

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

```

36 • ALTER TABLE trades_uniform ADD PRIMARY KEY (stock_symbol, time);
37 • ALTER TABLE trades_fractal ADD PRIMARY KEY (stock_symbol, time);
38
39 • SELECT stock_symbol, time, quantity, price,
40     SUM(quantity) AS total_quantity,
41     AVG(price) AS avg_price
42 FROM trades_uniform
43 WHERE time BETWEEN 100000 AND 800000
44     AND price > 100
45     AND quantity BETWEEN 500 AND 8000
46 GROUP BY stock_symbol, time, quantity, price
47 ORDER BY total_quantity DESC, avg_price ASC;
48
49 • SELECT stock_symbol, time, quantity, price,
50     SUM(quantity) AS total_quantity,
51     AVG(price) AS avg_price
52 FROM trades_fractal
53 WHERE time BETWEEN 100000 AND 800000
54     AND price > 100
55     AND quantity BETWEEN 500 AND 8000
56 GROUP BY stock_symbol, time, quantity, price
57 ORDER BY total_quantity DESC, avg_price ASC;

```

stock_symbol	time	quantity	price	total_quantity	avg_price
s11173	31649	8000	117.00	8000	117.000000
s56500	388151	8000	119.00	8000	119.000000
s13620	571762	8000	123.00	8000	123.000000
s5773	256410	8000	127.00	8000	127.000000
s6937	490873	8000	136.00	8000	136.000000

Result 12 x

Output

Action Output

#	Time	Action	Message	Duration / Fetch
46	16:04:07	SELECT stock_symbol, time, quantity, price, SUM(quantity) AS total_quantity, AVG(price) AS avg_price FROM trades_uniform WHERE time ...	1000 row(s) returned	7.047 sec / 0.000 sec
47	16:04:20	SELECT stock_symbol, time, quantity, price, SUM(quantity) AS total_quantity, AVG(price) AS avg_price FROM trades_fractal WHERE time ...	1000 row(s) returned	7.156 sec / 0.000 sec

Non Clustered

```

62     -- Remove Clustering and add index
63 • ALTER TABLE trades_uniform DROP PRIMARY KEY;
64 • ALTER TABLE trades_fractal DROP PRIMARY KEY;
65 • CREATE INDEX idx_uniform_nonclustered ON trades_uniform (stock_symbol, time);
66 • CREATE INDEX idx_fractal_nonclustered ON trades_fractal (stock_symbol, time);
67
68     -- seconds (Uniform Non-Clustered)
69 • SELECT stock_symbol, time, quantity, price,
70     SUM(quantity) AS total_quantity,
71     AVG(price) AS avg_price
72 FROM trades_uniform
73 WHERE time BETWEEN 100000 AND 800000
74     AND price > 100
75     AND quantity BETWEEN 500 AND 8000
76 GROUP BY stock_symbol, time, quantity, price
77 ORDER BY total_quantity DESC, avg_price ASC;
78
79     -- seconds (Fractal Non-Clustered)
80 • SELECT stock_symbol, time, quantity, price,
81     SUM(quantity) AS total_quantity,
82     AVG(price) AS avg_price
83 FROM trades_fractal
84 WHERE time BETWEEN 100000 AND 800000
85     AND price > 100
86     AND quantity BETWEEN 500 AND 8000
87 GROUP BY stock_symbol, time, quantity, price
88 ORDER BY total_quantity DESC, avg_price ASC;

```

stock_symbol	time	quantity	price	total_quantity	avg_price
s11173	31649	8000	117.00	8000	117.000000
s56500	388151	8000	119.00	8000	119.000000
s13620	571762	8000	123.00	8000	123.000000
s5773	256410	8000	127.00	8000	127.000000
s6937	490873	8000	136.00	8000	136.000000

Result 14 x

Output

Action Output

#	Time	Action	Message	Duration / Fetch
52	16:11:32	SELECT stock_symbol, time, quantity, price, SUM(quantity) AS total_quantity, AVG(price) AS avg_price FROM trades_uniform WHERE time ...	1000 row(s) returned	8.641 sec / 0.000 sec
53	16:11:48	SELECT stock_symbol, time, quantity, price, SUM(quantity) AS total_quantity, AVG(price) AS avg_price FROM trades_fractal WHERE time ...	1000 row(s) returned	8.656 sec / 0.000 sec

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.

