Задание 1. B-tree индексы в PostgreSQL

1. Запустите БД через docker compose в ./src/docker-compose.yml:

2. Выполните запрос для поиска книги с названием 'Oracle Core' и получите план выполнения:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title = 'Oracle Core';
```

План выполнения:

	QUERY PLAN
1	Seq Scan on t_books (cost=0.00..3100.00 rows=1 width=33) (actual time=26.589..26.591 rows=1 loops=1)
2	Filter: ((title)::text = 'Oracle Core'::text)
3	Rows Removed by Filter: 149999
4	Planning Time: 0.334 ms
5	Execution Time: 26.615 ms

Объясните результат: отсутствие индекса приводит к полному сканированию (`Seq Scan`)

3. Создайте B-tree индексы:

```
CREATE INDEX t_books_title_idx ON t_books(title);
CREATE INDEX t_books_active_idx ON t_books(is_active);
```

Результат:

```
workshop.public> CREATE INDEX t_books_title_idx ON t_books(title)
[2024-11-26 15:08:07] completed in 547 ms
workshop.public> CREATE INDEX t_books_active_idx ON t_books(is_active)
[2024-11-26 15:08:07] completed in 106 ms
```

4. Проверьте информацию о созданных индексах:

```
SELECT schemaname, tablename, indexname, indexdef
FROM pg_catalog.pg_indexes
WHERE tablename = 't_books';
```

Результат:

	schemaname	tablename	indexname	indexdef
1	public	t_books	t_books_id_pk	CREATE UNIQUE INDEX t_books_id_pk ON public.t_books USING btree (book_id)
2	public	t_books	t_books_title_idx	CREATE INDEX t_books_title_idx ON public.t_books USING btree (title)
3	public	t_books	t_books_active_idx	CREATE INDEX t_books_active_idx ON public.t_books USING btree (is_active)

Объясните результат: после выполнения команды из `п.3`, в таблице появились 2 новых индекса (2 и 3 строки)

5. Обновите статистику таблицы:

```
ANALYZE t_books;
```

Результат:

```
workshop.public> ANALYZE t_books
[2024-11-26 15:11:29] completed in 287 ms
```

6. Выполните запрос для поиска книги 'Oracle Core' и получите план выполнения:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title = 'Oracle Core';
```

План выполнения:

```
QUERY PLAN
Index Scan using t_books_title_idx on t_books  (cost=0.42..8.44 rows=1 width=33) (actual time=1.061..1.064 rows=1 loops=1)
  Index Cond: ((title)::text = 'Oracle Core'::text)
Planning Time: 1.216 ms
Execution Time: 1.093 ms
```

Объясните результат: использован индекс `t_books_title_idx`, поэтому время исполнения уменьшилось примерно в 26 раз

7. Выполните запрос для поиска книги по book_id и получите план выполнения:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE book_id = 18;
```

План выполнения:

```
QUERY PLAN
Index Scan using t_books_id_pk on t_books  (cost=0.42..8.44 rows=1 width=33) (actual time=2.199..2.203 rows=1 loops=1)
  Index Cond: (book_id = 18)
Planning Time: 0.152 ms
Execution Time: 2.235 ms
```

Объясните результат: использован индекс `PRIMARY KEY`. Видно, что использование такого индекса сильно уменьшает время планирования (примерно в 12 раз)

8. Выполните запрос для поиска активных книг и получите план выполнения:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE is_active = true;
```

План выполнения:

	QUERY PLAN
1	Seq Scan on t_books (cost=0.00..2725.00 rows=75205 width=33) (actual time=0.011..23.381 rows=75142 loops=1)
2	Filter: is_active
3	Rows Removed by Filter: 74858
4	Planning Time: 0.155 ms
5	Execution Time: 26.802 ms

Объясните результат: был выполнен поиск по всей таблице (`Seq Scan`), индекс `t_books_active_idx` не использовался. Это произошло, потому что слишком много строк (примерно половина) удовлетворяют условию.

9. Посчитайте количество строк и уникальных значений:

```
SELECT
    COUNT(*) as total_rows,
    COUNT(DISTINCT title) as unique_titles,
    COUNT(DISTINCT category) as unique_categories,
    COUNT(DISTINCT author) as unique_authors
FROM t_books;
```

Результат:

	total_rows	unique_titles	unique_categories	unique_authors
1	150000	150000	6	1003

10. Удалите созданные индексы:

```
DROP INDEX t_books_title_idx;
DROP INDEX t_books_active_idx;
```

Результат:

```
workshop.public> DROP INDEX t_books_title_idx
[2024-11-26 16:26:47] completed in 26 ms
workshop.public> DROP INDEX t_books_active_idx
[2024-11-26 16:26:47] completed in 8 ms
```

11. Основываясь на предыдущих результатах, создайте индексы для оптимизации следующих запросов: a. `WHERE title = $1 AND category = $2` b. `WHERE title = $1` c. `WHERE category = $1 AND author = $2` d. `WHERE author = $1 AND book_id = $2`

Созданные индексы:

```
CREATE INDEX t_books_title_category_idx ON t_books(title, category);
CREATE INDEX t_books_category_author_idx ON t_books(category, author);
```

```
CREATE INDEX t_books_author_bookid_idx ON t_books(author, book_id);
```

Объясните ваше решение: [Ваше объяснение]

12. Протестируйте созданные индексы.

