B-tree индексы в PostgreSQL

Владимир Ситников

Performance engineer
PgJDBC, JMeter committer
Член программных комитетов
JPoint, Joker, Heisenbug, DevOops, SmartData

VladimirSitnikv
VladimirSitnikv
sitnikov.vladimir@gmail.com

Что будет на докладе

- Алгоритмы работы B-Tree индексов
- Применение индексов в запросах
- Применение многоколоночных индексов

Зачем же нужны индексы?

Индексы нужны для

Индексы нужны для быстрой выборки

Варианты использования индексов

- поиск по ключу: `where id = ?`
- поиск по составному ключу: `where order_id = ? and line_id = ?`
- поиск по диапазону: `where created_when > ?`
- поиск по диапазону: `where created_when beetween ? and ?`
- top N: `order by created_when`
- обработка `foreign keys`

Примеры

https://use-the-index-luke.com/

По-хорошему, нужно понять цель:

- оптимизируем чтение (DML страдает)
- оптимизируем модификацию (SELECT страдает)
- оптимизируем всё вместе (засучите рукава)

Какой должен быть индекс?

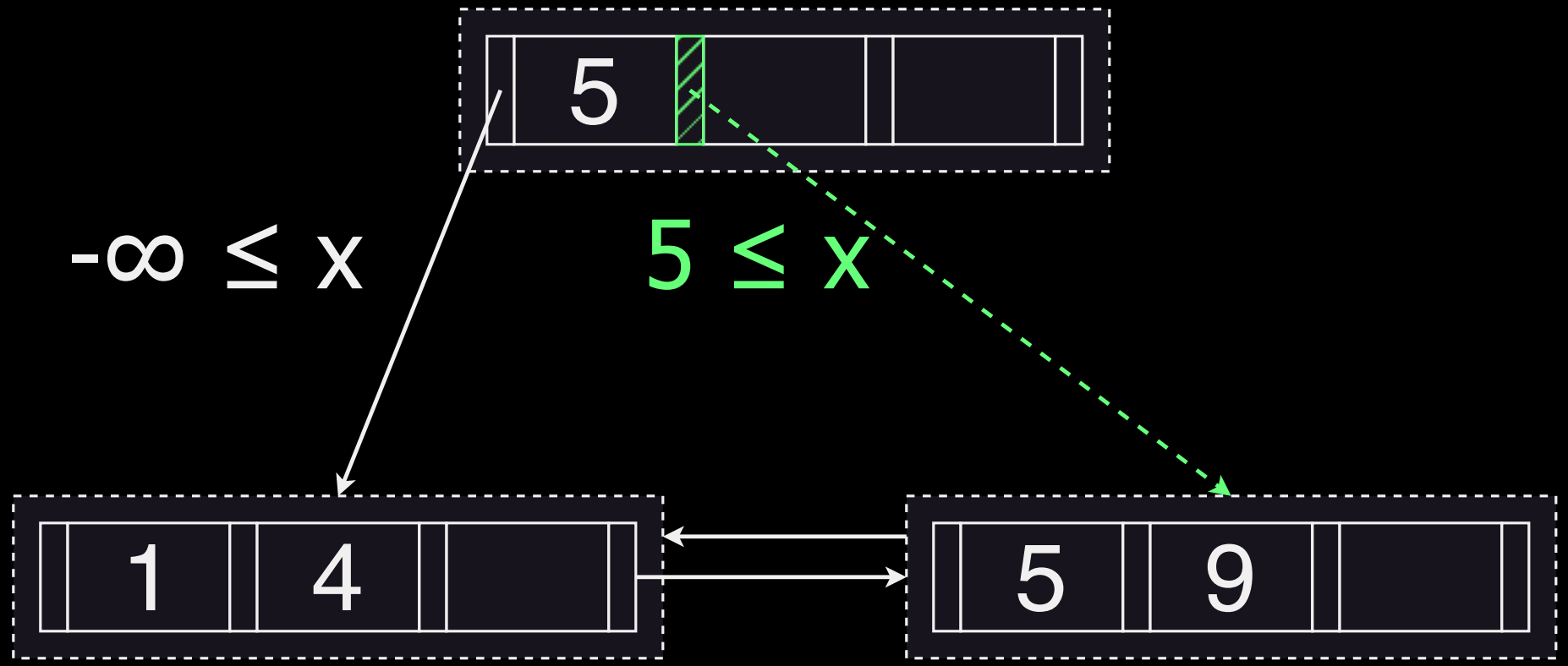
- Просто индекс. Какие вопросы?
- Всегда оптимальный

Критерии оптимальности

- Эффективная работа на больших данных
- Эффективная работа на маленьких данных
- Эффективный многопользовательский доступ

Индекс может не поместиться в ОЗУ

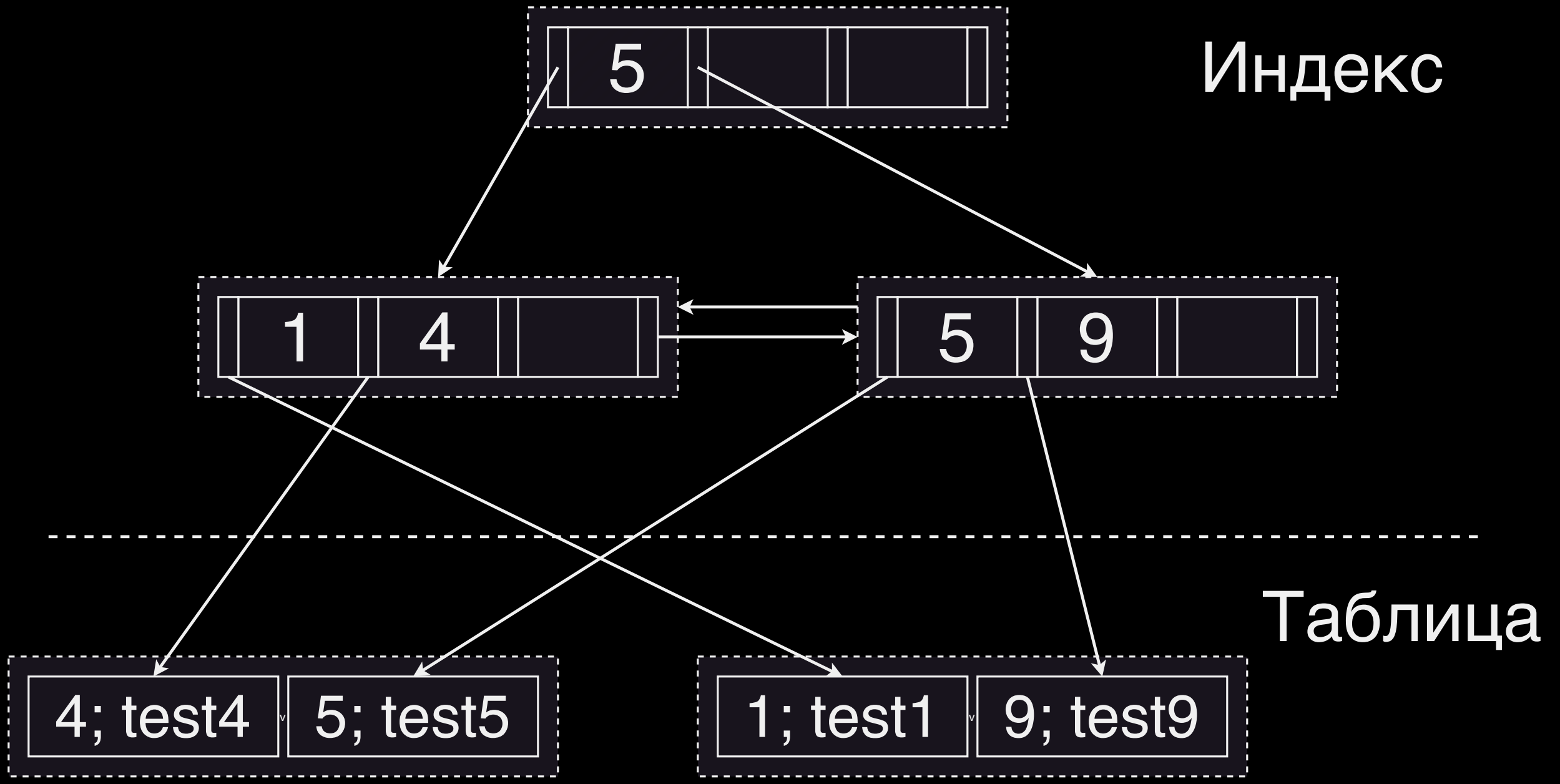
Сначала была страница


5
-∞ ≤ x
5 ≤ x
1
4
5
9

```
   from generate_series(1, 1000000) as g(x);

vacuum analyze users;


select id, name
  from users
 where id = 1;

  id   name
|  1 | test1 |
```

Откуда взялось name в результатах?

```
explain analyze
  select id, name
    from users
   where id = 1;

Index Scan using users_pkey on users  (cost=0.42..8.44 rows=1 width=18)
    (actual time=0.033..0.035 rows=1 loops=1)
  Index Cond: (id = 1)
Planning Time: 0.108 ms
Execution Time: 0.061 ms


  id   name
|  1 | test1 |
```


5
Индекс
1
4
5
9
Таблица
4; test4
5; test5
1; test1
9; test9

- Индекс отсортирован
- На каждом уровне страницы связаны в список
- Листья ссылаются на таблицу
- В PostgreSQL таблица не сортирована

Сколько длится чтение из памяти?

