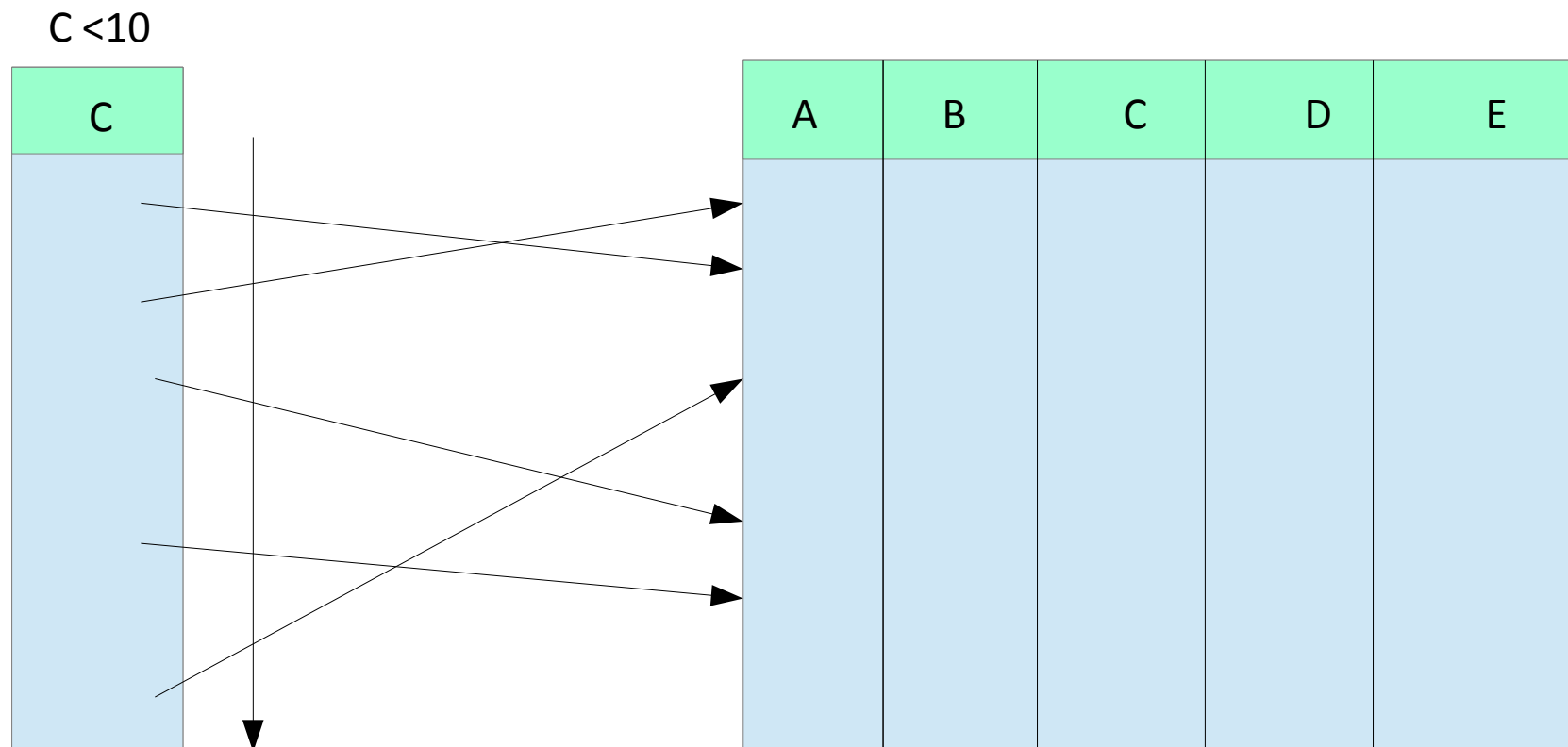
Простейший индекс (Index-only Scan)

