2022-Fall
Database System

학번	2015150220
이름	이윤수

Lab - Indexing 2

[Exercise 1]

- Create two indexes. Create indexes on attribute “recordid” in “table_btree” and “table_hash”.

Create b-tree in “table_btree.recordid”

Create hash index in “table_hash.recordid”

```
postgres=# CREATE INDEX btree_idx on table_btree USING BTREE (recordid);
CREATE INDEX
postgres=# CREATE INDEX hash_idx on table_hash USING HASH(recordid);
CREATE INDEX
```

[Exercise 2]

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

- SELECT * FROM table_btree WHERE recordid=10001;

```
postgres=# EXPLAIN ANALYZE SELECT * FROM table_btree WHERE recordid=10001;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using btree_idx on table_btree  (cost=0.43..8.45 rows=1 width=49) (actual time=1.865..1.868 rows=1 loops=1)
   Index Cond: (recordid = 10001)
 Planning Time: 5.720 ms
 Execution Time: 2.992 ms
(4 rows)
```

- SELECT * FROM table_hash WHERE recordid=10001;

```
postgres=# EXPLAIN ANALYZE SELECT * FROM table_hash WHERE recordid=10001;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using hash_idx on table_hash  (cost=0.00..8.02 rows=1 width=49) (actual time=0.501..0.504 rows=1 loops=1)
   Index Cond: (recordid = 10001)
 Planning Time: 2.426 ms
 Execution Time: 0.544 ms
(4 rows)
```

Table_hash 에서의 hash_idx 에 의한 index scan 이 table_btree 에서의 btree_idx 에 의한 index scan 보다 더 빠르다.

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

- SELECT * FROM table_btree WHERE recordid>250 AND recordid<550;

```
postgres=# EXPLAIN ANALYZE SELECT * FROM table_btree WHERE recordid>250 AND recordid<550;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using btree_idx on table_btree  (cost=0.43..17.50 rows=303 width=49) (actual time=0.449..1.896 rows=299 loops=1)
   Index Cond: ((recordid > 250) AND (recordid < 550))
 Planning Time: 1.362 ms
 Execution Time: 1.966 ms
(4 rows)
```

- SELECT * FROM table_hash WHERE recordid>250 AND recordid<550;

```
postgres=# EXPLAIN ANALYZE SELECT * FROM table_hash WHERE recordid>250 AND recordid<550;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..166593.10 rows=1 width=49) (actual time=4.857..4186.059 rows=299 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on table_hash  (cost=0.00..165593.00 rows=1 width=49) (actual time=2767.306..4160.193 rows=100 loops=3)
         Filter: ((recordid > 250) AND (recordid < 550))
         Rows Removed by Filter: 3333234
 Planning Time: 1.218 ms
 Execution Time: 4188.280 ms
(8 rows)
```

Range query 의 경우 hash_idx 보다 btree_idx 를 이용한 index_scan 이 더 빠르다.

Hash_idx 의 경우 index_scan 이 sequential scan 보다 더 느리다는 것을 알 수 있다.

[Exercise 3]

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

- Update “recordid” from 9,999,997 to 9,999,998

```
postgres=# EXPLAIN ANALYZE UPDATE table_btree SET recordid = 9999998 WHERE recordid = 9999997;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Update on table_btree  (cost=0.43..8.45 rows=0 width=0) (actual time=2.821..2.823 rows=0 loops=1)
   ->  Index Scan using btree_idx on table_btree  (cost=0.43..8.45 rows=1 width=10) (actual time=1.720..1.725 rows=1 loops=1)
         Index Cond: (recordid = 9999997)
 Planning Time: 1.555 ms
 Execution Time: 9.608 ms
(5 rows)

postgres=# EXPLAIN ANALYZE UPDATE table_noindex SET recordid = 9999998 WHERE recordid = 9999997;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Update on table_noindex  (cost=0.00..228092.36 rows=0 width=0) (actual time=1377.627..1377.627 rows=0 loops=1)
   ->  Seq Scan on table_noindex  (cost=0.00..228092.36 rows=1 width=10) (actual time=1377.053..1377.055 rows=1 loops=1)
         Filter: (recordid = 9999997)
         Rows Removed by Filter: 9999999
 Planning Time: 0.329 ms
 Execution Time: 1377.684 ms
(6 rows)
```

