CS3120 Database Management Systems Laboratory

Assignment-9

Mayank Singla
111901030

Q1. Consider the following queries to the 'payment' table:

a. Fetch all the transactions where the transaction amount is always greater than $3.

b. Fetch all the details of the transactions performed by the customer with id = '380'.

Create a partial index on the search keys to make the queries efficient. Justify why a partial index is used here by comparing the execution time of the above queries with and without using the index.

a) Execution time without any index.

```
dvdrental=# EXPLAIN ANALYZE SELECT * FROM payment WHERE amount > 3;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Seq Scan on payment  (cost=10000000000.00..10000001941.45 rows=8038 width=435) (actual
 time=51.983..55.167 rows=8038 loops=1)
   Filter: (amount > '3'::numeric)
   Rows Removed by Filter: 6558
 Planning Time: 0.138 ms
 JIT:
   Functions: 4
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 0.969 ms, Inlining 7.510 ms, Optimization 32.255 ms, Emission 11.
989 ms, Total 52.723 ms
 Execution Time: 56.363 ms
(9 rows)
```

Creating the partial index

```
dvdrental=# CREATE INDEX pay_amt ON payment(amount) WHERE amount > 3;
CREATE INDEX
dvdrental=#
```

Execution time after creating the index

```
dvdrental=# EXPLAIN ANALYZE SELECT * FROM payment WHERE amount > 3;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on payment  (cost=138.48..1997.96 rows=8038 width=435) (actual time=0.536..3.010 rows=8038 loops=1)
   Recheck Cond: (amount > '3'::numeric)
   Heap Blocks: exact=1308
   ->  Bitmap Index Scan on pay_amt  (cost=0.00..136.47 rows=8038 width=0) (actual time=0.379..0.379 rows=8038 loops=1)
 Planning Time: 0.147 ms
 Execution Time: 3.258 ms
(6 rows)
```

b) Since there is already a default index present in the table, I am temporarily dropping that index to measure the correct execution time without any index.

solution.sql U

dbms-lab > labwork > assignment_9 > solution.sql > ...

```
7   BEGIN;
    ▶ Run SQL
8   DROP INDEX idx_fk_customer_id;
    ▶ Run SQL
9   EXPLAIN ANALYZE SELECT * FROM payment WHERE customer_id = '380';
    ▶ Run SQL
10  ROLLBACK;
```

Execution time without any index.

```
dvdrental=# \i solution.sql
BEGIN
DROP INDEX
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on payment  (cost=10000000000.00..10000001941.45 rows=35 width=435) (actual time=48.261..50.205 rows=35 loops=1)
   Filter: (customer_id = '380'::smallint)
   Rows Removed by Filter: 14561
 Planning Time: 0.119 ms
 JIT:
   Functions: 4
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 0.920 ms, Inlining 6.081 ms, Optimization 29.727 ms, Emission 12.191 ms, Total 48.919 ms
 Execution Time: 51.170 ms
(9 rows)

ROLLBACK
```

Creating the partial index.

```
dvdrental=# CREATE INDEX pay_cust_id ON payment(customer_id) WHERE customer_id = '380';
CREATE INDEX
dvdrental=#
```

Execution time with index.

```
dvdrental=# \i solution.sql
BEGIN
DROP INDEX
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on payment  (cost=8.32..133.95 rows=35 width=435) (actual time=0.013..0.035 rows=35 loops=1)
   Recheck Cond: (customer_id = '380'::smallint)
   Heap Blocks: exact=17
   ->  Bitmap Index Scan on pay_cust_id  (cost=0.00..8.32 rows=35 width=0) (actual time=0.005..0.005 rows=35 loops=1)
 Planning Time: 0.099 ms
 Execution Time: 0.052 ms
(6 rows)

ROLLBACK
```

We can observe that by using a partial index, the execution time has been reduced significantly in both cases. We are using partial indexing here because we are querying only certain parts of the data table which satisfy the `WHERE` clause. Hence, it is the best option to make use of the partial index so that indexing is applied efficiently and the execution time improves significantly.

