Задание №10

Шалаев Алексей БПИ 222

https://github.com/AlexeyShalaev/hse-se-db/tree/master/hw-10

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
Seq Scan on t_books (cost=0.00..3099.00 rows=1 width=33) (actual time=8.386..8.387 rows=1 loops=1)
Filter: ((title)::text = 'Oracle Core'::text)
Rows Removed by Filter: 149999
Planning Time: 0.056 ms
Execution Time: 8.403 ms

Объясните результат:

- Запрос использует Seq Scan, перебирая все строки таблицы, так как индекс на столбце title отсутствует.
- Было обработано 150,000 строк, из которых только одна соответствовала условию.
- Последовательное сканирование заняло 8.4 мс, что относительно долго для выборки одной строки в большой таблице.
- Это происходит из-за большого объема данных и необходимости проверять каждую строку вручную.
- Создание индекса на title значительно ускорит выполнение запроса.

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:

```
postgres.public> CREATE INDEX t_books_title_idx ON t_books(title)
[2024-11-20 12:16:57] completed in 361 ms
postgres.public> CREATE INDEX t_books_active_idx ON t_books(is_active)
[2024-11-20 12:16:57] completed in 55 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
public	t_books	t_books_id_pk	CREATE UNIQUE INDEX t_books_id_pk ON public.t_books USING btree (book_id)
public	t_books	t_books_title_idx	CREATE INDEX t_books_title_idx ON public.t_books USING btree (title)
public	t_books	t_books_active_idx	CREATE INDEX t_books_active_idx ON public.t_books USING btree (is_active)

Объясните результат:

- Запрос возвращает список индексов таблицы t_books, включая их схему, название и структуру.
- Индекс t_books_id_pk — это уникальный индекс на book_id, автоматически созданный для первичного ключа.
- Индекс t_books_title_idx создан для ускорения поиска по столбцу title.
- Индекс t_books_active_idx создан для оптимизации запросов, использующих условие на is_active.
- Все индексы используют структуру B-tree, которая подходит для равенств, диапазонов и сортировок.

5. Обновите статистику таблицы:

```
ANALYZE t_books;
```

Результат:

```
postgres.public> ANALYZE t_books
[2024-11-20 12:19:01] completed in 175 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=0.037..0.038 rows=1 loops=1)
Index Cond: ((title)::text = 'Oracle Core'::text)
Planning Time: 0.220 ms
Execution Time: 0.049 ms

Объясните результат:

- Запрос использует Index Scan, благодаря созданному индексу t_books_title_idx на столбце title.
- Индекс позволил PostgreSQL напрямую найти нужное значение, избегая полного перебора строк таблицы.
- Стоимость выполнения запроса уменьшилась до cost=0.42..8.44, что значительно быстрее, чем последовательное сканирование.
- Время выполнения запроса существенно сократилось до 0.049 мс, так как индекс ускоряет выборку.
- Использование индекса подтверждает, что он эффективно оптимизировал поиск записи по title.

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=0.938..0.940 rows=1 loops=1)
Index Cond: (book_id = 18)
Planning Time: 0.084 ms
Execution Time: 0.956 ms

Объясните результат:

- Запрос использует Index Scan, благодаря уникальному индексу t_books_id_pk на столбце book_id, созданному автоматически для первичного ключа.

- Индекс позволил PostgreSQL быстро найти строку с book_id = 18, минуя полный перебор таблицы.
- План выполнения показывает низкую стоимость cost=0.42..8.44, так как индексная структура позволяет сразу обратиться к нужной записи.
- Время выполнения запроса составило 0.956 мс, что эффективно для таблицы с большим количеством записей.
- Использование индекса на первичный ключ подтверждает его оптимальность для точного поиска по уникальным значениям.

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
Seq Scan on t_books (cost=0.00..2724.00 rows=75045 width=33) (actual time=0.008..9.913 rows=75131 loops=1)
Filter: is_active
Rows Removed by Filter: 74869
Planning Time: 0.074 ms
Execution Time: 11.845 ms

Объясните результат:

- Запрос использует Seq Scan, несмотря на наличие индекса t_books_active_idx, потому что большое количество строк (75131) соответствует условию is_active = true.
- PostgreSQL решил, что последовательное сканирование эффективнее, так как при высоком проценте совпадений индексное сканирование может быть медленнее.
- Rows Removed by Filter: 74869 показывает, что почти половина строк была отфильтрована, но оставшиеся строки все же значительны по объему.
- Общее время выполнения запроса составило 11.845 мс, что выше, чем при использовании индекса для выборки небольшого количества записей.
- Индексы эффективнее для выборки редких значений, а для частых лучше подходят другие подходы, например, таблица разделения по активным и неактивным записям.

