Database 6th Assignment Report

Query Optimization

학과 : 컴퓨터학과
이름 : 정이든
학번 : 2018380603

I. DB Setup for Equivalent SQL Statements

```
template1=# \i /Users/jeong-ideun/Downloads/table1-1.dump
SET
SET
SET
SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
COPY 10000000
template1=# SET enable_bitmapscan=false;
SET
template1=# SET max_parallel_workers_per_gather=0;
SET
```

새로운 DB인 Template1에 table1.dump 데이터 파일을 import시켜주었습니다. 또한 이전 과제와 같이 2개의 SET 명령어로 기본 Setup을 해주었습니다.

II. Equivalent SQL Statements

Select "unsorted" from table1 where the "unsorted' value is 967, 968, 969 (Use the DISTINCT for the question a, b, c *except d*)

a. Make an SQL statement using BETWEEN and AND operator
b. Make an SQL statement using IN operator
c. Make an SQL statement using = and OR operator
d. Make an SQL statement using UNION operator

Q1. EXPLAIN ANALYZE your SQL statements and discuss the results of the queries.

```
template1=# EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted between 967 and 969;
                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Unique  (cost=253093.64..253093.72 rows=16 width=4) (actual time=1861.400..1861.408 rows=3 loops=1)
   ->  Sort  (cost=253093.64..253093.68 rows=16 width=4) (actual time=1861.397..1861.399 rows=18 loops=1)
         Sort Key: unsorted
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on table1  (cost=0.00..253093.32 rows=16 width=4) (actual time=144.605..1861.339 rows=18 loops=1)
               Filter: ((unsorted >= 967) AND (unsorted <= 969))
               Rows Removed by Filter: 9999982
 Planning Time: 0.110 ms
 Execution Time: 1861.446 ms
(9 rows)

template1=#
```

a) EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted between 967 AND 969;

unsorted 속성을 퀵 정렬 알고리즘으로 정렬한 후, condition을 만족하는 tuple을 찾기 위해 table1 처음부터 끝까지 읽어야하므로 Seq Scan이 사용되었다. No index의 4개 Query 중에서 가장 실행 시간이 빠른 것을 알 수 있다.

```
template1=# EXPLAIN ANALYZE select distinct unsorted from table1 where unsorted in
(967, 968, 969);
                                                                    QUERY PLAN

------------------------------------------------------------------------------------
-------------------------------------------------
 Unique  (cost=161386.97..161387.07 rows=19 width=4) (actual time=2051.346..2051.35
7 rows=3 loops=1)
   ->  Sort  (cost=161386.97..161387.02 rows=19 width=4) (actual time=2051.345..205
1.347 rows=18 loops=1)
         Sort Key: unsorted
         Sort Method: quicksort  Memory: 25kB
         ->  Gather  (cost=1000.00..161386.57 rows=19 width=4) (actual time=618.518
..2053.219 rows=18 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Parallel Seq Scan on table1  (cost=0.00..160384.67 rows=8 width=
4) (actual time=330.600..2045.338 rows=6 loops=3)
                     Filter: (unsorted = ANY ('{967,968,969}'::integer[]))
                     Rows Removed by Filter: 3333327
 Planning Time: 0.234 ms
 Execution Time: 2055.547 ms
(12 rows)
```

b) EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted in (967, 968, 969)

위의 a와 비슷한 이유로 no index의 경우에는 순차적으로 접근할 수 밖에 없기 때문에 Seq Scan을 통해 Query를 수행하였다.

```
template1=# EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted = 967
OR unsorted = 968 OR unsorted = 969;
                                                              QUERY PLAN

------------------------------------------------------------------------------------------------
-----------------------------------
 Unique  (cost=278093.77..278093.87 rows=19 width=4) (actual time=2075.144..2075.153 r
ows=3 loops=1)
   ->  Sort  (cost=278093.77..278093.82 rows=19 width=4) (actual time=2075.142..2075.1
44 rows=18 loops=1)
         Sort Key: unsorted
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on table1  (cost=0.00..278093.37 rows=19 width=4) (actual time=1
61.300..2075.084 rows=18 loops=1)
               Filter: ((unsorted = 967) OR (unsorted = 968) OR (unsorted = 969))
               Rows Removed by Filter: 9999982
 Planning Time: 0.101 ms
 Execution Time: 2075.191 ms
(9 rows)

template1=#
```

c) EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted = 967 OR unsorted = 968 OR unsorted = 969;

