Terminal Information:

```
riceanj@cloudshell:~ (cosmic-talent-364620)$ gcloud sql connect musicreal --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 200033
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use test;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------+
| Tables_in_test |
+----------------+
| Login          |
| Post           |
| Profile        |
| Song           |
| UserRef        |
+----------------+
5 rows in set (0.01 sec)

mysql>
```

DDL Commands:

```
CREATE TABLE `Login` (
  `userName` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  PRIMARY KEY (`userName`),
  UNIQUE KEY `username_UNIQUE` (`userName`),
  CONSTRAINT `userName` FOREIGN KEY (`userName`) REFERENCES `Profile`
(`userName`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

CREATE TABLE `Post` (
  `postId` int NOT NULL,
  `time` date NOT NULL,
  `songId` varchar(255) NOT NULL,
  `userName` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`postId`),
```

```
  KEY `userName_post_idx` (`userName`) /*!80000 INVISIBLE */,
  KEY `song_post_idx` (`songId`),
  CONSTRAINT `song` FOREIGN KEY (`songId`) REFERENCES `Song` (`songId`),
  CONSTRAINT `userName_post` FOREIGN KEY (`userName`) REFERENCES `Profile`
(`userName`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

CREATE TABLE `Profile` (
  `userName` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  PRIMARY KEY (`userName`),
  UNIQUE KEY `userName_UNIQUE` (`userName`),
  UNIQUE KEY `email_UNIQUE` (`email`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

CREATE TABLE `Song` (
  `songId` varchar(255) NOT NULL,
  `title` varchar(255) NOT NULL,
  `artist` varchar(255) NOT NULL,
  `plays` int DEFAULT NULL,
  PRIMARY KEY (`songId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

CREATE TABLE `UserRef` (
  `userToken` varchar(255) NOT NULL,
  `userName` varchar(45) NOT NULL,
  PRIMARY KEY (`userToken`),
  KEY `userNameRef_idx` (`userName`),
  CONSTRAINT `userNameRef` FOREIGN KEY (`userName`) REFERENCES `Profile`
(`userName`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Proof of 1000 Rows (count query screen shot):

Login:

```
1  select count(*) from test.Login
```

100% 32:1

Result Grid Filter Rows: Search

count(*)
1001

```
| Peppa995  | zgtbypud |     1 |
| Peppa996  | tigxuxej |     1 |
| Peppa997  | nnvmyiuv |     1 |
| Peppa998  | hhdukofo |     1 |
| Peppa999  | hbheqzqo |     1 |
+-----------+----------+-------+
1001 rows in set (0.00 sec)
```

Profile:

```
1  select count(*) from test.Profile
```

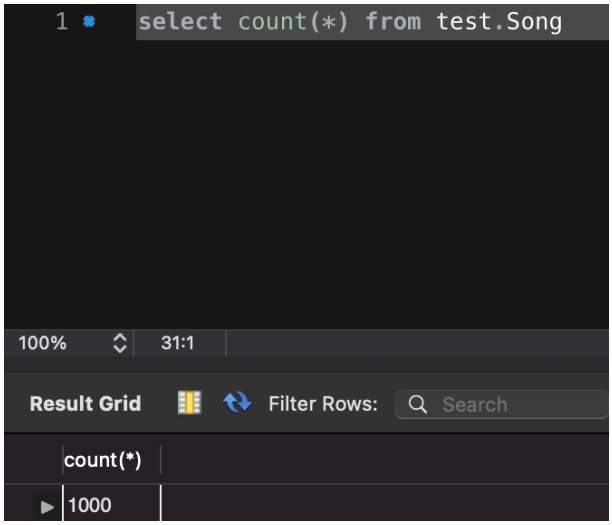
100% 34:1

Result Grid Filter Rows: Search

count(*)
1001

```
| Peppa992  |      1 |
| Peppa993  |      1 |
| Peppa994  |      1 |
| Peppa995  |      1 |
| Peppa996  |      1 |
| Peppa997  |      1 |
| Peppa998  |      1 |
| Peppa999  |      1 |
+-----------+--------+
1001 rows in set (0.00 sec)
```

Song:




```
| ZTWl1ktEAh |      1 |
| Zu2EVVYvgh |      1 |
| ZuOyMZgGQZ |      1 |
| ZuwTxLBYNK |      1 |
| ZV3HNNUKwv |      1 |
| ZVqUScsvpQ |      1 |
| zVtsUlG4mT |      1 |
| zWldbfFSDh |      1 |
| zWR1ngYI82 |      1 |
+------------+--------+
1000 rows in set (0.00 sec)
```

Queries:

Get User's Post History:

```
SELECT DISTINCT songList.userName, s.songId, s.title, s.artist, po.time
FROM test.Post po NATURAL JOIN test.Song s NATURAL JOIN (
```

```
                        SELECT po2.userName as userName
                        FROM test.Post po2
                        WHERE ('Peppa145' = po2.userName)
                        ) as songList
ORDER BY po.time DESC;
```

Results:

```
mysql> SELECT DISTINCT songList.userName, s.songId, s.title, s.artist, po.time FROM test.Post po NATURAL JOIN test.Song s NATURAL JOIN (                        SELECT po2.userName as userName
                    FROM test.Post po2                      WHERE ("Peppa145" = po2.userName)                       ) as songList ORDER BY po.time DESC;
+----------+------------+------------+--------+------------+
| userName | songId     | title      | artist | time       |
+----------+------------+------------+--------+------------+
| Peppa145 | 1HeXLaKEbO | ecgkamifxu | George | 2022-10-19 |
+----------+------------+------------+--------+------------+
1 row in set (0.00 sec)
```

Get Today's Posts:

```
SELECT po2.userName, s.songId, s.title, s.artist, t.time
FROM Post po2 NATURAL JOIN Song s NATURAL JOIN
    (SELECT po.time as time
    FROM Post po NATURAL JOIN Profile pr NATURAL JOIN Song s
    GROUP BY po.time
    HAVING  po.time = CAST(CURDATE() as Date)) as t;
```

Results:

```
mysql> SELECT po2.userName, s.songId, s.title, s.artist, t.time
    -> FROM Post po2 NATURAL JOIN Song s NATURAL JOIN
    ->     (SELECT po.time as time
    ->     FROM Post po NATURAL JOIN Profile pr NATURAL JOIN Song s
    ->     GROUP BY po.time
    ->     HAVING  po.time = CAST(CURDATE() as Date)) as t;
+----------+------------+------------+--------------+------------+
| userName | songId     | title      | artist       | time       |
+----------+------------+------------+--------------+------------+
| Peppa697 | 07p8iZbho1 | ytsvqcsljn | George       | 2022-10-21 |
| Peppa582 | 0vgUwpAvGY | wvcwzskjfq | Mr. Elephant | 2022-10-21 |
| Peppa898 | 1pJHgAaVRH | xvyutnolyf | ZuZu         | 2022-10-21 |
| Peppa308 | 2PZh3HT3jV | nrgybmaatb | Mr. Elephant | 2022-10-21 |
| Peppa859 | 3s8vnlEtfs | imolvezlcu | ZuZu         | 2022-10-21 |
| Peppa416 | 6yklzBFu0h | oymwjreefr | George       | 2022-10-21 |
| Peppa666 | 7Aoub3mykY | izzwnbflnb | ZuZu         | 2022-10-21 |
| Peppa680 | 7K4HQ3d0AD | cyyvlheqpf | Mr. Elephant | 2022-10-21 |
| Peppa322 | A8XJjjRGTs | lahporudlb | Candy Cat    | 2022-10-21 |
| Peppa401 | ahcLKBIPp0 | ecksagwgci | Candy Cat    | 2022-10-21 |
| Peppa133 | BmdsrmVCqh | cqwjqjexxi | Mr. Elephant | 2022-10-21 |
| Peppa723 | f3Et2hLltG | juvabnastk | ZuZu         | 2022-10-21 |
| Peppa609 | gDe9D6jniu | lqzbodogyy | ZaZa         | 2022-10-21 |
| Peppa778 | GS5sKSWcwf | mjadaajbkd | George       | 2022-10-21 |
| Peppa97  | GsRD5aLLmX | ncohknifdf | ZuZu         | 2022-10-21 |
```

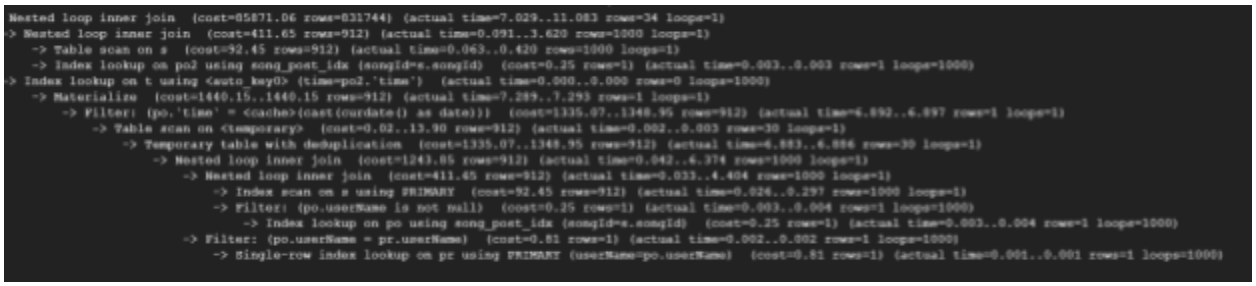
<u>Indexes:</u>

Get today's post query:

- **Without indexes:**

