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
Bitmap Heap Scan on t_books  (cost=12.00..16.01 rows=1 width=33) (actual time=0.008..0.008 rows=0 loops=1)
  Recheck Cond: (category IS NULL)
  ->  Bitmap Index Scan on t_books_brin_cat_idx  (cost=0.00..12.00 rows=1 width=0) (actual time=0.007..0.007 rows=0 loops=1)
        Index Cond: (category IS NULL)
Planning Time: 0.339 ms
Execution Time: 0.026 ms
```

Объясните результат: использование `Bitmap Heap Scan` в начале говорит о том, что сначала индекс не был использован, а после речека стал использоваться, т.к. теперь там `Bitmap Index Scan`

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=12.16..2336.73 rows=1 width=33) (actual time=15.618..15.620 rows=0 loops=1)
  Recheck Cond: ((category)::text = 'INDEX'::text)
  Rows Removed by Index Recheck: 150000
  Filter: ((author)::text = 'SYSTEM'::text)
  Heap Blocks: lossy=1225
  ->  Bitmap Index Scan on t_books_brin_cat_idx  (cost=0.00..12.16 rows=73305 width=0) (actual time=0.056..0.057 rows=12250 loops=1)
        Index Cond: ((category)::text = 'INDEX'::text)
Planning Time: 0.361 ms
Execution Time: 15.678 ms
```

Объясните результат (обратите внимание на bitmap scan): в начале используется `Bitmap Heap Scan`, видимо потому что движок, решил, что это оптимальнее, чем использовать индекс, созданных в п. 6. После того, как он отфильтровал значения по `author`, стал использовать `Bitmap Index Scan`, т.е. категорию искал уже по индексу

8. Получите список уникальных категорий:

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

План выполнения:

```
QUERY PLAN
Sort  (cost=3100.11..3100.12 rows=5 width=7) (actual time=48.396..48.398 rows=6 loops=1)
  Sort Key: category
  Sort Method: quicksort  Memory: 25kB
  ->  HashAggregate  (cost=3100.00..3100.05 rows=5 width=7) (actual time=48.382..48.385 rows=6 loops=1)
        Group Key: category
        Batches: 1  Memory Usage: 24kB
        ->  Seq Scan on t_books  (cost=0.00..2725.00 rows=150000 width=7) (actual time=0.004..12.721 rows=150000 loops=1)
Planning Time: 0.157 ms
Execution Time: 48.454 ms
```

Объясните результат: индекс не отсортировал данные, поэтому пришлось тратить время на сортировку (т.к. запрос содержит `DISTINCT`), а затем использовался самый простой `Seq Scan`, потому что после сортировке это быстрее

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=3100.03..3100.05 rows=1 width=8) (actual time=12.951..12.953 rows=1 loops=1)
  ->  Seq Scan on t_books  (cost=0.00..3100.00 rows=14 width=0) (actual time=12.947..12.948 rows=0 loops=1)
        Filter: ((author)::text ~~ 'S%'::text)
        Rows Removed by Filter: 150000
Planning Time: 0.167 ms
Execution Time: 12.982 ms
```

Объясните результат: использован `Seq Scan`, т.к. созданный индекс не позволял выполнить поиск строк с нужным суффиксом

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=3476.88..3476.89 rows=1 width=8) (actual time=45.486..45.487 rows=1 loops=1)
  ->  Seq Scan on t_books  (cost=0.00..3475.00 rows=750 width=0) (actual time=45.477..45.480 rows=1 loops=1)
        Filter: (lower((title)::text) ~~ 'o%'::text)
        Rows Removed by Filter: 149999
Planning Time: 0.371 ms
Execution Time: 45.527 ms
```

Объясните результат: ситуация аналогична п. 9, движок выполняет последовательное сканирование

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=12.16..2336.73 rows=1 width=33) (actual time=1.283..1.284 rows=0 loops=1)
  Recheck Cond: (((category)::text = 'INDEX'::text) AND ((author)::text = 'SYSTEM'::text))
  Rows Removed by Index Recheck: 8844
  Heap Blocks: lossy=73
  ->  Bitmap Index Scan on t_books_brin_cat_auth_idx  (cost=0.00..12.16 rows=73305 width=0) (actual time=0.023..0.024 rows=730 loops=1)
        Index Cond: (((category)::text = 'INDEX'::text) AND ((author)::text = 'SYSTEM'::text))
Planning Time: 0.198 ms
Execution Time: 1.311 ms
```

Объясните результат: в итоге используется `Bitmap Index Scan`, потому что в п. 13 был создан составной индекс по интересующим нас колонкам

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=0.019..0.020 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=0.015..0.015 rows=1 loops=1)
        Index Cond: (to_tsvector('english'::regconfig, (title)::text) @@ '''expert'''::tsquery)
Planning Time: 0.329 ms
Execution Time: 0.049 ms
```