위의 쿼리 플랜을 보니 마찬가지로 DBMS가 가장 먼저 “unsorted” Attribute를 Quick Sorting 알고리즘을 이용하여 Sort하였다. 그리고 정렬된 “unsorted” 속성을 Seq Scan로 처음부터 linear scan하여 조건에 맞는 튜플을 찾는다.

(a)랑 플랜이 상당히 비슷하지만 이 쿼리가 실행 시간이 조금 더 걸린 이유는 Seq Scan을 하면서 각 unsorted Attribute의 값이 967, 968, 969 인지 아닌지 총 3번 필터링을 해야하기 때문이라고 생각한다.

```
template1=# explain analyze (select unsorted from table1 where unsorted = 967) UNION (select unsorted
from table1 WHERE unsorted = 968) UNION (SELECT unsorted FROM table1 WHERE unsorted = 969);
                                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------------
-------------------------
 HashAggregate  (cost=684280.10..684280.28 rows=18 width=4) (actual time=4942.037..4942.038 rows=3 loo
ps=1)
   Group Key: table1.unsorted
   ->  Append  (cost=0.00..684280.06 rows=18 width=4) (actual time=914.931..4941.910 rows=18 loops=1)
         ->  Seq Scan on table1  (cost=0.00..228093.26 rows=6 width=4) (actual time=914.930..1655.463
rows=1 loops=1)
               Filter: (unsorted = 967)
               Rows Removed by Filter: 9999999
         ->  Seq Scan on table1 table1_1  (cost=0.00..228093.26 rows=6 width=4) (actual time=125.636..
1643.871 rows=5 loops=1)
               Filter: (unsorted = 968)
               Rows Removed by Filter: 9999995
         ->  Seq Scan on table1 table1_2  (cost=0.00..228093.26 rows=6 width=4) (actual time=185.826..
1642.556 rows=12 loops=1)
               Filter: (unsorted = 969)
               Rows Removed by Filter: 9999988
 Planning Time: 0.250 ms
 Execution Time: 4942.144 ms
(14 rows)

template1=#
```

d) EXPLAIN ANALYZE (SELECT unsorted FROM table1 WHERE unsorted = 967) UNION (SELECT unsorted FROM table1 WHERE unsorted = 968) UNION (SELECT unsorted FROM table1 WHERE unsorted = 969);

이 쿼리는 UNION operator를 사용하여 만들어진 쿼리이다. 그런데 우리는 교과서에서 Set Operations는 hash index를 활용하는게 훨씬 효율적이라고 하였는데 이 쿼리에서는 Index scan이 아닌 Seq Scan을 하였다. 그 이유는 현재 table1은 no index 상태이기때문이다. 그러므로 3개의 Operand가 조건에 맞는지 확인하기 위해서 처음부터 끝까지 Seq Scan 해야하고, 그것들을 UNION하므로 append까지 해야한다. 총 3번의 Seq Scan을 해야하므로 4개의 쿼리 중에서 실행 시간이 거의 제일 느리다.

Q2) Create an Btree index on “unsorted” column and repeat Q1.

```
template1=# SET max_parallel_workers_per_gather=0;
SET
template1=# SET enable_bitmapscan=false;
SET
template1=# CREATE INDEX b_index on table1 using btree (unsorted);
CREATE INDEX
template1=# \d table1
                  Table "public.table1"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 sorted   | integer       |           |          |
 unsorted | integer       |           |          |
 rndm     | integer       |           |          |
 dummy    | character(40) |           |          |
Indexes:
    "b_index" btree (unsorted)

template1=#
```

