DataBase Assignment

practice #5

정보대학 컴퓨터학과 2017320108 고재영

Exercise #1.

1. Make (and execute) three queries each of which uses seq scan index scan and index only scan respectively. (use the 'explain analyze')

```
postgres=# ₩d table1;
                  "public.table1" 테이블
  필드명  |     종류      | Collation | NULL허용 | 초기값
----------+---------------+-----------+----------+--------
 sorted   | integer       |           |          |
 unsorted | integer       |           |          |
 rndm     | integer       |           |          |
 dummy    | character(40) |           |          |
인덱스들:
    "sort_index" btree (sorted)
    "unsort_index" btree (unsorted)
```

위를 보면 알 수 있듯이, sort_index와 unsort_index, 두 개의 index를 생성해 놓았다.

(1) sequential scan한 query

```
postgres=# explain analyze
postgres-# select *
postgres-# from table1;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Seq Scan on table1  (cost=0.00..203093.00 rows=10000000 width=53) (actual time=0.027..2603.691 rows=10000000 loops=1)
 Planning Time: 0.086 ms
 Execution Time: 3156.538 ms
(3개 행)
```

```
postgres=# explain analyze
postgres-# select *
postgres-# from table1
postgres-# where rndm < 1500 and sorted < 200;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Index Scan using sort_index on table1  (cost=0.43..46.46 rows=14 width=53) (actual time=0.026..0.302 rows=17 loops=1)
   Index Cond: (sorted < 200)
   Filter: (rndm < 1500)
   Rows Removed by Filter: 983
 Planning Time: 0.218 ms
 Execution Time: 0.328 ms
(6개 행)
```

```
postgres=# explain analyze
postgres-# select sorted
postgres-# from table1
postgres-# where sorted < 1000;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
--------
 Index Only Scan using sort_index on table1  (cost=0.43..199.04 rows=5006 width=4) (actual time=0.018..2.463 rows=5000 l
oops=1)
   Index Cond: (sorted < 1000)
   Heap Fetches: 5000
 Planning Time: 0.183 ms
 Execution Time: 2.726 ms
(5개 행)
```

2. Make two queries using clustered index and non clustered index. Compare their execution times.

SQL Shell (psql)

```
postgres=# select *
postgres-# from table1;
 sorted  | unsorted | rndm  |                    dummy
---------+----------+-------+-----------------------------------------
      0  |  1491963 | 16948 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      0  |   880963 | 36170 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      0  |  1978898 | 53354 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      0  |   578184 | 52251 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      0  |   796195 | 96338 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      1  |   162073 | 83176 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      1  |  1544691 | 61606 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      1  |  1980901 | 12320 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      1  |   996608 | 51659 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      1  |   427610 | 16195 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      2  |    90777 | 41470 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
      2  |   406856 | 69077 | 'abcdefghijklmnopqrstuvwxyzabcdefgh'
```

위를 보면 알 수 있듯이, table1의 데이터는 sorted란 attribute에 대해 정렬되어 있음을 알 수 있다. 사전에 sort_index는 sorted란 attribute에 대해서 clustered 되어 있기 때문에,

```
postgres=# explain analyze
postgres-# select sorted, rndm
postgres-# from table1
postgres-# where rndm < 2000;
                                                          QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Seq Scan on table1  (cost=0.00..228093.00 rows=192665 width=8) (actual time=0.029..2774.397 rows=200139 loops=1)
   Filter: (rndm < 2000)
   Rows Removed by Filter: 9799861
 Planning Time: 0.105 ms
 Execution Time: 2784.738 ms
(5개 행)
```

위와 같은 query문으로 clustered index를 이용한 작업을 할 수 있다.

```
postgres=# explain analyze
postgres-# select unsorted, rndm
postgres-# from table1
postgres-# where rndm < 2000;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on table1  (cost=0.00..228093.00 rows=192665 width=8) (actual time=0.030..2776.042 rows=200139 loops=1)
   Filter: (rndm < 2000)
   Rows Removed by Filter: 9799861
 Planning Time: 0.115 ms
 Execution Time: 2786.359 ms
(5개 행)
```

마찬가지로 위의 query는 unsorted를 이용함으로, non-clustered index를 이용한 query문임을 알 수 있다.

query plan에서 볼 수 있듯이, execution time을 비교해보면 clustered index를 이용한 전자가 약 2ms 더 빠른 것을 알 수 있다. 이는 sequential scan에 있어서 clustered index를 이용하는 것이 non-clustered, 즉 secondary index를 이용하는 것보다 efficient하다는 것을 알 수 있다.