Каждый программист должен знать эти числа

https://gist.github.com/hellerbarde/2843375

```
Main memory reference ..... 100 ns
SSD random read ........... 150,000 ns  = 150 µs
Read 1 MB sequentially from SSD ..... 1,000,000 ns  =   1 ms
Disk seek ................. 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
```

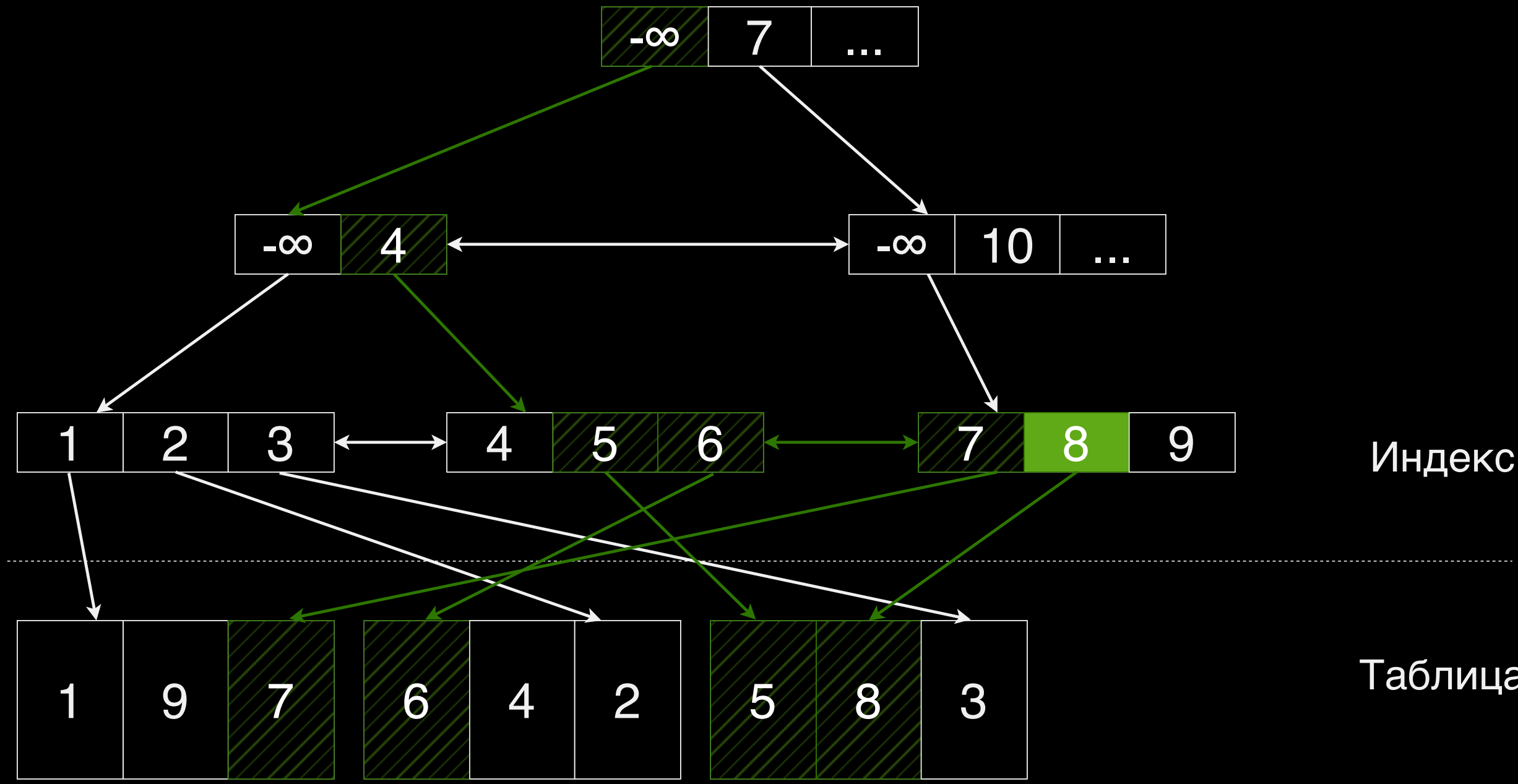
- Диск намного медленнее памяти
- Скорость выборки определяется количеством случайных чтений

```
explain
  (analyze, costs off, buffers)
  select id, name
    from users
   where name = 'test1';

Seq Scan on users (actual time=0.016..63.787 rows=1 loops=1)
  Filter: ((name)::text = 'test1'::text)
  Rows Removed by Filter: 999999
  Buffers: shared hit=6369
Execution Time: 60.402 ms

Index Scan using name__users on users (actual time=0.047..0.048 rows=1 loops=1)
  Index Cond: ((name)::text = 'test1'::text)
  Buffers: shared hit=1 read=3
Execution Time: 0.065 ms
```

- Типичный размер страницы: 8 килобайт
- На страницу помещается ~100 записей
- Высота индекса это $4.5 = \mathrm{Log}_{100}(1'000'000'000)$

Поиск по диапазону

```
explain (analyze, costs off, buffers)
  select id, name
    from users
    where id ⩾ 5 and id ⩽ 8;

Index Scan using users_pkey on users (actual time=0.036..0.041 rows=2 loops=1)
  Index Cond: ((id ⩾ 5) AND (id ⩽ 8))
  Buffers: shared hit=4
Execution Time: 0.080 ms
```