“unsorted” column의 B tree index인 b_index를 만들어주었습니다.

```
template1=# EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted
 between 967 AND 969;
                                                                  QUERY PLAN

------------------------------------------------------------------------------
------------------------------------------
 Unique  (cost=0.43..68.79 rows=16 width=4) (actual time=0.038..0.190 rows=3 lo
ops=1)
   ->  Index Only Scan using b_index on table1  (cost=0.43..68.75 rows=16 width
=4) (actual time=0.036..0.181 rows=18 loops=1)
         Index Cond: ((unsorted >= 967) AND (unsorted <= 969))
         Heap Fetches: 18
 Planning Time: 0.396 ms
 Execution Time: 0.213 ms
(6 rows)

template1=#
```

a) unsorted의 Index가 없었던 1-(a)와 비교도 안되게 빠르다. search key가 “unsorted”인 B tree 인덱스 b_index만을 이용하여 unsorted값이 967, 969사이의 값에 바로 접근할 수 있기 때문에 정말 빠른 모습이다. Index Only Scan을 사용하는 이유는 table1이 unsorted로 정렬되어 있지는 않기 때문이다.

```
template1=# EXPLAIN ANALYZE select distinct unsorted from table1 where unsorted in
 (967, 968, 969);
                                                                    QUERY PLAN

-------------------------------------------------------------------------------------
-----------------------------------------
 Unique  (cost=0.43..89.68 rows=19 width=4) (actual time=0.730..2.331 rows=3 loops
=1)
   ->  Index Only Scan using b_index on table1  (cost=0.43..89.63 rows=19 width=4)
 (actual time=0.727..2.314 rows=18 loops=1)
         Index Cond: (unsorted = ANY ('{967,968,969}'::integer[]))
         Heap Fetches: 18
 Planning Time: 1.183 ms
 Execution Time: 2.406 ms
(6 rows)
```

b) EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted in (967, 968, 969);

이 쿼리 또한 1-(b)에 비해 실행이 정말 빠름을 알 수 있다. 왜냐하면 B tree index의 경우 특정 값에 해당하는 tuple들의 값들만 찾아내면 되기 때문에 index only scan을 통해 쿼리를 비교적 빠르게 수행한 모습이다.

```
template1=# EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted = 967
OR unsorted = 968 OR unsorted = 969;
                                                        QUERY PLAN

-------------------------------------------------------------------------------------
----------------------------------
 Unique  (cost=278093.40..278093.50 rows=19 width=4) (actual time=2094.122..2094.130 r
ows=3 loops=1)
   ->  Sort  (cost=278093.40..278093.45 rows=19 width=4) (actual time=2094.119..2094.1
21 rows=18 loops=1)
         Sort Key: unsorted
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on table1  (cost=0.00..278093.00 rows=19 width=4) (actual time=1
61.460..2094.052 rows=18 loops=1)
               Filter: ((unsorted = 967) OR (unsorted = 968) OR (unsorted = 969))
               Rows Removed by Filter: 9999982
 Planning Time: 0.153 ms
 Execution Time: 2094.168 ms
(9 rows)

template1=#
```

c) **EXPLAIN ANALYZE SELECT DISTINCT unsorted FROM table1 WHERE unsorted = 967 OR unsorted = 968 OR unsorted = 969;**

