Новые возможности Btree

Лубенникова А.В.
Фролков И.А.

www.postgrespro.ru

Общий принцип

- **Чем меньше данных — тем лучше**
  - Вырожденный случай — одна строка в одной таблице с одной колонкой
  - Так не бывает :-)
- Почему?
  - Меньше ввод-вывод
  - Эффективней используется кеш

Кто виноват

- Таблицы
  - От 23 байт на заголовок строки
  - Если колонка nullable — один бит в еще один байт
  - Выравнивание!
- Индексы
  - 8 байт на строку
  - Да, выравнивание

Что делать?

- Таблицы
  - Секционировать
  - Сжатие
- Сейчас сжатия таблиц нет, но зато есть паллиатив — toast
  - Сжимаются значения переменной длины
    - Строки
    - Bytea
    - Массивы!

Исходная таблица

```
create table usr_action_raw as
 select n%5000 as usr_id,
   (array['READ','READ',
          'SAVE','DELETE'])[n%3+1]
    as action,
   '2016-01-01'::timestamptz
    +make_interval(secs:=n)
    as added
from
 generate_series(1,10000000) as gs(n)
```

Исходная таблица

```
create index uar_usr_id on
 usr_action_raw(usr_id)

create index uar_usr_added on
usr_action_raw(added);

create index uar_usr_action on
 usr_action_raw(action)
```

Исходная таблица

- Размеры
  - Таблица — 498 M
  - Индексы — три по 214M
- Итого - 1.1GB

Массивы типов

- ```
  create type usr_action as(
    action text,
    added timestamptz,
    ….
  );
  ```
- ```
  create table usr_log(
    usr_id int not null references usr(id),
    min_added timestamptz not null,
    max_added timestamptz not null,
    actions usr_action[]
  )
  ```

Индексы

```
create index ul_usr_id on
usr_log(usr_id);
create index ul_usr_added on usr_log
using gist(tstzrange(min_added,
max_added));
```

Что получилось

- Таблица — 376 КБ
  - Toast — 79MB
  - Индексы — 305 КБ
- Итого — 80 МБ

Postgres PROFESSIONAL

А что с индексами?

- Во-первых, индекс по usr_id стал маленьким
- Во-вторых, индекс по tstzrange(min_added, max_added) тоже небольшой
- В-третьих, можно построить индекс не по всем значениям action

```
create or replace function
   get_action_values(actions anyarray)
   returns text[] as
$code$
select array_agg(distinct un.action)
     from unnest(actions) as un;
$code$
language sql
immutable
```

- **Размер — 32 КБ**

Индексы

```
create index usr_actions
   on usr using gin((get_usr_actions(actions));
```

- Запрос

```
select u.usr_id, un.*
  from usr_log u, unnest(u.actions) as un
 where u.usr_id=3000
   and 'READ'=any(get_action_values(actions));
```

```
Nested Loop  (cost=0.28..10.57 rows=100 width=44) (actual
time=2.499..3.441 rows=2000 loops=1)
  ->  Index Scan using ul_usr_id on usr_log u  (cost=0.28..8.56
rows=1 width=22) (actual time=1.944..1.946 rows=1 loops=1)
        Index Cond: (usr_id = 3000)
        Filter: ('READ'::text = ANY (get_action_values(actions)))
  ->  Function Scan on unnest un  (cost=0.00..1.00 rows=100
width=40) (actual time=0.551..0.850 rows=2000 loops=1)
Planning time: 0.286 ms
Execution time: 3.707 ms
```

Используем

```
Prepare sth(int,int,timestamptz,
timestampz) as
select u.usr_id, un.*
  from usr_log u, unnest(u.actions) as un
where u.usr_id between $1 and $2
  and tstzrange(u.min_added, u.max_added)
      &&
      tstzrange($3,$4)
  and un.added between $3 and $4;
```

- `execute sth(1000,1100,'2016-01-01 00:00:00','2016-01-01 01:20:00');`

```
Nested Loop  (actual time=3.881..90.529 rows=11216 loops=1)
  ->  Bitmap Heap Scan on usr_log u  (actual time=3.175..3.294
rows=101 loops=1)
        Recheck Cond: ((tstzrange(min_added, max_added) &&
tstzrange($3, $4)) AND (usr_id >= $1) AND (usr_id <= $2))
        Heap Blocks: exact=45
        ->  BitmapAnd  (actual time=3.152..3.152 rows=0 loops=1)
              ->  Bitmap Index Scan on ul_usr_added (actual
time=3.086..3.086 rows=5000 loops=1)
                    Index Cond: (tstzrange(min_added, max_added)
&& tstzrange($3, $4))
              ->  Bitmap Index Scan on ul_usr_id  (actual
time=0.051..0.051 rows=101 loops=1)
                    Index Cond: ((usr_id >= $1) AND (usr_id <=
$2))
  ->  Function Scan on unnest un  (cost=0.00..1.50 rows=1
width=40) (actual time=0.373..0.821 rows=111 loops=101)
        Filter: ((added >= $3) AND (added <= $4))
                 Rows Removed by Filter: 1889
Execution time: 91.377 ms
```

Что еще можно сделать?

