Задание №11

Шалаев Алексей БПИ 222

https://github.com/AlexeyShalaev/hse-se-db/tree/master/hw-11

Задание 1: BRIN индексы и bitmap-сканирование

1. Удалите старую базу данных, если есть:

```
docker compose down
```

2. Поднимите базу данных из src/docker-compose.yml:

```
docker compose down && docker compose up -d
```

3. Обновите статистику:

```
ANALYZE t_books;
```

4. Создайте BRIN индекс по колонке category:

```
CREATE INDEX t_books_brin_cat_idx ON t_books USING brin(category);
```

5. Найдите книги с NULL значением category:

```
EXPLAIN ANALYZE
SELECT * 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..7.97 rows=1 width=33) (actual time=0.013..0.014 rows=1 loops=1)
Planning Time: 2.168 ms
Execution Time: 0.033 ms

Объясните результат:
BRIN индекс позволяет эффективно находить строки с `NULL`, ограничивая поиск только релевантными блоками данных. В результате запрос выполняется быстро, используя `Index Scan` с низкими затратами на обработку. Это демонстрирует эффективность BRIN индекса для слабо селективных и упорядоченных данных.

6. Создайте BRIN индекс по автору:

```
CREATE INDEX t_books_brin_author_idx ON t_books USING brin(author);
```

7. Выполните поиск по категории и автору:

```
EXPLAIN ANALYZE
SELECT * FROM t_books
WHERE category = 'INDEX' AND author = 'SYSTEM';
```

План выполнения:

QUERY PLAN
Bitmap Heap Scan on t_books (cost=17.94..251.76 rows=1 width=33) (actual time=0.980..0.981 rows=0 loops=1)
Recheck Cond: (((author)::text = 'SYSTEM'::text) AND ((category)::text = 'INDEX'::text))
-> BitmapAnd (cost=17.94..17.94 rows=73 width=0) (actual time=0.974..0.975 rows=0 loops=1)
-> Bitmap Index Scan on t_books_author_title_index (cost=0.00..5.54 rows=149 width=0) (actual time=0.973..0.973 rows=0 loops=1)
Index Cond: ((author)::text = 'SYSTEM'::text)
-> Bitmap Index Scan on t_books_brin_cat_idx (cost=0.00..12.16 rows=73530 width=0) (never executed)
Index Cond: ((category)::text = 'INDEX'::text)
Planning Time: 0.143 ms
Execution Time: 0.997 ms

Объясните результат (обратите внимание на bitmap scan):
Запрос использует **Bitmap Heap Scan**, который объединяет результаты двух **Bitmap Index Scan** операций: по `author` и `category`. BRIN индексы быстро находят релевантные блоки данных для каждого условия, а затем объединяются с помощью `BitmapAnd`. Это снижает количество сканируемых строк, но в данном случае строки, соответствующие обоим условиям, отсутствуют, поэтому итоговый результат пустой.

8. Получите список уникальных категорий:

```
EXPLAIN ANALYZE
SELECT DISTINCT category
FROM t_books
ORDER BY category;
```

План выполнения:

QUERY PLAN
Sort (cost=3100.18..3100.19 rows=6 width=7) (actual time=28.708..28.709 rows=7 loops=1)
Sort Key: category
Sort Method: quicksort Memory: 25kB
-> HashAggregate (cost=3100.04..3100.10 rows=6 width=7) (actual time=28.694..28.696 rows=7 loops=1)
Group Key: category
Batches: 1 Memory Usage: 24kB
-> Seq Scan on t_books (cost=0.00..2725.03 rows=150003 width=7) (actual time=0.007..7.065 rows=150003 loops=1)
Planning Time: 0.098 ms
Execution Time: 28.736 ms

Объясните результат:
Запрос выполняет последовательное сканирование (**Seq Scan**) таблицы `t_books`, чтобы собрать все строки и определить уникальные значения категории через **HashAggregate**. Затем результат сортируется с использованием **quicksort** для упорядочивания категорий. Несмотря на относительно большой объем данных (150003 строк), операция эффективна, так как агрегация и сортировка используют небольшую память (~25 kB).

9. Подсчитайте книги, где автор начинается на 'S':