유일하게 이 쿼리만 실행 시간이 매우 느리다. 심지어 1-(c)보다 느린 것을 확인 할 수 있는다. 내가 생각하기에 이러한 결과가 도출된 이유는 다음과 같다. BETWEEN AND operator나 IN operator는 해당 범위의 값을 하나의 단위로 인식하여 index를 통해 Index Only Scan하여 해당 값을 바로 찾는 operator인 것 같고, = OR operator로 묶인 조건들은 범위로 검사하는 것이 아니라, 각 값과 일치하는지 일일이 확인하기에 수행 시간이 너무 길게 되어서 Seq Scan으로 수행되지 않았나 조심스럽게 추측해본다. (이 Query에 경우 tuple 당 967, 968, 969와 같은지 최대 3번의 확인절차가 필요하다.)

```
template1=# EXPLAIN ANALYZE (SELECT unsorted FROM table1 WHERE unsorted = 967) UNION (SELECT unsorted
FROM table1 WHERE unsorted = 968) UNION (SELECT unsorted FROM table1 WHERE unsorted = 969);
                                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------
-------------------------------------
 HashAggregate  (cost=85.93..86.11 rows=18 width=4) (actual time=0.145..0.147 rows=3 loops=1)
   Group Key: table1.unsorted
   ->  Append  (cost=0.43..85.89 rows=18 width=4) (actual time=0.040..0.120 rows=18 loops=1)
         ->  Index Only Scan using b_index on table1  (cost=0.43..28.54 rows=6 width=4) (actual time=0
.039..0.040 rows=1 loops=1)
               Index Cond: (unsorted = 967)
               Heap Fetches: 1
         ->  Index Only Scan using b_index on table1 table1_1  (cost=0.43..28.54 rows=6 width=4) (actu
al time=0.019..0.029 rows=5 loops=1)
               Index Cond: (unsorted = 968)
               Heap Fetches: 5
         ->  Index Only Scan using b_index on table1 table1_2  (cost=0.43..28.54 rows=6 width=4) (actu
al time=0.020..0.044 rows=12 loops=1)
               Index Cond: (unsorted = 969)
               Heap Fetches: 12
 Planning Time: 0.226 ms
 Execution Time: 0.236 ms
(14 rows)

template1=#
```

d) EXPLAIN ANALYZE (SELECT unsorted FROM table1 WHERE unsorted = 967) UNION (SELECT unsorted FROM table1 WHERE unsorted = 968) UNION (SELECT unsorted FROM table1 WHERE unsorted = 969);

이 쿼리의 Index Cond를 보면, (unsorted = 967) 이런 식으로 하나의 unsorted 값에 대한 Index를 B-tree를 통해 찾을 수 있기 때문에 Index Only Scan으로 조건을 만족하는 레코드를 찾는다. 그리고 찾은 각각을 Append 해준다. b_index를 사용함으로써 1-(d)보다 실행 시간이 훨씬 빨라졌다.

Q3) Create an Hash index on “unsorted” column and repeat Q1.

```
template1=# create index h_index on table1 using hash(unsorted);
CREATE INDEX
template1=# \d table1
                    Table "public.table1"
  Column   |     Type       | Collation | Nullable | Default
-----------+----------------+-----------+----------+---------
 sorted    | integer        |           |          |
 unsorted  | integer        |           |          |
 rndm      | integer        |           |          |
 dummy     | character(40)  |           |          |
Indexes:
    "h_index" hash (unsorted)
```

마찬가지로 hash index를 만들고, 정상적으로 생성된 것을 확인한다.

```
template1=# EXPLAIN ANALYZE select distinct unsorted from table1 where unsorted between
967 AND 969;
                                                        QUERY PLAN

-------------------------------------------------------------------------------------------
-------------------------------
 Unique  (cost=253093.32..253093.40 rows=16 width=4) (actual time=2043.313..2043.323 row
s=3 loops=1)
   ->  Sort  (cost=253093.32..253093.36 rows=16 width=4) (actual time=2043.310..2043.312
 rows=18 loops=1)
         Sort Key: unsorted
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on table1  (cost=0.00..253093.00 rows=16 width=4) (actual time=165
.679..2043.206 rows=18 loops=1)
               Filter: ((unsorted >= 967) AND (unsorted <= 969))
               Rows Removed by Filter: 9999982
 Planning Time: 0.152 ms
 Execution Time: 2043.410 ms
(9 rows)
```

a) Hash index는 특정 값을 검색할 때는 빠른 성능을 내지만, 키값 자체가 변환되어 bucket에 저장되기 때문에 범위를 검색하거나 원본 값으로 정렬할 수 없다. 그렇기에 이 Query의 범위 검사에 대해서 Seq Scan를 이용하여 실행한 것을 볼 수 있다.

```
template1=# EXPLAIN ANALYZE select distinct unsorted from table1 where unsorted in (967,
968, 969);
                                                        QUERY PLAN

------------------------------------------------------------------------------------------
------------------------------
 Unique  (cost=240593.40..240593.50 rows=19 width=4) (actual time=2175.928..2175.936 rows
=3 loops=1)
   ->  Sort  (cost=240593.40..240593.45 rows=19 width=4) (actual time=2175.926..2175.928
rows=18 loops=1)
         Sort Key: unsorted
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on table1  (cost=0.00..240593.00 rows=19 width=4) (actual time=171.
044..2175.853 rows=18 loops=1)
               Filter: (unsorted = ANY ('{967,968,969}'::integer[]))
               Rows Removed by Filter: 9999982
 Planning Time: 0.127 ms
 Execution Time: 2175.970 ms
(9 rows)
```