- action вполне может быть повторяющимся
  - сделать enum. 4 байта на значение
  - сделать справочник. 2 байта на значение
  - На достаточно большом объеме особого смысла не имеет.

Предварительные выводы

- Компрессия нужна!
- Просто так ее нет
- Приходится прибегать к нестандартным возможностям на грани трюков
  - Да, это все несложно, но это предъявляет повышенные требования к DBA/разработчикам

Postgres PROFESSIONAL

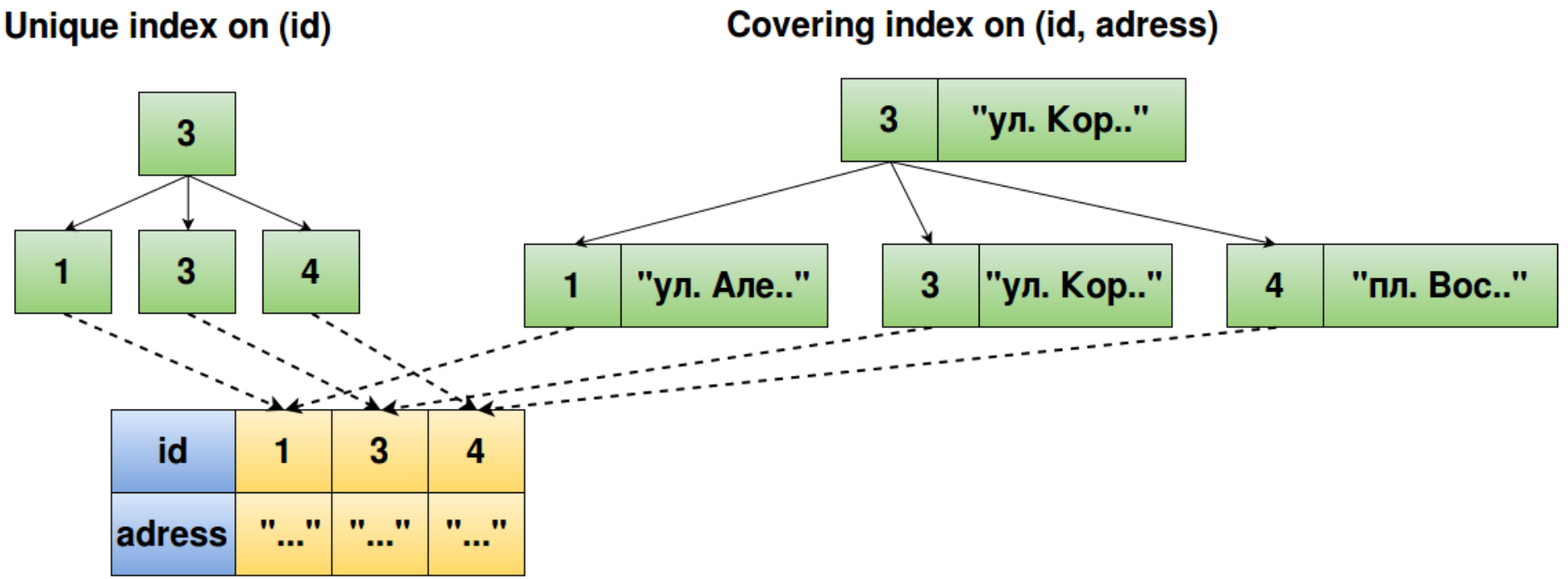
Надо проще

Покрывающие индексы

- Зайдем издалека...
- Получать данные из индекса, не обращаясь в таблицу бывает значительно быстрее
- Для этого можно создавать покрывающие индексы
- Но от индексов, поддерживающих уникальность или первичные ключи всё равно никуда не деться