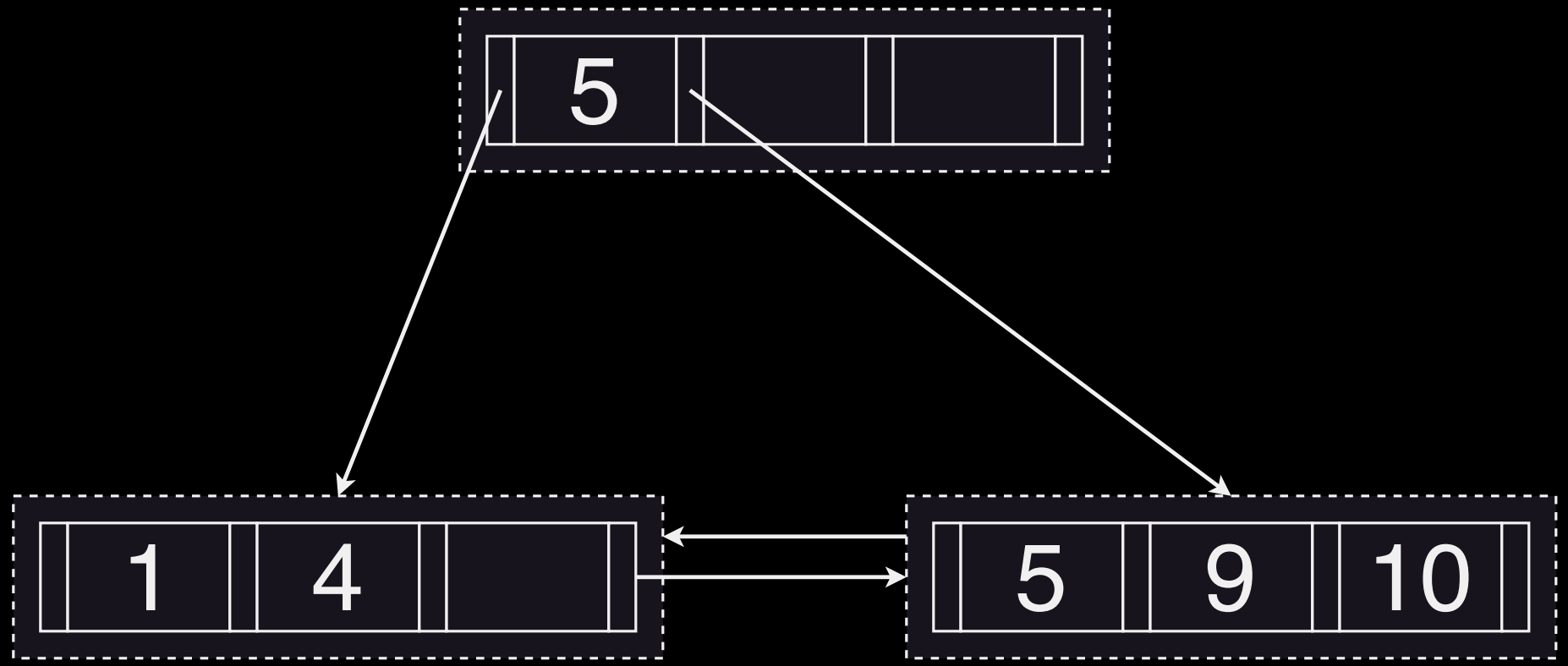

-∞
7
...
-∞
4
-∞
10
...
1
2
3
4
5
6
7
8
9
Индекс
1
9
7
6
4
2
5
8
3
Таблица

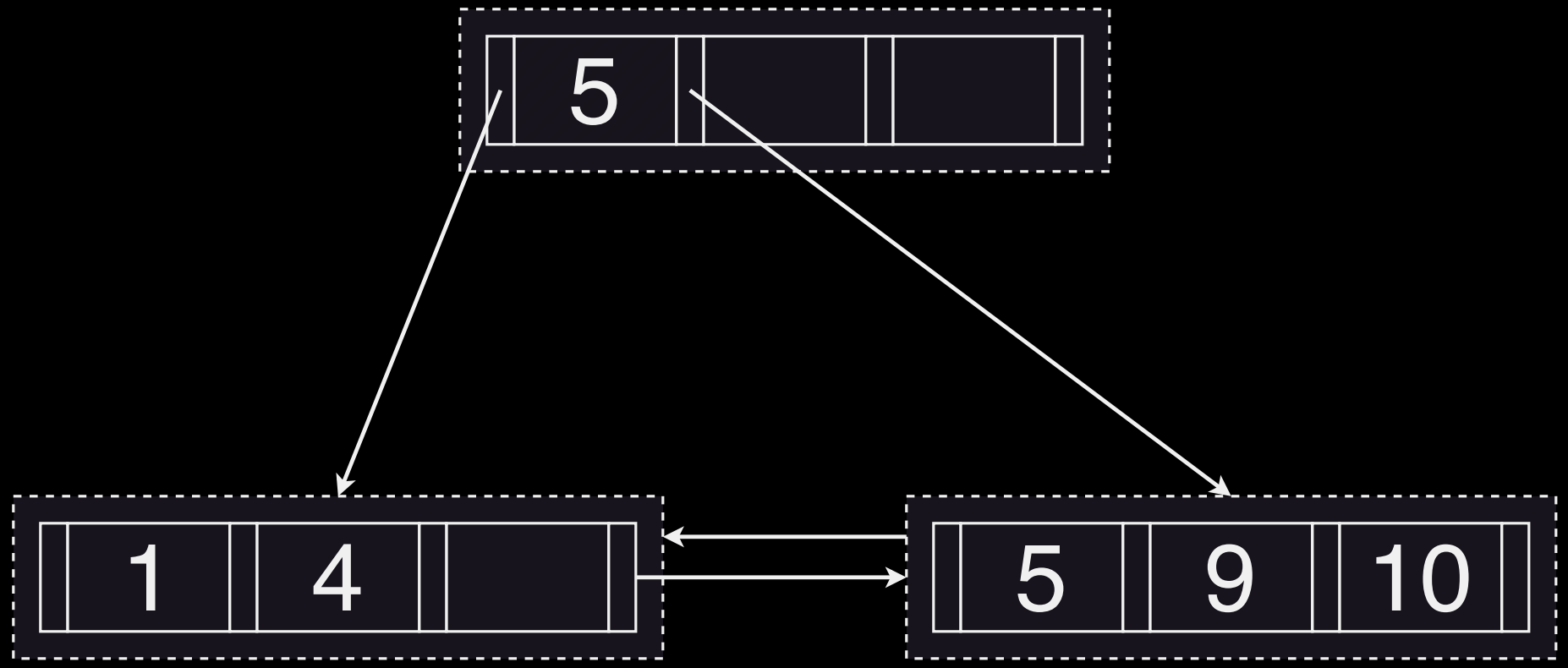
- База может сканировать диапазоны по возрастанию и убыванию
- "Заходим" в индекс лишь один раз, далее следуем по списку
- За данными по-прежнему ходим в таблицу

индекс не хранит информацию о видимости строк

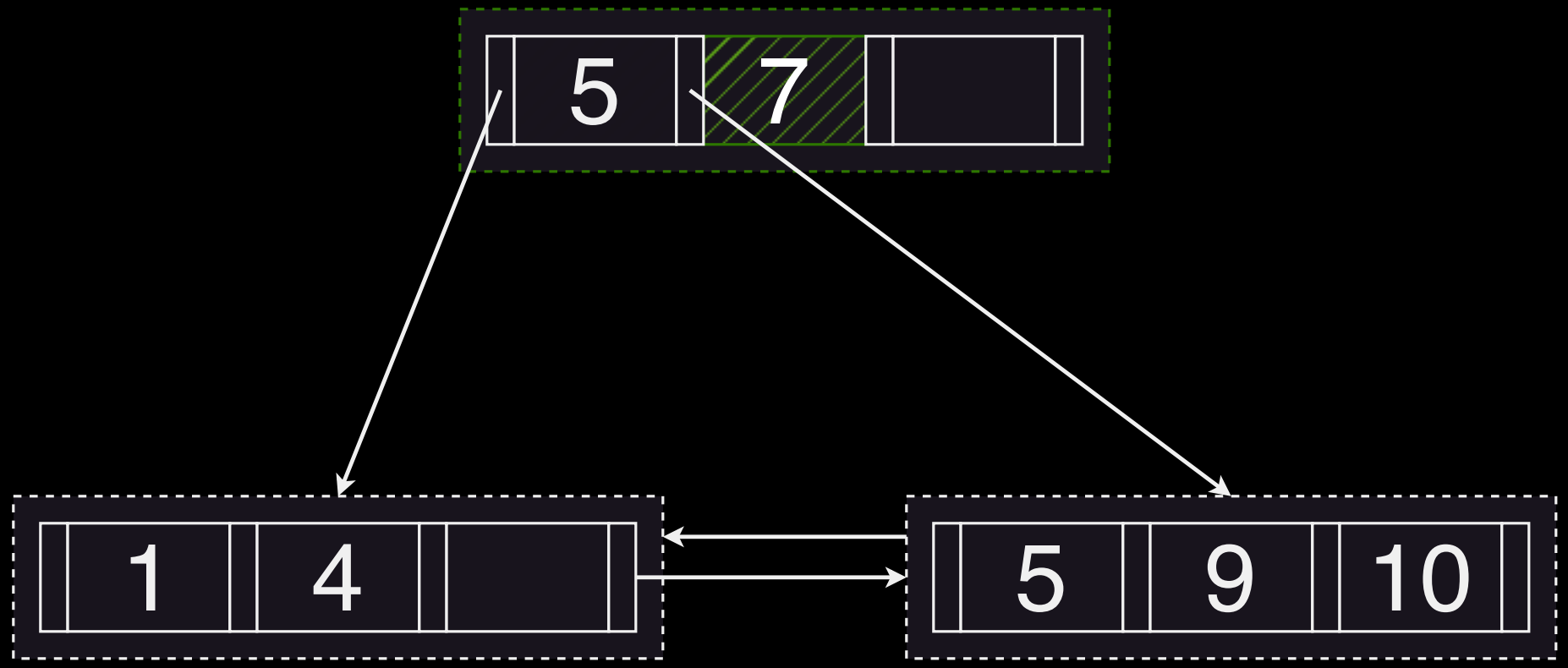
```
explain (analyze, costs off, buffers)
  select id
    from users
    where id ⩾ 5 and id ⩽ 8;

Index Only Scan using users_pkey on users (actual time=0.015..0.018 rows=4 loops=1)
  Index Cond: ((id ⩾ 5) AND (id ⩽ 8))
  Heap Fetches: 4
  Buffers: shared hit=4
Execution Time: 0.044 ms
```