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



Простейший индекс++

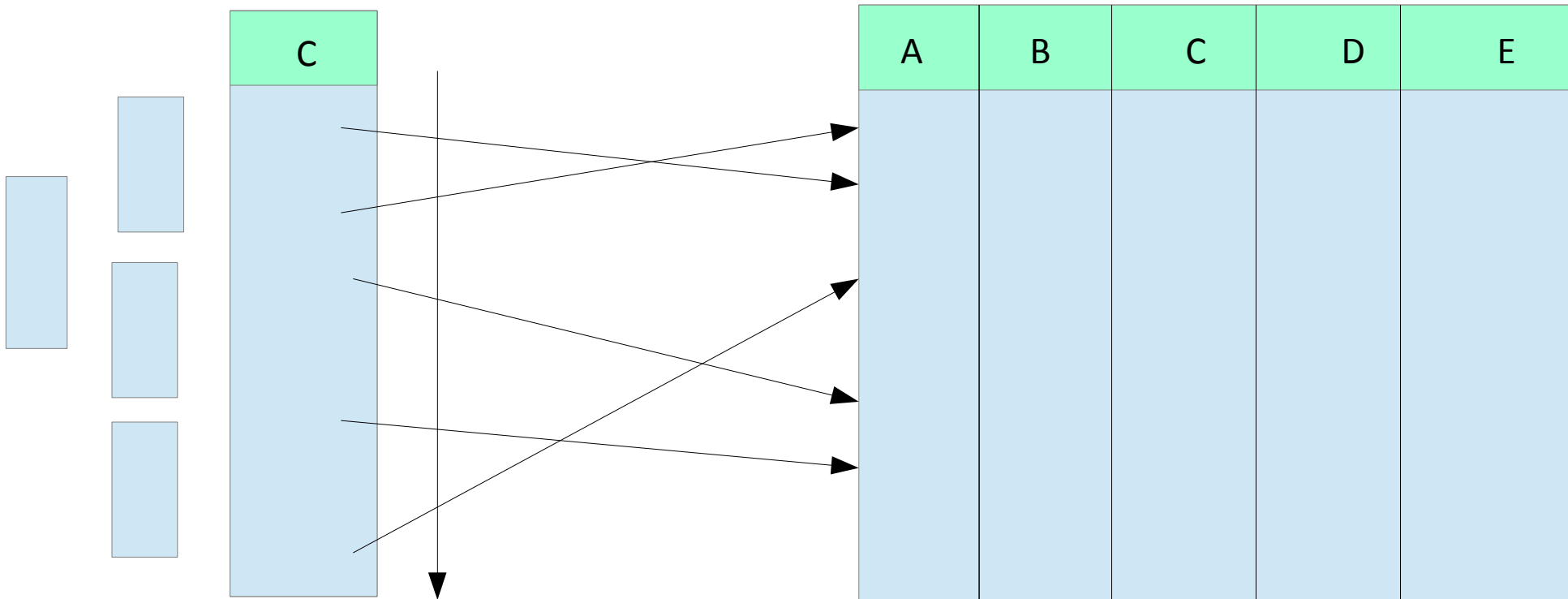
- Упорядочиваем — получаем быстрый поиск, ускоряем ORDER BY, но случайное чтение таблицы.