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

	Planning	Execution	Total	Planning	Execution	Total
Index	Time	Time	Time	Time	Time	Time
Type	(Uniform, milliseconds)	(Uniform, milliseconds)	(Uniform, seconds)	(Fractal, milliseconds)	(Fractal, milliseconds)	(Fractal, seconds)
Non-Covering 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=470037 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial GroupAggregate  (cost=189177.68..194217.30 rows=201585 width=63) (actual time=1665.395..3297.630 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) (actual 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=468152 loops=1)
          Workers Planned: 2
          Workers Launched: 2
            -> Partial GroupAggregate (cost=188091.74..192896.61 rows=192195 width=63) (actual time=1689.259..3321.099 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) (actual 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=470037 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=2402.413..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) (actual 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=198202 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..254099.26 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

- 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
Without Indexing	4.875	3.563
Difference	4.515	0.766

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

```

37 GROUP BY user_id
38 ORDER BY frequency DESC;
39
-- Indexed Queries
40
41 • CREATE INDEX idx_user_id ON uniform_data(user_id);
42 • CREATE INDEX idx_user_id2 ON skewed_data(user_id);
43
44 • SELECT user_id, COUNT(*) AS frequency
45 FROM uniform_data
46 WHERE user_id BETWEEN 100000 AND 200000
47 GROUP BY user_id
48 ORDER BY frequency DESC;
49
50 • SELECT user_id, COUNT(*) AS frequency
51 FROM skewed_data
52 WHERE user_id BETWEEN 1 AND 1000
53 GROUP BY user_id
54 ORDER BY frequency DESC;

```

user_id	frequency
3	1402388
5	1400393
2	1400279
1	1399154
4	1399128

Action	Time	Action	Message	Duration / Fetch
81	17:37:00	USE advdb	0 row(s) affected	0.000 sec
82	17:37:00	CREATE TABLE uniform_data (`id` BIGINT PRIMARY KEY, `user_id` BIGINT NOT NULL)	0 row(s) affected	0.031 sec
83	17:37:00	CREATE TABLE skewed_data (`id` BIGINT PRIMARY KEY, `user_id` BIGINT NOT NULL)	0 row(s) affected	0.032 sec
84	17:37:09	LOAD DATA INFILE C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\uniform_data_thumb2.csv INTO TABLE uniform_data FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\n';	100000000 row(s) affected Records: 100000000 Deleted: 0 Skipped: 0 Warnings: 0	42.015 sec
85	17:37:51	LOAD DATA INFILE C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\skewed_data_thumb2.csv INTO TABLE skewed_data FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\n';	100000000 row(s) affected Records: 100000000 Deleted: 0 Skipped: 0 Warnings: 0	42.360 sec
86	17:38:33	SELECT user_id, COUNT(*) AS frequency FROM uniform_data WHERE user_id BETWEEN 100000 AND 200000 GROUP BY user_id ORDER BY frequency DESC;	1000 row(s) returned	4.075 sec / 0.000 sec
87	17:38:38	SELECT user_id, COUNT(*) AS frequency FROM skewed_data WHERE user_id BETWEEN 1 AND 1000 GROUP BY user_id ORDER BY frequency DESC;	1000 row(s) returned	3.563 sec / 0.000 sec
88	17:39:41	CREATE INDEX idx_user_id ON uniform_data(user_id)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	18.438 sec
89	17:39:40	CREATE INDEX idx_user_id2 ON skewed_data(user_id)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	18.625 sec
90	17:39:18	SELECT user_id, COUNT(*) AS frequency FROM uniform_data WHERE user_id BETWEEN 100000 AND 200000 GROUP BY user_id ORDER BY frequency DESC;	1000 row(s) returned	0.969 sec / 0.000 sec
91	17:39:19	SELECT user_id, COUNT(*) AS frequency FROM skewed_data WHERE user_id BETWEEN 1 AND 1000 GROUP BY user_id ORDER BY frequency DESC;	1000 row(s) returned	2.797 sec / 0.000 sec

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.