둘 다 recordid 가 9,999,997 인 record 를 찾는다. 이때 Table_btree 는 btree_idx 를 활용한 idex scan 이 쓰이는 반면, table_noindex 는 seq scand 이 쓰인다. 이에 따라 전자가 후자에 비해 더 빠르게 실행되는 것을 확인할 수 있다.

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

- Increase “recordid” fields by 100% whose value is greater than 8,000,000

```
postgres=# EXPLAIN ANALYZE UPDATE table_btree SET recordid = recordid * 2.0 WHERE recordid > 8000000;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Update on table_btree  (cost=0.43..91809.51 rows=0 width=0) (actual time=23953.348..23953.349 rows=0 loops=1)
   ->  Index Scan using btree_idx on table_btree  (cost=0.43..91809.51 rows=1983763 width=10) (actual time=2.139..2838.847 rows=1999999 loops=1)
         Index Cond: (recordid > 8000000)
 Planning Time: 0.233 ms
 Execution Time: 23953.938 ms
(5 rows)

postgres=# EXPLAIN ANALYZE UPDATE table_noindex SET recordid = recordid * 2.0 WHERE recordid > 8000000;
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Update on table_noindex  (cost=0.00..243112.06 rows=0 width=0) (actual time=13582.780..13582.781 rows=0 loops=1)
   ->  Seq Scan on table_noindex  (cost=0.00..243112.06 rows=2002626 width=10) (actual time=706.409..1998.523 rows=1999999 loops=1)
         Filter: (recordid > 8000000)
         Rows Removed by Filter: 8000001
 Planning Time: 0.076 ms
 Execution Time: 13582.822 ms
(6 rows)
```

이 경우는 오히려 table_noindex 에서의 seq scan 이 더 빨랐다. WHERE 절이 단순한 조건일 경우 index 를 타고 찾는 것보다 순차탐색을 하며 체크하는 것이 더 빠를 수 있다는 것을 알 수 있다.

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

- Increase all “recordid” fields by 10%

```
postgres=# EXPLAIN ANALYZE UPDATE table_btree SET recordid = recordid * 1.10;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Update on table_btree  (cost=0.00..298754.93 rows=0 width=0) (actual time=150288.612..150288.613 rows=0 loops=1)
   ->  Seq Scan on table_btree  (cost=0.00..298754.93 rows=10002453 width=10) (actual time=0.538..18847.249 rows=10000000 loops=1)
 Planning Time: 1.354 ms
 Execution Time: 150288.683 ms
(4 rows)

postgres=# EXPLAIN ANALYZE UPDATE table_noindex SET recordid = recordid * 1.10;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Update on table_noindex  (cost=0.00..298722.24 rows=0 width=0) (actual time=30413.281..30413.282 rows=0 loops=1)
   ->  Seq Scan on table_noindex  (cost=0.00..298722.24 rows=10000585 width=10) (actual time=0.827..6397.755 rows=10000000 loops=1)
 Planning Time: 2.279 ms
 Execution Time: 30413.361 ms
(4 rows)
```

Table_noindex 의 seq scan 이 훨씬 빠르다. 전체 레코드를 모두 탐색하는 경우에는 Index scan 보다 seq scan 이 더 빠르다는 것을 알 수 있다.

[Exercise 4]

a. Find all points within a rectangle ((1,1), (10,10)) on the tables “test0” and “test1”.