```
-> Nested loop inner join  (cost=85871.06 rows=831744) (actual time=7.419..11.500 rows=34 loops=1)
    -> Nested loop inner join  (cost=411.65 rows=912) (actual time=0.129..3.648 rows=1000 loops=1)
        -> Table scan on s  (cost=92.45 rows=912) (actual time=0.063..0.416 rows=1000 loops=1)
        -> Index lookup on po2 using song_post_idx (songId=s.songId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1000)
    -> Index lookup on t using <auto_key0> (time=po2.`time`)  (actual time=0.000..0.000 rows=0 loops=1000)
        -> Materialize  (cost=1440.15..1440.15 rows=912) (actual time=7.583..7.588 rows=1 loops=1)
            -> Filter: (po.`time` = <cache>(cast(curdate() as date)))  (cost=1335.07..1348.95 rows=912) (actual time=7.146..7.151 rows=1 loops=1)
                -> Table scan on <temporary>  (cost=0.02..13.90 rows=912) (actual time=0.001..0.003 rows=30 loops=1)
                    -> Temporary table with deduplication  (cost=1335.07..1348.95 rows=912) (actual time=7.138..7.141 rows=30 loops=1)
                        -> Nested loop inner join  (cost=1243.85 rows=912) (actual time=0.079..6.733 rows=1000 loops=1)
                            -> Nested loop inner join  (cost=411.65 rows=912) (actual time=0.068..4.744 rows=1000 loops=1)
                                -> Index scan on s using PRIMARY  (cost=92.45 rows=912) (actual time=0.055..0.322 rows=1000 loops=1)
                                -> Filter: (po.userName is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=1000)
                                    -> Index lookup on po using song_post_idx (songId=s.songId)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=1000)
                            -> Filter: (po.userName = pr.userName)  (cost=0.81 rows=1) (actual time=0.002..0.002 rows=1 loops=1000)
                                -> Single-row index lookup on pr using PRIMARY (userName=po.userName)  (cost=0.81 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
```

- **With index on Post(time):**
    - We tried to index t.time because we are grouping by the time. When using the EXPLAIN ANALYZE command, we were able to see a decrease in most of the actual times with index, compared to that without the time index. Also, looking at the overall run time: without indexes, the run time was 0.02 seconds; with indexes, the run time was 0.01 seconds. However, this was not consistent when different runs. Considering it was a very minute difference and inconsistent, it probably means that the index did not make that much of a difference. This is pretty understandable considering that our query does not require excessive searching through the data. Additionally, the attributes we are selecting are already indexed since they are primary and foreign keys. It is possible that indexing time would speed up the process if we had more data, but considering our dataset only had one row that matched the current Date, the indexing did not make a huge difference.



- **With index on Song(title):**
    - We chose to try to index the title from the Song table because in order for a Post to occur, we also need the song title since that is one of the attributes we are displaying. Looking through all the titles would take a great amount of time so we decided to try indexing it. However, it did not make much difference because of songId. SongId is connected to the song's information including the title. Since songId is a primary key and already is being indexed, it is already being performed at max performance.

