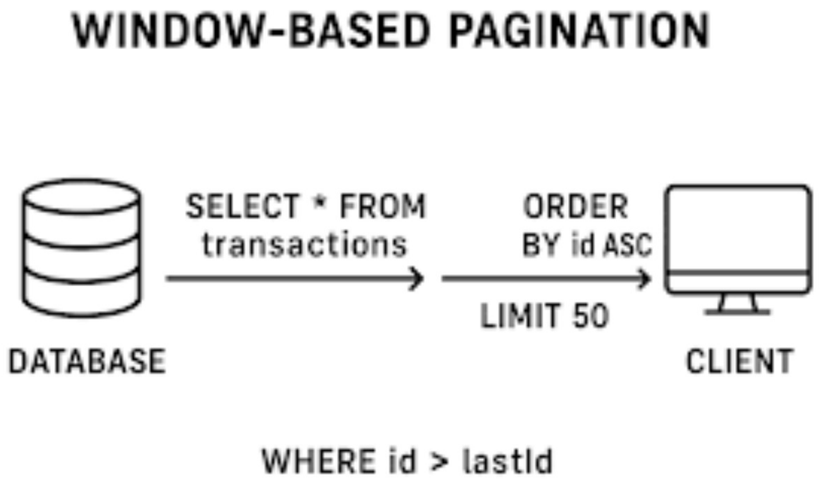
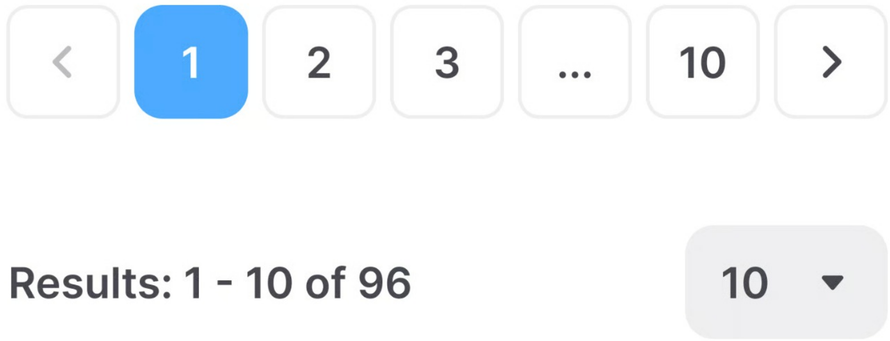
Делаем pagination эффективным





https://struchkov.dev/blog/ru/seek-method-or-keyset-pagination/
https://use-the-index-luke.com/no-offset

