Advanced DB - Homework 2 - pvg2018, sc10670

https://github.com/PranavGrandhi/advDB-HW2

Question 1: Fractal Distribution and Queries

Steps:

1. **Run `distr.py`**: This generates a `fractal_distribution.txt` file.
2. **Run `gen_data.py`**: This creates the `trades.csv` data file.
3. **Run `queries.py`**:
    - The script reads the `trades.csv` file.
    - Executes 4 queries (`query_a`, `query_b`, `query_c`, `query_d`).
    - Saves the results to:
        - `query_a.csv`
        - `query_b.csv`
        - `query_c.csv`
        - `query_d.csv`
    - Outputs the execution times on the command line.

Notes:

- The `typescript` file with logs saved in `crunchy1` is provided in `Queries_Q1.zip`.

Question 2:

Rule of Thumb1

1. **Non-covering clustered index** should give better performance for a multipoint query than a **Non-covering Non-clustered index**.

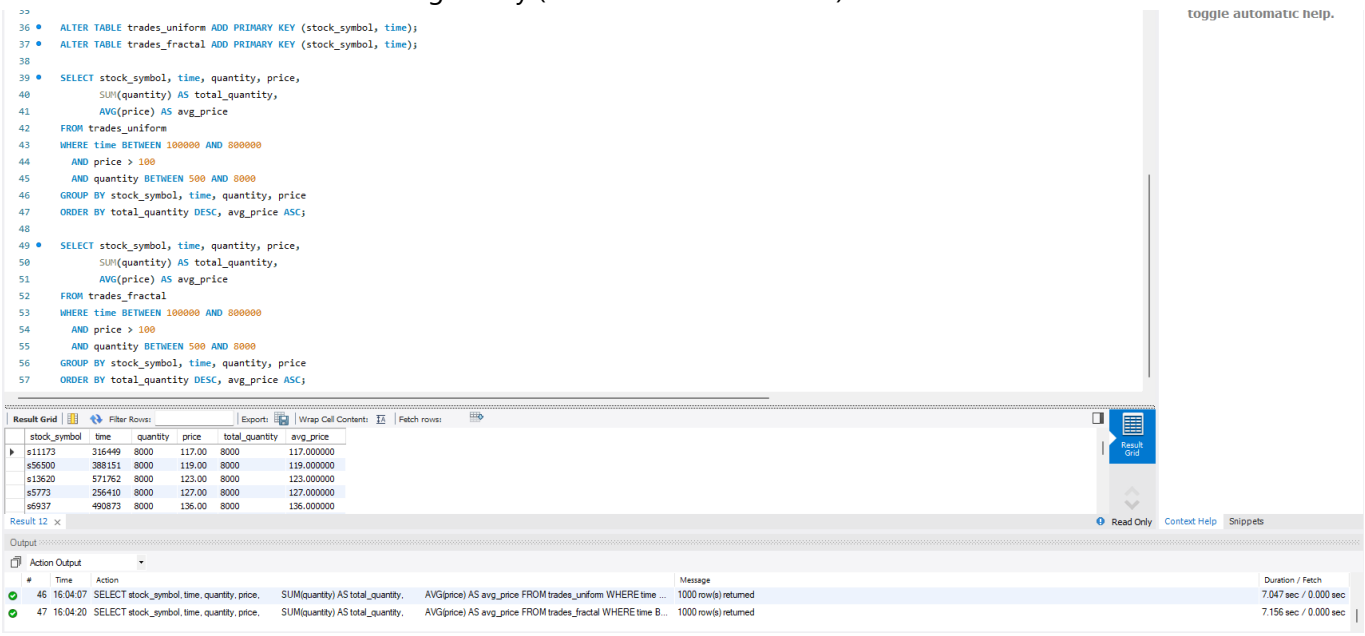
Distributions:

- **Fractal Distribution**: Using the same dataset from Question 1.
- **Uniform Distribution**: Run `gen_uniform_distr.py` to create `uniform_trades.csv`.