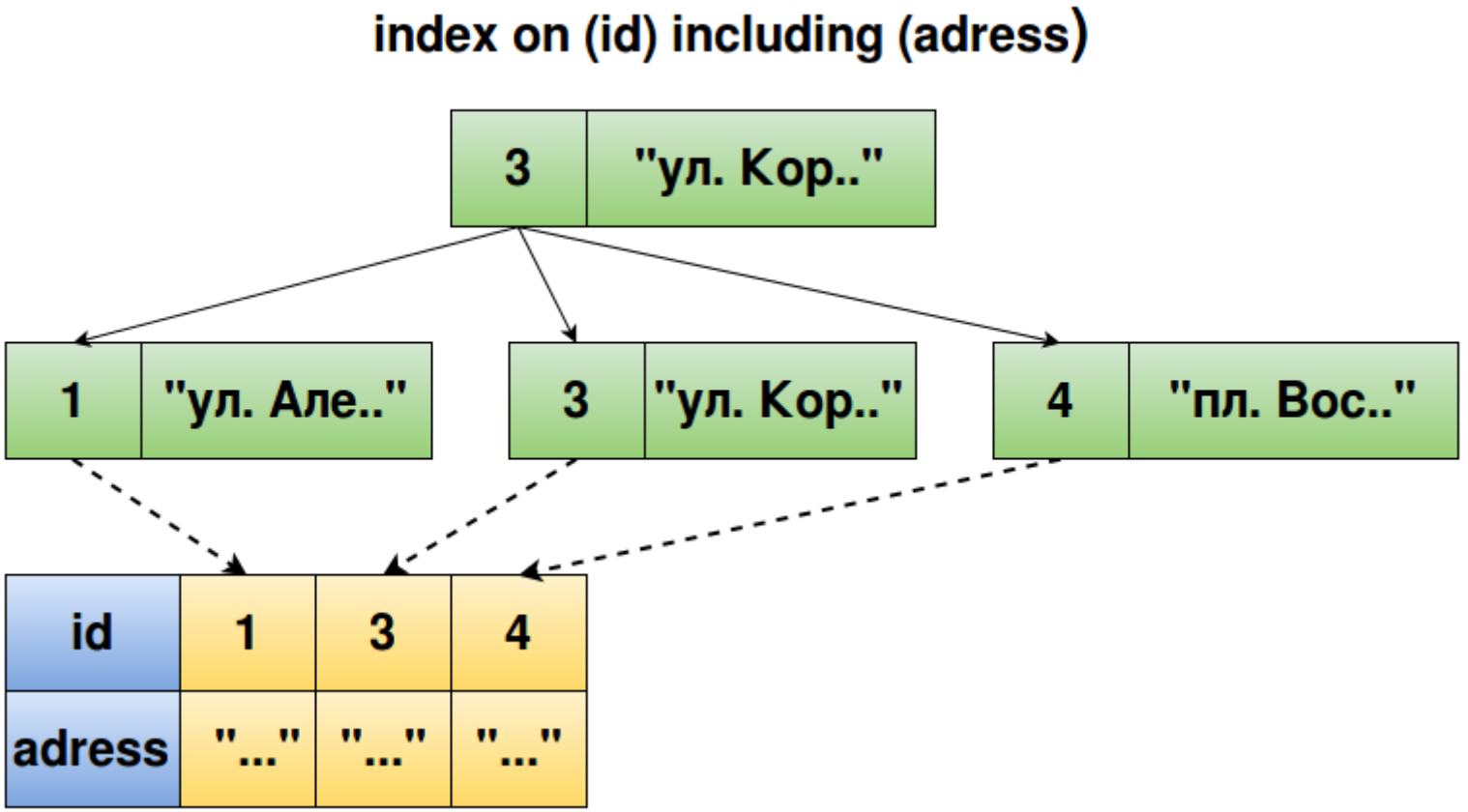
Покрывающие индексы

- В итоге имеем такую конструкцию:



Индекс с включенными столбцами

- А так она должна выглядеть:

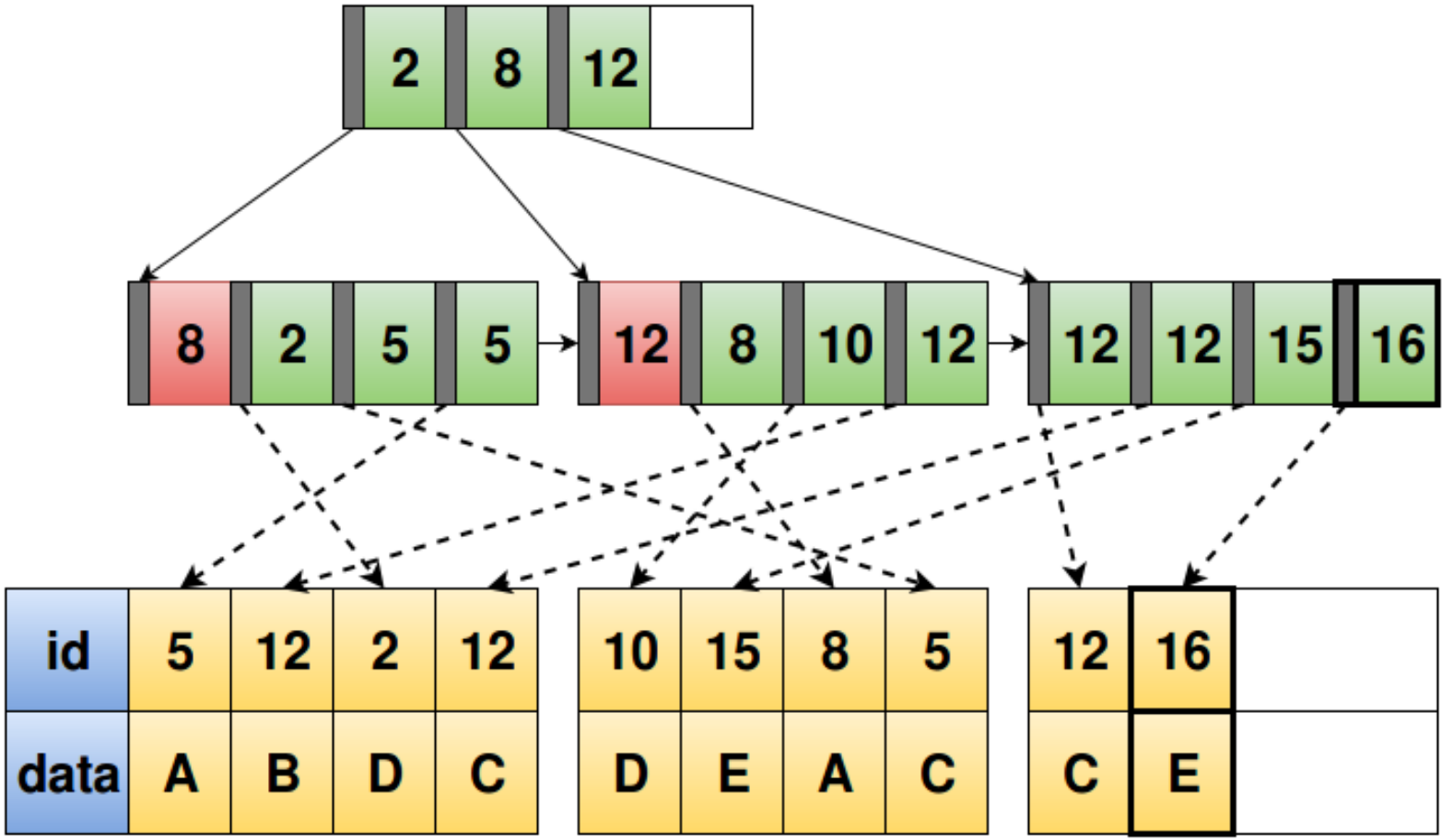


Индекс с включенными столбцами

```
CREATE UNIQUE INDEX ON tbl (a) INCLUDING (b);
CREATE INDEX ON tbl (a) INCLUDING (b);

CREATE TABLE tbl(c1 int,c2 int, c3 text, c4 box,
UNIQUE(c1,c2)INCLUDING(c3,c4));

CREATE TABLE tbl (c1 int, c2 int, c3 text, c4 box);
CREATE UNIQUE INDEX tbl_idx_unique ON tbl
using btree(c1, c2) INCLUDING(c3,c4);
ALTER TABLE tbl add UNIQUE USING INDEX
tbl_idx_unique;

CREATE TABLE tbl (c1 int, c2 int, c3 text, c4 box);
ALTER TABLE tbl add UNIQUE(c1,c2) INCLUDING(c3,c4);
```