Крутим дерево

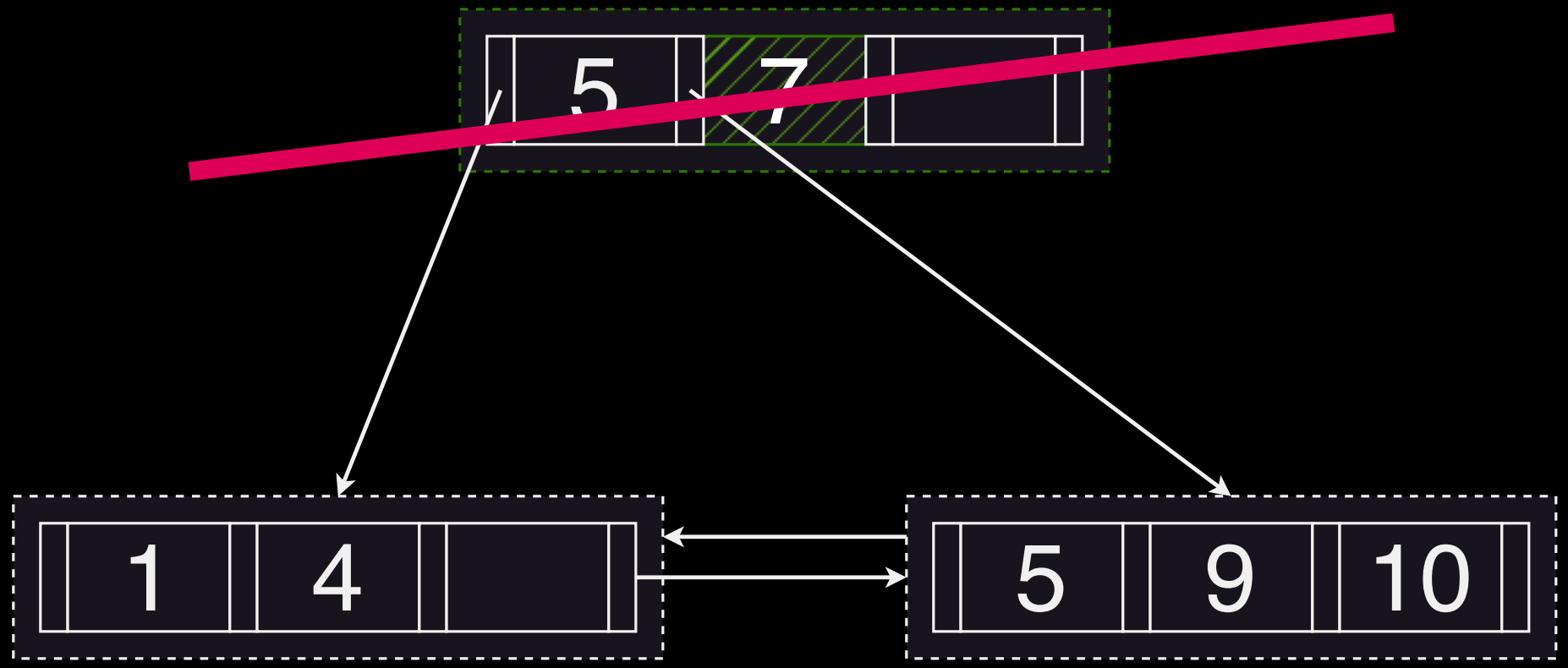

5
1
4
5
9
10



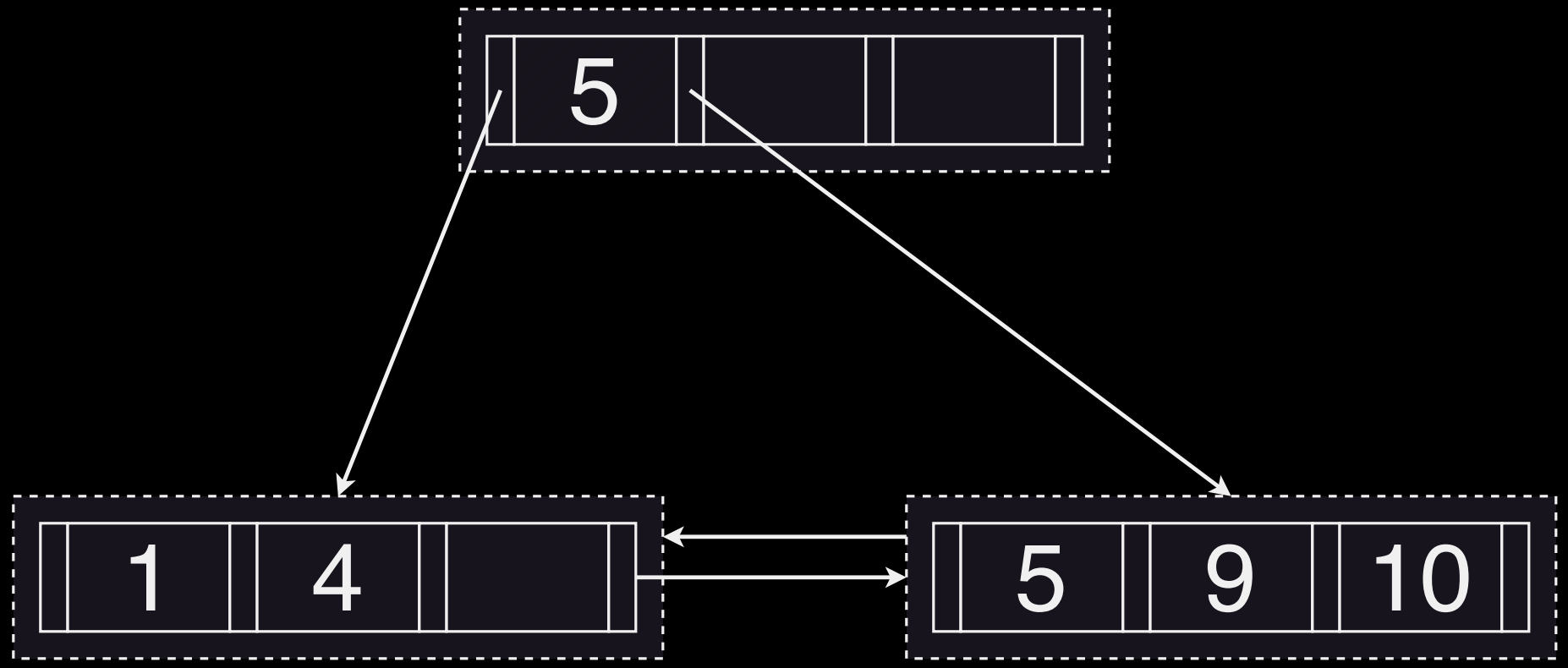
Добавляем 7



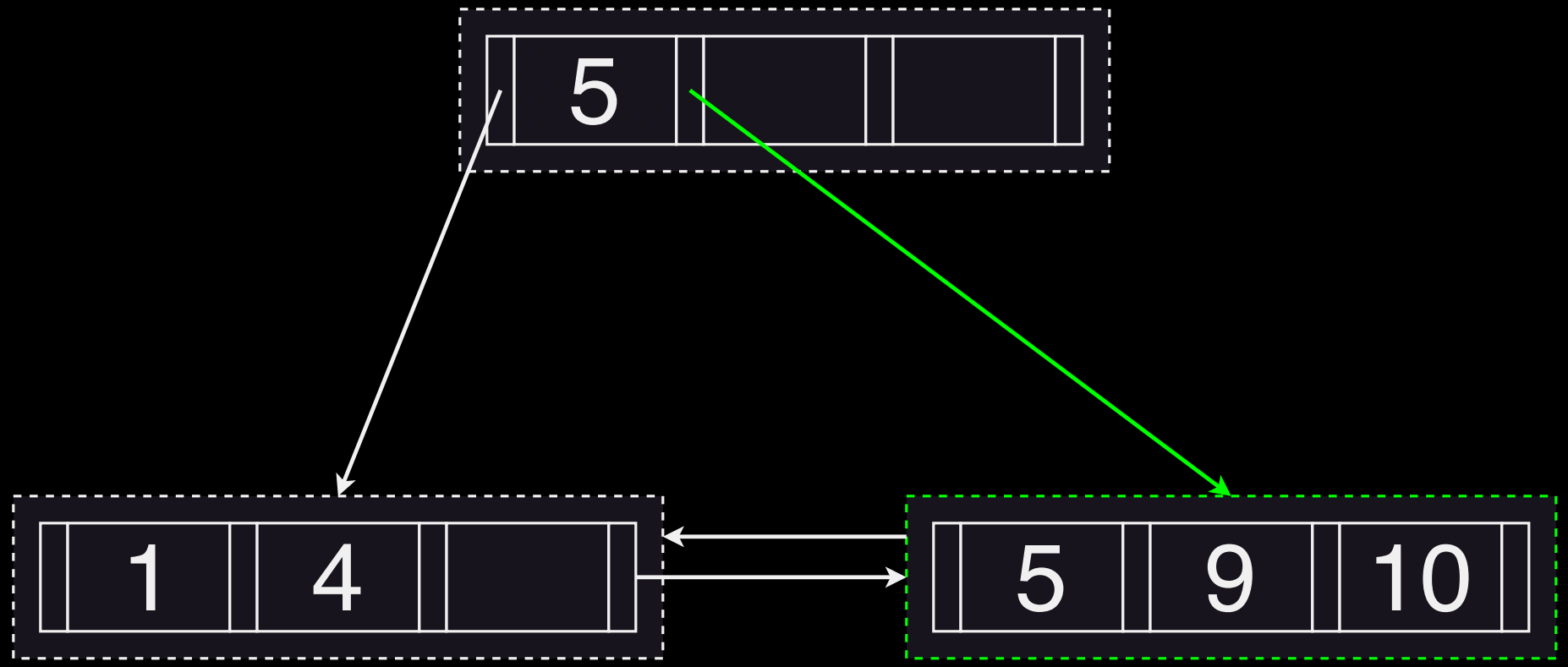
Добавляем 7



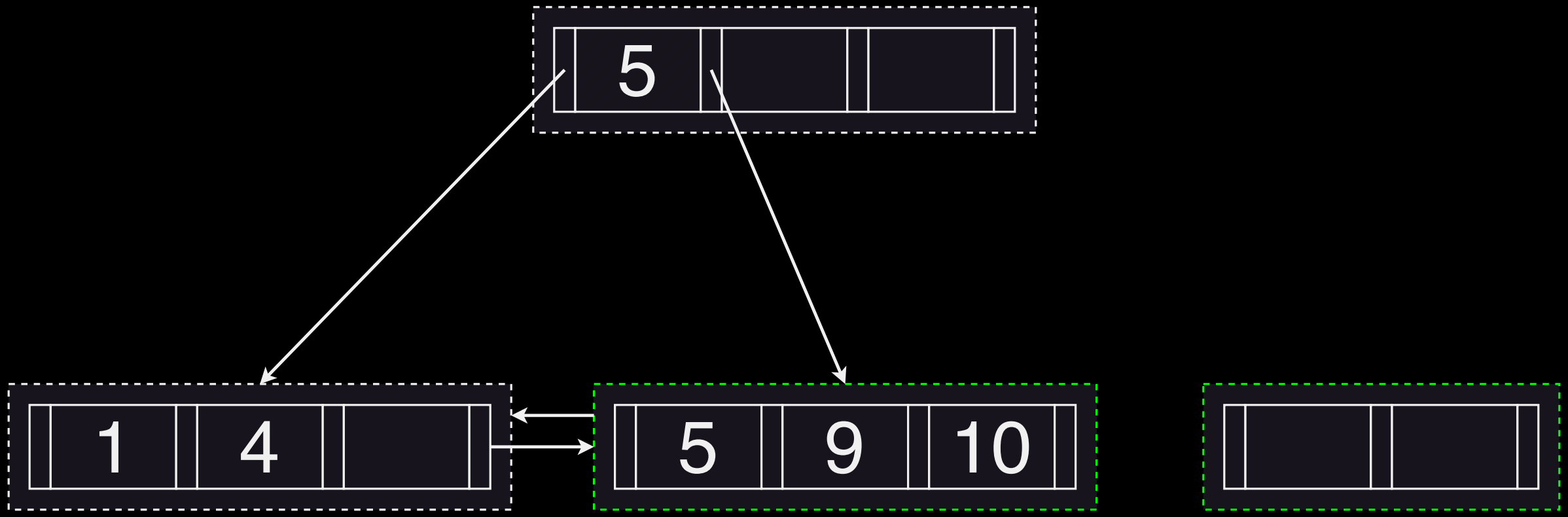
С корня НЕ начинаем



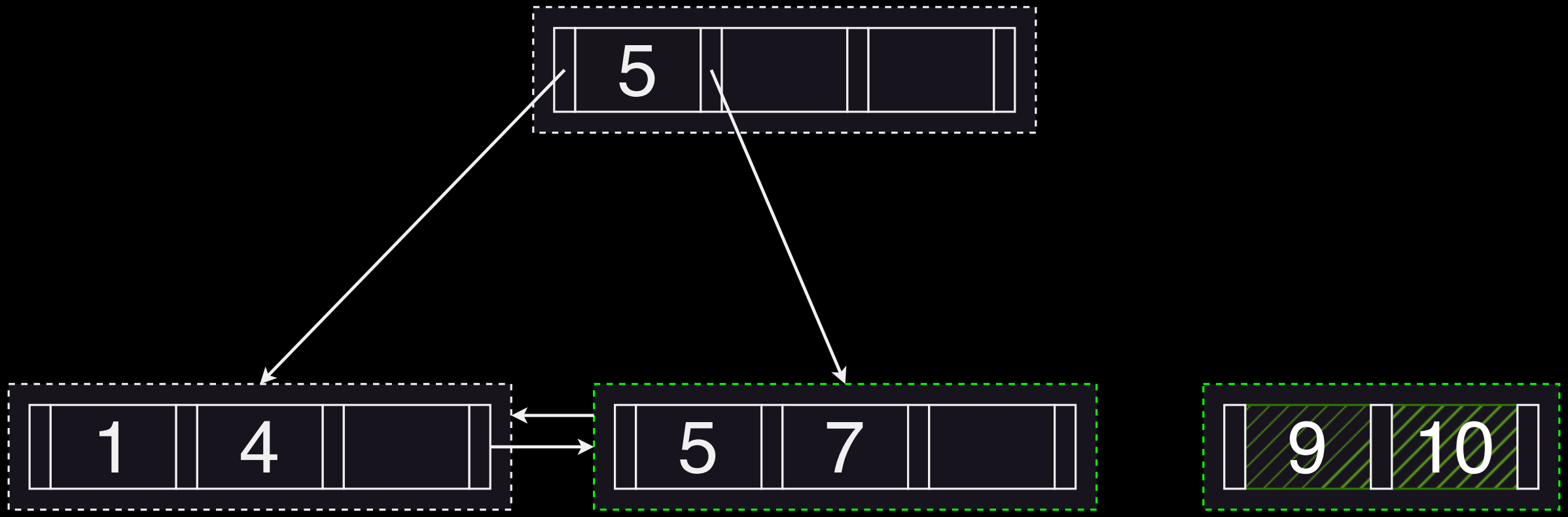
Добавляем 7



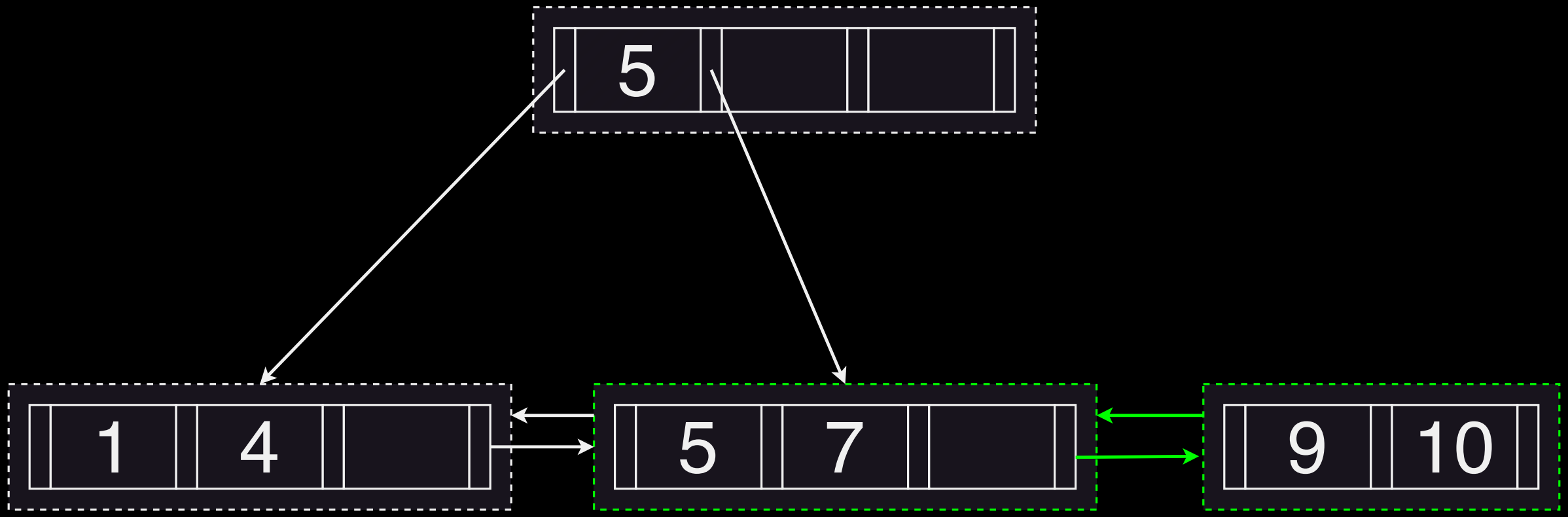
Сначала ищем куда попадает 7



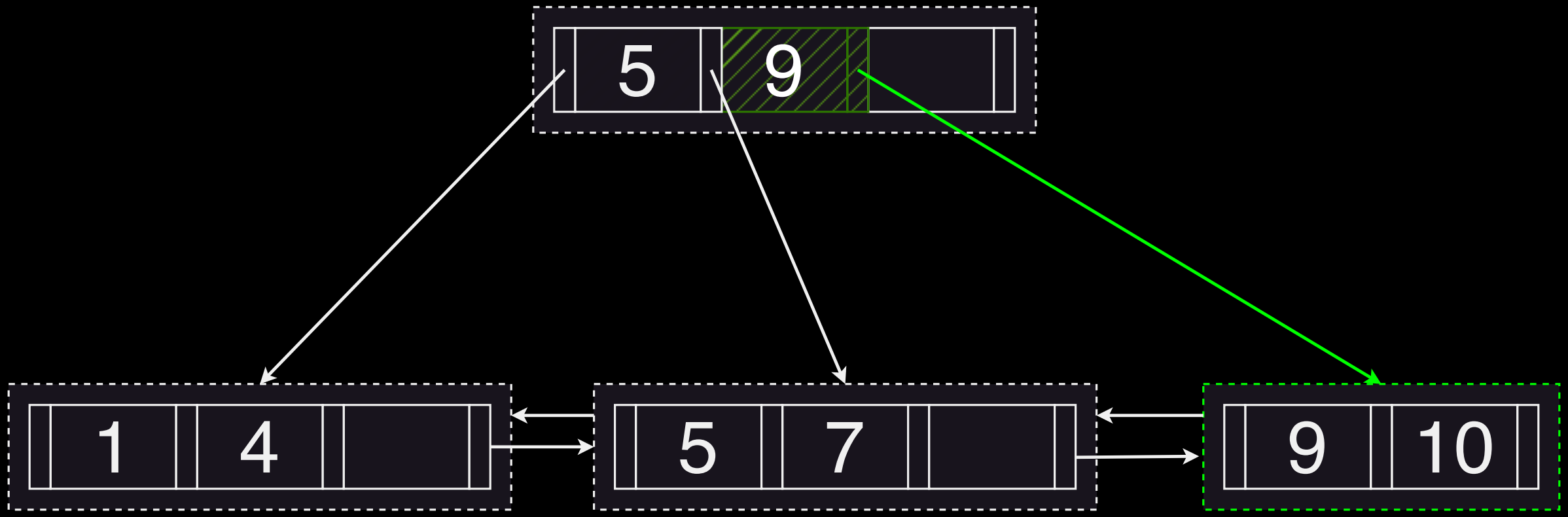