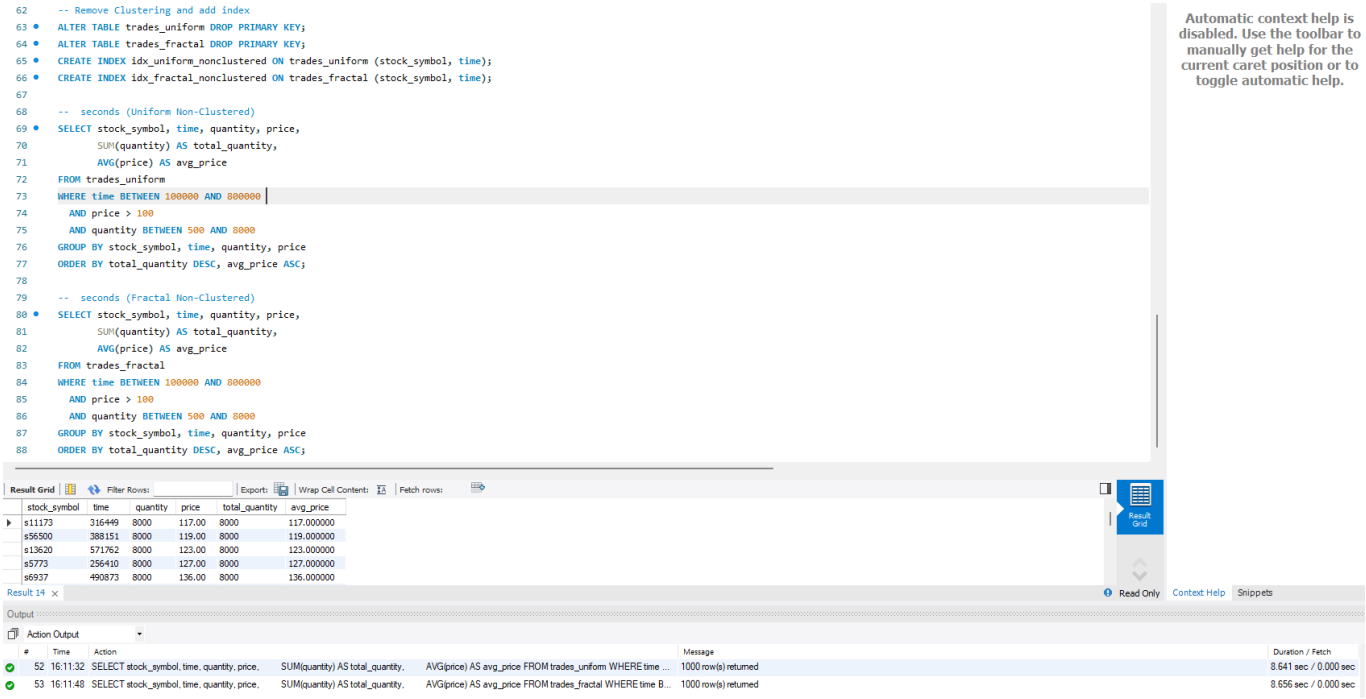
Run thumb1.sql to get these results for Rule of Thumb1 in MySQL

Index Type	Uniform Distribution (seconds)	Fractal Distribution (seconds)
Non-Covering Clustered Index	7.047	7.156
Non-Covering Non-Clustered Index	8.641	8.656
Difference	1.594	1.500

Here are the screenshots of running locally (times are in the bottom) Clustered



Non Clustered



I have included codes for Postgres Queries in the Postgres_Thumb1.s file. These queries are supposed to be run on the shell one by one so as to get the result.