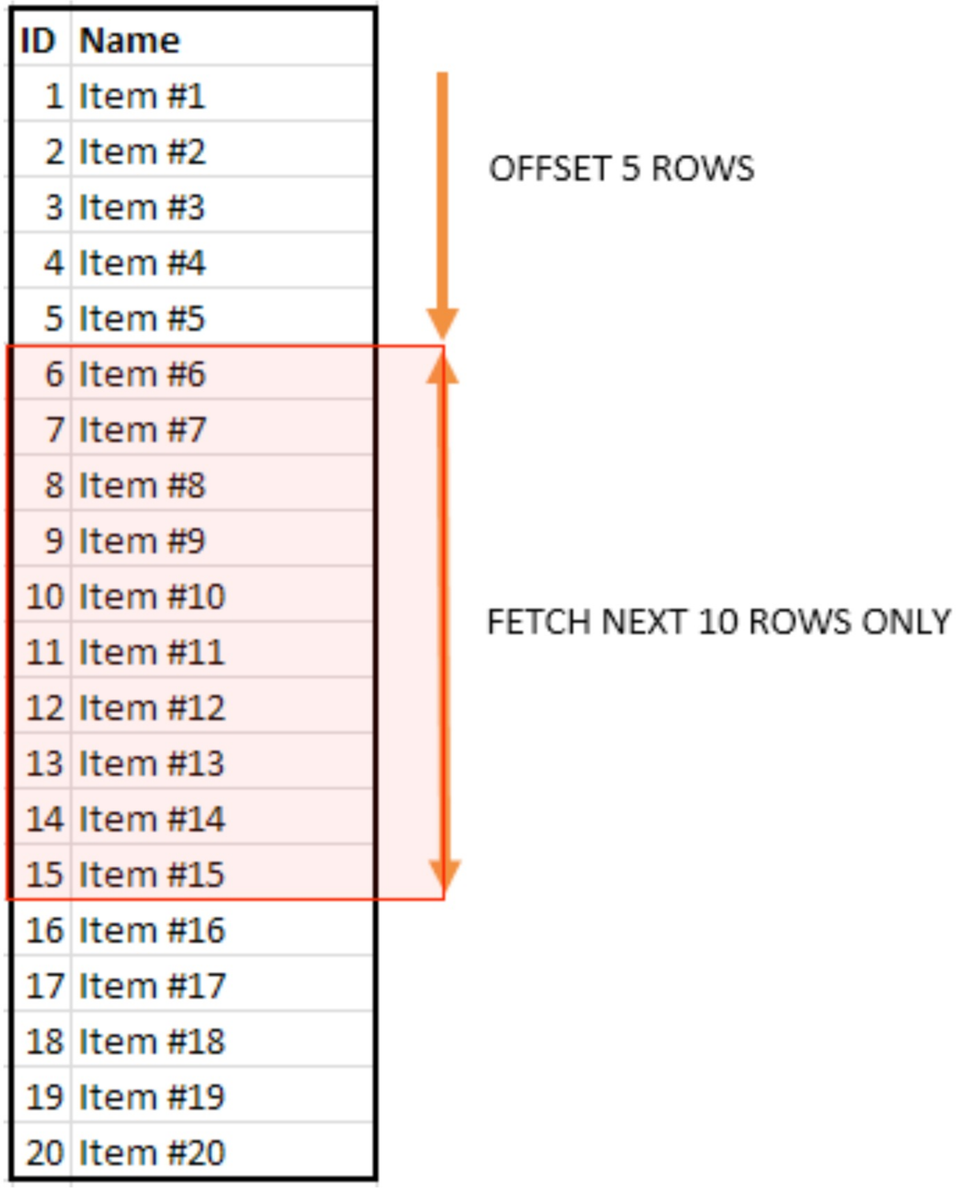
Данных слишком много, чтобы достать за раз нужен способ навигации

Демо пример

```sql
CREATE TABLE orders (
    id BIGSERIAL PRIMARY KEY,
    user_id BIGINT NOT NULL,
    created_at TIMESTAMP NOT NULL
);
```

Как выглядит решение в лоб

```sql
SELECT id, created_at
FROM orders
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET 1000;
```



Проверим как это работает, когда данных стало больше

```
1    EXPLAIN ANALYZE SELECT id, created_at
2    FROM orders
3    ORDER BY created_at DESC, id DESC
4    LIMIT 20 OFFSET 300000;
```

Как работает

Пропускаем первые N строк и берем следующую страницу

```
1    Limit  (cost=10700.36..10701.07 rows=20 width=16)
         (actual time=74.967..74.970 rows=20 loops=1)
2      ->  Index Only Scan using idx_orders_created_id on
           orders  (cost=0.42..15539.65 rows=435682 width
           =16) (actual time=0.174..65.693 rows=300020
           loops=1)
3            Heap Fetches: 0
4    Planning Time: 1.128 ms
5    Execution Time: 75.023 ms
```

Плюсы

- Простая реализация
- Легко интегрировать page в UI
- Легко реализовать сортировки любых видов
- Идеальное решение для маленьких таблиц (несколько десятков тысяч записей)