Compare an index scan and seq scan

(1) Index scan

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test0 WHERE x>=1 and x<=10 and y>=1 and y<=10;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..15832.23 rows=1289 width=20) (actual time=0.665..146.216 rows=1238 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test0  (cost=0.00..14703.33 rows=537 width=20) (actual time=0.275..136.595 rows=413 loops=3)
         Filter: ((x >= '1'::double precision) AND (x <= '10'::double precision) AND (y >= '1'::double precision) AND (y <= '10'::double precision))
         Rows Removed by Filter: 332921
 Planning Time: 0.197 ms
 Execution Time: 146.425 ms
(8 rows)
```

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test1 WHERE p<@ box '((1,1),(10,10))';
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using test_rtree_idx on test1  (cost=0.29..3757.78 rows=1000 width=20) (actual time=0.051..1.794 rows=1304 loops=1)
   Index Cond: (p <@ '(10,10),(1,1)'::box)
 Planning Time: 0.139 ms
 Execution Time: 1.994 ms
(4 rows)
```

(2) Seq scan

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test0 WHERE x>=1 and x<=10 and y>=1 and y<=10;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..15832.23 rows=1289 width=20) (actual time=0.632..59.104 rows=1238 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test0  (cost=0.00..14703.33 rows=537 width=20) (actual time=0.295..51.315 rows=413 loops=3)
         Filter: ((x >= '1'::double precision) AND (x <= '10'::double precision) AND (y >= '1'::double precision) AND (y <= '10'::double precision))
         Rows Removed by Filter: 332921
 Planning Time: 0.205 ms
 Execution Time: 59.278 ms
(8 rows)
```

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test1 WHERE p<@ box '((1,1),(10,10))';
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..12678.33 rows=1000 width=20) (actual time=3.146..312.172 rows=1304 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test1  (cost=0.00..11578.33 rows=417 width=20) (actual time=2.129..302.614 rows=435 loops=3)
         Filter: (p <@ '(10,10),(1,1)'::box)
         Rows Removed by Filter: 332899
 Planning Time: 0.133 ms
 Execution Time: 312.447 ms
(8 rows)
```

Test0 테이블의 경우 인덱스 스캔을 사용해도 자동으로 seq scan 이 수행된다.

즉 index scan 보다 seq scan 이 더 효율적인 것을 알 수 있다.

한편 Test1 테이블의 경우 index scan 을 사용하는 것이 seq scan 보다 더 빠르다.

또한 test1 의 index scan 보다 test0 의 seq scan 이 더 빠르다.

b. Find all boxes overlapped with rectangles ((0,0), (1,1)) and ((9,9), (10,10)) at the same time on the table “test2”.

(1) Index scan

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test2 WHERE testbox && box'((0,0),(1,1))' and testbox && box '((9,9),(10,10))';
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using test_box_idx on test2  (cost=0.41..104.91 rows=25 width=36) (actual time=0.267..8.990 rows=1425 loops=1)
   Index Cond: ((testbox && '(1,1),(0,0)'::box) AND (testbox && '(10,10),(9,9)'::box))
 Planning Time: 0.148 ms
 Execution Time: 9.236 ms
(4 rows)
```

(2) Seq scan

```
postgres=# set enable_indexscan=false;
SET
postgres=# EXPLAIN ANALYZE SELECT * FROM test2 WHERE testbox && box'((0,0),(1,1))' and testbox && box '((9,9),(10,10))';
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..15586.50 rows=25 width=36) (actual time=11.038..564.668 rows=1425 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test2  (cost=0.00..14584.00 rows=10 width=36) (actual time=2.325..529.008 rows=475 loops=3)
         Filter: ((testbox && '(1,1),(0,0)'::box) AND (testbox && '(10,10),(9,9)'::box))
         Rows Removed by Filter: 332858
 Planning Time: 2.763 ms
 Execution Time: 565.677 ms
(8 rows)
```

Index scan 이 seq scan 에 비해 더 빠르다.

c. Find 10 nearest points to (0,0) on the tables “test0” and “test1”.

(1) Index Scan