Index Type	Planning Time (Uniform, milliseconds)	Execution Time (Uniform, milliseconds)	Total Time (Uniform, seconds)	Planning Time (Fractal, milliseconds)	Execution Time (Fractal, milliseconds)	Total Time (Fractal, seconds)
Non-Covering Clustered Index	10.149	13547.375	13.557524	3.432	13503.324	13.506756

Index Type	Planning Time (Uniform, milliseconds)	Execution Time (Uniform, milliseconds)	Total Time (Uniform, seconds)	Planning Time (Fractal, milliseconds)	Execution Time (Fractal, milliseconds)	Total Time (Fractal, seconds)
Non-Covering Non-Clustered Index	19.289	14270.868	14.290157	57.844	14237.571	14.295415
Difference			0.732633			0.788659

Below are the runs for the Postgres queries on Crunchy:

Uniform Distribution/Clustered

```
                                                                                QUERY PLAN

------------------------------------------------------------------------------------------------------------------------
---------------------------------------------
 Sort  (cost=316714.13..317923.64 rows=483803 width=63) (actual time=10732.557..12411.067 rows=470037 loops=1)
   Sort Key: (sum(quantity)) DESC, (avg(price))
   Sort Method: external merge  Disk: 25368kB
   ->  Finalize GroupAggregate  (cost=190177.70..252840.31 rows=483803 width=63) (actual time=1702.504..8926.413 rows
=470037 loops=1)
         Group Key: stock_symbol, "time"
         ->  Gather Merge  (cost=190177.70..241753.15 rows=403170 width=63) (actual time=1702.430..5439.377 rows=4700
37 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial GroupAggregate  (cost=189177.68..194217.30 rows=201585 width=63) (actual time=1665.395..32
97.630 rows=156679 loops=3)
                     Group Key: stock_symbol, "time"
                     ->  Sort  (cost=189177.68..189681.64 rows=201585 width=23) (actual time=1665.329..2183.321 rows=
156679 loops=3)
                           Sort Key: stock_symbol, "time"
                           Sort Method: external merge  Disk: 5536kB
                           Worker 0:  Sort Method: external merge  Disk: 5288kB
                           Worker 1:  Sort Method: external merge  Disk: 5312kB
                           ->  Parallel Seq Scan on trades_uniform  (cost=0.00..167280.00 rows=201585 width=23) (actu
al time=0.435..1191.903 rows=156679 loops=3)
                                 Filter: (("time" >= 100000) AND ("time" <= 800000) AND (price > '100'::numeric) AND
(quantity >= 500) AND (quantity <= 8000))
                                 Rows Removed by Filter: 3176654
 Planning Time: 10.149 ms
 Execution Time: 13547.375 ms
(20 rows)
```

Fractal Distribution/Clustered

```
                                                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------------
--------------------------------------------
 Sort  (cost=309575.74..310728.91 rows=461267 width=63) (actual time=10715.727..12529.905 rows=468152 loops=1)
   Sort Key: (sum(quantity)) DESC, (avg(price))
   Sort Method: external merge  Disk: 25264kB
   ->  Finalize GroupAggregate  (cost=189091.76..248835.49 rows=461267 width=63) (actual time=1703.825..8901.576 rows
=468152 loops=1)
         Group Key: stock_symbol, "time"
         ->  Gather Merge  (cost=189091.76..238264.78 rows=384390 width=63) (actual time=1703.760..5446.401 rows=4681
52 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial GroupAggregate  (cost=188091.74..192896.61 rows=192195 width=63) (actual time=1689.259..33
21.099 rows=156051 loops=3)
                     Group Key: stock_symbol, "time"
                     ->  Sort  (cost=188091.74..188572.22 rows=192195 width=23) (actual time=1689.166..2210.687 rows=
156051 loops=3)
                           Sort Key: stock_symbol, "time"
                           Sort Method: external merge  Disk: 5600kB
                           Worker 0:  Sort Method: external merge  Disk: 5376kB
                           Worker 1:  Sort Method: external merge  Disk: 5104kB
                           ->  Parallel Seq Scan on trades_fractal  (cost=0.00..167280.00 rows=192195 width=23) (actu
al time=0.343..1221.364 rows=156051 loops=3)
                                 Filter: (("time" >= 100000) AND ("time" <= 800000) AND (price > '100'::numeric) AND
(quantity >= 500) AND (quantity <= 8000))
                                 Rows Removed by Filter: 3177283
[ Planning Time: 3.432 ms
 Execution Time: 13503.324 ms
(20 rows)
```