b) 위의 문제와 유사하게, 이 Query에서도 Hash index인 경우에 유사한 이유로 Seq Scan을 하게 된다. 왜냐하면 Hash index의 경우에 범위로 검사하는 것은 아니지만, 특정 값을 찾아내는 것이 아닌 각 값들을 따로 비교해가며 찾아야하기 때문에 Seq Scan 더 빠르다고 DBMS가 판단하여 Seq Scan으로 이 Query를 실행하였다.

```
template1=# EXPLAIN ANALYZE select distinct unsorted from table1 where unsorted=967 OR
unsorted=968 OR unsorted=969;
                                                        QUERY PLAN

------------------------------------------------------------------------------------------
------------------------------
 Unique  (cost=278093.40..278093.50 rows=19 width=4) (actual time=2100.588..2100.597 ro
ws=3 loops=1)
   ->  Sort  (cost=278093.40..278093.45 rows=19 width=4) (actual time=2100.585..2100.58
8 rows=18 loops=1)
         Sort Key: unsorted
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on table1  (cost=0.00..278093.00 rows=19 width=4) (actual time=16
4.965..2100.505 rows=18 loops=1)
               Filter: ((unsorted = 967) OR (unsorted = 968) OR (unsorted = 969))
               Rows Removed by Filter: 9999982
 Planning Time: 0.128 ms
 Execution Time: 2100.634 ms
(9 rows)
```

c) a, b에서와 같이 이 Query에 대해서도 동일한 이유로 Seq Scan을 하게 된다.

```
template1=# EXPLAIN ANALYZE (select unsorted from table1 where unsorted=967) UNION (select unsorted
 from table1 where unsorted = 968) UNION (select unsorted from table1 where unsorted = 969);
                                                                  QUERY PLAN

------------------------------------------------------------------------------------------------------------
-----------------------------------
 HashAggregate  (cost=84.63..84.81 rows=18 width=4) (actual time=0.168..0.169 rows=3 loops=1)
   Group Key: table1.unsorted
   ->  Append  (cost=0.00..84.59 rows=18 width=4) (actual time=0.036..0.096 rows=18 loops=1)
         ->  Index Scan using h_index on table1  (cost=0.00..28.11 rows=6 width=4) (actual time=0.0
35..0.037 rows=1 loops=1)
               Index Cond: (unsorted = 967)
         ->  Index Scan using h_index on table1 table1_1  (cost=0.00..28.11 rows=6 width=4) (actual
 time=0.019..0.028 rows=5 loops=1)
               Index Cond: (unsorted = 968)
         ->  Index Scan using h_index on table1 table1_2  (cost=0.00..28.11 rows=6 width=4) (actual
 time=0.007..0.026 rows=12 loops=1)
               Index Cond: (unsorted = 969)
 Planning Time: 0.290 ms
 Execution Time: 0.273 ms
(11 rows)
```

d) 이 쿼리는 특정 값을 찾는 것이기 때문에 Hash Index의 장점을 그대로 이용할 수 있다. 각 특정 unsorted값을 찾은 뒤 각 Bucket에서 내가 원하는 키 값을 Linear Scan하여 찾은 후 UNION하는 것이기 때문에 Index Scan이 이용되었다.