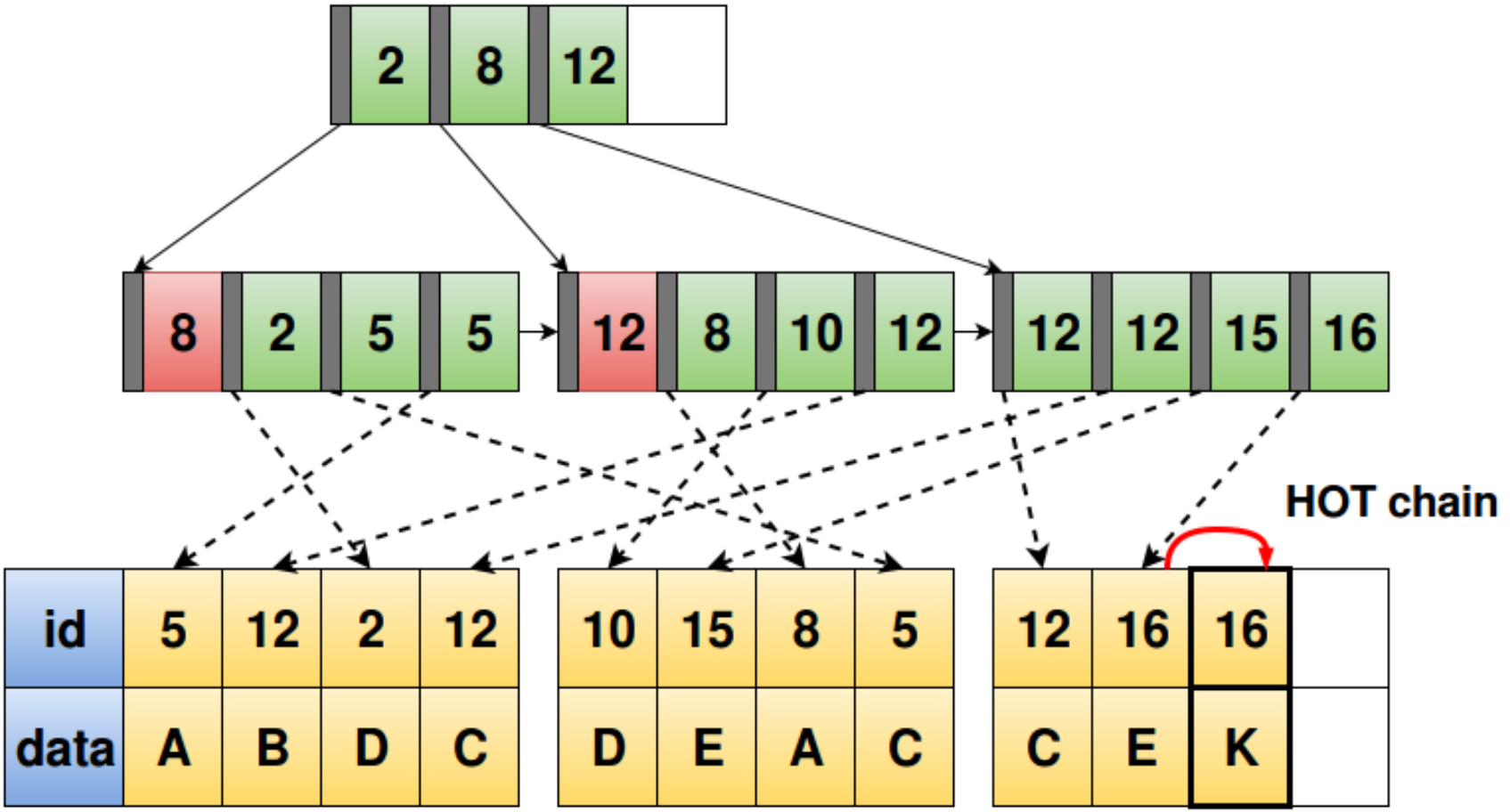
HOT-обновления

```
UPDATE tbl SET data = K WHERE id = 16;
```