Uniform Distribution/Non Clustered

```
                                                                                    QUERY PLAN

----------------------------------------------------------------------------------------------------
---------------------------------------------------------------
 Sort  (cost=317110.75..318299.96 rows=475685 width=63) (actual time=11644.086..13282.170 rows=47
0037 loops=1)
   Sort Key: (sum(quantity)) DESC, (avg(price))
   Sort Method: external merge  Disk: 25368kB
   ->  Finalize GroupAggregate  (cost=189785.44..254369.49 rows=475685 width=63) (actual time=240
2.413..9827.377 rows=470037 loops=1)
         Group Key: stock_symbol, "time", quantity, price
         ->  Gather Merge  (cost=189785.44..241486.36 rows=396404 width=63) (actual time=2402.370
..6207.556 rows=470037 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial GroupAggregate  (cost=188785.42..194731.48 rows=198202 width=63) (actu
al time=2248.428..3961.938 rows=156679 loops=3)
                     Group Key: stock_symbol, "time", quantity, price
                     ->  Sort  (cost=188785.42..189280.92 rows=198202 width=23) (actual time=2248
.363..2835.504 rows=156679 loops=3)
                           Sort Key: stock_symbol, "time", quantity, price
                           Sort Method: external merge  Disk: 5872kB
                           Worker 0:  Sort Method: external merge  Disk: 5256kB
                           Worker 1:  Sort Method: external merge  Disk: 5016kB
                           ->  Parallel Seq Scan on trades_uniform  (cost=0.00..167280.00 rows=19
8202 width=23) (actual time=12.770..1136.336 rows=156679 loops=3)
                                 Filter: (("time" >= 100000) AND ("time" <= 800000) AND (price >
'100'::numeric) AND (quantity >= 500) AND (quantity <= 8000))
                                 Rows Removed by Filter: 3176654
 Planning Time: 19.289 ms
 Execution Time: 14270.868 ms
(20 rows)
```

Fractal Distribution/ Non Clustered

```
                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=316478.58..317662.91 rows=473730 width=63) (actual time=11366.936..13050.872 rows=468152 loops=1)
   Sort Key: (sum(quantity)) DESC, (avg(price))
   Sort Method: external merge  Disk: 25264kB
   ->  Finalize GroupAggregate  (cost=189690.46..254009.26 rows=473730 width=63) (actual time=2355.243..9558.719 rows=468152 loops=1)
         Group Key: stock_symbol, "time", quantity, price
         ->  Gather Merge  (cost=189690.46..241179.05 rows=394776 width=63) (actual time=2355.186..5963.823 rows=468152 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Partial GroupAggregate  (cost=188690.44..194612.08 rows=197388 width=63) (actual time=2240.829..3951.350 rows=156051 loops=3)
                     Group Key: stock_symbol, "time", quantity, price
                     ->  Sort  (cost=188690.44..189183.91 rows=197388 width=23) (actual time=2240.769..2829.924 rows=156051 loops=3)
                           Sort Key: stock_symbol, "time", quantity, price
                           Sort Method: external merge  Disk: 5376kB
                           Worker 0:  Sort Method: external merge  Disk: 5376kB
                           Worker 1:  Sort Method: external merge  Disk: 5328kB
                           ->  Parallel Seq Scan on trades_fractal  (cost=0.00..167280.00 rows=197388 width=23) (actual time=7.677..1144.793 rows=156051 loops=3)
                                 Filter: (("time" >= 100000) AND ("time" <= 800000) AND (price > '100'::numeric) AND (quantity >= 500) AND (quantity <= 8000))
                                 Rows Removed by Filter: 3177283
 Planning Time: 57.844 ms
 Execution Time: 14237.571 ms
(20 rows)
```

Rule of Thumb2

1. **Indexes** are most effective when they have high **selectivity** (i.e., the indexed column contains many unique values).