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
150000	150000	6	1003

10. Удалите созданные индексы:

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

Результат:

```
postgres.public> DROP INDEX t_books_title_idx
[2024-11-20 12:23:33] completed in 9 ms
postgres.public> DROP INDEX t_books_active_idx
[2024-11-20 12:23:33] completed in 4 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:

```
-- a. Оптимизация WHERE title = $1 AND category = $2
CREATE INDEX t_books_title_category_idx ON t_books(title, category);

-- b. Оптимизация WHERE title = $1
CREATE INDEX t_books_title_idx ON t_books(title);

-- c. Оптимизация WHERE category = $1 AND author = $2
CREATE INDEX t_books_category_author_idx ON t_books(category, author);

-- d. Оптимизация WHERE author = $1 AND book_id = $2
CREATE INDEX t_books_author_book_id_idx ON t_books(author, book_id);
```

Объясните ваше решение:

- **Запрос a (WHERE title = $1 AND category = $2):**
  Комбинированный индекс на (title, category) позволяет эффективно отфильтровать записи по обоим условиям одновременно, сокращая необходимость дополнительной фильтрации.
- **Запрос b (WHERE title = $1):**
  Индекс на title уже был создан ранее. Он оптимизирует запрос, так как позволяет сразу найти записи по значению title.
- **Запрос c (WHERE category = $1 AND author = $2):**
  Комбинированный индекс на (category, author) упорядочивает данные, ускоряя фильтрацию по обоим столбцам.
- **Запрос d (WHERE author = $1 AND book_id = $2):**
  Индекс на (author, book_id) позволяет PostgreSQL использовать упорядоченные данные для быстрого поиска по первому и второму столбцу в одном запросе.

Все индексы построены с учетом порядка столбцов, где первый столбец используется в качестве основного фильтра. Комбинированные индексы особенно эффективны для запросов с несколькими условиями.

12. Протестируйте созданные индексы.

Результаты тестов:

a. `WHERE title = $1 AND category = $2`

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

QUERY PLAN
Index Scan using t_books_category_author_idx on t_books (cost=0.29..8.23 rows=1 width=33) (actual time=0.021..0.021 rows=0 loops=1)
Index Cond: ((category)::text = 'Database'::text)
Filter: ((title)::text = 'Oracle Core'::text)
Planning Time: 0.309 ms
Execution Time: 0.035 ms

b. `WHERE title = $1`

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

QUERY PLAN

QUERY PLAN
Index Scan using t_books_title_idx on t_books (cost=0.42..8.44 rows=1 width=33) (actual time=0.090..0.091 rows=1 loops=1)
Index Cond: ((title)::text = 'Oracle Core'::text)
Planning Time: 0.079 ms
Execution Time: 0.104 ms

c. `WHERE category = $1 AND author = $2`

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE category = 'Databases' AND author = 'Tom Lane';
```

QUERY PLAN
Index Scan using t_books_category_author_idx on t_books (cost=0.29..8.31 rows=1 width=33) (actual time=0.023..0.023 rows=1 loops=1)
Index Cond: (((category)::text = 'Databases'::text) AND ((author)::text = 'Tom Lane'::text))
Planning Time: 0.078 ms
Execution Time: 0.037 ms