Q2. Fetch the transaction ID and the payment date of the customer with ID = '341' and staff ID = '1' using the 'payment' table. Create a multi-column index on the search keys to make the queries efficient.

a. Justify the decision over the order chosen to create the index by comparing the execution time of the above queries with and without using the index.

b. Change the order of creation of the multi-column index and showcase the change in the execution times which are encountered.

a) Since there is already a default index present in the table, I am temporarily dropping that index to measure the correct execution time without any index.

solution.sql U

dbms-lab > labwork > assignment_9 > solution.sql > ...

```
5   BEGIN;
    ▶ Run SQL
6   DROP INDEX idx_fk_customer_id;
    ▶ Run SQL
7   DROP INDEX idx_fk_staff_id;
    ▶ Run SQL
8   EXPLAIN ANALYZE
9   SELECT transaction_id,
10      payment_date
11  FROM payment
12  WHERE customer_id = '341'
13      AND staff_id = '1';
    ▶ Run SQL
14  ROLLBACK;
```

Execution time without any index.

```
dvdrental=# \i solution.sql
BEGIN
DROP INDEX
DROP INDEX
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on payment  (cost=10000000000.00..10000001977.94 rows=11 width=24) (actual time=45.969..47.471 rows=8 loops=1)
   Filter: ((customer_id = '341'::smallint) AND (staff_id = '1'::smallint))
   Rows Removed by Filter: 14588
 Planning Time: 0.111 ms
 JIT:
   Functions: 4
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 0.896 ms, Inlining 5.905 ms, Optimization 29.128 ms, Emission 10.514 ms, Total 46.444 ms
 Execution Time: 48.419 ms
(9 rows)

ROLLBACK
```

As the number of distinct `customer_ids` is more than `staff_ids`, we will prioritize `customer_id` first while creating the multi-column index.

```
dvdrental=# SELECT COUNT(DISTINCT customer_id), COUNT(DISTINCT staff_id) FROM payment;
 count | count
-------+-------
   599 |     2
(1 row)
```

Creating the multi-column index.

```
dvdrental=# CREATE INDEX pay_cust_staff_id ON payment(customer_id, staff_id);
CREATE INDEX
dvdrental=#
```

Execution time with index.

```
dvdrental=# \i solution.sql
BEGIN
DROP INDEX
DROP INDEX
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on payment  (cost=4.40..45.95 rows=11 width=24) (actual time=0.014..0.020 rows=8 loops=1)
   Recheck Cond: ((customer_id = '341'::smallint) AND (staff_id = '1'::smallint))
   Heap Blocks: exact=5
   ->  Bitmap Index Scan on pay_cust_staff_id  (cost=0.00..4.40 rows=11 width=0) (actual time=0.007..0.008 rows=8 loops=1)
         Index Cond: ((customer_id = '341'::smallint) AND (staff_id = '1'::smallint))
 Planning Time: 0.089 ms
 Execution Time: 0.037 ms
(7 rows)

ROLLBACK
```

We can observe that the execution time has decreased significantly.

b) Creating the index in reversed order of the search keys and dropping the previous index.

```
dvdrental=# DROP INDEX pay_cust_staff_id;
DROP INDEX
dvdrental=# CREATE INDEX pay_cust_staff_id2 ON payment(staff_id, customer_id);
CREATE INDEX
dvdrental=#
```

Execution time with the new order of search keys in the multi-column index.

```
dvdrental=# \i solution.sql
BEGIN
DROP INDEX
DROP INDEX
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Index Scan using pay_cust_staff_id2 on payment  (cost=0.29..43.11 rows=11 width=24) (actual time=0.021..0.027 rows=8 loops=1)
   Index Cond: ((staff_id = '1'::smallint) AND (customer_id = '341'::smallint))
 Planning Time: 0.300 ms
 Execution Time: 0.067 ms
(4 rows)

ROLLBACK
```

We can observe that the execution time has been increased from the previous case due to order reversal which was also expected as we are now having redundant data in our index tree.