Добавляем пустую страницу



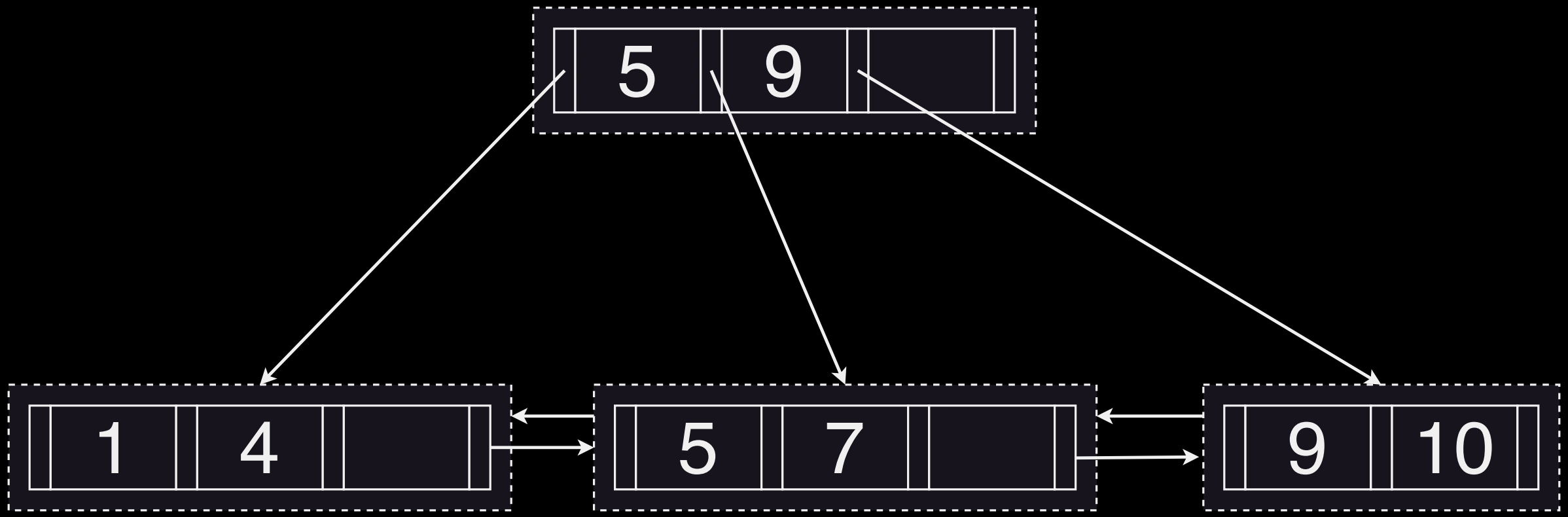
Переносим данные

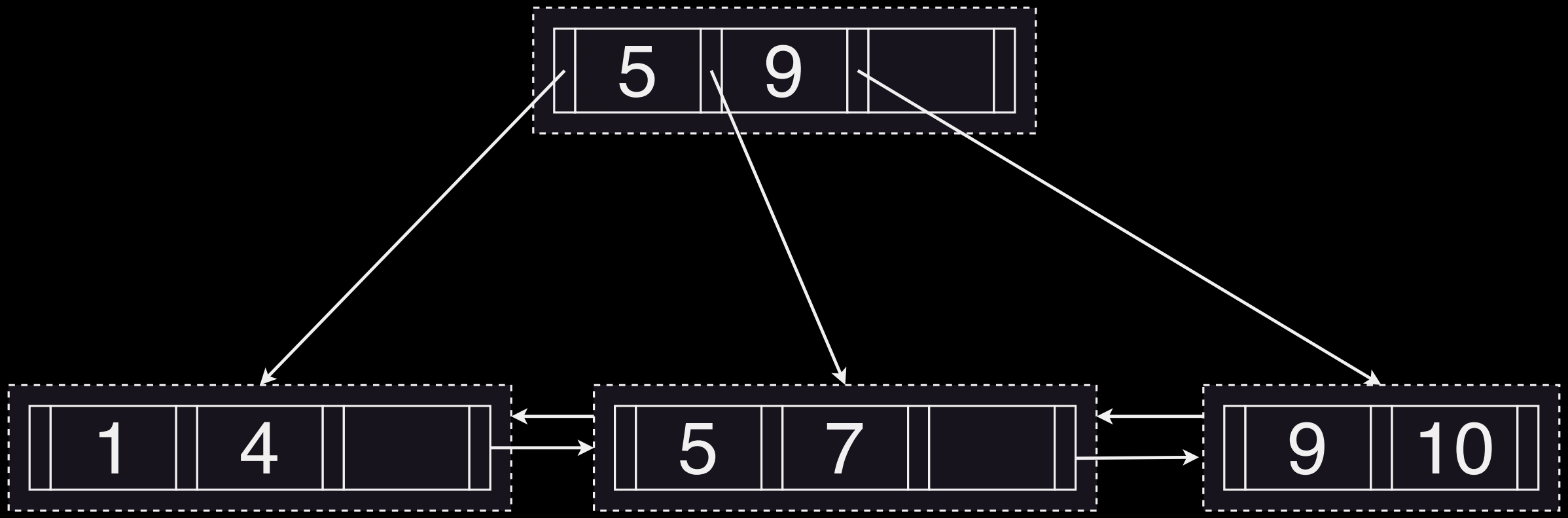


Связываем с индексом

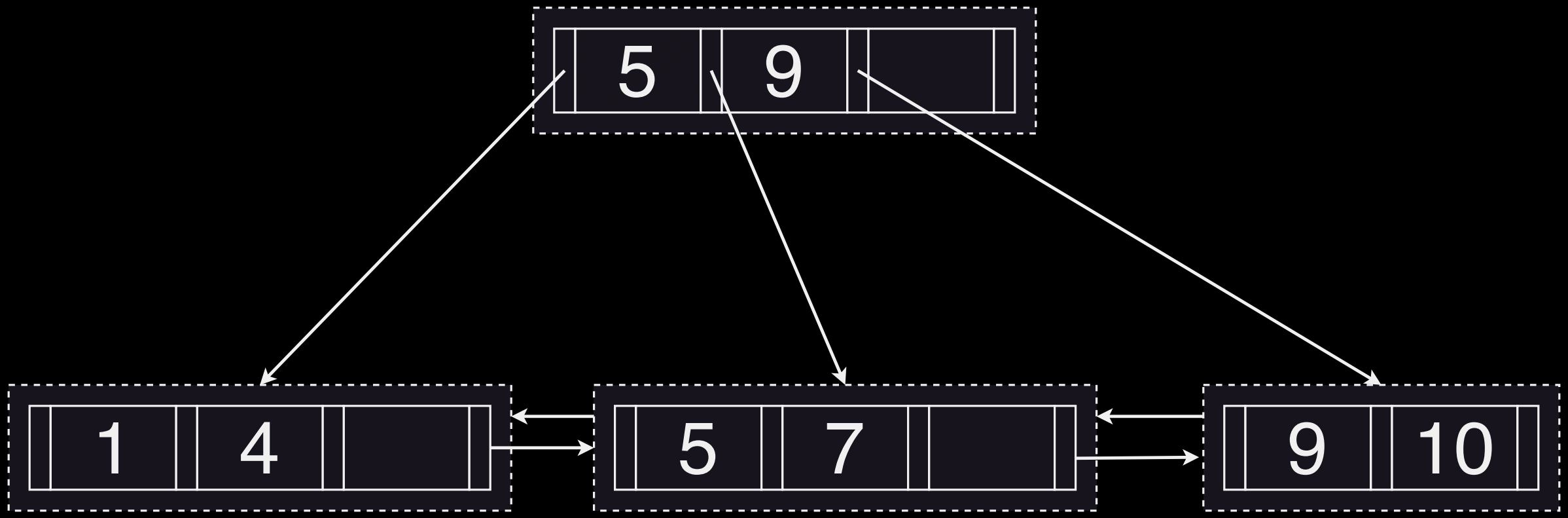


Добавляем запись в родительскую страницу


5
9
1
4
5
7
9
10



На практике в странице 100+ записей



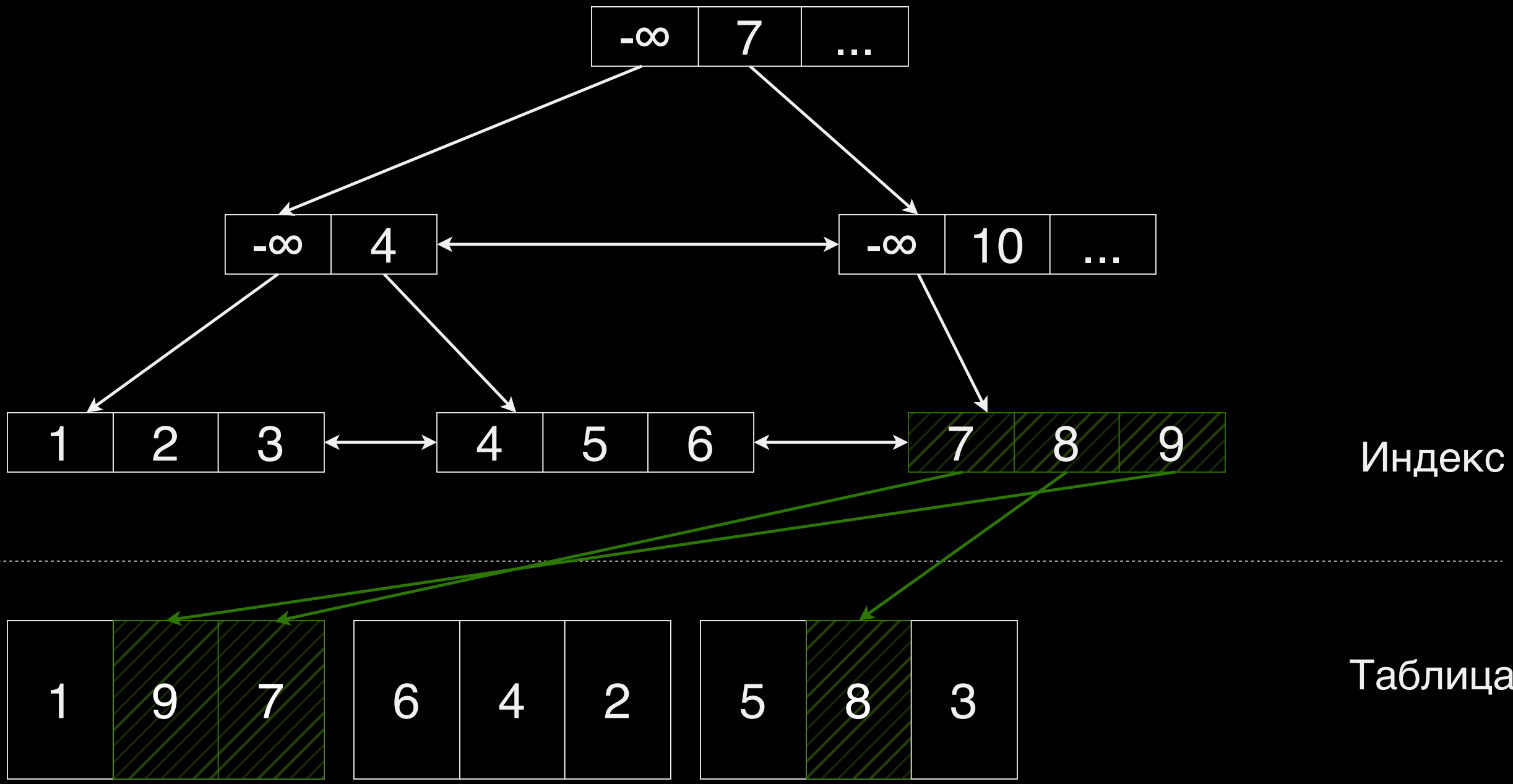
На 1 млн строк нужно всего 3 уровня

UUID vs bigint

```sql
create table users (