Q4) Compare each SQL statements' performances on three cases(no index, Btree index, hash Index)

*(Searh Method의 Reason는 Q1~Q3에 있으므로 성능적으로만 비교하였습니다.)

a) 비교하자면 no index case는 전부 Seq Scan로 세 케이스 중 가장 느린 축에 속하였고, Btree index case는 Index Only using으로 가장 빨랐으며, 마지막으로 Hash Index case는 조건문이 범위 값이기 때문 Seq Scan으로 느린 성능을 보였다.

b) no index case는 말할 것도 없이 Seq Scan으로 가장 느린 축에 속하였고, Btree index는 Index Only Using으로 가장 성능이 좋았으며, hash index는 Seq Scan으로 여전히 느렸다.

c) 모든 case 에서 Seq Scan을 하였다. 조금 특별한 점은 B tree index에서 다소 예상치 못한 결과가 나온 것이다. 이에 대한 설명은 앞에서 언급했으니 넘어가도록 한다.

d) No index의 경우 각각 Seq Scan을 거쳐 Union operator를 합치는 append를 하다 보니 성능이 가장 좋지 않았고, 나머지 두 Query에 대해서는 Btree index가 Hash index보다 조금 더 나은 성능을 보였다. 이는 Query Plan이 Btree는 Index Only Scan, Hash Index는 Index Scan을 썼기 때문이다. Hash의 경우 Uniform하고 Random 성질을 가진 Bucket들을 통해 값들에 대한 index를 갖게 되는데, Bucket 내에 정확히 똑같은 값들만 채워져 있는 것이 아니기 때문에 찾고자 하는 값에 근접한 Bucket에 접근하여 해당 값을 찾으므로 Index Scan을 하게 된다. 그러므로 실행 시간이 조금 더 걸린 것 같다.

III. DB Setup for Query Plan

```
template1=# SET max_parallel_workers_per_gather=0;
SET
template1=# SET enable_bitmapscan=false;
SET
template1=# CREATE TABLE pool1 (val integer);
CREATE TABLE
template1=# INSERT INTO pool1(val) SELECT random()*500 FROM (SELECT generate_series(1, 5000000))
 AS T;
INSERT 0 5000000
template1=# CREATE TABLE pool2 (val integer);
CREATE TABLE
template1=# INSERT INTO pool2(val) SELECT random()*500 FROM (SELECT generate_series(1, 5000000))
AS T;
INSERT 0 5000000
```

PPT의 명령어로 테이블과 tuple을 추가하였다. 현재 template1 DB에는 table1, pool1, pool2 총 세 개의 Table이 있다.

IV. Query Plan

1. Following queries have different syntax but return same result. You have to use UNION ALL operator!(different from Equi-SQL statement problem)

a) Union tables(poo1, poo2) and then perfrom aggregation with COUNT function

```
template1=# select count(*) from ((select * from pool1) UNION ALL (select * from pool2)) AS T;
  count
----------
 10000000
(1 row)
```

b) Perfrom aggregation with COUNT function on each table, and then aggregate them again with SUM function on the union of the aggregated results.

```
template1=# select sum(count) from ((select count(*) from pool1) UNION ALL (select count(*) from
 pool2)) as T;
   sum
----------
 10000000
(1 row)

template1=#
```

Q6) Write the queries and use EXPLAIN ANALYZE to see how the query execution is actually planned

```
template1=# set max_parallel_workers_per_gather=0;
SET
template1=# set enalbe_bitmapscan=false;
ERROR:  unrecognized configuration parameter "enalbe_bitmapscan"
template1=# set enable_bitmapscan=false;
SET
template1=# EXPLAIN ANALYZE select count(*) from ((select * from pool1) UNION ALL (select * from
pool2)) as T;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------
----------------------------
 Aggregate  (cost=219248.00..219248.01 rows=1 width=8) (actual time=3485.421..3485.421 rows=1 loo
ps=1)
   ->  Append  (cost=0.00..194248.00 rows=10000000 width=0) (actual time=1.317..2514.844 rows=100
00000 loops=1)
         ->  Seq Scan on pool1  (cost=0.00..72124.00 rows=5000000 width=0) (actual time=1.316..76
2.228 rows=5000000 loops=1)
         ->  Seq Scan on pool2  (cost=0.00..72124.00 rows=5000000 width=0) (actual time=0.022..71
7.917 rows=5000000 loops=1)
 Planning Time: 0.144 ms
 Execution Time: 3485.481 ms
(6 rows)
```

```
template1=# set max_parallel_workers_per_gather=0;
      SET
template1=# set enable_bitmapscan=false;
      SET
template1=# EXPLAIN ANALYZE select sum(count) from ((select count(*) from pool1) UNION ALL
(select count (*) from pool2)) as T;
                                                          QUERY PLAN

------------------------------------------------------------------------------------------------------
----------------------------------------
 Aggregate  (cost=169248.06..169248.07 rows=1 width=32) (actual time=2259.804..2259.804 row
s=1 loops=1)
   ->  Append  (cost=84624.00..169248.05 rows=2 width=8) (actual time=1146.373..2259.791 ro
ws=2 loops=1)
         ->  Aggregate  (cost=84624.00..84624.01 rows=1 width=8) (actual time=1146.372..114
6.372 rows=1 loops=1)
               ->  Seq Scan on pool1  (cost=0.00..72124.00 rows=5000000 width=0) (actual ti
me=1.127..643.956 rows=5000000 loops=1)
         ->  Aggregate  (cost=84624.00..84624.01 rows=1 width=8) (actual time=1113.416..111
3.416 rows=1 loops=1)
               ->  Seq Scan on pool2  (cost=0.00..72124.00 rows=5000000 width=0) (actual ti
me=0.044..619.506 rows=5000000 loops=1)
 Planning Time: 0.159 ms
 Execution Time: 2259.908 ms
(8 rows)
```