d. `WHERE author = $1 AND book_id = $2`

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE author = 'Tom Lane' AND book_id = 2025;
```

QUERY PLAN
Index Scan using t_books_author_book_id_idx on t_books (cost=0.42..8.44 rows=1 width=33) (actual time=0.025..0.026 rows=1 loops=1)
Index Cond: (((author)::text = 'Tom Lane'::text) AND (book_id = 2025))
Planning Time: 0.083 ms
Execution Time: 0.042 ms

Объясните результаты:

- Индексы значительно ускорили выполнение запросов, особенно когда порядок полей в индексе совпадает с условиями запроса.
- Запрос а продемонстрировал, что порядок полей в индексе критичен для оптимальной производительности.

- Индексы b, c, и d показали наилучшие результаты, так как их структура полностью соответствует запросам.
- Для дальнейшей оптимизации запроса a стоит создать индекс (title, category).

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..3099.00 rows=15 width=33) (actual time=68.579..68.580 rows=0 loops=1)
Filter: ((title)::text ~~* 'Relational%'::text)
Rows Removed by Filter: 150000
Planning Time: 0.111 ms
Execution Time: 68.595 ms

Объясните результат:

- **Тип сканирования:**
  Используется Seq Scan (последовательное сканирование), так как условие ILIKE 'Relational%' не может быть эффективно обработано с использованием существующих индексов.
- **Фильтрация:**
  Для каждой строки выполняется проверка на соответствие шаблону Relational%, что требует полного перебора всех строк таблицы (150,000 строк).
- **Время выполнения:**
  В результате последовательного сканирования и проверки каждой строки общее время выполнения составило 68.595 мс, что значительно больше, чем при использовании индекса.
- **Причина отсутствия индекса:**
  PostgreSQL не может использовать обычный B-tree индекс для условий ILIKE, так как они чувствительны к регистронезависимому сравнению.

14. Создайте функциональный индекс:

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

Результат:

```
postgres.public> CREATE INDEX t_books_up_title_idx ON t_books(UPPER(title))
[2024-11-20 12:47:18] completed in 343 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..3474.00 rows=750 width=33) (actual time=39.883..39.884 rows=0 loops=1)
Filter: (upper((title)::text) ~~ 'RELATIONAL%'::text)
Rows Removed by Filter: 150000
Planning Time: 0.289 ms
Execution Time: 39.899 ms

Объясните результат: Время уменьшилось в ~1.7 раза.

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..3099.00 rows=15 width=33) (actual time=63.776..63.778 rows=1 loops=1)
Filter: ((title)::text ~~* '%Core%'::text)
Rows Removed by Filter: 149999
Planning Time: 0.109 ms
Execution Time: 63.796 ms

Объясните результат: [Ваше объяснение]

17. Попробуйте удалить все индексы:

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

Результат: Остался один индекс: `t_books_id_pk`.

Объясните результат: Удалили все индексы, кроме PK.

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:

```
EXPLAIN ANALYZE
SELECT * FROM t_books WHERE reverse(title) LIKE reverse('%Core');
```

QUERY PLAN
Seq Scan on t_books (cost=0.00..3474.00 rows=750 width=33) (actual time=21.002..21.003 rows=1 loops=1)
Filter: (reverse((title)::text) ~~ 'eroC%'::text)
Rows Removed by Filter: 149999
Planning Time: 0.191 ms
Execution Time: 21.013 ms

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

QUERY PLAN
Bitmap Heap Scan on t_books (cost=21.57..76.77 rows=15 width=33) (actual time=0.015..0.016 rows=1 loops=1)
Recheck Cond: ((title)::text ~~* '%Core%'::text)
Heap Blocks: exact=1
-> Bitmap Index Scan on t_books_trgm_idx (cost=0.00..21.56 rows=15 width=0) (actual time=0.009..0.009 rows=1 loops=1)
Index Cond: ((title)::text ~~* '%Core%'::text)
Planning Time: 0.193 ms
Execution Time: 0.032 ms

Объясните результаты:

- **Индекс reverse():**
  Подходит для строгого суффиксного поиска (например, reverse(title) LIKE 'Core%'), но неудобен для произвольного поиска, так как требует вызова функции reverse().
- **Триграммный индекс (pg_trgm):**
  Универсальное решение для поиска подстрок в любом месте строки. Поддерживает запросы с % в начале, середине или конце шаблона. Рекомендуется использовать триграммный индекс (pg_trgm) для произвольного поиска, так как он значительно уменьшает время выполнения запросов.