  id bigint primary key,
  name varchar(200)
);

vs

create table users (
  id uuid primary key,
  name varchar(200)
);
```

- bigint — 8 байт
- uuid — 16 байт
- Значит индекс по uuid будет хуже, как же ещё?
- Рассмотрим массовую вставку

```
insert into users(id_bigint, name)
  select nextval('serial'), 'test' || g.x
    from generate_series(1, 1000) as g(x);

insert into users(id_uuid, name)
  select gen_random_uuid(), 'test' || g.x
    from generate_series(1, 1000) as g(x);
```


-∞
7
...
-∞
4
-∞
10
...
1
2
3
4
5
6
7
8
9
Индекс
1
9
7
6
4
2
5
8
3
Таблица

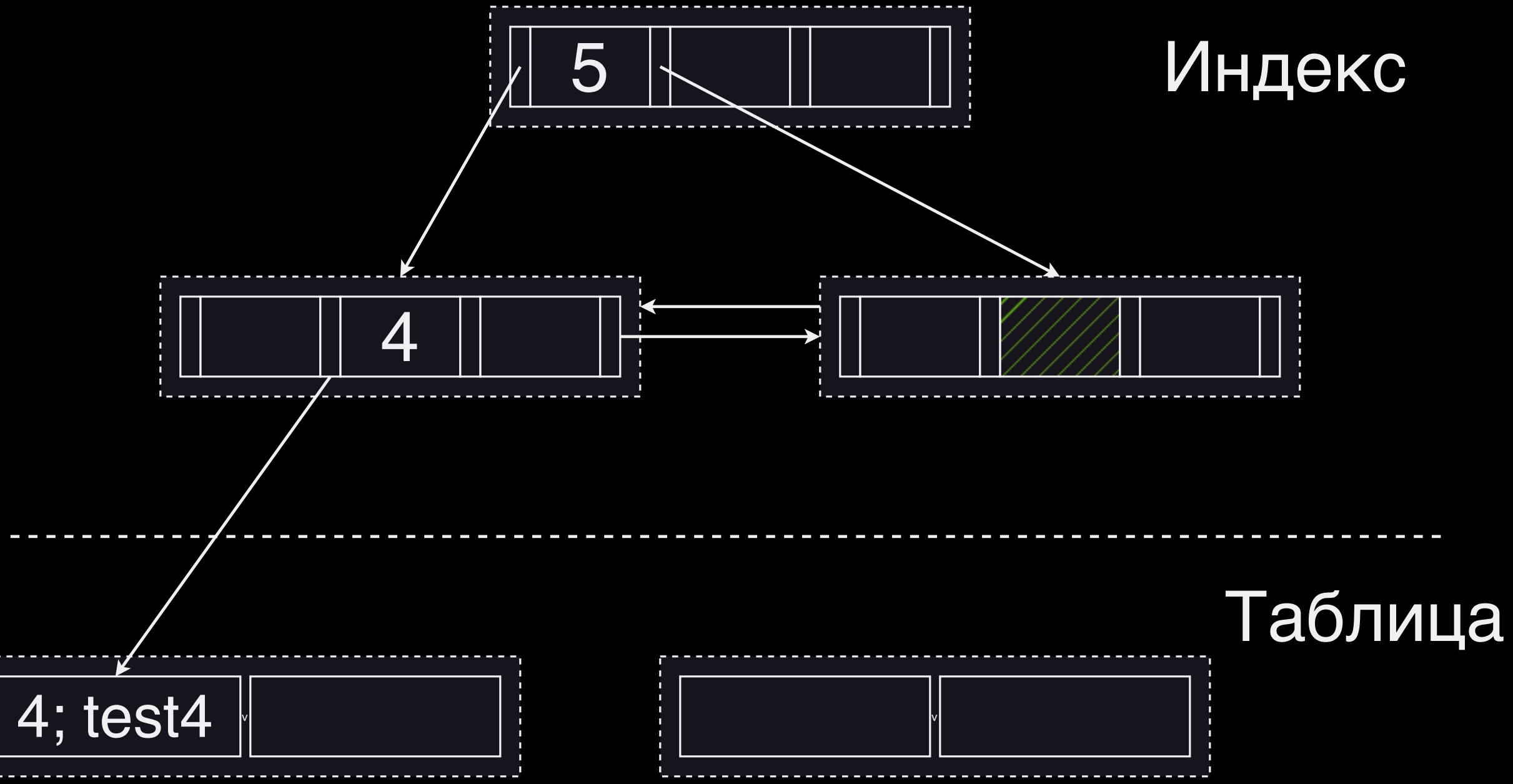
- Случайные UUID будут попадать в случайные места индекса
- Это вызовет больше чтений с диска
- А за счёт full_page_writes будет гораздо больше записей на диск

https://www.2ndquadrant.com/en/blog/on-the-impact-of-full-page-writes/

Что делать с UUID'ами?