```
EXPLAIN ANALYZE
SELECT COUNT(*)
FROM t_books
WHERE author LIKE 'S%';
```

План выполнения:

QUERY PLAN
Aggregate (cost=3100.08..3100.09 rows=1 width=8) (actual time=11.603..11.605 rows=1 loops=1)
-> Seq Scan on t_books (cost=0.00..3100.04 rows=15 width=0) (actual time=11.600..11.601 rows=0 loops=1)
Filter: ((author)::text ~~ 'S%'::text)
Rows Removed by Filter: 150003
Planning Time: 0.116 ms

QUERY PLAN
Execution Time: 11.622 ms

Объясните результат:
Запрос использует **Seq Scan** для последовательного чтения всей таблицы `t_books` (150003 строк) из-за отсутствия индекса, подходящего для фильтрации по условию `LIKE 'S%'`. Фильтр проверяет каждую строку на соответствие шаблону, удаляя нерелевантные строки. Поскольку строки с авторами, начинающимися на `S`, отсутствуют, итоговое значение равно нулю, а время выполнения составило ~11.6 мс.

10. Создайте индекс для регистронезависимого поиска:

```
CREATE INDEX t_books_lower_title_idx ON t_books(LOWER(title));
```

11. Подсчитайте книги, начинающиеся на 'O':

```
EXPLAIN ANALYZE
SELECT COUNT(*)
FROM t_books
WHERE LOWER(title) LIKE 'o%';
```

План выполнения:

QUERY PLAN
Aggregate (cost=3476.92..3476.93 rows=1 width=8) (actual time=43.390..43.392 rows=1 loops=1)
-> Seq Scan on t_books (cost=0.00..3475.05 rows=750 width=0) (actual time=43.382..43.385 rows=1 loops=1)
Filter: (lower((title)::text) ~~ 'o%'::text)
Rows Removed by Filter: 150002
Planning Time: 0.299 ms
Execution Time: 43.416 ms

Объясните результат:
Хотя был создан индекс `t_books_lower_title_idx`, запрос все еще использует **Seq Scan** вместо индекса. Это происходит потому, что функция `LOWER(title)` используется внутри фильтра `LIKE`, и PostgreSQL не может эффективно применить индекс для этого условия. В результате все 150003 строки последовательно сканируются, проверяются на соответствие шаблону `LIKE 'o%'`, что приводит к длительному времени выполнения (~43.4 мс). Индекс мог бы быть использован, если бы условие точно соответствовало формату индексации.

12. Удалите созданные индексы:

```
DROP INDEX t_books_brin_cat_idx;
DROP INDEX t_books_brin_author_idx;
DROP INDEX t_books_lower_title_idx;
```

13. Создайте составной BRIN индекс:

```
CREATE INDEX t_books_brin_cat_auth_idx ON t_books
USING brin(category, author);
```

14. Повторите запрос из шага 7:

```
EXPLAIN ANALYZE
SELECT * FROM t_books
WHERE category = 'INDEX' AND author = 'SYSTEM';
```

План выполнения:

QUERY PLAN
Bitmap Heap Scan on t_books (cost=5.54..428.26 rows=1 width=33) (actual time=0.012..0.012 rows=0 loops=1)
Recheck Cond: ((author)::text = 'SYSTEM'::text)
Filter: ((category)::text = 'INDEX'::text)
-> Bitmap Index Scan on t_books_author_title_index (cost=0.00..5.54 rows=149 width=0) (actual time=0.009..0.009 rows=0 loops=1)
Index Cond: ((author)::text = 'SYSTEM'::text)
Planning Time: 0.142 ms
Execution Time: 0.021 ms

Объясните результат:
Составной BRIN индекс `t_books_brin_cat_auth_idx` позволяет PostgreSQL эффективно сузить диапазон блоков для поиска по двум колонкам. Однако, в данном запросе используется **Bitmap Index Scan**, но только по колонке `author`, так как фильтр по `category` применяется на этапе **Filter**. Это связано с тем, что BRIN индексы работают с блоками, и точные соответствия зависят от порядка фильтров в запросе. В данном случае строк, удовлетворяющих обоим условиям, не найдено, что подтверждается быстрым временем выполнения (~0.021 мс).

Задание 2

1. Удалите старую базу данных, если есть:

```
docker compose down
```

2. Поднимите базу данных из src/docker-compose.yml:

```
docker compose down && docker compose up -d
```

3. Обновите статистику:

```
ANALYZE t_books;
```

4. Создайте полнотекстовый индекс:

```
CREATE INDEX t_books_fts_idx ON t_books
USING GIN (to_tsvector('english', title));
```

5. Найдите книги, содержащие слово 'expert':

```
EXPLAIN ANALYZE
SELECT * FROM t_books
WHERE to_tsvector('english', title) @@ to_tsquery('english', 'expert');
```

План выполнения:

QUERY PLAN
Bitmap Heap Scan on t_books (cost=21.03..1336.08 rows=750 width=33) (actual time=0.013..0.014 rows=1 loops=1)
Recheck Cond: (to_tsvector('english'::regconfig, (title)::text) @@ '''expert'''::tsquery)
Heap Blocks: exact=1
-> Bitmap Index Scan on t_books_fts_idx (cost=0.00..20.84 rows=750 width=0) (actual time=0.009..0.009 rows=1 loops=1)
Index Cond: (to_tsvector('english'::regconfig, (title)::text) @@ '''expert'''::tsquery)
Planning Time: 1.581 ms

QUERY PLAN

Execution Time: 0.032 ms

Объясните результат:
Создание полнотекстового индекса с использованием GIN позволяет PostgreSQL эффективно обрабатывать запросы с условиями на текстовые совпадения. Запрос использует **Bitmap Index Scan** по индексу `t_books_fts_idx`, чтобы найти релевантные строки, содержащие слово `'expert'`. Затем выполняется **Bitmap Heap Scan** для проверки соответствия условий. Благодаря индексации запрос выполняется быстро (0.032 мс), поскольку проверка текста производится непосредственно через индекс, а не через последовательное сканирование всей таблицы.

6. Удалите индекс:

```
DROP INDEX t_books_fts_idx;
```

7. Создайте таблицу lookup:

```
CREATE TABLE t_lookup (
     item_key VARCHAR(10) NOT NULL,
     item_value VARCHAR(100)
);
```

8. Добавьте первичный ключ:

```
ALTER TABLE t_lookup
ADD CONSTRAINT t_lookup_pk PRIMARY KEY (item_key);
```

9. Заполните данными:

```
INSERT INTO t_lookup
SELECT
     LPAD(CAST(generate_series(1, 150000) AS TEXT), 10, '0'),
     'Value_' || generate_series(1, 150000);
```

10. Создайте кластеризованную таблицу:

```
CREATE TABLE t_lookup_clustered (
     item_key VARCHAR(10) PRIMARY KEY,
     item_value VARCHAR(100)
);
```

11. Заполните её теми же данными:

```
INSERT INTO t_lookup_clustered
SELECT * FROM t_lookup;

CLUSTER t_lookup_clustered USING t_lookup_clustered_pkey;
```

12. Обновите статистику:

```
ANALYZE t_lookup;
ANALYZE t_lookup_clustered;
```

13. Выполните поиск по ключу в обычной таблице:

```
EXPLAIN ANALYZE
SELECT * FROM t_lookup WHERE item_key = '0000000455';
```

План выполнения:

QUERY PLAN
Index Scan using t_lookup_pk on t_lookup (cost=0.42..8.44 rows=1 width=23) (actual time=0.014..0.014 rows=1 loops=1)
Index Cond: ((item_key)::text = '0000000455'::text)
Planning Time: 0.089 ms
Execution Time: 0.025 ms

Объясните результат:
Запрос использует **Index Scan**, так как по колонке `item_key` задан первичный ключ, автоматически создающий B-Tree индекс. Индекс позволяет PostgreSQL эффективно находить строку с ключом `'0000000455'` без необходимости полного сканирования таблицы. Быстрое время выполнения (0.025 мс) обусловлено прямым обращением к индексу и минимальным объемом данных для проверки условия.

14. Выполните поиск по ключу в кластеризованной таблице:

```
EXPLAIN ANALYZE
SELECT * FROM t_lookup_clustered WHERE item_key = '0000000455';
```

План выполнения:

QUERY PLAN
Index Scan using t_lookup_clustered_pkey on t_lookup_clustered (cost=0.42..8.44 rows=1 width=23) (actual time=0.018..0.018 rows=1 loops=1)
Index Cond: ((item_key)::text = '0000000455'::text)
Planning Time: 0.147 ms
Execution Time: 0.030 ms

Объясните результат:
Кластеризация таблицы упорядочила данные в соответствии с индексом `t_lookup_clustered_pkey`, что минимизирует фрагментацию. Однако, запрос все равно использует **Index Scan**, так как поиск происходит по ключу `item_key`. В данном случае кластеризация не дает значительного выигрыша в производительности, так как индексация уже обеспечивает быстрый доступ. Время выполнения (0.030 мс) немного больше, чем для обычной таблицы, из-за дополнительных накладных расходов на упорядоченные данные.

15. Создайте индекс по значению для обычной таблицы:

```
CREATE INDEX t_lookup_value_idx ON t_lookup(item_value);
```

16. Создайте индекс по значению для кластеризованной таблицы:

```
CREATE INDEX t_lookup_clustered_value_idx
ON t_lookup_clustered(item_value);
```

17. Выполните поиск по значению в обычной таблице:

```
EXPLAIN ANALYZE
SELECT * FROM t_lookup WHERE item_value = 'T_BOOKS';
```

План выполнения:

QUERY PLAN
Index Scan using t_lookup_value_idx on t_lookup (cost=0.42..8.44 rows=1 width=23) (actual time=0.035..0.036 rows=0 loops=1)
Index Cond: ((item_value)::text = 'T_BOOKS'::text)
Planning Time: 0.289 ms
Execution Time: 0.055 ms

Объясните результат:
Запрос использует **Index Scan** по индексу `t_lookup_value_idx`, который создан для ускорения поиска по колонке `item_value`. Индекс позволяет быстро проверить строки на соответствие значению `'T_BOOKS'`, минуя полное сканирование таблицы. Однако в данном случае результирующих строк нет (rows=0), так как указанное значение отсутствует в данных. Время выполнения (0.055 мс) остается небольшим благодаря индексации.

18. Выполните поиск по значению в кластеризованной таблице:

```
EXPLAIN ANALYZE
SELECT * FROM t_lookup_clustered WHERE item_value = 'T_BOOKS';
```

План выполнения:

QUERY PLAN
Index Scan using t_lookup_clustered_value_idx on t_lookup_clustered (cost=0.42..8.44 rows=1 width=23) (actual time=0.031..0.032 rows=0 loops=1)
Index Cond: ((item_value)::text = 'T_BOOKS'::text)
Planning Time: 0.173 ms
Execution Time: 0.043 ms

Объясните результат:
Запрос использует **Index Scan** по индексу `t_lookup_clustered_value_idx`, специально созданному для поиска по колонке `item_value`. Индекс позволяет PostgreSQL быстро определить отсутствие строки с указанным значением `'T_BOOKS'`, избегая полного сканирования таблицы. Благодаря индексации время выполнения остается минимальным (0.043 мс). Кластеризация таблицы не влияет на этот запрос, так как порядок данных не связан с колонкой `item_value`.

19. Сравните производительность поиска по значению в обычной и кластеризованной таблицах:

Сравнение производительности:

1. **Время выполнения**:
    - Обычная таблица: **0.055 мс**.
    - Кластеризованная таблица: **0.043 мс**.
    - Кластеризованная таблица немного быстрее (~20%), что может быть связано с более компактным расположением данных из-за кластеризации, уменьшившим количество операций ввода-вывода.
2. **План выполнения**:
    - В обоих случаях использовался **Index Scan**, поскольку индексы по `item_value` оптимизируют поиск по значению.
3. **Влияние кластеризации**:

- Кластеризация сама по себе не улучшает производительность для колонок, не связанных с кластеризующим индексом (`item_key`), но может незначительно повлиять на общую производительность за счет улучшенного порядка хранения данных.

Итог:

Разница в производительности минимальна, так как поиск выполняется через индекс в обоих случаях. Однако кластеризация может быть полезна для запросов, которые учитывают упорядоченность данных по кластеризующему индексу.

Задание 3

1. Создайте таблицу с большим количеством данных:

```
CREATE TABLE test_cluster AS
SELECT
    generate_series(1,1000000) as id,
    CASE WHEN random() < 0.5 THEN 'A' ELSE 'B' END as category,
    md5(random()::text) as data;
```

2. Создайте индекс:

```
CREATE INDEX test_cluster_cat_idx ON test_cluster(category);
```

3. Измерьте производительность до кластеризации:

```
EXPLAIN ANALYZE
SELECT * FROM test_cluster WHERE category = 'A';
```

План выполнения:

QUERY PLAN
Bitmap Heap Scan on test_cluster (cost=59.17..7696.73 rows=5000 width=68) (actual time=13.424..94.731 rows=500395 loops=1)
Recheck Cond: (category = 'A'::text)
Heap Blocks: exact=8334
-> Bitmap Index Scan on test_cluster_cat_idx (cost=0.00..57.92 rows=5000 width=0) (actual time=12.302..12.303 rows=500395 loops=1)
Index Cond: (category = 'A'::text)
Planning Time: 0.190 ms
Execution Time: 107.584 ms

Объясните результат:
Запрос использует **Bitmap Index Scan** для нахождения блоков с `category = 'A'` и **Bitmap Heap Scan** для извлечения данных, что эффективно благодаря индексу. Однако из-за случайного распределения строк требуется обработать много блоков (8334), что увеличивает время выполнения до ~107 мс. Кластеризация могла бы уменьшить фрагментацию и улучшить производительность.

4. Выполните кластеризацию:

```
CLUSTER test_cluster USING test_cluster_cat_idx;
```

Результат:

```
postgres.public> CLUSTER test_cluster USING test_cluster_cat_idx
[2024-12-04 15:44:29] completed in 763 ms
```

5. Измерьте производительность после кластеризации:

```
EXPLAIN ANALYZE
SELECT * FROM test_cluster WHERE category = 'A';
```

План выполнения:

QUERY PLAN
Bitmap Heap Scan on test_cluster (cost=5565.48..20139.89 rows=499233 width=39) (actual time=14.496..75.043 rows=500395 loops=1)
Recheck Cond: (category = 'A'::text)
Heap Blocks: exact=4170
-> Bitmap Index Scan on test_cluster_cat_idx (cost=0.00..5440.67 rows=499233 width=0) (actual time=13.966..13.967 rows=500395 loops=1)
Index Cond: (category = 'A'::text)
Planning Time: 0.286 ms
Execution Time: 88.820 ms

Объясните результат:
Кластеризация упорядочила строки по индексу `test_cluster_cat_idx`, что уменьшило фрагментацию данных. После кластеризации количество блоков для обработки снизилось с 8334 до 4170, что ускорило чтение данных. В результате общее время выполнения сократилось до ~88 мс.

6. Сравните производительность до и после кластеризации:

Сравнение производительности:

1. **До кластеризации**:
   - Время выполнения: **107.584 мс**.
   - Количество обработанных блоков: **8334**.
2. **После кластеризации**:

- Время выполнения: **88.820 мс** (ускорение ~17%).
- Количество обработанных блоков: **4170** (уменьшение почти в 2 раза).

3. **Причина улучшения**:

   - После кластеризации строки с `category = 'A'` стали расположены последовательно, что уменьшило количество операций ввода-вывода для чтения данных.

Итог:

Кластеризация значительно уменьшает фрагментацию данных, улучшая производительность запросов, особенно при поиске по индексированным колонкам.