	Planning	Execution	Total	Planning	Execution	Total
Index	Time	Time	Time	Time	Time	Time
Type	(Uniform, milliseconds)	(Uniform, milliseconds)	(Uniform, seconds)	(Skewed, milliseconds)	(Skewed, milliseconds)	(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 distributions

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. ---

```

CREATE INDEX idx_friends_person2 ON UniqueFriends(person2);
CREATE INDEX idx_like_person ON UniqueLike(person);
CREATE INDEX idx_like_person_artist ON UniqueLike(person, artist);

SELECT DISTINCT
    f.person1 AS u1,
    f.person2 AS u2,
    l.artist AS a
FROM
    UniqueFriends f
        INNER JOIN UniqueLike l ON f.person2 = l.person
        LEFT JOIN UniqueLike ll ON f.person1 = ll.person AND l.artist = ll.artist
WHERE
    u1 < u2
    AND ll.artist IS NULL;
    
```

Output

Action	Time	Action	Message	Duration / Fetch
98	19.27.08	SET GLOBAL wait_timeout = 600	0 rows(affected)	0.000 sec
99	19.27.08	SET GLOBAL interactive_timeout = 600	0 rows(affected)	0.000 sec
100	19.27.08	SET GLOBAL net_read_timeout = 600	0 rows(affected)	0.000 sec
101	19.27.08	SET GLOBAL net_write_timeout = 600	0 rows(affected)	0.000 sec
102	19.27.08	USE advb	0 rows(affected)	0.000 sec
103	19.27.08	CREATE TABLE Friends (person1 INT, person2 INT)	0 rows(affected)	0.016 sec
104	19.27.08	CREATE TABLE Likes (person INT, artist INT)	0 rows(affected)	0.015 sec
105	19.27.08	LOAD DATA INFILE 'C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\Friends.csv' INTO TABLE Friends FIELDS ...	750000 rows(affected) Records: 750000 Deleted: 0 Skipped: 0 Warnings: 0	2.719 sec
106	19.27.10	LOAD DATA INFILE 'C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\Likes.csv' INTO TABLE Likes FIELDS TER...	150000 rows(affected) Records: 150000 Deleted: 0 Skipped: 0 Warnings: 0	0.562 sec
107	19.27.11	INSERT INTO Friends (person1, person2) SELECT person2, person1 FROM Friends t1 LEFT JOIN Friends t2 ON t1.person2 = t2.person1	749332 rows(affected) Records: 749332 Duplicates: 0 Warnings: 0	4.891 sec
108	19.27.16	CREATE TABLE UniqueFriends AS SELECT DISTINCT person1, person2 FROM Friends	149870 rows(affected) Records: 149870 Duplicates: 0 Warnings: 0	11.265 sec
109	19.27.27	DROP TABLE Friends	0 rows(affected)	0.063 sec
110	19.27.27	CREATE TABLE UniqueLike AS SELECT DISTINCT person, artist FROM Likes	148813 rows(affected) Records: 148813 Duplicates: 0 Warnings: 0	1.109 sec
111	19.27.28	DROP TABLE Likes	0 rows(affected)	0.016 sec
112	19.27.28	CREATE INDEX idx_friends_person1 ON UniqueFriends(person1)	0 rows(affected) 0 Duplicates: 0 Warnings: 0	0.519 sec
113	19.27.34	CREATE INDEX idx_friends_person2 ON UniqueFriends(person2)	0 rows(affected) 0 Duplicates: 0 Warnings: 0	4.047 sec
114	19.27.38	CREATE INDEX idx_like_person ON UniqueLike(person)	0 rows(affected) 0 Duplicates: 0 Warnings: 0	0.421 sec
115	19.27.38	CREATE INDEX idx_like_person_artist ON UniqueLike(person, artist)	0 rows(affected) 0 Duplicates: 0 Warnings: 0	0.516 sec
116	19.27.39	SELECT DISTINCT (person1 AS u1, person2 AS u2, artist AS a) FROM UniqueFriends t1 INNER JOIN Unique...	730810 rows(affected)	28.672 sec
117	19.28.07	DROP INDEX idx_friends_person1 ON UniqueFriends	0 rows(affected) 0 Duplicates: 0 Warnings: 0	0.031 sec
118	19.28.07	DROP INDEX idx_friends_person2 ON UniqueFriends	0 rows(affected) 0 Duplicates: 0 Warnings: 0	0.016 sec
119	19.28.07	DROP INDEX idx_like_person ON UniqueLike	0 rows(affected) 0 Duplicates: 0 Warnings: 0	0.015 sec
120	19.28.07	DROP INDEX idx_like_person_artist ON UniqueLike	0 rows(affected) 0 Duplicates: 0 Warnings: 0	0.016 sec

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.