HW2 Q2

1. PostgreSQL Suggestion 1: Clustered Index

For PostgreSQL, execution times are recorded by the built in `\timing` functionality.

Suggestion Effective Case: Range scan intensive access pattern

Uses the `datagen1.py` to generate a dataset. For this case, we use the range scan query

```
SELECT * FROM mytable WHERE id BETWEEN 3000 AND 3100;
```

Then create the table of no clustered indexing applied by `CREATE TABLE mytable (id INT, value INT );` And then `COPY mytable FROM '/users/yliang/test_data1.csv' DELIMITER ',' CSV HEADER;`.

Then we use **Clustered Index** by firstly creating an index on a new copy of the table, `CREATE INDEX idx_mytable_id ON mytable2 (id);` Then use the command `CLUSTER mytable2 USING idx_mytable_id;` to cluster the id explicitly. Then we also load data to `mytable2`.

Now, `mytable` does not use clustered index as `mytable2` uses it. And when we execute the same query, the results are

```
postgres=# SELECT * FROM mytable WHERE id BETWEEN 3000 AND 3100;
Time: 138.168 ms
postgres=# SELECT * FROM mytable2 WHERE id BETWEEN 3000 AND 3100;
Time: 6.459 ms
```

We can see that there is **a huge improvement** on the performance.

Suggestion Ineffective Case: Discrete insert intensive access pattern

We firstly use `datagen2.py` to generate some **non-consecutive data for insertion** to our original tables.

This time, we use the insertion query

```
COPY mytable FROM '/users/yliang/test_data2.csv' DELIMITER ',' CSV
HEADER;
```

and the results are

```
postgres=# COPY mytable FROM '/users/yliang/test_data2.csv' DELIMITER ',' CSV HEADER;
COPY 100000
Time: 79.419 ms
postgres=# COPY mytable2 FROM '/users/yliang/test_data2.csv' DELIMITER ',' CSV HEADER;
COPY 100000
Time: 731.540 ms
```

We can see that, this time, the clustered indexed table `mytable2` has a much lower performance, it's 10x slower than not using the suggestion. So if the user access pattern consists of frequent non-consecutive insertions, then using clustered indexing has adverse effects.

2. PosgreSQL Suggestion 2: Larger RAM Buffer Size

For this rule, we can change the ram buffer size by changing the `sahred_buffers` parameter in the config file.

Suggestion Effecive Case: Most access to Popular Data

Now when the buffer size is only 128kB. Even after repeated querrying the same "popular" data, the execution time does not decrease.

```
postgres=# SELECT * FROM mytable WHERE id BETWEEN 3000 AND 9000;
Time: 162.448 ms
postgres=# SELECT * FROM mytable WHERE id BETWEEN 3000 AND 9000;
Time: 170.591 ms
```

But when we increase the buffer size to 2GB.

```
postgres=# SELECT * FROM mytable WHERE id BETWEEN 3000 AND 9000;
Time: 120.164 ms
postgres=# SELECT * FROM mytable WHERE id BETWEEN 3000 AND 9000;
Time: 98.148 ms
postgres=# SELECT * FROM mytable WHERE id BETWEEN 3000 AND 9000;
Time: 99.425 ms
```

This time, we can see after executing it several times, accessing the popular data becomes much faster and stable, this is because we have a larger buffer.

Suggestion Ineffective Case: Accessing in a uniform distribution

In this case there is no "popular" data becuase each query accesses different segments of data scattered in the database. We give some queries that don't have popular data:

```
SELECT * FROM mytable
WHERE (id BETWEEN 30 AND 40)
   OR (id BETWEEN 1000 AND 1100)
   OR (id BETWEEN 2000 AND 2100)
   OR (id BETWEEN 9000 AND 9100);

SELECT * FROM mytable
WHERE (id BETWEEN 50 AND 150)
   OR (id BETWEEN 600 AND 700)
   OR (id BETWEEN 2500 AND 2600)
   OR (id BETWEEN 8000 AND 8100);
```

```
SELECT * FROM mytable
WHERE (id BETWEEN 100 AND 200)
   OR (id BETWEEN 500 AND 550)
   OR (id BETWEEN 3000 AND 3100)
   OR (id BETWEEN 9500 AND 9600);
```

With a tiny `128kB` buffer, the results are

```
postgres=# SELECT * FROM mytable
postgres-# WHERE (id BETWEEN 30 AND 40)
postgres-#    OR (id BETWEEN 1000 AND 1100)
postgres-#    OR (id BETWEEN 2000 AND 2100)
postgres-#    OR (id BETWEEN 9000 AND 9100);
Time: 126.469 ms
postgres=# SELECT * FROM mytable
postgres-# WHERE (id BETWEEN 50 AND 150)
postgres-#    OR (id BETWEEN 600 AND 700)
postgres-#    OR (id BETWEEN 2500 AND 2600)
postgres-#    OR (id BETWEEN 8000 AND 8100);
Time: 122.407 ms
postgres=# SELECT * FROM mytable
postgres-# WHERE (id BETWEEN 100 AND 200)
postgres-#    OR (id BETWEEN 500 AND 550)
postgres-#    OR (id BETWEEN 3000 AND 3100)
postgres-#    OR (id BETWEEN 9500 AND 9600);
Time: 120.598 ms
```

With a increased buffer of `2GB` mimicing the decision that we bought more RAM,

```
postgres=# SELECT * FROM mytable
postgres-# WHERE (id BETWEEN 30 AND 40)
postgres-#    OR (id BETWEEN 1000 AND 1100)
postgres-#    OR (id BETWEEN 2000 AND 2100)
postgres-#    OR (id BETWEEN 9000 AND 9100);
```

```
Time: 142.503 ms
postgres=# SELECT * FROM mytable
postgres-# WHERE (id BETWEEN 50 AND 150)
postgres-#    OR (id BETWEEN 600 AND 700)
postgres-#    OR (id BETWEEN 2500 AND 2600)
postgres-#    OR (id BETWEEN 8000 AND 8100);
Time: 132.034 ms
postgres=# SELECT * FROM mytable
postgres-# WHERE (id BETWEEN 100 AND 200)
postgres-#    OR (id BETWEEN 500 AND 550)
postgres-#    OR (id BETWEEN 3000 AND 3100)
postgres-#    OR (id BETWEEN 9500 AND 9600);
Time: 126.391 ms
```

Looking at the execution time, we can see that there is **no improving performance** after several queries, because all the queries are pretty disjoint when there is no "hot data", and thus the larger buffer would be that much useful if the access is uniformly distributed to access the stuff in the entire DB.

3. SQLite Suggestion 1: Clustered Index

To show with and witout clustered index, we use normal table creation for without, and use primary key creation for with. Original table data is generated by `datagen3.py`

Positive Case: Accessing patterns are range scans

So we use the range query

```
SELECT * FROM mytable WHERE id BETWEEN 3000 AND 3100;
```

And the results

```
Without Clustered Indexing:
Run Time: real 0.801 user 0.732282 sys 0.068277
With Clustered Indexing:
Run Time: real 0.001 user 0.000000 sys 0.001114
```

We can see that **there is an improvement of a order of magnitude** here when clustered indexing is enabled.

Negative Case: Accessing patterns involves non-consecutive inserts

We use the queries to insert discrete data, which cause a lot of rearranging in the database, so the performance of clustered indexing will be slower. The results are

```
Without Clustered Indexing:
~0.1 second
With Clustered Indexing:
~7 seconds
```

4. SQLite Suggestion 2: Larger RAM Buffer Size

The results are very similar to what we had for postgres for exactly the same reasons.