B-tree индекс

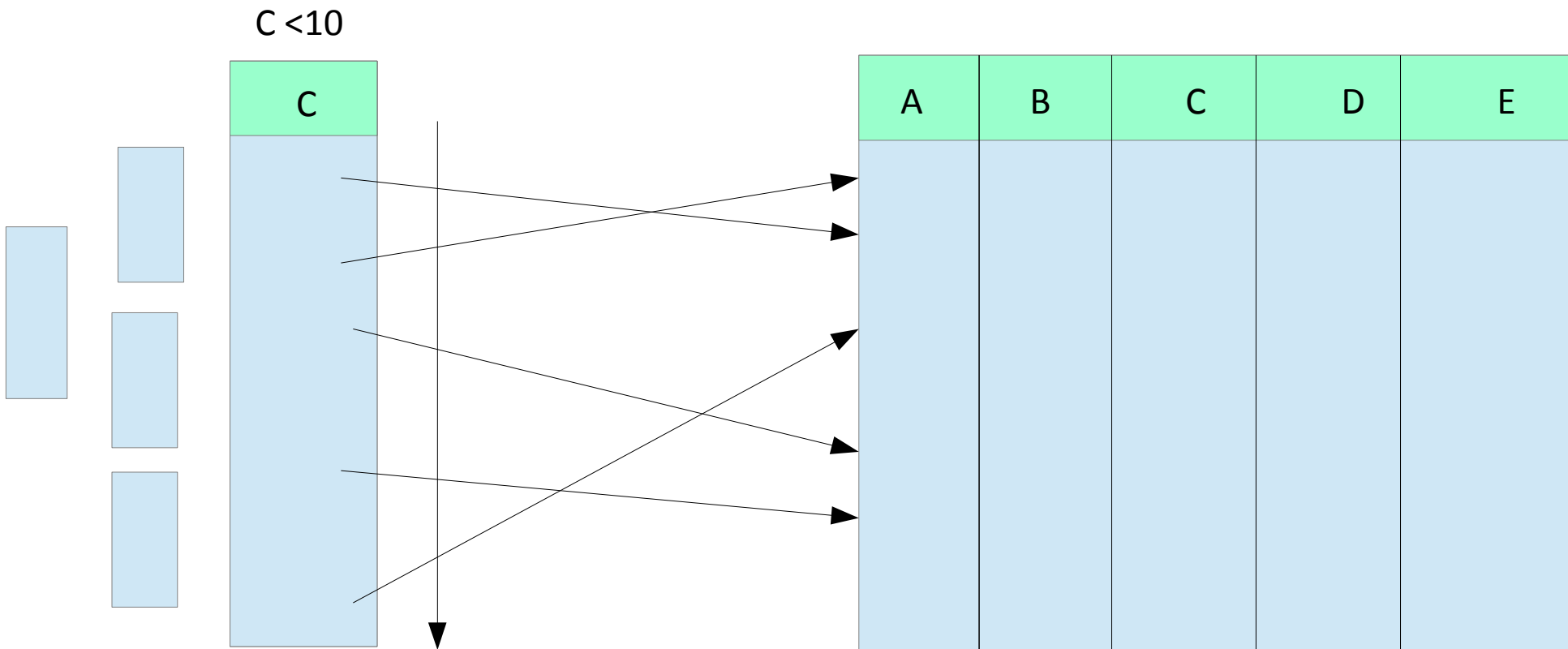
- Строим дерево — уменьшаем чтение индекса при быстром поиске