Объясните результат: запрос эффективно использует GIN-индекс `t_books_fts_idx` для поиска, минимизируя операции с таблицей

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=0.012..0.013 rows=1 loops=1)
  Index Cond: ((item_key)::text = '0000000455'::text)
Planning Time: 0.085 ms
Execution Time: 0.023 ms
```

Объясните результат: использован `Index Scan` по первичному ключу

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=1.967..1.969 rows=1 loops=1)
  Index Cond: ((item_key)::text = '0000000455'::text)
Planning Time: 0.208 ms
Execution Time: 1.995 ms
```

Объясните результат: использован `Index Scan` по первичному ключу

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=0.016..0.017 rows=0 loops=1)
  Index Cond: ((item_value)::text = 'T_BOOKS'::text)
Planning Time: 0.153 ms
Execution Time: 0.026 ms
```

Объясните результат: использован `Index Scan` по созданному в п.15 индексу

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=0.019..0.019 rows=0 loops=1)
  Index Cond: ((item_value)::text = 'T_BOOKS'::text)
Planning Time: 0.175 ms
Execution Time: 0.034 ms
```

Объясните результат: использован `Index Scan` по созданному в п.16 индексу

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

Сравнение: поиск по значению в обоих случаях очень быстрый. В случае обычной таблицы время планирования меньше на 13 процентов, время исполнения - меньше на 23 процента, что является приличной разницей

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;
```

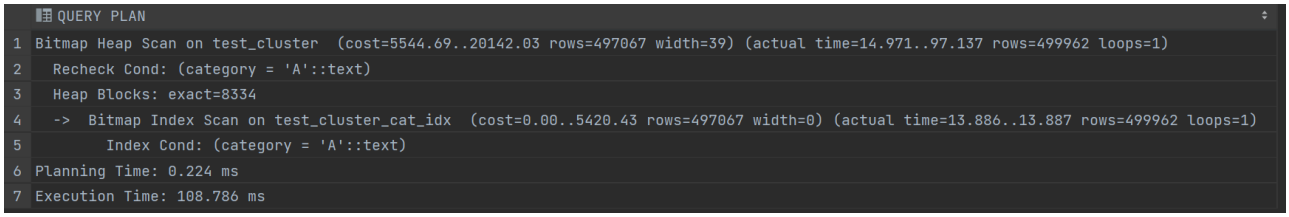
2. Создайте индекс:

```
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:



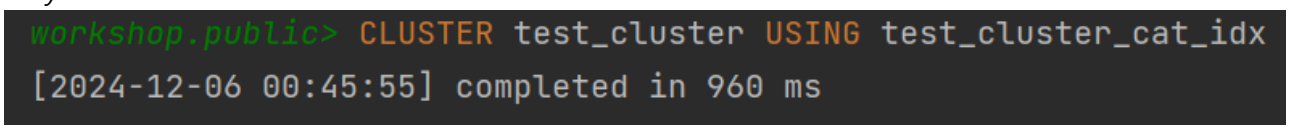
```
QUERY PLAN
Bitmap Heap Scan on test_cluster  (cost=5544.69..20142.03 rows=497067 width=39) (actual time=14.971..97.137 rows=499962 loops=1)
  Recheck Cond: (category = 'A'::text)
  Heap Blocks: exact=8334
  ->  Bitmap Index Scan on test_cluster_cat_idx  (cost=0.00..5420.43 rows=497067 width=0) (actual time=13.886..13.887 rows=499962 loops=1)
        Index Cond: (category = 'A'::text)
Planning Time: 0.224 ms
Execution Time: 108.786 ms
```

Объясните результат: используется `Bitmap Index Scan` по созданному в п. 2 индексу. Время выполнения (аж?) 108.768мс

4. Выполните кластеризацию:

```
CLUSTER test_cluster USING test_cluster_cat_idx;
```

Результат:



```
workshop.public> CLUSTER test_cluster USING test_cluster_cat_idx
[2024-12-06 00:45:55] completed in 960 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=5544.69..20092.03 rows=497067 width=39) (actual time=13.088..63.384 rows=499962 loops=1)
  Recheck Cond: (category = 'A'::text)
  Heap Blocks: exact=4167
  ->  Bitmap Index Scan on test_cluster_cat_idx  (cost=0.00..5420.43 rows=497067 width=0) (actual time=12.599..12.599 rows=499962 loops=1)
        Index Cond: (category = 'A'::text)
Planning Time: 0.208 ms
Execution Time: 73.767 ms
```

Объясните результат: использован `Bitmap Index Scan` по созданному тому же индексу. Теперь время выполнения 73.767мс

6. Сравните производительность до и после кластеризации:

Сравнение: после кластеризации время планирования стало совсем чуть-чуть меньше, а вот время выполнения сократилось на целых 30 процентов. Это демонстрирует, что кластеризация действительно дает серьезный прирост в производительности