HOT-обновления

```
UPDATE tbl SET data = K WHERE id = 16;
```

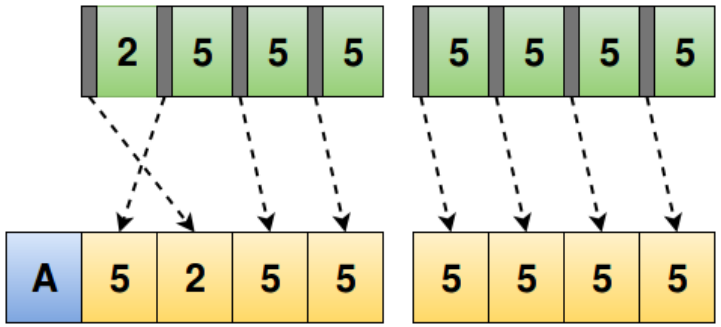


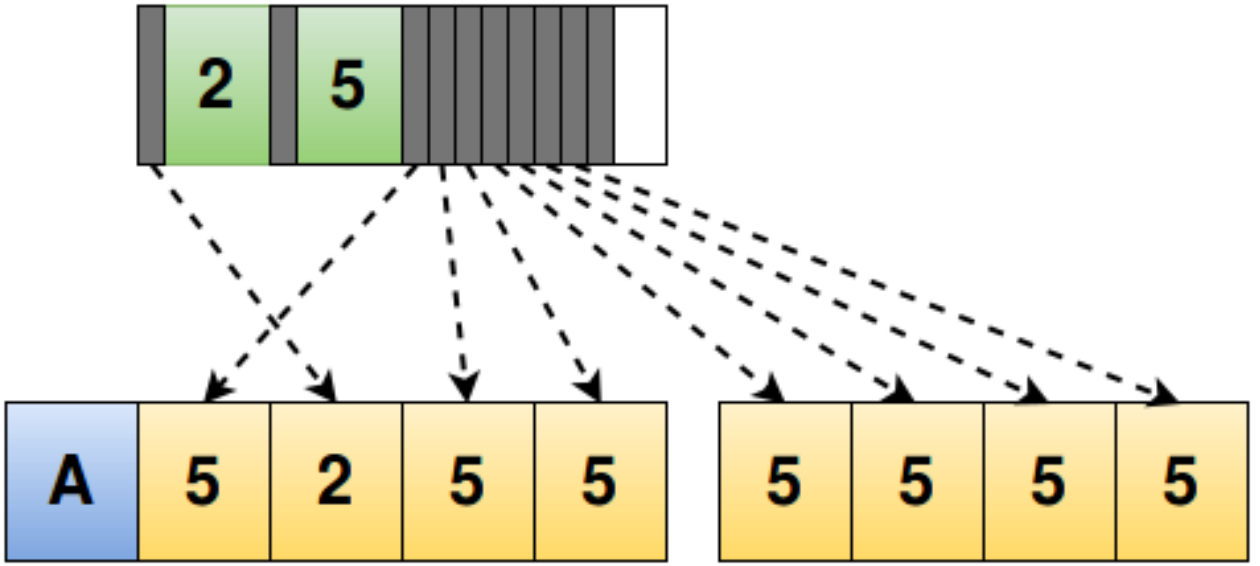
Индекс с включенными столбцами

- ✔ Меньше индексов — меньше издержек
  - ✔ Общий размер индексов меньше
  - ✔ Быстрее выполняется вставка и обновление
  - ✔ Быстрее планирование и поиск
- ✔ Для включенных столбцов не нужно задавать порядок сортировки

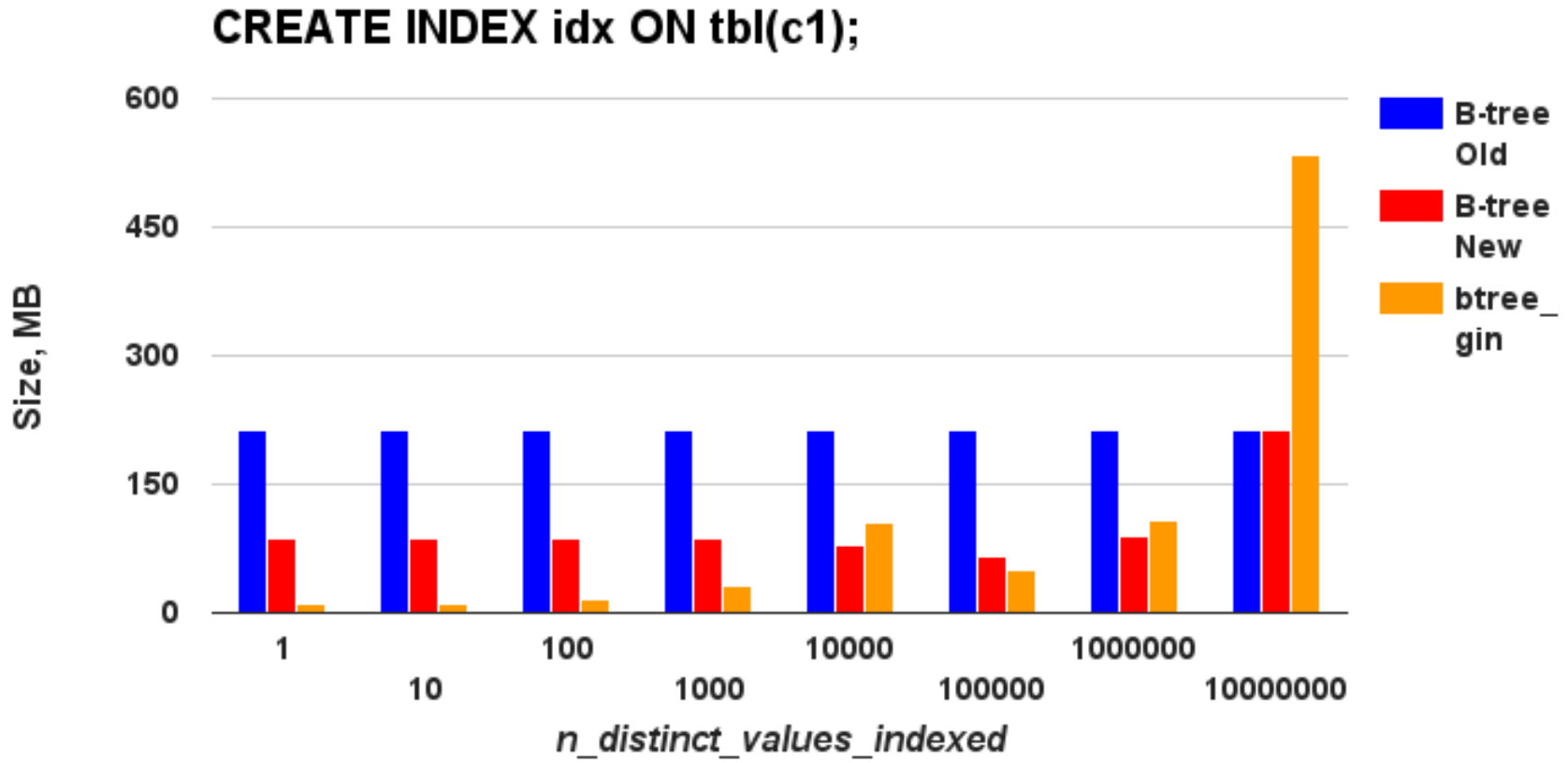
- ✗ Удаление любого столбца приводит к удалению индекса
- ✗ HOT-обновления не работают для проиндексированных столбцов