$C < 10$



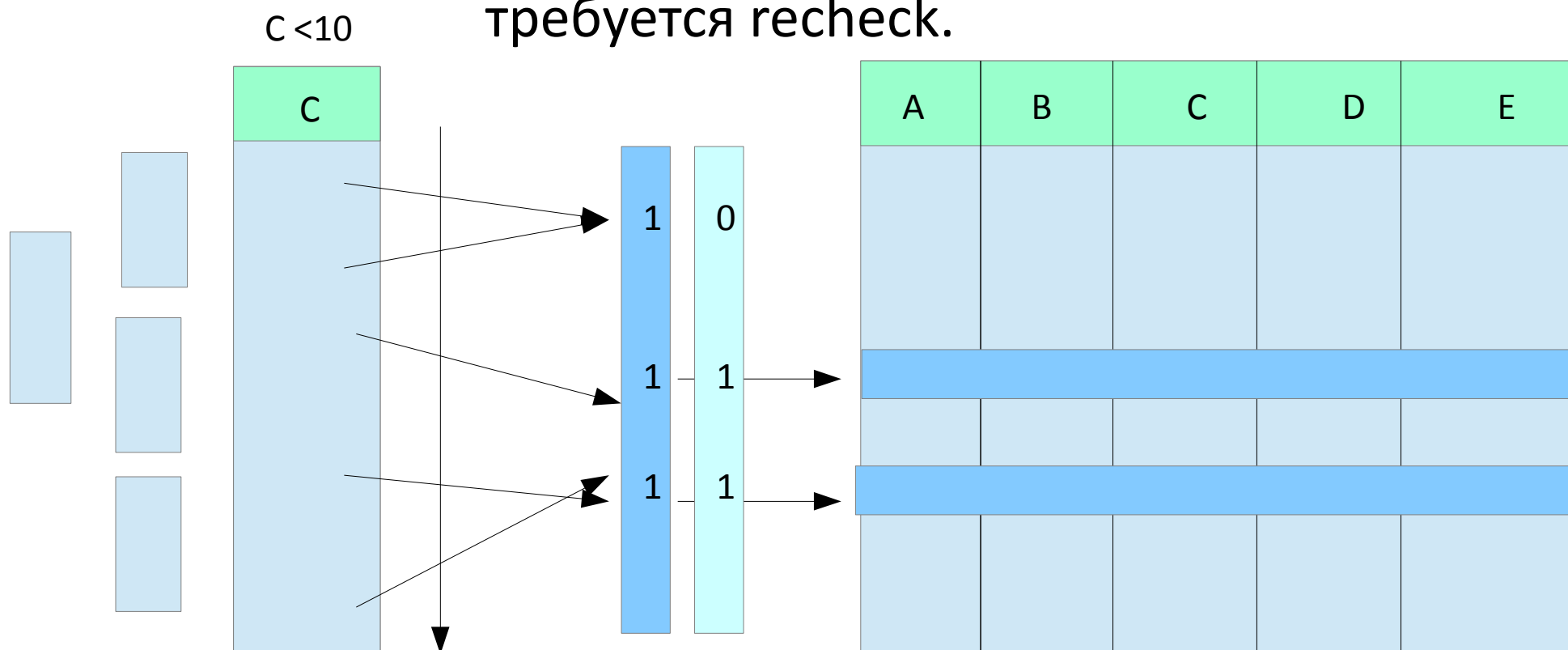
GiST,GIN,SP-GiST индексы

- Дерево — шаблон с API, поддержка произвольных типов данных



Bitmap index scan

- Результат Index scan сортируем, строим в памяти bitmap index и читаем таблицу. Можно комбинировать индексы. Иногда требуется recheck.



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
- Требуется 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»);

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 FIRST
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

```
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)
```

```
=# \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)
```

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	create
apod	gin_apod_fts_idx	0	2712 kB	1616 kB	1754	CREATE INDEX gin_apod_fts_idx ON
events	e_date_id	4	14 MB	3352 kB	151643	CREATE INDEX e_date_id ON even
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)

Выбор способа сканирования таблицы (нет статистики)

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

```
SELECT * FROM test1 WHERE v BETWEEN 0.1 AND 0.9      Нет статистики!  
Bitmap Heap Scan on test1  (cost=107.61..5701.58 rows=5000 width=12)  
(actual time=63.206..205.054 rows=800235 loops=1)  
  Recheck Cond: ((v >= 0.1::double precision) AND (v <= 0.9::double precision))  
  Rows Removed by Index Recheck: 152799  
  Buffers: shared hit=7596  
-> Bitmap Index Scan on test1_v_idx  (cost=0.00..106.36 rows=5000 width=0) (actual time=63.206..205.054 rows=800235 loops=1)  
    Index Cond: ((v >= 0.1::double precision) AND (v <= 0.9::double precision))  
    Buffers: shared hit=2190  
Total runtime: 239.768 ms
```

Выбор способа сканирования таблицы (нет статистики)

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.500001;      Нет статистики!
```

```
Bitmap Heap Scan on test1 (cost=107.61..5701.58 rows=5000 width=12)
```

```
(actual time=0.014..0.014 rows=1 loops=1)
```

```
Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.500001::double precision))
```

```
Buffers: shared hit=4
```

```
-> Bitmap Index Scan on test1_v_idx (cost=0.00..106.36 rows=5000 width=0) (actual time=0.011
```

```
Index Cond: ((v >= 0.5::double precision) AND (v <= 0.500001::double precision))
```

```
Buffers: shared hit=3
```

```
Total runtime: 0.035 ms
```

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;      Нет статистики!
```

```
Bitmap Heap Scan on test1 (cost=107.61..5701.58 rows=5000 width=12)
```

```
(actual time=12.368..30.688 rows=99951 loops=1)
```

```
Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
```

```
Buffers: shared hit=5681 read=1
```

```
-> Bitmap Index Scan on test1_v_idx (cost=0.00..106.36 rows=5000 width=0) (actual time=11.649
```

```
Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
```

```
Buffers: shared hit=275 read=1
```

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=801140 width=12)
```

```
(actual time=0.007..159.966 rows=800235 loops=1)
```

```
Filter: ((v >= 0.1::double precision) AND (v <= 0.9::double precision))
```

```
Rows Removed by Filter: 199765
```

```
Buffers: shared hit=5406
```

```
Total runtime: 196.179 ms
```

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.500001;
```

```
Index Scan using test1_v_idx on test1 (cost=0.00..8.38 rows=1 width=12)
```

```
(actual time=0.018..0.019 rows=1 loops=1)
```

```
Index Cond: ((v >= 0.5::double precision) AND (v <= 0.500001::double precision))
```

```
Buffers: shared hit=4
```

```
Total runtime: 0.037 ms
```

Влияние random_page_cost

```
SET random_page_cost = 1;
```

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
```

```
Index Scan using test1_v_idx on test1 (cost=0.00..7666.22 rows=99349 width=12)
```