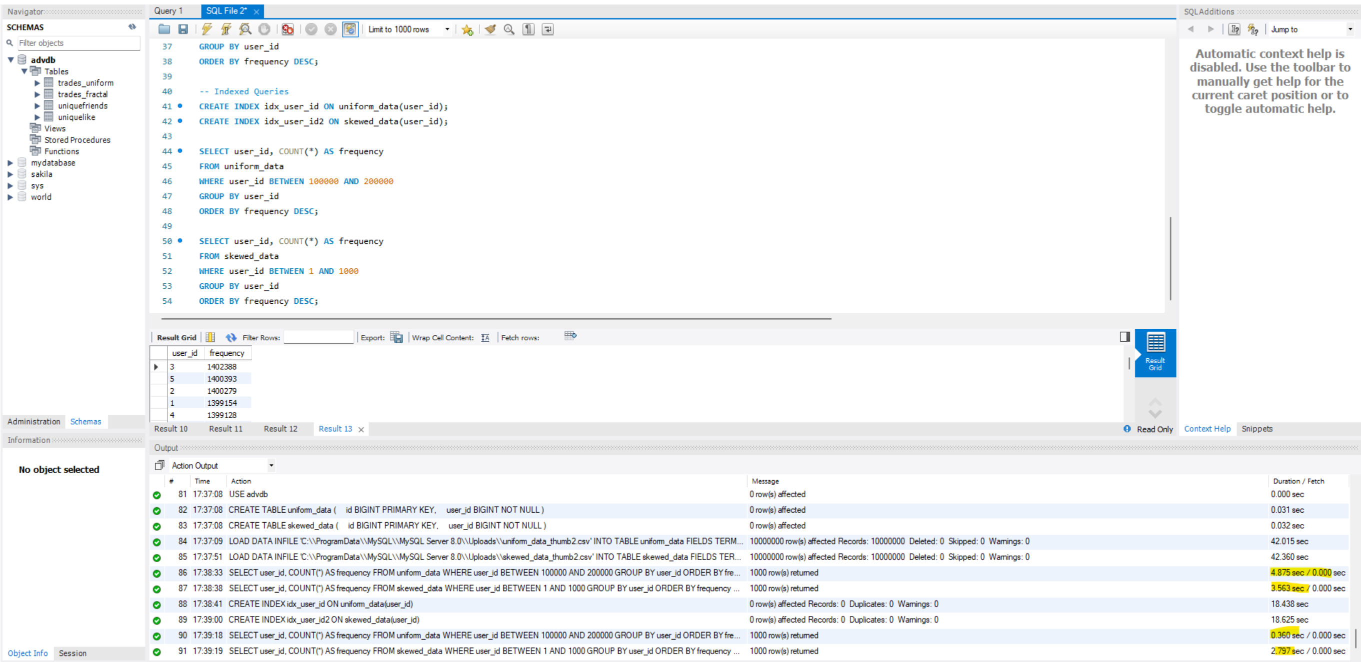
Distributions:

- **Uniform Distribution**: Uniformly distributed dataset of size 10 million numbers.
- **Skewed Distribution**: Skewed distribution of data where 70% of the numbers are between 1-5, 20% of the data are between 6-15, and 10% of the data in 16-1000
- Run `gen_Data_thumb2.py` to create `uniform_data_thumb2.csv` and `skewed_data_thumb2.csv`.