3. Execute and compare the following two queries. Explain why their query plans are different.

- SELECT sorted , rndm FROM table1 where sorted > 1999231 and rndm = 1005;
- SELECT sorted , rndm FROM table1 where sorted < 1999231 and rndm = 1005;

```
postgres=# explain analyze
postgres-# select sorted, rndm
postgres-# from table1
postgres-# where sorted > 1999231 and rndm = 1005;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Index Scan using sort_index on table1  (cost=0.43..161.13 rows=1 width=8) (actual time=1.627..1.627 rows=0 loops=1)
   Index Cond: (sorted > 1999231)
   Filter: (rndm = 1005)
   Rows Removed by Filter: 3840
 Planning Time: 0.209 ms
 Execution Time: 1.686 ms
(6개 행)


postgres=# explain analyze
postgres-# select sorted, rndm
postgres-# from table1
postgres-# where sorted < 1999231 and rndm = 1005;
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on table1  (cost=0.00..253093.00 rows=103 width=8) (actual time=30.523..2956.493 rows=114 loops=1)
   Filter: ((sorted < 1999231) AND (rndm = 1005))
   Rows Removed by Filter: 9999886
 Planning Time: 0.176 ms
 Execution Time: 2956.561 ms
(5개 행)
```

일단 두 가지를 비교해보자면, 전자의 경우에는 index scan을 이용한 반면에 후자의 경우에는 sequential scan을 이용했다. execution time을 비교한 결과에도, 자릿수가 몇 개 이상이 차이나는 상이한 차이점을 보인다.

이러한 차이점이 발생하는 이유는 바로 다음 query를 통해 찾을 수 있었다.

```
postgres=# select count(*)
postgres-# from table1
postgres-# where sorted > 1999231;
 count
-------
  3840
(1개 행)


postgres=# select count(*)
postgres-# from table1;
  count
----------
 10000000
(1개 행)


postgres=# select count(*)
postgres-# from table1
postgres-# where sorted < 1999231;
  count
---------
 9996155
(1개 행)
```

사실 이렇게 count로 찾을 필요없이, 이전 두 query에서 [rows removed by filter] 결과를 참고하면 알 수 있지만, 전자의 경우에는 전체 rows 중에서 1%보다도 미만의 데이터를 탐색하기에 index scan이 훨씬 빠른 반면, 후자의 경우는 사실상 전체 rows를 읽는 게 효율적이므로 sequential scan을 수행하게 된다.

Exercise #2.

1. Create two indexes

• Create indexes on attribute “recordid” in “table_btree” and “table_hash”

• Create “b-tree” in “table_btree.recordid” column

• Create “hash index” in “table_hash.recordid” column

• Type “₩h create index” for detailed index creation syntax

• Use a method name “btree” for creating b-tree and “hash” for creating hash index

```
hw5=# ₩d table_btree
               "public.table_btree" 테이블
  필드명   |     종류      | Collation | NULL허용 | 초기값
----------+---------------+-----------+----------+--------
 recordid | integer       |           |          |
 rndm     | integer       |           |          |
 dummy    | character(40) |           |          |
인덱스들:
    "btr_index" btree (recordid)


hw5=# ₩d table_hash
                "public.table_hash" 테이블
  필드명   |     종류      | Collation | NULL허용 | 초기값
----------+---------------+-----------+----------+--------
 recordid | integer       |           |          |
 rndm     | integer       |           |          |
 dummy    | character(40) |           |          |
인덱스들:
    "hsh_index" btree (recordid)
```

인덱스 이름을 각각 btr_index와 hsh_index라고 지정함.

2. Run two queries. And compare the query execution plan and total execution time

- Select * from table_btree where recordid = 10001;
- Select * from table_hash where recordid = 10001;

```
hw5=# explain analyze
hw5-# select *
hw5-# from table_btree
hw5-# where recordid = 10001;
                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Index Scan using btr_index on table_btree  (cost=0.43..8.45 rows=1 width=49) (actual time=3.854..3.856 rows=1 loops=1)
   Index Cond: (recordid = 10001)
 Planning Time: 5.880 ms
 Execution Time: 4.336 ms
(4개 행)


hw5=# explain analyze
hw5-# select *
hw5-# from table_hash
hw5-# where recordid = 10001;
                                                  QUERY PLAN
----------------------------------------------------------------------------------------------------------------------
 Index Scan using hsh_index on table_hash  (cost=0.43..8.45 rows=1 width=49) (actual time=3.630..3.632 rows=1 loops=1)
   Index Cond: (recordid = 10001)
 Planning Time: 5.205 ms
 Execution Time: 3.647 ms
(4개 행)
```

btree는 본질적으로 ordered index방식인 반면, hash는 ordered sequence와 상관없기 때문에, 검색시간에 있어 btree가 더 빠르다.

3. Run two queries. And compare the query execution plan and total execution time

- Select * from table_btree where recordid > 250 and recordid < 550;

• Select * from table_hash where recordid > 250 and recordid < 550;

```
hw5=# explain analyze
hw5-# select *
hw5-# from table_btree
hw5-# where recordid > 250 and recordid < 550;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using btr_index on table_btree  (cost=0.43..17.41 rows=299 width=49) (actual time=0.113..2.554 rows=299 loops=1)
   Index Cond: ((recordid > 250) AND (recordid < 550))
 Planning Time: 3.416 ms
 Execution Time: 2.602 ms
(4개 행)


hw5=# explain analyze
hw5-# select *
hw5-# from table_hash
hw5-# where recordid > 250 and recordid < 550;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using hsh_index on table_hash  (cost=0.43..17.47 rows=302 width=49) (actual time=0.138..2.038 rows=299 loops=1)
   Index Cond: ((recordid > 250) AND (recordid < 550))
 Planning Time: 3.953 ms
 Execution Time: 2.084 ms
(4개 행)
```

btree는 본질적으로 ordered index방식인 반면, hash는 ordered sequence와 상관없기 때문에, 검색시간에 있어 btree가 더 빠르다.