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title = 'Oracle Core' AND category =
'Databases';

EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title = 'Oracle Core';

EXPLAIN ANALYZE
SELECT * FROM t_books WHERE category = 'Databases' AND author = 'Jonathan
Lewis';

EXPLAIN ANALYZE
SELECT * FROM t_books WHERE author = 'Jonathan Lewis' AND book_id = 3001;
```

Результаты тестов:

QUERY PLAN
1 Index Scan using t_books_title_category_idx on t_books (cost=0.42..8.44 rows=1 width=33) (actual time=2.040..2.044 rows=1 loops=1)
2 Index Cond: (((title)::text = 'Oracle Core'::text) AND ((category)::text = 'Databases'::text))
3 Planning Time: 2.786 ms
4 Execution Time: 2.117 ms

QUERY PLAN
1 Index Scan using t_books_title_category_idx on t_books (cost=0.42..8.44 rows=1 width=33) (actual time=0.030..0.032 rows=1 loops=1)
2 Index Cond: ((title)::text = 'Oracle Core'::text)
3 Planning Time: 0.132 ms
4 Execution Time: 0.050 ms

QUERY PLAN
1 Index Scan using t_books_category_author_idx on t_books (cost=0.29..8.31 rows=1 width=33) (actual time=2.977..2.999 rows=1 loops=1)
2 Index Cond: (((category)::text = 'Databases'::text) AND ((author)::text = 'Jonathan Lewis'::text))
3 Planning Time: 0.341 ms
4 Execution Time: 3.048 ms

QUERY PLAN
1 Index Scan using t_books_author_bookid_idx on t_books (cost=0.42..8.44 rows=1 width=33) (actual time=3.185..3.187 rows=1 loops=1)
2 Index Cond: (((author)::text = 'Jonathan Lewis'::text) AND (book_id = 3001))
3 Planning Time: 0.079 ms
4 Execution Time: 3.203 ms

Объясните результаты: первый, третий и четвертый запрос ожидаемо используют созданные комбинированные индексы. Второй запрос использует индекс, созданный *как бы* для первого, потому что он выполняет поиск по `title`

13. Выполните регистронезависимый поиск по началу названия:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title ILIKE 'Relational%';
```

План выполнения:

```
QUERY PLAN
Seq Scan on t_books  (cost=0.00..3100.00 rows=15 width=33) (actual time=109.934..109.935 rows=0 loops=1)
  Filter: ((title)::text ~~* 'Relational%'::text)
  Rows Removed by Filter: 150000
Planning Time: 5.386 ms
Execution Time: 109.956 ms
```

Объясните результат: используется `Seq Scan`, т.к. созданный индекс не поддерживает операцию поиска по регистронезависимому шаблону

14. Создайте функциональный индекс:

```
CREATE INDEX t_books_up_title_idx ON t_books(UPPER(title));
```

Результат:

```
workshop.public> CREATE INDEX t_books_up_title_idx ON t_books(UPPER(title))
[2024-11-26 16:45:09] completed in 405 ms
```

15. Выполните запрос из шага 13 с использованием UPPER:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE UPPER(title) LIKE 'RELATIONAL%';
```

План выполнения:

```
QUERY PLAN
Seq Scan on t_books  (cost=0.00..3475.00 rows=750 width=33) (actual time=58.136..58.137 rows=0 loops=1)
  Filter: (upper((title)::text) ~~ 'RELATIONAL%'::text)
  Rows Removed by Filter: 150000
Planning Time: 0.085 ms
Execution Time: 58.158 ms
```

Объясните результат: все равно используется `Seq Scan`. Я подозреваю, что проблема в функции `LIKE`, т.к. если выполнить

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE UPPER(title) = 'RELATIONAL';
```

```
QUERY PLAN
Index Scan using t_books_up_title_idx on t_books  (cost=0.42..8.44 rows=1 width=33) (actual time=1.598..1.599 rows=0 loops=1)
  Index Cond: (upper((title)::text) = 'RELATIONAL'::text)
Planning Time: 0.219 ms
Execution Time: 1.637 ms
```

индекс используется

16. Выполните поиск подстроки:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title ILIKE '%Core%';
```

План выполнения:

```
QUERY PLAN
Seq Scan on t_books  (cost=0.00..3100.00 rows=15 width=33) (actual time=103.267..103.272 rows=1 loops=1)
  Filter: ((title)::text ~~* '%Core%'::text)
  Rows Removed by Filter: 149999
Planning Time: 0.389 ms
Execution Time: 103.334 ms
```

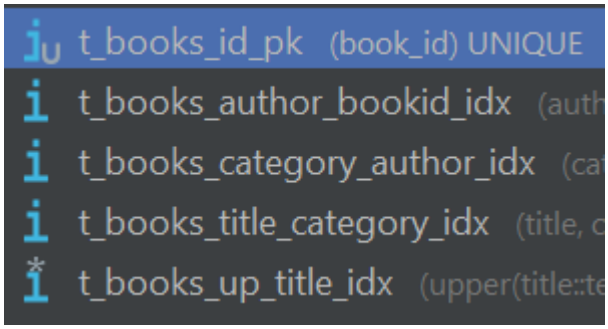
Объясните результат: аналогично заданию 13, не поддерживается регистронезависимый поиск

17. Попробуйте удалить все индексы:

```
DO $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN (SELECT indexname FROM pg_indexes
              WHERE tablename = 't_books'
              AND indexname != 'books_pkey')
    LOOP
        EXECUTE 'DROP INDEX ' || r.indexname;
    END LOOP;
END $$;
```

Результат:

```
[2BP01] ERROR: cannot drop index t_books_id_pk because constraint t_books_id_pk on table t_books requires it
Подсказка: You can drop constraint t_books_id_pk on table t_books instead.
Где: SQL statement "DROP INDEX t_books_id_pk"
PL/pgSQL function inline_code_block line 9 at EXECUTE
```



Объясните результат: Postgres не дает нам удалить индекс, связанный с первичным ключом, а из-за того, что он в списке индексов первый, остальные тоже остались нетронутыми

18. Создайте индекс для оптимизации суффиксного поиска:

```
-- Вариант 1: с reverse()
CREATE INDEX t_books_rev_title_idx ON t_books(reverse(title));
```

```
-- Вариант 2: с триграммами
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE INDEX t_books_trgm_idx ON t_books USING gin (title gin_trgm_ops);
```

Результаты тестов:

```
workshop.public> CREATE INDEX t_books_rev_title_idx ON t_books(reverse(title))
[2024-11-26 17:06:45] completed in 381 ms
```

	QUERY PLAN
1	Seq Scan on t_books (cost=0.00..3100.00 rows=15 width=33) (actual time=112.558..112.563 rows=1 loops=1)
2	Filter: ((title)::text ~~* '%Core%'::text)
3	Rows Removed by Filter: 149999
4	Planning Time: 1.191 ms
5	Execution Time: 112.586 ms

```
workshop.public> CREATE EXTENSION IF NOT EXISTS pg_trgm
[2024-11-26 17:08:13] completed in 53 ms
workshop.public> CREATE INDEX t_books_trgm_idx ON t_books USING gin (title gin_trgm_ops)
[2024-11-26 17:08:13] completed in 519 ms
```

	QUERY PLAN
1	Bitmap Heap Scan on t_books (cost=21.57..76.78 rows=15 width=33) (actual time=0.074..0.075 rows=1 loops=1)
2	Recheck Cond: ((title)::text ~~* '%Core%'::text)
3	Heap Blocks: exact=1
4	-> Bitmap Index Scan on t_books_trgm_idx (cost=0.00..21.56 rows=15 width=0) (actual time=0.063..0.063 rows=1 loops=1)
5	Index Cond: ((title)::text ~~* '%Core%'::text)
6	Planning Time: 0.506 ms
7	Execution Time: 0.108 ms

Объясните результаты: индекс с `reverse()` нам ничего не дал, а вот триграммы вызвали `Bitmap Heap Scan`, что ускорило запрос примерно в 1120 раз

19. Выполните поиск по точному совпадению:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title = 'Oracle Core';
```

План выполнения:

	QUERY PLAN
1	Index Scan using t_books_title_category_idx on t_books (cost=0.42..8.44 rows=1 width=33) (actual time=0.026..0.027 rows=1 loops=1)
2	Index Cond: ((title)::text = 'Oracle Core'::text)
3	Planning Time: 0.238 ms
4	Execution Time: 0.048 ms

Объясните результат: использован индекс `t_books_title_category_idx`, т.к. он в себе содержит `title`

20. Выполните поиск по началу названия:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE title ILIKE 'Relational%';
```

План выполнения:

```
QUERY PLAN
Bitmap Heap Scan on t_books  (cost=95.15..150.36 rows=15 width=33) (actual time=0.062..0.062 rows=0 loops=1)
  Recheck Cond: ((title)::text ~~* 'Relational%'::text)
  Rows Removed by Index Recheck: 1
  Heap Blocks: exact=1
  ->  Bitmap Index Scan on t_books_trgm_idx  (cost=0.00..95.15 rows=15 width=0) (actual time=0.049..0.049 rows=1 loops=1)
        Index Cond: ((title)::text ~~* 'Relational%'::text)
Planning Time: 0.258 ms
Execution Time: 0.088 ms
```

Объясните результат: использован `Bitmap Heap Scan` благодаря триграммам

21. Создайте свой пример индекса с обратной сортировкой:

```
CREATE INDEX t_books_desc_idx ON t_books(title DESC);
```

Тестовый запрос:

```
EXPLAIN ANALYZE
SELECT * FROM t_books ORDER BY title DESC;
```

План выполнения:

```
QUERY PLAN
Index Scan using t_books_desc_idx on t_books  (cost=0.42..9070.73 rows=150000 width=33) (actual time=0.095..63.124 rows=150000 loops=1)
Planning Time: 0.118 ms
Execution Time: 71.402 ms
```

Объясните результат: используется созданный индекс, который предоставляет данные уже отсортированными, что сильно уменьшает время выполнения запроса

Задание 2: Специальные случаи использования индексов

Партиционирование и специальные случаи использования индексов

1. Удалите прошлый инстанс PostgreSQL - `docker-compose down` в папке `src` и запустите новый: `docker-compose up -d`.

2. Создайте партиционированную таблицу и заполните её данными:

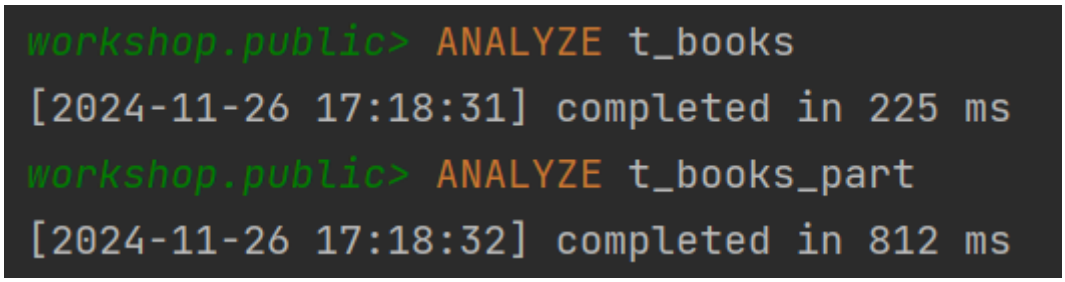
```
-- Создание партиционированной таблицы
CREATE TABLE t_books_part (
    book_id     INTEGER      NOT NULL,
    title       VARCHAR(100) NOT NULL,
    category    VARCHAR(30),
```

```
    author      VARCHAR(100) NOT NULL,
    is_active   BOOLEAN      NOT NULL
) PARTITION BY RANGE (book_id);

-- Создание партиций
CREATE TABLE t_books_part_1 PARTITION OF t_books_part
    FOR VALUES FROM (MINVALUE) TO (50000);

CREATE TABLE t_books_part_2 PARTITION OF t_books_part
    FOR VALUES FROM (50000) TO (100000);

CREATE TABLE t_books_part_3 PARTITION OF t_books_part
    FOR VALUES FROM (100000) TO (MAXVALUE);

-- Копирование данных из t_books
INSERT INTO t_books_part
SELECT * FROM t_books;
```

3. Обновите статистику таблиц:

```
ANALYZE t_books;
ANALYZE t_books_part;
```

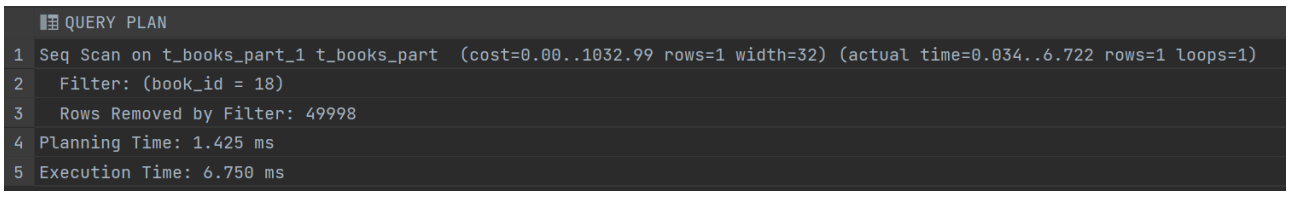
Результат:



4. Выполните запрос для поиска книги с id = 18:

```
EXPLAIN ANALYZE
SELECT * FROM t_books_part WHERE book_id = 18;
```

План выполнения:



Объясните результат: система выбрала патрицию, в которой находятся значения `book_id = 18` и выполнила по ней `Seq Scan` вместо поиска по индексу, т.к. `PRIMARY KEY` не "унаследовался" патрициями от своего родителя

5. Выполните поиск по названию книги:

```
EXPLAIN ANALYZE
SELECT * FROM t_books_part
WHERE title = 'Expert PostgreSQL Architecture';
```

План выполнения:

```
QUERY PLAN
Append  (cost=0.00..3100.01 rows=3 width=33) (actual time=7.145..20.161 rows=1 loops=1)
  ->  Seq Scan on t_books_part_1  (cost=0.00..1032.99 rows=1 width=32) (actual time=7.143..7.144 rows=1 loops=1)
        Filter: ((title)::text = 'Expert PostgreSQL Architecture'::text)
        Rows Removed by Filter: 49998
  ->  Seq Scan on t_books_part_2  (cost=0.00..1034.00 rows=1 width=33) (actual time=6.652..6.652 rows=0 loops=1)
        Filter: ((title)::text = 'Expert PostgreSQL Architecture'::text)
        Rows Removed by Filter: 50000
  ->  Seq Scan on t_books_part_3  (cost=0.00..1033.01 rows=1 width=34) (actual time=6.354..6.354 rows=0 loops=1)
        Filter: ((title)::text = 'Expert PostgreSQL Architecture'::text)
        Rows Removed by Filter: 50001
Planning Time: 0.393 ms
Execution Time: 20.209 ms
```

Объясните результат: т.к. патрицирование было сделано по `book_id` подходящие под запрос значения могли попасть в каждую патрицию, поэтому происходит `Seq Scan` по каждой патриции

6. Создайте партиционированный индекс:

```
CREATE INDEX ON t_books_part(title);
```

Результат:

```
workshop.public> CREATE INDEX ON t_books_part(title)
[2024-11-26 17:25:38] completed in 318 ms
```

7. Повторите запрос из шага 5:

```
EXPLAIN ANALYZE
SELECT * FROM t_books_part
WHERE title = 'Expert PostgreSQL Architecture';
```

План выполнения:

```
QUERY PLAN
Append  (cost=0.29..24.94 rows=3 width=33) (actual time=0.062..0.128 rows=1 loops=1)
  ->  Index Scan using t_books_part_1_title_idx on t_books_part_1  (cost=0.29..8.31 rows=1 width=32) (actual time=0.061..0.062 rows=1 loops=1)
        Index Cond: ((title)::text = 'Expert PostgreSQL Architecture'::text)
  ->  Index Scan using t_books_part_2_title_idx on t_books_part_2  (cost=0.29..8.31 rows=1 width=33) (actual time=0.026..0.026 rows=0 loops=1)
        Index Cond: ((title)::text = 'Expert PostgreSQL Architecture'::text)
  ->  Index Scan using t_books_part_3_title_idx on t_books_part_3  (cost=0.29..8.31 rows=1 width=34) (actual time=0.037..0.037 rows=0 loops=1)
        Index Cond: ((title)::text = 'Expert PostgreSQL Architecture'::text)
Planning Time: 0.697 ms
Execution Time: 0.163 ms
```

Объясните результат: всё аналогично `п.5`, но используется `Index Scan`, т.к. мы явно создали индекс по `title`. В каждой патриции создался свой индекс.

8. Удалите созданный индекс:

```
DROP INDEX t_books_part_title_idx;
```

Результат:

```
workshop.public> DROP INDEX t_books_part_title_idx
[2024-11-26 17:27:10] completed in 15 ms
```

9. Создайте индекс для каждой партиции:

```
CREATE INDEX ON t_books_part_1(title);
CREATE INDEX ON t_books_part_2(title);
CREATE INDEX ON t_books_part_3(title);
```

Результат:

```
workshop.public> CREATE INDEX ON t_books_part_1(title)
[2024-11-26 17:27:41] completed in 287 ms
workshop.public> CREATE INDEX ON t_books_part_2(title)
[2024-11-26 17:27:41] completed in 72 ms
workshop.public> CREATE INDEX ON t_books_part_3(title)
[2024-11-26 17:27:41] completed in 63 ms
```

10. Повторите запрос из шага 5:

```
EXPLAIN ANALYZE
SELECT * FROM t_books_part
WHERE title = 'Expert PostgreSQL Architecture';
```

План выполнения:

```
QUERY PLAN
Append  (cost=0.29..24.94 rows=3 width=33) (actual time=0.026..0.052 rows=1 loops=1)
  ->  Index Scan using t_books_part_1_title_idx on t_books_part_1  (cost=0.29..8.31 rows=1 width=32) (actual time=0.025..0.025 rows=1 loops=1)
        Index Cond: ((title)::text = 'Expert PostgreSQL Architecture'::text)
  ->  Index Scan using t_books_part_2_title_idx on t_books_part_2  (cost=0.29..8.31 rows=1 width=33) (actual time=0.013..0.013 rows=0 loops=1)
        Index Cond: ((title)::text = 'Expert PostgreSQL Architecture'::text)
  ->  Index Scan using t_books_part_3_title_idx on t_books_part_3  (cost=0.29..8.31 rows=1 width=34) (actual time=0.011..0.011 rows=0 loops=1)
        Index Cond: ((title)::text = 'Expert PostgreSQL Architecture'::text)
Planning Time: 0.562 ms
Execution Time: 0.078 ms
```

Объясните результат: ситуация полностью как в п.7, только теперь мы руками создали индексы в отдельных патрициях

11. Удалите созданные индексы:

```
DROP INDEX t_books_part_1_title_idx;
DROP INDEX t_books_part_2_title_idx;
DROP INDEX t_books_part_3_title_idx;
```

Результат:

```
workshop.public> DROP INDEX t_books_part_1_title_idx
[2024-11-26 17:30:37] completed in 9 ms
workshop.public> DROP INDEX t_books_part_2_title_idx
[2024-11-26 17:30:37] completed in 7 ms
workshop.public> DROP INDEX t_books_part_3_title_idx
[2024-11-26 17:30:37] completed in 4 ms
```

12. Создайте обычный индекс по book_id:

```
CREATE INDEX t_books_part_idx ON t_books_part(book_id);
```

Результат:

```
workshop.public> CREATE INDEX t_books_part_idx ON t_books_part(book_id)
[2024-11-26 17:31:05] completed in 89 ms
```

13. Выполните поиск по book_id:

```
EXPLAIN ANALYZE
SELECT * FROM t_books_part WHERE book_id = 11011;
```

План выполнения:

```
   QUERY PLAN
1  Index Scan using t_books_part_1_book_id_idx on t_book…
2    Index Cond: (book_id = 11011)
3  Planning Time: 0.633 ms
4  Execution Time: 0.085 ms
```

Объясните результат: произошел поиск по индексу только в патриции, в которой могут находиться строки с `book_id = 11011`

14. Создайте индекс по полю is_active:

```
CREATE INDEX t_books_active_idx ON t_books(is_active);
```

Результат:

```
workshop.public> CREATE INDEX t_books_active_idx ON t_books(is_active)
[2024-11-26 17:32:58] completed in 90 ms
```

15. Выполните поиск активных книг с отключенным последовательным сканированием:

```
SET enable_seqscan = off;
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE is_active = true;
SET enable_seqscan = on;
```

План выполнения:

	QUERY PLAN
1	Bitmap Heap Scan on t_books (cost=850.47..2830.82 ro…
2	Recheck Cond: is_active
3	Heap Blocks: exact=1224
4	-> Bitmap Index Scan on t_books_active_idx (cost=…
5	Index Cond: (is_active = true)
6	Planning Time: 0.306 ms
7	Execution Time: 17.939 ms

Объясните результат: используется `Bitmap Heap Scan` с последующим `Bitmap Index Scan` по созданному индексу

16. Создайте составной индекс:

```
CREATE INDEX t_books_author_title_index ON t_books(author, title);
```

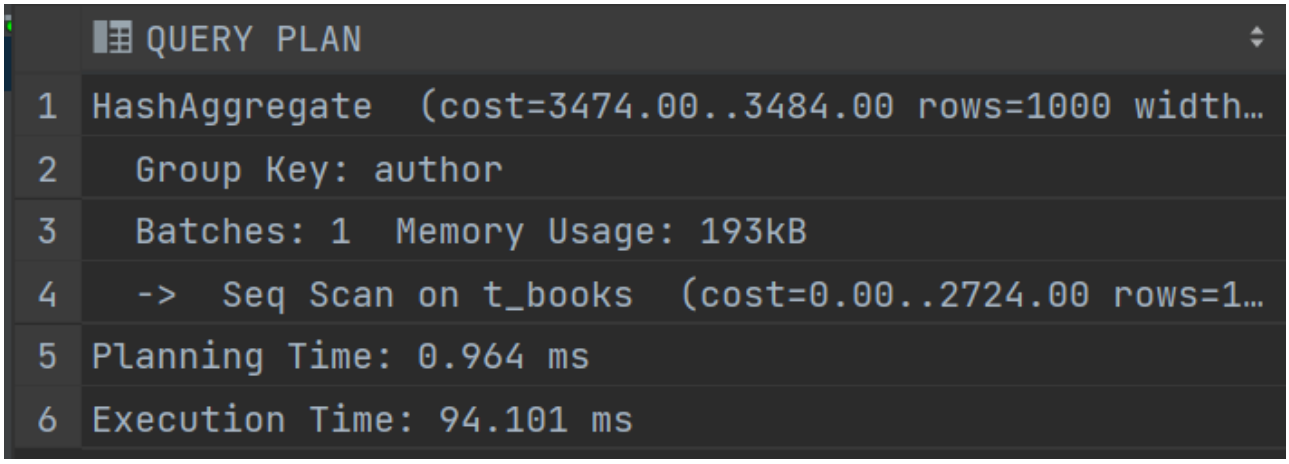
Результат:

```
workshop.public> CREATE INDEX t_books_author_title_index ON t_books(author, title)
[2024-11-26 17:34:50] completed in 448 ms
```

17. Найдите максимальное название для каждого автора:

```
EXPLAIN ANALYZE
SELECT author, MAX(title)
FROM t_books
GROUP BY author;
```

План выполнения:



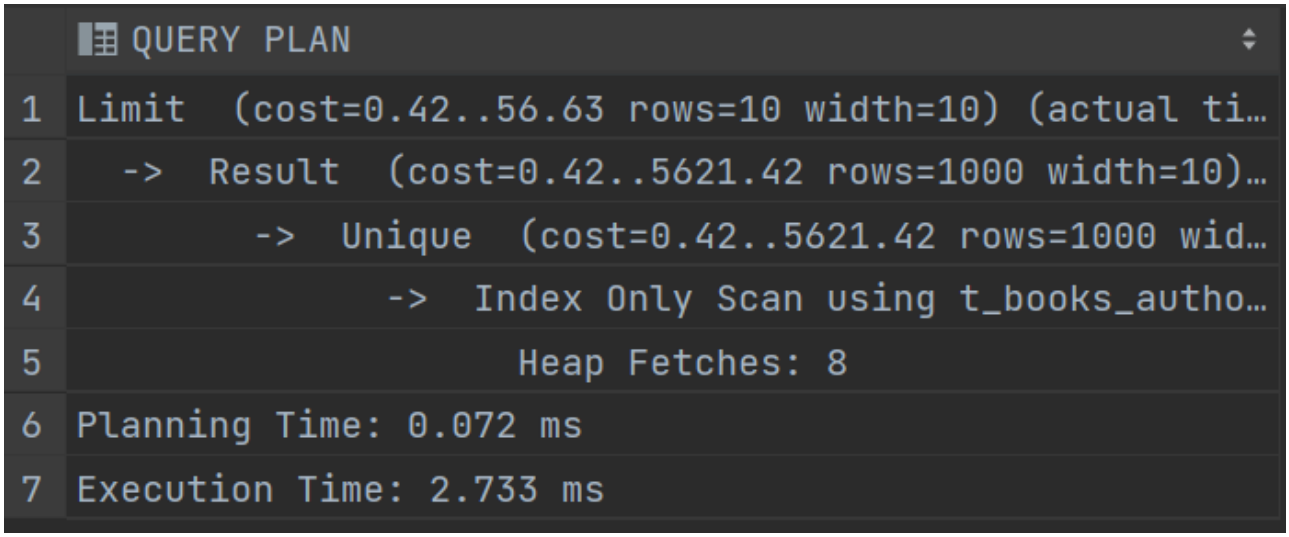
```
QUERY PLAN
1 HashAggregate  (cost=3474.00..3484.00 rows=1000 width…
2   Group Key: author
3   Batches: 1  Memory Usage: 193kB
4   ->  Seq Scan on t_books  (cost=0.00..2724.00 rows=1…
5 Planning Time: 0.964 ms
6 Execution Time: 94.101 ms
```

Объясните результат: индекс не используется, т.к. он не покрывает агрегацию, т.е. функцию `MAX(title)`, он был бы полезен для запросов с `... WHERE author = $1 AND title = $2`

18. Выберите первых 10 авторов:

```
EXPLAIN ANALYZE
SELECT DISTINCT author
FROM t_books
ORDER BY author
LIMIT 10;
```

План выполнения:



```
QUERY PLAN
1 Limit  (cost=0.42..56.63 rows=10 width=10) (actual ti…
2   ->  Result  (cost=0.42..5621.42 rows=1000 width=10)…
3         ->  Unique  (cost=0.42..5621.42 rows=1000 wid…
4               ->  Index Only Scan using t_books_autho…
5                     Heap Fetches: 8
6 Planning Time: 0.072 ms
7 Execution Time: 2.733 ms
```

Объясните результат: здесь индекс как раз помогает, т.к. предоставляет сортировку по парам `(author, title)`

19. Выполните поиск и сортировку:

```
EXPLAIN ANALYZE
SELECT author, title
FROM t_books
WHERE author LIKE 'T%'
ORDER BY author, title;
```

План выполнения:

```
QUERY PLAN
Sort  (cost=3099.29..3099.33 rows=15 width=21) (actual time=18.668..18.670 rows=1 loops=1)
  Sort Key: author, title
  Sort Method: quicksort  Memory: 25kB
  ->  Seq Scan on t_books  (cost=0.00..3099.00 rows=15 width=21) (actual time=18.652..18.654 rows=1 loops=1)
        Filter: ((author)::text ~~ 'T%'::text)
        Rows Removed by Filter: 149999
Planning Time: 1.809 ms
Execution Time: 18.695 ms
```

Объясните результат: здесь индекс не помогает, т.к. сначала отобрались строки, подходящие под условие `LIKE`

20. Добавьте новую книгу:

```
INSERT INTO t_books (book_id, title, author, category, is_active)
VALUES (150001, 'Cookbook', 'Mr. Hide', NULL, true);
COMMIT;
```

Результат:

```
workshop.public> INSERT INTO t_books (book_id, title, author, category, is_active)
                 VALUES (150001, 'Cookbook', 'Mr. Hide', NULL, true)
[2024-11-26 17:43:57] 1 row affected in 7 ms
workshop.public> COMMIT
[2024-11-26 17:43:57] [25P01] there is no transaction in progress
[2024-11-26 17:43:57] completed in 26 ms
```

21. Создайте индекс по категории:

```
CREATE INDEX t_books_cat_idx ON t_books(category);
```

Результат:

```
workshop.public> CREATE INDEX t_books_cat_idx ON t_books(category)
[2024-11-26 17:44:31] completed in 129 ms
```

22. Найдите книги без категории:

```
EXPLAIN ANALYZE
SELECT author, title
FROM t_books
WHERE category IS NULL;
```

План выполнения:

```
QUERY PLAN
Index Scan using t_books_cat_idx on t_books  (cost=0.29…
  Index Cond: (category IS NULL)
Planning Time: 0.257 ms
Execution Time: 0.043 ms
```

Объясните результат: использован индекс по категориям. Категорий всего 6 штук + `NULL`, поэтому индекс эффективно помогает ускорить выполнение запроса

23. Создайте частичные индексы:

```
DROP INDEX t_books_cat_idx;
CREATE INDEX t_books_cat_null_idx ON t_books(category) WHERE category IS
NULL;
```

Результат:

```
workshop.public> DROP INDEX t_books_cat_idx
[2024-11-26 17:47:22] completed in 6 ms
workshop.public> CREATE INDEX t_books_cat_null_idx ON t_books(category) WHERE category IS NULL
[2024-11-26 17:47:22] completed in 29 ms
```

24. Повторите запрос из шага 22:

```
EXPLAIN ANALYZE
SELECT author, title
FROM t_books
WHERE category IS NULL;
```

План выполнения:

```
QUERY PLAN
Execution Time: 0.027 ms
```

Объясните результат: частичный индекс хранит как раз все строки, которые находит запрос, и только их, поэтому эффективно просто возвращается индекс

25. Создайте частичный уникальный индекс:

```sql
CREATE UNIQUE INDEX t_books_selective_unique_idx
ON t_books(title)
WHERE category = 'Science';

-- Протестируйте его
INSERT INTO t_books (book_id, title, author, category, is_active)
VALUES (150002, 'Unique Science Book', 'Author 1', 'Science', true);

-- Попробуйте вставить дубликат
INSERT INTO t_books (book_id, title, author, category, is_active)
VALUES (150003, 'Unique Science Book', 'Author 2', 'Science', true);

-- Но можно вставить такое же название для другой категории
INSERT INTO t_books (book_id, title, author, category, is_active)
VALUES (150004, 'Unique Science Book', 'Author 3', 'History', true);
```

Результат:

```
workshop.public> CREATE UNIQUE INDEX t_books_selective_unique_idx
                     ON t_books(title)
                     WHERE category = 'Science'
[2024-11-26 17:51:14] completed in 113 ms
```

```
workshop.public> INSERT INTO t_books (book_id, title, author, category, is_active)
                   VALUES (150002, 'Unique Science Book', 'Author 1', 'Science', true)
[2024-11-26 17:51:57] 1 row affected in 6 ms
```

```
[23505] ERROR: duplicate key value violates unique constraint "t_books_selective_unique_idx"
Подробности: Key (title)=(Unique Science Book) already exists.
```

```
workshop.public> INSERT INTO t_books (book_id, title, author, category, is_active)
                   VALUES (150004, 'Unique Science Book', 'Author 3', 'History', true)
[2024-11-26 17:52:55] 1 row affected in 6 ms
```

Объясните результат: индекс обеспечивает уникальность значений только для строк, где `category = 'Science'`. Первая вставка прошла успешно, потому что книги с таким названием в категории `'Science'` еще не было. Дубликат нам вставить не дали, т.к. книга с таким названием в категории `'Science'` уже есть, и такая вставка нарушила бы уникальность. Третья вставка прошла успешно, т.к. несмотря на то, что книга с таким названием уже есть, мы вставляем книгу другой категории, а индекс нам этого не запрещает