Run thumb2.sql to get these results for Rule of Thumb2 in MySQL

Index Type	Uniform Distribution (seconds)	Skewed Distribution (seconds)
With Indexing	0.360	2.797
Wihtout Indexing	4.875	3.563
Difference	4.515	0.766

Here are the screenshots of running locally (times are in the bottom)



I have included codes for Postgres Queries in the Postgres_Thumb2.s file. These queries are supposed to be run on the shell one by one so as to get the result.

Index Type	Planning Time (Uniform, milliseconds)	Execution Time (Uniform, milliseconds)	Total Time (Uniform, seconds)	Planning Time (Skewed, milliseconds)	Execution Time (Skewed, milliseconds)	Total Time (Skewed, seconds)
With Indexing	2.438	5648.564	5.651002	1.903	19941.123	19.943026
Without Indexing	19.289	9148.476	9.167762	1.474	22200.418	22.201892
Difference			3.51676			2.258866

Uniform Distribution/Including Indexing

```
sc10670=# CREATE INDEX idx_user_id ON uniform_data(user_id);
CREATE INDEX
sc10670=# CREATE INDEX idx_user_id2 ON skewed_data(user_id);
CREATE INDEX
sc10670=# EXPLAIN ANALYZE
SELECT user_id, COUNT(*) AS frequency
FROM uniform_data
WHERE user_id BETWEEN 100000 AND 200000
GROUP BY user_id
ORDER BY frequency DESC;
                                                                          QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------
------------------------------
 Sort  (cost=106332.73..107886.60 rows=621547 width=16) (actual time=5229.736..5448.095 rows=99997 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: external merge  Disk: 2552kB
   ->  GroupAggregate  (cost=0.43..35900.31 rows=621547 width=16) (actual time=0.978..4964.843 rows=99997 loops=1)
         Group Key: user_id
         ->  Index Only Scan using idx_user_id on uniform_data  (cost=0.43..24659.96 rows=1004976 width=8) (actual time=0.928.
.2431.084 rows=999158 loops=1)
               Index Cond: ((user_id >= 100000) AND (user_id <= 200000))
               Heap Fetches: 0
 Planning Time: 2.438 ms
 Execution Time: 5648.564 ms
(10 rows)

sc10670=# EXPLAIN ANALYZE
SELECT user_id, COUNT(*) AS frequency
FROM skewed_data
WHERE user_id BETWEEN 1 AND 1000
GROUP BY user_id
ORDER BY frequency DESC;
```

Skewed Distribution/Without Indexing

```
sc10670=# EXPLAIN ANALYZE
SELECT user_id, COUNT(*) AS frequency
FROM skewed_data
WHERE user_id BETWEEN 1 AND 1000
GROUP BY user_id
ORDER BY frequency DESC;
                                                              QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------
------------------------------------
 Sort  (cost=138732.89..138735.27 rows=951 width=16) (actual time=19936.806..19939.001 rows=1000 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: quicksort  Memory: 71kB
   ->  Finalize GroupAggregate  (cost=138444.91..138685.85 rows=951 width=16) (actual time=19912.477..19934.356 rows=1000 loop
s=1)
         Group Key: user_id
         ->  Gather Merge  (cost=138444.91..138666.83 rows=1902 width=16) (actual time=19912.433..19924.414 rows=3000 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Sort  (cost=137444.89..137447.27 rows=951 width=16) (actual time=19900.879..19903.455 rows=1000 loops=3)
                     Sort Key: user_id
                     Sort Method: quicksort  Memory: 71kB
                     Worker 0:  Sort Method: quicksort  Memory: 71kB
                     Worker 1:  Sort Method: quicksort  Memory: 71kB
                     ->  Partial HashAggregate  (cost=137388.33..137397.85 rows=951 width=16) (actual time=19894.984..19897.77
2 rows=1000 loops=3)
                           Group Key: user_id
                           Batches: 1  Memory Usage: 193kB
                           Worker 0:  Batches: 1  Memory Usage: 193kB
                           Worker 1:  Batches: 1  Memory Usage: 193kB
                           ->  Parallel Seq Scan on skewed_data  (cost=0.00..116555.00 rows=4166667 width=8) (actual time=72.7
80..10166.368 rows=3333333 loops=3)
                                 Filter: ((user_id >= 1) AND (user_id <= 1000))
 Planning Time: 1.903 ms
 Execution Time: 19941.123 ms
(22 rows)
```