4. Update a single “recordid” field in “table_btree”. And update a single “recordid” field in “table_noindex”. Then find a difference

Hint) Update recordid 9,999,997 to 9,999,998

```
hw5=# update table_btree
hw5-# set recordid = 9999997
hw5-# where recordid = 9999998;
UPDATE 1
hw5=# update table_noindex
hw5-# set recordid = 9999997
hw5-# where recordid = 9999998;
UPDATE 1
```

```
hw5=# explain analyze
hw5-# select *
hw5-# from table_btree
hw5-# where recordid > 100 and recordid < 800;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using btr_index on table_btree  (cost=0.56..69.44 rows=1044 width=49) (actual time=0.413..3.416 rows=635 loops=1)
   Index Cond: ((recordid > 100) AND (recordid < 800))
 Planning Time: 0.243 ms
 Execution Time: 3.483 ms
(4개 행)


hw5=# explain analyze
hw5-# select *
hw5-# from table_noindex
hw5-# where recordid > 100 and recordid < 800;
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on table_noindex  (cost=0.00..446182.22 rows=1 width=49) (actual time=711.937..2078.237 rows=635 loops=1)
   Filter: ((recordid > 100) AND (recordid < 800))
   Rows Removed by Filter: 9999365
 Planning Time: 2.351 ms
 Execution Time: 2078.284 ms
(5개 행)
```

#4, #5, #6 모두 다음의 difference를 찾을 수 있다. -

recordid에 대해 btr_index가 있는 table_btree에 비해, index없이 sequential scan을 해야하는 table_noindex는 무지막지하게 더 긴 execution time을 가진다.

5. Update 2,000,000 “recordid” fields in “table_btree”. And update 2,000,000 “recordid” fields in “table_noindex”. Then find a difference

Hint) Raise “recordid” fields 100% whose value is greater than 8,000,000 (This query will update 2,000,000 records)

```
hw5=# update table_btree
hw5-# set recordid = recordid * 2
hw5-# where recordid > 8000000;
UPDATE 1999999
hw5=# update table_noindex
hw5-# set recordid = recordid * 2
hw5-# where recordid > 8000000;
UPDATE 1999999
```

```
hw5=# explain analyze
hw5-# select *
hw5-# from table_btree
hw5-# where recordid > 250 and recordid < 550;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using btr_index on table_btree  (cost=0.56..31.50 rows=447 width=49) (actual time=0.018..0.094 rows=272 loops=1)
   Index Cond: ((recordid > 250) AND (recordid < 550))
 Planning Time: 0.146 ms
 Execution Time: 0.124 ms
(4개 행)


hw5=# explain analyze
hw5-# select *
hw5-# from table_noindex
hw5-# where recordid > 250 and recordid < 550;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Seq Scan on table_noindex  (cost=0.00..446182.22 rows=1 width=49) (actual time=812.744..2161.346 rows=272 loops=1)
   Filter: ((recordid > 250) AND (recordid < 550))
   Rows Removed by Filter: 9999728
 Planning Time: 1.753 ms
 Execution Time: 2161.376 ms
(5개 행)
```

6. Update all “recordid” fields in “table_btree”. And update all “recordid” fields in “table_noindex”. Then find a difference

Hint) Raise all “recordid” fields 10%

```
hw5=# update table_btree
hw5-# set recordid = recordid * 1.1
hw5-# ;
UPDATE 10000000
hw5=#
hw5=# update table_noindex
hw5-# set recordid = recordid * 1.1;
UPDATE 10000000
hw5=# explain analyze
hw5-# select *
hw5-# from table_btree;
```

```
hw5=# explain analyze
hw5-# select *
hw5-# from table_btree
hw5-# where recordid > 250 and recordid < 550;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using btr_index on table_btree  (cost=0.56..31.50 rows=447 width=49) (actual time=0.695..4.092 rows=272 loops=1)
   Index Cond: ((recordid > 250) AND (recordid < 550))
 Planning Time: 9.844 ms
 Execution Time: 4.151 ms
(4개 행)
```

```
hw5=# explain analyze
hw5-# select *
hw5-# from table_noindex
hw5-# where recordid > 250 and recordid < 550;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on table_noindex  (cost=0.00..446182.22 rows=1 width=49) (actual time=729.482..2108.653 rows=272 loops=1)
   Filter: ((recordid > 250) AND (recordid < 550))
   Rows Removed by Filter: 9999728
 Planning Time: 0.111 ms
 Execution Time: 2108.711 ms
(5개 행)
```