Эффективное хранение дубликатов

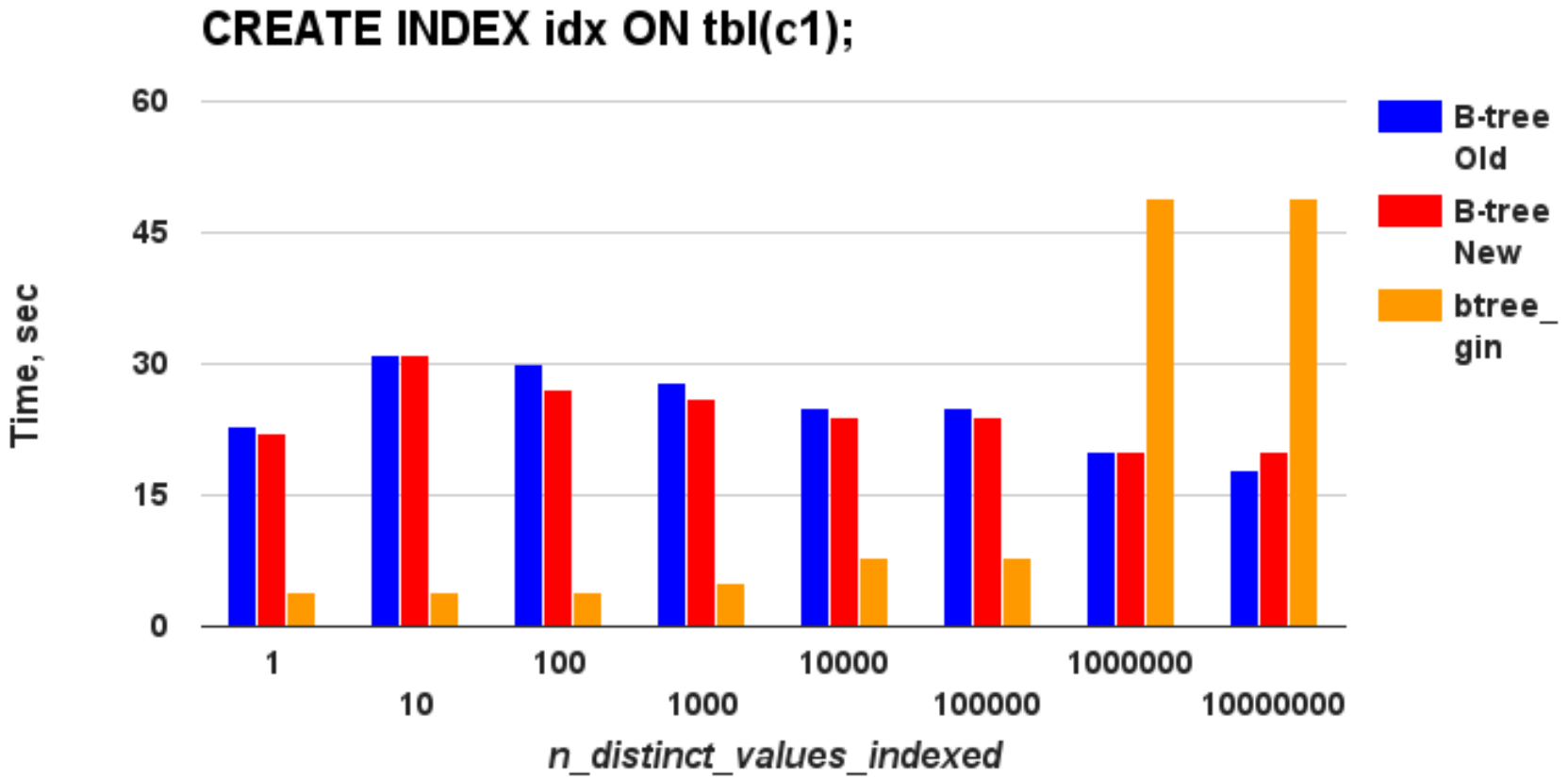




Размер индекса

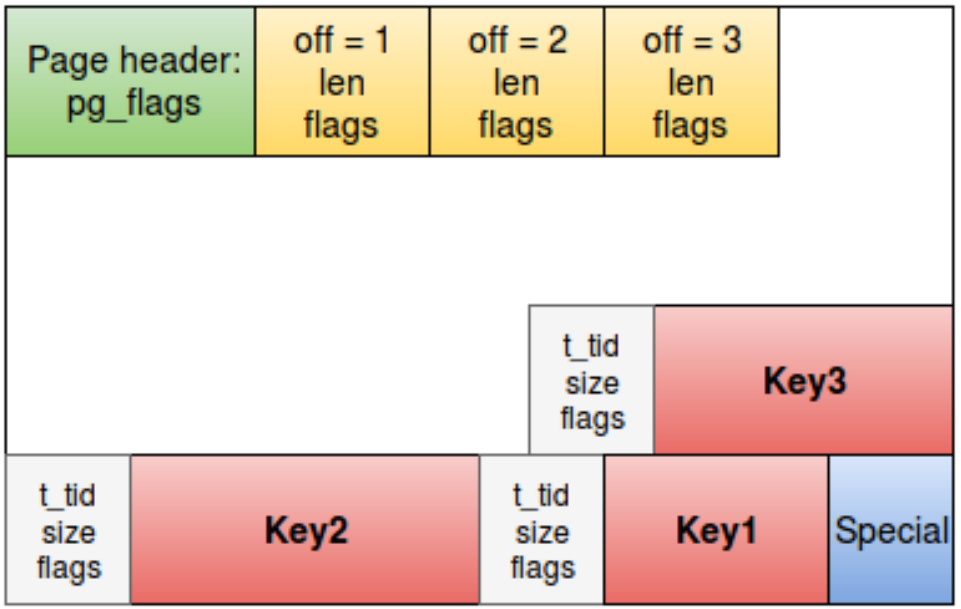


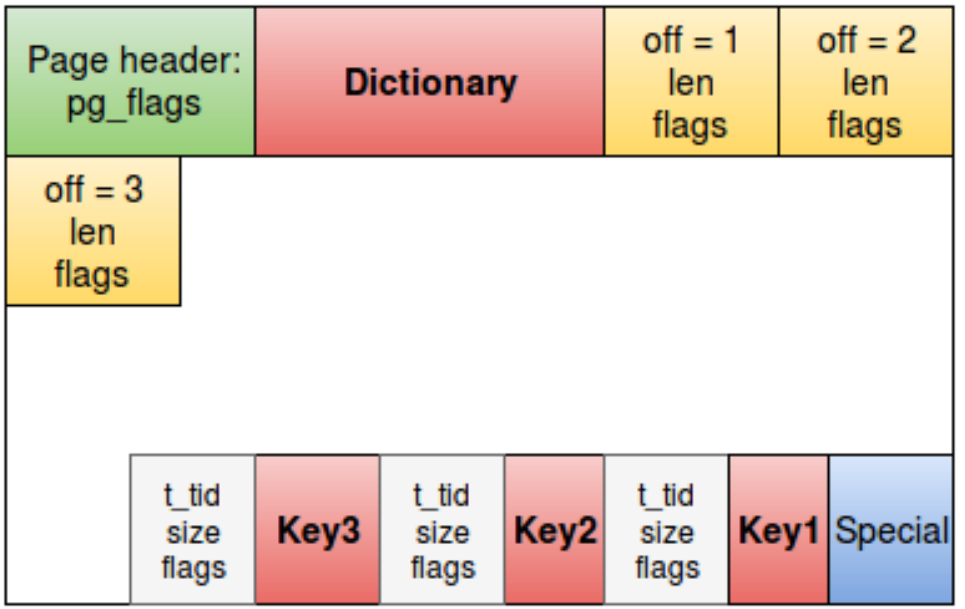
Скорость построения



Страничное сжатие

- Словарь для каждой страницы
- Каждая запись сжимается отдельно
- Заголовки записей не сжимаются





Страничное сжатие

- Не зависит от данных
- Подходит всем страницам со стандартной укладкой
- Страницы бинарно совместимы с прошлой версией
- Страницы сжаты и на диске, и в памяти
- Записи разжимаются при обращении к данным

Страничное сжатие индексов

- При создании индекса

```
CREATE INDEX ON tbl (id, data)
WITH(compression=true);
```

- При пересоздании индекса

```
CREATE INDEX CONCURRENTLY ON tbl (id, data)
WITH(compression=true);
```

- Перед разделением страниц
- При выполнении vacuum (или autovacuum)

Страничное сжатие таблиц

- При загрузке данных функцией COPY
- При выполнении VACUUM FULL и CLUSTER
- При выполнении vacuum

Что предстоит доработать?

- Использовать другой алгоритм сжатия
- Внести изменения в планировщик
- Продумать эвристики для работы со словарем

Postgres PROFESSIONAL

Светлое будущее

- Index-Organized-Tables (первичные индексы)
- KNN (поиск N ближайших соседей) для B-tree
  - Несложная задачка для начинающих
- Bulk update для индексов
- Индексы для секционированных таблиц
  - pg_pathman
  - Глобальные индексы
  - Секционированные индексы

Postgres PROFESSIONAL

Спасибо за внимание!

Вопросы? Предложения? Пожелания?

Проверка целостности индексов

- CREATE EXTENSION amcheck;
- `bt_index_check(index regclass);`
- `bt_index_parent_check(index regclass);`