Description)

pool1과 pool2 모두 index가 존재하지 않기 때문에 두 Query 모두 UNION에 해당하는 두 select 구문들은 Seq Scan을 수행하게 된다. 다만 두 Query 간에 수행 시간 차이가 나는 이유는 연산에 참여하는 tuple의 수와 관련이 있다고 생각했다. 첫번째 Query는 UNION ALL에 해당하는 두 select 문에서 5,000,000개씩 총 10,000,000개의 tuple이 참여한다. 또한 UNION을 마치면 count을 통해 10,000,000개 tuple을 세게 된다. 따라서 총 20,000,000의 tuple을 보게 되는 것이다. 반면 두번째 Query는 UNION ALL에 해당하는 두 select 문에서 5,000,000개씩 총 10,000,000개의 tuple을 세게 되고, 이 두 count 값을 UNION하여 count라는 attribute에 두 개의 5,000,000을 저장하게 된다. 또한 UNION ALL의 결과로 나온 count 값을 sum하여 총 tuple의 수를 구하게 된다. 이렇게 하면 두 Query의 결과는 서로 동일하나 두번째 Query에서 살펴본 tuple의 수는 10,000,000개로 첫번째 Query보다 절반의 tuple만 보고 동일한 결과를 낼 수 있다. 따라서 두번째 Query가 더 짧은 수행 시간을 낸다고 생각했다.

2. Following queries also return the same result but can be written in different ways. You have to use UNION ALL operator!(different from Equi-SQL statement problem)

a) SELECT tuple WHERE value is above 250 on each table and then union them

```
template1=# SELECT count(*) from ((select * from pool1 where val > 250) UNION ALL (select * from
pool2 where val > 250)) AS T;

  count
---------
 4991776
(1 row)
```

b) Union two tables and SELECT tuples WHERE value is above 250.

```
template1=# select count(*) from ((select * from pool1) UNION ALL (select * from pool2)) as T
where T.val > 250;
  count
---------
 4991776
(1 row)

template1=#
```

두 Query 결과가 같은 것을 알 수 있다.