Q3. Fetch the movie name and the year in which it was released of all the movies which have a rent greater than $3 and have a running time of at least 100 minutes.

a. Create a covering index to cover any particular attribute of your choice inside the index. Compare the execution time of the fetch query with and without using this index.

b. Create a multi-column index over the same data set and compare the execution times of this newly created index with the covering index created in the previous question. Explain the query plan.

a) Execution time without any index.

```
dvdrental=# EXPLAIN ANALYZE SELECT title, release_year FROM film WHERE rental_rate > '3' AND length > '100';
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on film  (cost=10000000000.00..10000000126.00 rows=205 width=19) (actual time=50.226..50.424 rows=209 loops=1)
   Filter: ((rental_rate > '3'::numeric) AND (length > '100'::smallint))
   Rows Removed by Filter: 791
 Planning Time: 0.144 ms
 JIT:
   Functions: 4
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 0.945 ms, Inlining 7.519 ms, Optimization 31.644 ms, Emission 10.948 ms, Total 51.057 ms
 Execution Time: 51.416 ms
(9 rows)
```

Creating the covering index using the `rental_rate` attribute and including the `length` attribute

```
dvdrental=# CREATE INDEX rental_rate_length ON film(rental_rate) INCLUDE(length);
CREATE INDEX
dvdrental=#
```

Execution time with covering index.

```
dvdrental=# EXPLAIN ANALYZE SELECT title, release_year FROM film WHERE rental_rate > '3' AND length > '100';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on film  (cost=14.85..130.89 rows=205 width=19) (actual time=0.087..0.324 rows=209 loops=1)
   Recheck Cond: (rental_rate > '3'::numeric)
   Filter: (length > '100'::smallint)
   Rows Removed by Filter: 127
   Heap Blocks: exact=58
   ->  Bitmap Index Scan on rental_rate_length  (cost=0.00..14.79 rows=336 width=0) (actual time=0.063..0.063 rows=336 loops=1)
         Index Cond: (rental_rate > '3'::numeric)
 Planning Time: 0.144 ms
 Execution Time: 0.365 ms
(9 rows)
```

We can observe that the execution time has decreased significantly.

b) Since the number of distinct values of the `length` attribute is more than that of the `rental_rate` attribute, we will give more priority to the `length` attribute while creating the multi-column index.

```
dvdrental=# SELECT COUNT(DISTINCT length), COUNT(DISTINCT rental_rate) FROM film;
 count | count
-------+-------
   140 |     3
(1 row)
```

Dropping the previous index and creating the multi-column index.

```
dvdrental=# DROP INDEX rental_rate_length;
DROP INDEX
dvdrental=# CREATE INDEX rental_rate_length2 ON film(length, rental_rate);
CREATE INDEX
dvdrental=#
```

Execution time with the multi-column index.

```
dvdrental=# EXPLAIN ANALYZE SELECT title, release_year FROM film WHERE rental_rate > '3' AND length > '100';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on film  (cost=22.42..138.33 rows=205 width=19) (actual time=0.062..0.138 rows=209 loops=1)
   Recheck Cond: ((length > '100'::smallint) AND (rental_rate > '3'::numeric))
   Heap Blocks: exact=55
   ->  Bitmap Index Scan on rental_rate_length2  (cost=0.00..22.36 rows=205 width=0) (actual time=0.052..0.053 rows=209 loops=1)
         Index Cond: ((length > '100'::smallint) AND (rental_rate > '3'::numeric))
 Planning Time: 0.065 ms
 Execution Time: 0.160 ms
(7 rows)
```

We can observe that the execution time has even decreased more as compared to the case when using the covering index. This tells us that the multi-column index is faster than the covering index.

The query plan in the case of the multi-column index is first doing a Bitmap Heap Scan on the table and filtering out the blocks satisfying the query condition and then using a Bitmap Index scan that we created, it is filtering out all the records that satisfy our query condition and then finally displaying the results. It also tells us about the number of records that were obtained from the result of the query execution (in this case 209). A bitmap is often used when we have multiple constraints because logical operations can be done very efficiently using bitmaps rather than operating on partially generated result data.