Uniform Distribution/Including Indexing

```
sc10670=# EXPLAIN ANALYZE
SELECT user_id, COUNT(*) AS frequency
FROM uniform_data
WHERE user_id BETWEEN 100000 AND 200000
GROUP BY user_id
ORDER BY frequency DESC;
                                                              QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------
--------
 Sort  (cost=345080.72..346634.58 rows=621543 width=16) (actual time=8314.985..8535.101 rows=99997 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: external merge  Disk: 2552kB
   ->  HashAggregate  (cost=260582.01..274648.71 rows=621543 width=16) (actual time=7567.132..8051.232 rows=99997 loops=1)
         Group Key: user_id
         Planned Partitions: 16  Batches: 17  Memory Usage: 4241kB  Disk Usage: 30896kB
         ->  Seq Scan on uniform_data  (cost=0.00..204052.90 rows=1004962 width=8) (actual time=91.263..4612.659 rows=999158 l
oops=1)
               Filter: ((user_id >= 100000) AND (user_id <= 200000))
               Rows Removed by Filter: 9000842
 Planning Time: 19.286 ms
 Execution Time: 9148.476 ms
(11 rows)

sc10670=# EXPLAIN ANALYZE
SELECT user_id, COUNT(*) AS frequency
FROM skewed_data
WHERE user_id BETWEEN 1 AND 1000
GROUP BY user_id
ORDER BY frequency DESC;
```

Skewed Distribution/ Without Indexing

```
sc10670=# EXPLAIN ANALYZE
SELECT user_id, COUNT(*) AS frequency
FROM skewed_data
WHERE user_id BETWEEN 1 AND 1000
GROUP BY user_id
ORDER BY frequency DESC;
                                                                            QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------
----------------------------------
 Sort  (cost=138733.93..138736.31 rows=951 width=16) (actual time=19351.391..22198.332 rows=1000 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: quicksort  Memory: 71kB
   ->  Finalize GroupAggregate  (cost=138445.95..138686.89 rows=951 width=16) (actual time=19326.888..22193.614 rows=1000 loop
s=1)
         Group Key: user_id
         ->  Gather Merge  (cost=138445.95..138667.87 rows=1902 width=16) (actual time=19326.848..22183.699 rows=3000 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Sort  (cost=137445.93..137448.30 rows=951 width=16) (actual time=19294.207..19296.723 rows=1000 loops=3)
                     Sort Key: user_id
                     Sort Method: quicksort  Memory: 71kB
                     Worker 0:  Sort Method: quicksort  Memory: 71kB
                     Worker 1:  Sort Method: quicksort  Memory: 71kB
                     ->  Partial HashAggregate  (cost=137389.37..137398.88 rows=951 width=16) (actual time=19288.305..19291.09
6 rows=1000 loops=3)
                           Group Key: user_id
                           Batches: 1  Memory Usage: 193kB
                           Worker 0:  Batches: 1  Memory Usage: 193kB
                           Worker 1:  Batches: 1  Memory Usage: 193kB
                           ->  Parallel Seq Scan on skewed_data  (cost=0.00..116556.09 rows=4166656 width=8) (actual time=0.36
5..9621.322 rows=3333333 loops=3)
                                 Filter: ((user_id >= 1) AND (user_id <= 1000))
 Planning Time: 1.474 ms
 Execution Time: 22200.418 ms
(22 rows)
```

Modifications to Rule of Thumbs that dont depend on data disributions

Rule of Thumb1

Original Rule: Non-covering clustered index should give better performance for a multipoint query than a non-covering non-clustered index.