Минусы

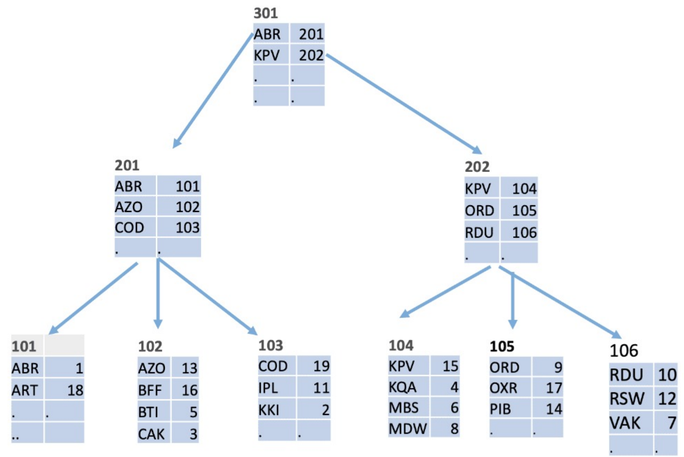
- Линейное время O(N) при больших offset
- Тяжелый на больших данных
- По индексу все равно читается все записи и фактически он не используется
- Может пропускать/повторять записи при вставках/удалениях (сдвиг страниц)

Проблема

Данных много, пагинация убивает базу

Решение

Использовать индексы и не пропускать записи, ищем записи после текущей




```sql
SELECT id, created_at
FROM orders
WHERE (created_at, id) < ('2025-10-07 15:10:28.82737', 287251)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

Проведем замеры

```
EXPLAIN ANALYZE SELECT id, created_at
FROM orders
WHERE (created_at, id) < ('2025-10-07 15:10:28.82737',
    287251)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

Как работает

Обычный индексный досутп

```
Limit  (cost=0.42..1.19 rows=20 width=16) (actual time=0.102..0.109 rows=20 loops=1)
  ->  Index Only Scan using idx_orders_created_id on orders  (cost=0.42..5170.52 rows=135434
      width=16) (actual time=0.098..0.104 rows=20 loops=1)
        Index Cond: (ROW(created_at, id) < ROW('2025-10-07 15:10:28.82737'::timestamp without
            time zone, 287251))
        Heap Fetches: 0
Planning Time: 0.383 ms
Execution Time: 0.197 ms
```

Плюсы

- Очень быстро, стабильное время O(log N + limit)
- Работает на огромных таблицах
- Детерминированный порядок
- Не страдает от вставок/удалений

Минусы

- Нельзя напрямую запросить конкретную страницу по номеру
- Требует хранения/передачи курсора
- Сложнее логика на клиенте

Когда используем

- Ленты, лонг-листы, activity feed
- Hot-path, где пользователи много листают
- Большие таблицы (миллионы/миллиарды строк)

Проблема

Компании же как-то решают эту проблему, гигантские таблицы с пагинацией, неужели платят за огромные базы

Решение

Используем гибридный подход, делаем "виртуальные" страницы, а работаем с курсором (считаем N страницу и дальше работаем с курсором)

```
1    SELECT created_at, id
2    FROM orders
3    ORDER BY created_at DESC, id DESC
4    LIMIT 1 OFFSET 500000 25 000
```

Проведем замеры

```
EXPLAIN ANALYZE SELECT created_at, id
FROM orders
ORDER BY created_at DESC, id DESC
LIMIT 1 OFFSET 500000
```

Как работает

Первый вариант, но берем только 1 элемент

```
Limit  (cost=10700.36..10700.39 rows=1 width=16) (actual time=57.330..57.331 rows=1
    loops=1)
  ->  Index Only Scan using idx_orders_created_id on orders  (cost=0.42..15539.65 rows
      =435682 width=16) (actual time=1.364..45.991 rows=300001 loops=1)
        Heap Fetches: 0
Planning Time: 0.291 ms
Execution Time: 57.381 ms
```

Плюсы

- Поддерживает UI с номерами страниц
- OFFSET используется редко, в cold-path (можно закешировать или посчитать заранее)
- Последующие страницы быстрые

Минусы

- Первый OFFSET все еще затратный
- Усложняет backend
- Логика на клиенте

Когда используем

- UI требует page N
- Большие таблицы, где OFFSET слишком дорог для всех страниц
- Компромисс между UX и производительностью