```
(actual time=0.035..63.101 rows=99951 loops=1)
```

```
Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
```

```
Buffers: shared hit=100200
```

```
Total runtime: 67.788 ms
```


Влияние work_mem

```
SET work_mem = '128 kB';
```

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
```

```
Bitmap Heap Scan on test1 (cost=2110.69..9006.92 rows=99349 width=12) (actual time=11.010..11.011)
```

```
Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
```

```
Rows Removed by Index Recheck: 688322
```

```
Buffers: shared hit=5682
```

```
-> Bitmap Index Scan on test1_v_idx (cost=0.00..2085.85 rows=99349 width=0) (actual time=11.010..11.011)
```

```
Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
```

```
Buffers: shared hit=276
```

```
Total runtime: 114.684 ms
```

maintenance_work_mem

```
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
```

Чем больше памяти, тем быстрее создастся индекс
(если она действительно есть)

CLUSTER

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
Bitmap Heap Scan on test1 (cost=2187.43..9137.44 rows=102934 width=12) (actual time=9.386..28.114)
  Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
  Buffers: shared hit=5682
  -> Bitmap Index Scan on test1_v_idx (cost=0.00..2161.70 rows=102934 width=0) (actual time=0.000..0.000)
    Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
    Buffers: shared hit=276
Total runtime: 33.688 ms
```

```
CLUSTER test1 USING test1_v_idx;
```

```
SELECT * FROM test1 WHERE v BETWEEN 0.5 AND 0.6;
Bitmap Heap Scan on test1 (cost=2187.43..9137.44 rows=102934 width=12) (actual time=7.014..15.114)
  Recheck Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
  Buffers: shared hit=816
  -> Bitmap Index Scan on test1_v_idx (cost=0.00..2161.70 rows=102934 width=0) (actual time=0.000..0.000)
    Index Cond: ((v >= 0.5::double precision) AND (v <= 0.6::double precision))
    Buffers: shared hit=276
Total runtime: 20.461 ms
```

Tablespace

Выделение индексов в отдельный tablespace позволяет оптимизировать нагрузку на 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!
```

```
BEGIN;
```

```
CREATE INDEX CONCURRENTLY test1_v1_idx2 ON test1(v1) TABLESPACE idx_tablespace;
```

```
DROP INDEX test1_v1_idx; - короткая эксклюзивная блокировка!
```

```
ALTER INDEX test1_v1_idx2 RENAME TO test1_v1_idx;
```

```
COMMIT;
```

Unlogged tables

Индексы по unlogged tables тоже unlogged

```
(wal_level = hot_standby)
```

```
# CREATE TABLE test AS (SELECT id, random() AS val FROM generate_series(1,5000000) id);
```

```
# CREATE INDEX test_idx ON test(val);
```

```
CREATE INDEX
```

```
Time: 9108,796 ms
```

```
# CREATE UNLOGGED TABLE test AS (SELECT id, random() AS val FROM  
generate_series(1,5000000) id);
```

```
# CREATE INDEX test_idx ON test(val);
```

```
CREATE INDEX
```

```
Time: 8592,537 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=0.010..86.150)
```

```
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)
```

```
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..86.150)
```

```
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) (actual
```

```
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) (actual
```

```
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));
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) (actual time=0.012..0.097 rows=20 loops=1)
```

```
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) (actual time=0.025..0.088 rows=20 loops=1)
```

```
Index Cond: (ROW(v1, v2) > ROW(9::double precision, 0.5::double precision))
```

```
Total runtime: 0.111 ms
```

Составной индекс, ORDER BY и LIMIT

```
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..0.038 rows=1000000 loops=1)  
Total runtime: 161.645 ms
```

Index only scan

```
ANALYZE test4;
```

```
SELECT v1, v2 FROM test4 WHERE v1 BETWEEN 0.5 AND 0.51;
```

```
Bitmap Heap Scan on test4 (cost=18.15..1709.51 rows=561 width=16) (actual time=0.109..0.455 rows=561)
```

```
Recheck Cond: ((v1 >= 0.5::double precision) AND (v1 <= 0.51::double precision))
```

```
-> Bitmap Index Scan on test4_v1_v2_idx (cost=0.00..18.01 rows=561 width=0) (actual time=0.000..0.000 rows=561)
```

```
Index Cond: ((v1 >= 0.5::double precision) AND (v1 <= 0.51::double precision))
```

```
Total runtime: 0.488 ms
```

```
SET enable_bitmapscan = OFF;
```

```
SET enable_indexscan = OFF;
```

```
SET enable_seqscan = OFF;
```

```
SELECT v1, v2 FROM test4 WHERE v1 BETWEEN 0.5 AND 0.51;
```

```
Index Only Scan using test4_v1_v2_idx on test4 (cost=10000000000.00..10000002175.62 rows=561 width=16)
```

```
Index Cond: ((v1 >= 0.5::double precision) AND (v1 <= 0.51::double precision))
```

```
Heap Fetches: 482
```

```
Total runtime: 0.593 ms
```