```
postgres=# SET enable_indexscan=true;
SET
postgres=# EXPLAIN ANALYZE SELECT * FROM ( SELECT id, x, y, ABS(x)+ABS(y) as dist FROM test0) as distance ORDER BY dist asc limit 10;
                                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=23665.72..23666.88 rows=10 width=28) (actual time=133.291..134.635 rows=10 loops=1)
   ->  Gather Merge  (cost=23665.72..120894.81 rows=833334 width=28) (actual time=133.289..134.630 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=22665.69..23707.36 rows=416667 width=28) (actual time=124.239..124.241 rows=8 loops=3)
               Sort Key: ((abs(test0.x) + abs(test0.y)))
               Sort Method: top-N heapsort  Memory: 26kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 26kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 26kB
               ->  Parallel Seq Scan on test0  (cost=0.00..13661.67 rows=416667 width=28) (actual time=0.089..68.236 rows=333333 loops=3)
 Planning Time: 0.267 ms
 Execution Time: 134.732 ms
(12 rows)

postgres=# EXPLAIN ANALYZE SELECT * FROM ( SELECT id, p,  p <-> point '(0,0)' as dist FROM test1) as distance ORDER BY dist asc limit 10;
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.29..0.98 rows=10 width=28) (actual time=3.317..4.344 rows=10 loops=1)
   ->  Index Scan using test_rtree_idx on test1  (cost=0.29..69732.29 rows=1000000 width=28) (actual time=3.314..4.336 rows=10 loops=1)
         Order By: (p <-> '(0,0)'::point)
 Planning Time: 0.139 ms
 Execution Time: 4.386 ms
(5 rows)
```

(2) Seq Scan

```
postgres=# SET enable_indexscan=false;
SET
postgres=# EXPLAIN ANALYZE SELECT * FROM ( SELECT id, x, y, ABS(x)+ABS(y) as dist FROM test0) as distance ORDER BY dist asc limit 10;
                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=23665.72..23666.88 rows=10 width=28) (actual time=128.377..129.837 rows=10 loops=1)
   ->  Gather Merge  (cost=23665.72..120894.81 rows=833334 width=28) (actual time=128.375..129.832 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=22665.69..23707.36 rows=416667 width=28) (actual time=121.246..121.248 rows=8 loops=3)
               Sort Key: ((abs(test0.x) + abs(test0.y)))
               Sort Method: top-N heapsort  Memory: 26kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 26kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 26kB
               ->  Parallel Seq Scan on test0  (cost=0.00..13661.67 rows=416667 width=28) (actual time=0.088..66.454 rows=333333 loops=3)
 Planning Time: 0.180 ms
 Execution Time: 129.893 ms
(12 rows)

postgres=# EXPLAIN ANALYZE SELECT * FROM ( SELECT id, p,  p <-> point '(0,0)' as dist FROM test1) as distance ORDER BY dist asc limit 10;
                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=21582.38..21583.55 rows=10 width=28) (actual time=132.236..133.781 rows=10 loops=1)
   ->  Gather Merge  (cost=21582.38..118811.47 rows=833334 width=28) (actual time=132.234..133.776 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=20582.36..21624.03 rows=416667 width=28) (actual time=125.073..125.075 rows=8 loops=3)
               Sort Key: ((test1.p <-> '(0,0)'::point))
               Sort Method: top-N heapsort  Memory: 26kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 26kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 26kB
               ->  Parallel Seq Scan on test1  (cost=0.00..11578.33 rows=416667 width=28) (actual time=0.085..72.426 rows=333333 loops=3)
 Planning Time: 0.185 ms
 Execution Time: 133.830 ms
(12 rows)
```

Test0 의 index scan 과 seq scan, 그리고 test1 의 seq scan 은 먼저 힙정렬을 한 다음 seq scan 을 하는 방식으로 수행됐다. 즉 test0 의 index scan 은 seq scan 보다 비효율적이다. 이때 세 가지 모두 129~135 ms 정도로 수행시간이 유사하다. 그런데 test1 의 Index scan 의 경우 수행시간이 4ms 로 나머지 세 방식에 비해 훨씬 빠르게 수행됐다.