Q7) Write the queries and use EXPLAIN ANALYZE to see how the query execution is actually planned

```
template1=# EXPLAIN ANALYZE select * from ((select * from pool1 where val > 250) UNION ALL (se
lect * from pool2 where val > 250)) as T;
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------------
-------------------------
 Append  (cost=0.00..244153.23 rows=4993682 width=4) (actual time=1.194..2234.308 rows=4991776
 loops=1)
   ->  Seq Scan on pool1  (cost=0.00..84624.00 rows=2503511 width=4) (actual time=1.193..864.8
83 rows=2495492 loops=1)
         Filter: (val > 250)
         Rows Removed by Filter: 2504508
   ->  Seq Scan on pool2  (cost=0.00..84624.00 rows=2490171 width=4) (actual time=0.036..846.2
25 rows=2496284 loops=1)
         Filter: (val > 250)
         Rows Removed by Filter: 2503716
 Planning Time: 0.166 ms
 Execution Time: 2500.459 ms
(9 rows)
```

```
template1=# EXPLAIN ANALYZE select * from ((select * from pool1) UNION ALL (select * from pool
2)) as T where T.val > 250;
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------------
-------------------------
 Append  (cost=0.00..194216.41 rows=4993682 width=4) (actual time=1.117..2217.640 rows=4991776
 loops=1)
   ->  Seq Scan on pool1  (cost=0.00..84624.00 rows=2503511 width=4) (actual time=1.116..866.5
26 rows=2495492 loops=1)
         Filter: (val > 250)
         Rows Removed by Filter: 2504508
   ->  Seq Scan on pool2  (cost=0.00..84624.00 rows=2490171 width=4) (actual time=0.021..829.4
22 rows=2496284 loops=1)
         Filter: (val > 250)
         Rows Removed by Filter: 2503716
 Planning Time: 0.151 ms
 Execution Time: 2482.949 ms
(9 rows)
```

Description)

pool1과 pool2 테이블의 각 tuple에 존재하는 val이라는 attribute값들은 랜덤으로 생성되었다. 또한 두 테이블에는 인덱스가 존재하지 않기 때문에 모두 Seq Scan후에 UNION operator을 통해 Append된 것을 알 수 있다. 추가로 두 Query에 대한 수행시간 비교는 단순히 250 이상이라는 값을 가진조건을 UNION operator을 쓰기 전, 후에 실행한 것으로 비교할 수 있다. 일단 예측 자체는 굉장히 헷갈렸다. 해당 Query들을 트리로 그렸을 때는 depth가 모두 3이었을 뿐 아니라 연산 수행에 있어서도 UNION 되는 tuple의 개수는 전자가 적지만 조건으로 거르는 횟수가 후자보다 더 많아서 어떤 것이 더 좋은 성능을 내는지 판단이 어려웠다. 결국 어차피 검사 자체는 모든 값에 대해서 250보다 큰지 확인하게 되므로 비슷하게 나오리라고 예상했다. 측정된 결과 역시 비슷하게 나온 것을 알 수 있었다.

Q8) Why does the user-level optimization important?

이번 실습을 통해, 서로 다른 operator들을 이용했지만 동일한 결과를 내놓는 Equivalent Query들에 대해서 공부할 수 있었다. 이런 Query들을 EXPLAIN ANALYZE을 통해 성능을 측정해본 결과로 index의 유무, 데이터의 분포 등 여러 조건에 따라서 각각 수행 시간이 달라짐을 알 수 있었다. 이런 수행 시간들이 서로 다른 것들에 대하여, 어떤 것은 나름 최선의 Query Plan을 썼음에도 정말 긴 수행 시간을 썼던 것을 확인할 수 있었고, 어떤 것은 이전에 긴 수행 시간을 썼던 Query와는 달리 비교도 안될 정도로 짧은 수행 시간을 가지며 요청을 처리해주는 것을 알 수 있었다. 이를 통해 내가 찾으려는 값에 대해서 attribute의 index유무, 데이터 분포 등을 미루어보아 어떤 operator을 써야할지 얕게나마 알아갈 수 있었다. 또한 Equivalent Query에서 operator을 달리했던 것과 반대로 사용하려는 Query에 대해서도 최적화를 한 것과 안한 것의 차이를 통해 수행 시간 비교를 할 수 있었다. 얼만큼의 tuple, attribute들이 요청을 처리하는데 참여하고 얼만큼의 연산을 썼는지에 따라 수행 시간이 천차만별임을 알 수 있었다. 따라서 내가 얻고자 하는 답에 걸맞는 연산과 적절한 Query 최적화가 이루어져야 짧은 수행 시간 안에 결과를 얻을 수 있음을 알 수 있었다. 심지어 데이터가 굉장히 많다면 이런 수행 시간들은 기하 급수적으로 증가할 것이기에 사용자가 Query을 이용할 때 최적화를 고려하여 사용하는 것이 중요하다고 생각한다. 따라서 User-Level의 Query Optimization은 정말 중요하다고 생각한다.