Index only scan

```
VACUUM test4;
```

```
SELECT v1, v2 FROM test4 WHERE v1 BETWEEN 0.5 AND 0.51;
```

```
Index Only Scan using test4_v1_v2_idx on test4 (cost=10000000000.00..10000000023.62 rows=561 w
```

```
Index Cond: ((v1 >= 0.5::double precision) AND (v1 <= 0.51::double precision))
```

```
Heap Fetches: 0
```

```
Total runtime: 0.173 ms
```

Когда индекс используется неправильно

```
CREATE TABLE test2 AS (SELECT id, polygon(20,circle(point(random(), random()),0.01)) AS p
FROM generate_series(1,1000000) 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=
-> Seq Scan on test2 t1 (cost=0.00..13000.00 rows=50000 width=36) (actual time=0.027..54.015
-> Bitmap Heap Scan on test2 t2 (cost=196.89..313.72 rows=83 width=36) (actual time=1.845..1
    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
```

Почему так?

oprname	oprrest	pg_operator oprjoin	oprleft	oprright
&&	arease1	areajoinse1	604	604 (polygon)

src/backend/utils/adt/geo_selffuncs.c

```
Datum  
arease1(PG_FUNCTION_ARGS)  
{  
    PG_RETURN_FLOAT8(0.005);  
}
```

```
Datum  
areajoinse1(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=1
```

```
    -> Seq Scan on test2 t1  (cost=0.00..13000.00 rows=50000 width=36) (actual time=0.005..20.801
```

```
    -> 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: 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.ru/~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;
```

```
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 (cost=135.70..6911.42 rows=7140 width=20) (actual time=1.707..21.039
```

```
Recheck Cond: ((v1 + v2) > 1.9::double precision)
```

```
-> Bitmap Index Scan on test7_v1_plus_v2_idx (cost=0.00..133.91 rows=7140 width=0) (actual t
```

```
Index Cond: ((v1 + v2) > 1.9::double precision)
```

```
Total runtime: 21.410 ms
```

Head Only Tuple (HOT)

```
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);
```

```
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
```

Вставка в индекс = delete, insert (MVCC)

Обновление таблицы приводит к обновлению **всех** индексов.

HOT (8.3+) позволяет не обновлять индексы по полям, которые не менялись.

```
=# 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

```
CREATE TABLE test9 AS (  
SELECT id, point(random(), random()) AS p FROM generate_series(1,1000000) id);
```

```
SELECT * FROM test9 ORDER BY p <-> point(0.5, 0.5) LIMIT 10;  
Limit (cost=41786.38..41786.41 rows=10 width=20) (actual time=331.506..331.509 rows=10 loops=1)  
-> Sort (cost=41786.38..44382.16 rows=1038310 width=20) (actual time=331.500..331.501 rows=10 loops=1)  
    Sort Key: ((p <-> '(0.5,0.5)'::point))  
    Sort Method: top-N heapsort Memory: 25kB  
    -> Seq Scan on test9 (cost=0.00..19348.88 rows=1038310 width=20) (actual time=0.019..2.019 rows=10 loops=1)  
Total runtime: 331.528 ms
```

```
CREATE INDEX test9_p_idx ON test9 USING gist (p);
```

```
SELECT * FROM test9 ORDER BY p <-> point(0.5, 0.5) LIMIT 10;  
Limit (cost=0.00..0.82 rows=10 width=20) (actual time=0.089..19.541 rows=10 loops=1)  
-> Index Scan using test9_p_idx on test9 (cost=0.00..81892.61 rows=1000000 width=20) (actual time=0.089..19.541 rows=10 loops=1)  
    Order By: (p <-> '(0.5,0.5)'::point)  
Total runtime: 19.575 ms
```

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)
```



Спасибо за Внимание !