- Использовать time-based UUID v7
- Запасаться SSD

Удаление данных


5
Индекс
4
4; test4
Таблица

Удаление из индекса

- Индекс не хранит информацию о видимости строк
- Поэтому удаление не требует обновления индекса
- Но в индексе копится мусор (bloat)
- Его собирает (auto-)vacuum

Удаление в PostgreSQL 14+

- Выполняется в момент разделения блоков
- "Bottom-up" deletion https://www.percona.com/blog/postgresql-14-b-tree-index-reduced-bloat-with-bottom-up-deletion/

Foreign keys

```sql
create table users (
  id bigint primary key,
  user_name varchar(200),
  project_id bigint,
  constraint project_id__users_fk
    foreign key (project_id)
    references projects(id)
);

create index project_id__users_ix on users(project_id);

create table projects (
  id bigint primary key,
  project_name varchar(200),
);
```

Foreign keys

- По умолчанию, индекс на `foreign key` не создаётся
- Без индекса удаление по FK будет блокировать таблицу

```
create table users (
  id bigint primary key,
  project_id bigint,
  constraint project_id__users_fk ...
);

id | project_id
 1 |          1
 2 |          2
 3 |          2
 4 |          2
```

Каждая запись в индексе уникальна за счёт ctid

Пусть все работают над разными проектами

```
  id bigint primary key,
  project_id bigint
);

create UNIQUE index id_project_id__users_ix
    on users(project_id)
    nulls not distinct

id | project_id
 1 |          1
 2 |       null
```

Обновление индексированной колонки

- Добавляется новая запись в индекс
- Старая остаётся мёртвым грузом (bloat)
- Новые версии добавляются во всех индексах, даже тех, которые логически не обновлялись

Обновление НЕиндексированной колонки

- Если обновляем неиндексированную колонку, и строка остаётся на месте
- Если строка не помещается, добавляется новая запись в индекс
- Старая остаётся мёртвым грузом (bloat)

Годы идут, backend растёт

Многоколоночные индексы

```sql
  -- 91% ACTIVE
  --  2% INACTIVE
  --  7% PENDING
);

select id
  from users
 where state = 'PENDING'
   and name = ?;

create index name_state__users on users(name, state);
create index state_name__users on users(state, name);
create index state_name__users on users(state, name) where state =/= 'ACTIVE';
create index state_name__users on users(state, name) where state is not null;
create index state_name__users on users(name) include(state);
```

```
create index ... on users(state, name);

select state, name, ctid
  from users
 order by state, name;
```

(state, name)

ACTIVE	test0
ACTIVE	test1
ACTIVE	test8
ACTIVE	test9

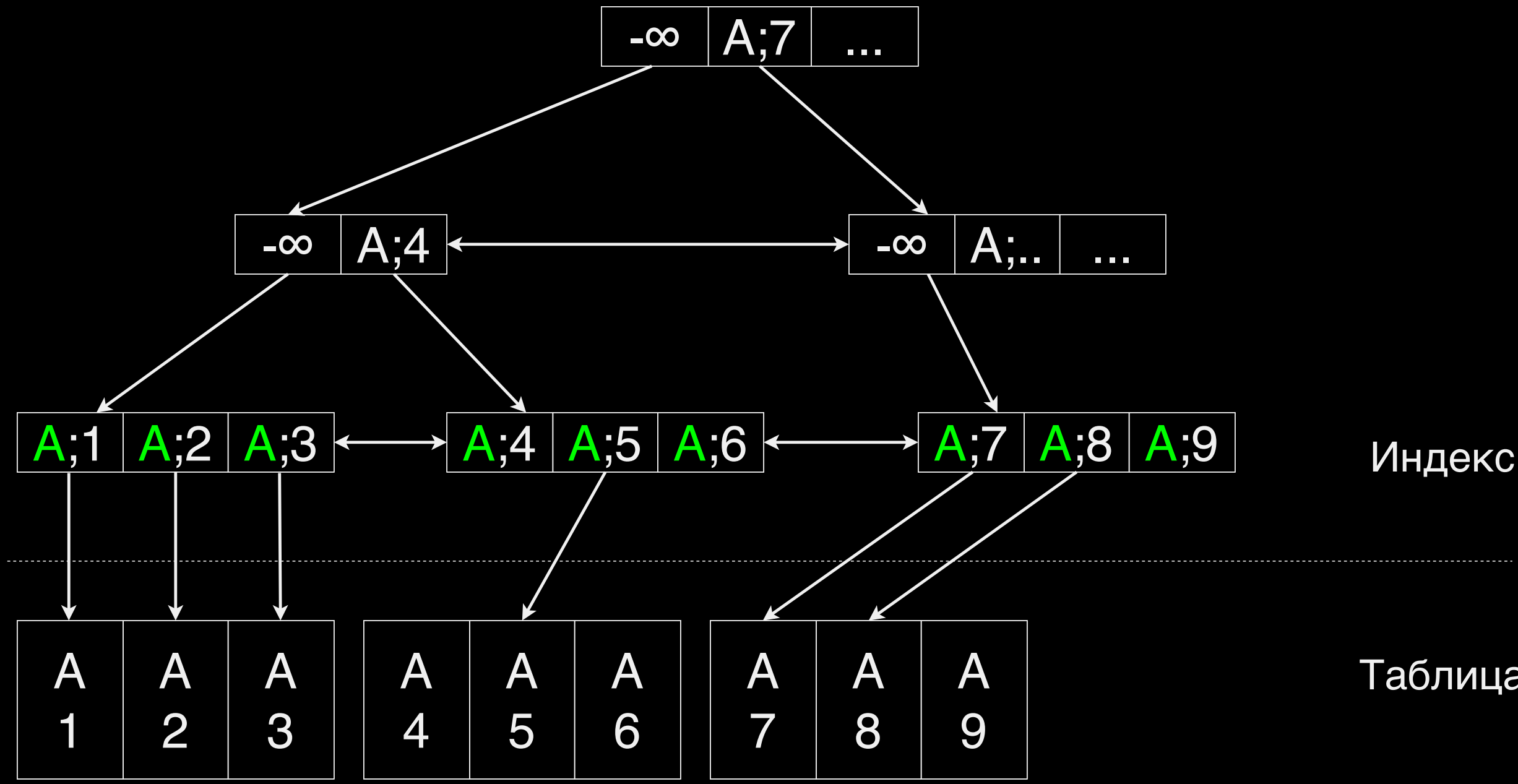
PENDING	test2
PENDING	test4
PENDING	test5
PENDING	test7

(state, name)	(name, state)
ACTIVE test0	test0 ACTIVE
ACTIVE test1	test1 ACTIVE
ACTIVE test8	test2 PENDING
ACTIVE test9	test4 PENDING
PENDING test2	test5 PENDING
PENDING test4	test7 PENDING
PENDING test5	test8 ACTIVE
PENDING test7	test9 ACTIVE

```
select min(state) min_state2
  from users
 where state > min_state1

select min(state) min_state3
  from users
 where state > min_state2

with recursive t as (
  select min(state) state from users
  union all
  select (select min(state) from users where state>t.state)
    from t
   where t.state is not null
)
select * from t
```

Из индекса по (state, name) можно выбирать уникальные значения `state`

https://wiki.postgresql.org/wiki/Loose_indexscan


-∞ A;7 ...
-∞ A;4
-∞ A;.. ...
A;1 A;2 A;3
A;4 A;5 A;6
A;7 A;8 A;9
Индекс
A 1
A 2
A 3
A 4
A 5
A 6
A 7
A 8
A 9
Таблица

```
create index name__users_ix on users(name)
  where state ≠ 'ACTIVE';
```

Ищем по диапазонам

```
select id, price
  from socks
 where id = $sock_id

   and date_from ⩾ 2003
   and 2003 < date_to
 order by date_from
 limit 1

create index from_to__socks
    on socks(id, date_from);
```

```
id | date_from | date_to | price
 1 |      2000 |    2002 |   100
 1 |      2002 |    2004 |   110
```

Запись истории, аудита

```
  id uuid,
  event_time timestamp with time zone,
  client_id bigint,
  message jsonb
)

create index client_id__events
    on events(
      extract(hour from event_time),
      client_id
    );

select *
  from events
 where client_id = $id
   and extract(hour from event_time) in (0, 1, 2, ..., 23)
```

Выводы

- Дружите с вашими индексами
- Учитывайте цель оптимизации
- Иногда имеющихся индексов достаточно

Владимир Ситников

Performance engineer
PgJDBC, JMeter committer
Член программных комитетов
JPoint, Joker, Heisenbug, DevOops, SmartData

VladimirSitnikv
VladimirSitnikv
sitnikov.vladimir@gmail.com