19. Выполните поиск по точному совпадению:

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

План выполнения:

QUERY PLAN
Bitmap Heap Scan on t_books (cost=116.57..120.58 rows=1 width=33) (actual time=0.036..0.037 rows=1 loops=1)
Recheck Cond: ((title)::text = 'Oracle Core'::text)
Heap Blocks: exact=1
-> Bitmap Index Scan on t_books_trgm_idx (cost=0.00..116.57 rows=1 width=0) (actual time=0.025..0.025 rows=1 loops=1)
Index Cond: ((title)::text = 'Oracle Core'::text)
Planning Time: 0.124 ms

QUERY PLAN
Execution Time: 0.053 ms

Объясните результат:

- **Тип сканирования:**
  Используется Bitmap Heap Scan, что означает, что PostgreSQL сначала выполняет поиск в триграммном индексе t_books_trgm_idx, а затем считывает соответствующие блоки из таблицы.
- **Условие:**
  Запрос с точным совпадением title = 'Oracle Core' полностью покрывается триграммным индексом.
  Bitmap Index Scan позволяет PostgreSQL быстро найти позицию строк, соответствующих условию.
- **Эффективность:**
  Index Cond указывает, что поиск был оптимизирован за счет использования индекса.
  Общее время выполнения составило 0.053 мс, включая обработку индекса и извлечение строки.

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=0.028..0.028 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=0.018..0.018 rows=1 loops=1)
Index Cond: ((title)::text ~~* 'Relational%'::text)
Planning Time: 0.128 ms
Execution Time: 0.044 ms

Объясните результат:

- **Тип сканирования:**
  Используется Bitmap Heap Scan, что означает, что PostgreSQL нашел соответствующие строки с помощью триграммного индекса t_books_trgm_idx и затем извлек данные из таблицы.

- **Условие:**
  Условие ILIKE 'Relational%' обработано с использованием триграммного индекса, который эффективно поддерживает поиск строк с любым префиксом.

- **Эффективность:**
  Bitmap Index Scan на триграммном индексе позволил быстро найти потенциальные совпадения.
  Recheck Cond выполняет дополнительную проверку на соответствие, так как триграммный индекс допускает ложные срабатывания.
  Время выполнения составило всего 0.044 мс, что показывает высокую эффективность триграммного индекса для префиксного поиска.

- **Результат:**
  Несмотря на использование ILIKE, запрос полностью оптимизирован за счет триграммного индекса.

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 LIMIT 10;
```

План выполнения:

QUERY PLAN
Limit (cost=0.42..1.02 rows=10 width=33) (actual time=0.174..0.178 rows=10 loops=1)
-> Index Scan using t_books_desc_idx on t_books (cost=0.42..9062.40 rows=150000 width=33) (actual time=0.174..0.177 rows=10 loops=1)
Planning Time: 0.267 ms
Execution Time: 0.187 ms

Объясните результат:
Индексы с обратной сортировкой эффективны для запросов с ORDER BY ... DESC, особенно в сочетании с ограничением (LIMIT). Это особенно полезно для обработки больших таблиц, когда нужно быстро получить несколько записей, отсортированных в обратном порядке.

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:

```
postgres.public> ANALYZE t_books
[2024-11-20 13:17:30] completed in 180 ms
```

```
postgres.public> ANALYZE t_books_part
[2024-11-20 13:17:30] completed in 522 ms
```

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

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

План выполнения:

QUERY PLAN
Seq Scan on t_books_part_1 t_books_part (cost=0.00..1032.99 rows=1 width=32) (actual time=0.008..2.367 rows=1 loops=1)
Filter: (book_id = 18)
Rows Removed by Filter: 49998
Planning Time: 0.250 ms
Execution Time: 2.380 ms

Объясните результат: Из-за партицирования время исполнения запроса увеличилось в 2 раза.

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=2.812..8.918 rows=1 loops=1)
-> Seq Scan on t_books_part_1 (cost=0.00..1032.99 rows=1 width=32) (actual time=2.811..2.812 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..1033.00 rows=1 width=33) (actual time=2.716..2.716 rows=0 loops=1)
Filter: ((title)::text = 'Expert PostgreSQL Architecture'::text)
Rows Removed by Filter: 50000

QUERY PLAN
-> Seq Scan on t_books_part_3 (cost=0.00..1034.01 rows=1 width=34) (actual time=3.384..3.384 rows=0 loops=1)
Filter: ((title)::text = 'Expert PostgreSQL Architecture'::text)
Rows Removed by Filter: 50001
Planning Time: 0.170 ms
Execution Time: 8.936 ms

Объясните результат: Поиск по партициям занимает чуть больше времени.

6. Создайте партиционированный индекс:

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

Результат: Создан индекс.

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=0.033..0.081 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=0.032..0.033 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=0.030..0.030 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=0.016..0.016 rows=0 loops=1)
Index Cond: ((title)::text = 'Expert PostgreSQL Architecture'::text)
Planning Time: 0.582 ms
Execution Time: 0.106 ms

Объясните результат: Время выполнения значительно сократилось.

8. Удалите созданный индекс:

```
DROP INDEX t_books_part_title_idx;
```

Результат: Индекс удалился.

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: Создались индексы.

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=0.019..0.048 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=0.019..0.019 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=0.016..0.016 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: 0.308 ms
Execution Time: 0.064 ms

Объясните результат: Поиск стал еще быстрее.

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: Индексы удалились.

12. Создайте обычный индекс по book_id:

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

Результат: Индекс создался.

13. Выполните поиск по book_id:

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

План выполнения:

QUERY PLAN
Index Scan using t_books_part_1_book_id_idx on t_books_part_1 t_books_part (cost=0.29..8.31 rows=1 width=32) (actual time=0.011..0.012 rows=1 loops=1)
Index Cond: (book_id = 11011)
Planning Time: 0.237 ms
Execution Time: 0.024 ms

Объясните результат: Поиск ускорился.

14. Создайте индекс по полю is_active:

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

Результат: Индекс создался.

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
Bitmap Heap Scan on t_books (cost=844.41..2823.11 rows=75370 width=33) (actual time=1.811..8.115 rows=75152 loops=1)
Recheck Cond: is_active
Heap Blocks: exact=1225
-> Bitmap Index Scan on t_books_active_idx (cost=0.00..825.57 rows=75370 width=0) (actual time=1.696..1.697 rows=75152 loops=1)
Index Cond: (is_active = true)
Planning Time: 0.173 ms
Execution Time: 10.034 ms

Объясните результат:
Индекс на is_active позволяет значительно ускорить запросы с частыми условиями, избегая полного сканирования таблицы. Использование Bitmap Index Scan эффективно обрабатывает большие объемы данных, минимизируя затраты.

16. Создайте составной индекс:

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

Результат: Индекс создался.

17. Найдите максимальное название для каждого автора:

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

План выполнения:

QUERY PLAN
HashAggregate (cost=3475.00..3485.01 rows=1001 width=42) (actual time=49.888..49.990 rows=1003 loops=1)
Group Key: author

QUERY PLAN
Batches: 1 Memory Usage: 193kB
-> Seq Scan on t_books (cost=0.00..2725.00 rows=150000 width=21) (actual time=0.005..6.750 rows=150000 loops=1)
Planning Time: 0.217 ms
Execution Time: 50.043 ms

Объясните результат:
Запрос эффективно использует HashAggregate, но последовательное сканирование таблицы замедляет выполнение. Составной индекс на (author, title DESC) ускорит поиск максимального значения и группировку.

18. Выберите первых 10 авторов:

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

План выполнения:

QUERY PLAN
Limit (cost=0.42..56.61 rows=10 width=10) (actual time=0.108..0.335 rows=10 loops=1)
-> Result (cost=0.42..5625.42 rows=1001 width=10) (actual time=0.107..0.333 rows=10 loops=1)
-> Unique (cost=0.42..5625.42 rows=1001 width=10) (actual time=0.107..0.332 rows=10 loops=1)
-> Index Only Scan using t_books_author_title_index on t_books (cost=0.42..5250.42 rows=150000 width=10) (actual time=0.105..0.258 rows=1345 loops=1)
Heap Fetches: 4
Planning Time: 0.080 ms
Execution Time: 0.349 ms

Объясните результат:
Запрос эффективно использует **Index Only Scan** на индексе `t_books_author_title_index`, извлекая уникальные значения авторов без необходимости полного сканирования таблицы. Лимитирование до 10 строк ускоряет выполнение, обеспечивая время всего **0.349 мс**.

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=3100.29..3100.33 rows=15 width=21) (actual time=11.594..11.595 rows=1 loops=1)
Sort Key: author, title
Sort Method: quicksort Memory: 25kB
-> Seq Scan on t_books (cost=0.00..3100.00 rows=15 width=21) (actual time=11.585..11.586 rows=1 loops=1)
Filter: ((author)::text ~~ 'T%'::text)
Rows Removed by Filter: 149999
Planning Time: 0.121 ms
Execution Time: 11.610 ms

Объясните результат:
Запрос выполняет **Seq Scan** для фильтрации авторов, соответствующих шаблону `LIKE 'T%'`, что замедляет выполнение. Сортировка выполняется с использованием **quicksort**, но создание индекса на `(author, title)` может существенно ускорить фильтрацию и упорядочивание.

20. Добавьте новую книгу:

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

Результат:

```
postgres.public> INSERT INTO t_books (book_id, title, author, category,
is_active)
                 VALUES (150001, 'Cookbook', 'Mr. Hide', NULL, true)
[2024-11-20 13:37:28] 1 row affected in 6 ms
postgres.public> COMMIT
[2024-11-20 13:37:28] [25P01] there is no transaction in progress
[2024-11-20 13:37:28] completed in 7 ms
```

21. Создайте индекс по категории:

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

Результат: Индекс создался.

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..8.13 rows=1 width=21) (actual time=0.023..0.024 rows=1 loops=1)
Index Cond: (category IS NULL)
Planning Time: 0.262 ms
Execution Time: 0.035 ms

Объясните результат:
Запрос эффективно использует **Index Scan** на индексе `t_books_cat_idx`, чтобы быстро найти строки, где `category IS NULL`. Это минимизирует время выполнения до **0.035 мс**, избегая полного сканирования таблицы.

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: Индексы созданы.

24. Повторите запрос из шага 22:

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

План выполнения:

QUERY PLAN
Index Scan using t_books_cat_null_idx on t_books (cost=0.12..7.96 rows=1 width=21) (actual time=0.008..0.009 rows=1 loops=1)
Planning Time: 0.198 ms
Execution Time: 0.019 ms

Объясните результат:
Запрос использует частичный индекс `t_books_cat_null_idx`, созданный специально для строк с `category IS NULL`. Это уменьшило затраты и время выполнения до **0.019 мс**, сделав запрос более быстрым и эффективным.

25. Создайте частичный уникальный индекс:

```
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:

```
postgres.public> CREATE UNIQUE INDEX t_books_selective_unique_idx
                 ON t_books(title)
                 WHERE category = 'Science'
[2024-11-20 13:40:25] completed in 70 ms
postgres.public> INSERT INTO t_books (book_id, title, author, category,
is_active)
                    VALUES (150002, 'Unique Science Book', 'Author 1',
'Science', true)
[2024-11-20 13:40:25] 1 row affected in 4 ms
postgres.public> INSERT INTO t_books (book_id, title, author, category,
is_active)
                    VALUES (150003, 'Unique Science Book', 'Author 2',
'Science', true)
[2024-11-20 13:40:25] [23505] ERROR: duplicate key value violates unique
constraint "t_books_selective_unique_idx"
[2024-11-20 13:40:25] Подробности: Key (title)=(Unique Science Book) already
```

```
exists.
postgres.public> INSERT INTO t_books (book_id, title, author, category,
is_active)
                 VALUES (150004, 'Unique Science Book', 'Author 3',
'History', true)
[2024-11-20 13:41:14] 1 row affected in 4 ms
```

Объясните результат:

1. **Частичный уникальный индекс:**
   Индекс `t_books_selective_unique_idx` обеспечивает уникальность значений `title`, но только для строк, где `category = 'Science'`.

2. **Первая вставка:**
   Строка с `title = 'Unique Science Book'` и `category = 'Science'` была успешно добавлена, так как это первая запись, соответствующая условию индекса.

3. **Попытка дублирования:**
   Попытка вставить другую строку с таким же `title` и `category = 'Science'` завершилась ошибкой, так как частичный индекс запретил дублирование.

4. **Иная категория:**
   Вставка строки с тем же `title`, но с другой категорией (`category = 'History'`), была успешной, так как частичный индекс не распространяется на строки, не соответствующие условию `category = 'Science'`.

5. **Вывод:**
   Частичный уникальный индекс позволяет гибко ограничивать уникальность только для определенного подмножества данных, обеспечивая как целостность, так и гибкость.