Modified Rule: Using a covering index for multipoint queries provides consistent performance benefits over non-covering indexes, regardless of whether the index is clustered or non-clustered and independent of data distribution.

Rule of Thumb2

Original Rule: Indexes are most effective when they have high selectivity (i.e., the indexed column contains many unique values).

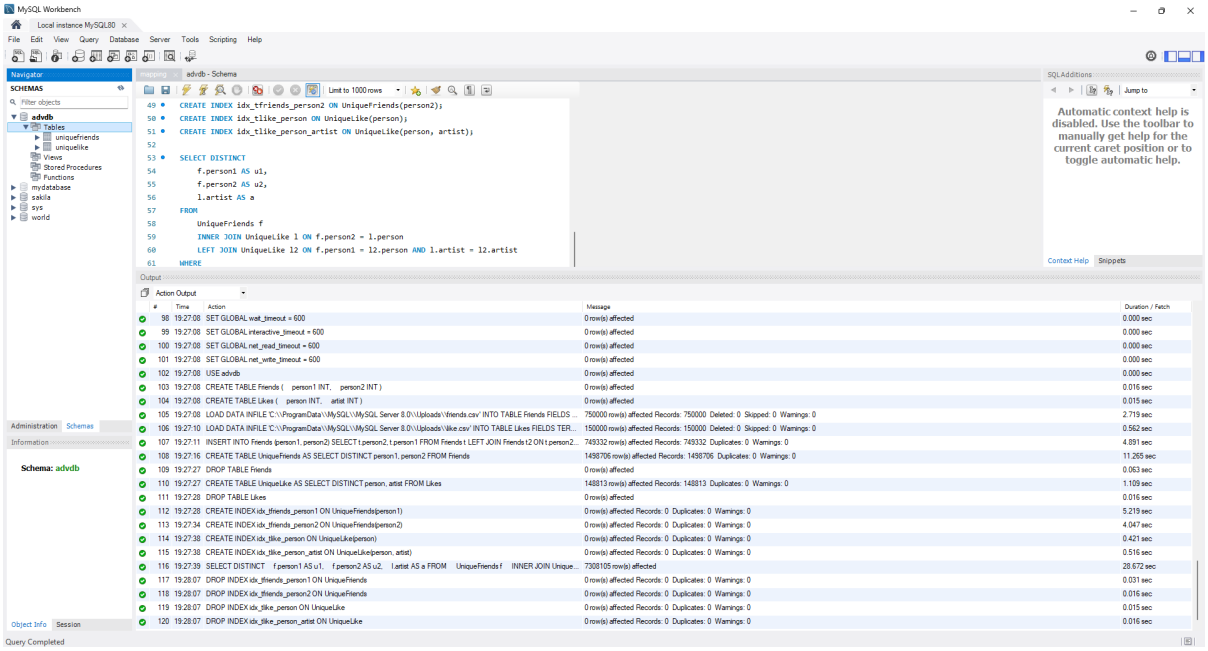
Modified Rule: Indexes are most effective when used in queries with highly selective predicates—that is, when queries filter out a large portion of the data—regardless of the uniqueness or distribution of values in the indexed column.

Question 3: Friends and Likes Queries in MySQL

Steps:

1. Download `friends.csv` and `like.csv` from the course website.
2. Run the `mapping.sql` script:
    - Ensure to update the file locations for `friends.csv` and `like.csv` in the script.
    - The script generates `result.csv` (output location can be modified in the script).
3. Verify Results:
    - Queries were tested locally using MySQL.

- A screenshot of the query execution is attached. ---



Files:

- `Result.zip`: Contains the zipped output result file (`result.csv`).

Attachments:

- `Queries_Q1.zip`: Contains the typescript file from Question 1.
- `Result.zip`: Contains the output file for Question 3.