```
-> Nested loop inner join  (cost=85871.06 rows=831744) (actual time=6.245..10.533 rows=34 loops=1)
    -> Nested loop inner join  (cost=411.65 rows=912) (actual time=0.056..3.757 rows=1000 loops=1)
        -> Table scan on s  (cost=92.45 rows=912) (actual time=0.037..0.390 rows=1000 loops=1)
        -> Index lookup on po2 using song_post_idx (songId=s.songId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1000)
    -> Index lookup on t using <auto_key0> (time=po2.`time`)  (actual time=0.000..0.000 rows=0 loops=1000)
        -> Materialize  (cost=1440.15..1440.15 rows=912) (actual time=6.594..6.600 rows=1 loops=1)
            -> Filter: (po.`time` = <cache>(cast(curdate() as date)))  (cost=1335.07..1348.95 rows=912) (actual time=6.149..6.151 rows=1 loops=1)
                -> Table scan on <temporary>  (cost=0.02..13.90 rows=912) (actual time=0.002..0.003 rows=30 loops=1)
                    -> Temporary table with deduplication  (cost=1335.07..1348.95 rows=912) (actual time=6.139..6.142 rows=30 loops=1)
                        -> Nested loop inner join  (cost=1243.85 rows=912) (actual time=0.045..5.798 rows=1000 loops=1)
                            -> Nested loop inner join  (cost=411.65 rows=912) (actual time=0.036..4.029 rows=1000 loops=1)
                                -> Index scan on s using title_idx  (cost=92.45 rows=912) (actual time=0.027..0.238 rows=1000 loops=1)
                                -> Filter: (po.userName is not null)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=1000)
                                    -> Index lookup on po using song_post_idx (songId=s.songId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1000)
                            -> Filter: (po.userName = pr.userName)  (cost=0.81 rows=1) (actual time=0.002..0.002 rows=1 loops=1000)
                                -> Single-row index lookup on pr using PRIMARY (userName=po.userName)  (cost=0.81 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
```

- **With index on Song(artist):**
    - Similar to title, we chose to try to index artists from the Song table because we will need the song's artist in each Post for a Post to occur. This attribute is also one that we are displaying. And, we also did not see much of a difference, most likely due to songId. Since songId is the primary key and is already indexed, it is already as efficient as possible.

```
| -> Nested loop inner join  (cost=85871.06 rows=831744) (actual time=5.993..9.712 rows=34 loops=1)
    -> Nested loop inner join  (cost=411.65 rows=912) (actual time=0.069..3.274 rows=1000 loops=1)
        -> Table scan on s  (cost=92.45 rows=912) (actual time=0.045..0.387 rows=1000 loops=1)
        -> Index lookup on po2 using song_post_idx (songId=s.songId)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=1000)
    -> Index lookup on t using <auto_key0> (time=po2.`time`)  (actual time=0.000..0.000 rows=0 loops=1000)
        -> Materialize  (cost=1440.15..1440.15 rows=912) (actual time=6.266..6.270 rows=1 loops=1)
            -> Filter: (po.`time` = <cache>(cast(curdate() as date)))  (cost=1335.07..1348.95 rows=912) (actual time=5.882..5.884 rows=1 loops=1)
                -> Table scan on <temporary>  (cost=0.02..13.90 rows=912) (actual time=0.002..0.004 rows=30 loops=1)
                    -> Temporary table with deduplication  (cost=1335.07..1348.95 rows=912) (actual time=5.869..5.872 rows=30 loops=1)
                        -> Nested loop inner join  (cost=1243.85 rows=912) (actual time=0.042..5.546 rows=1000 loops=1)
                            -> Nested loop inner join  (cost=411.65 rows=912) (actual time=0.034..3.812 rows=1000 loops=1)
                                -> Index scan on s using artist_idx  (cost=92.45 rows=912) (actual time=0.024..0.225 rows=1000 loops=1)
                                -> Filter: (po.userName is not null)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1000)
                                    -> Index lookup on po using song_post_idx (songId=s.songId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1000)
                            -> Filter: (po.userName = pr.userName)  (cost=0.81 rows=1) (actual time=0.001..0.002 rows=1 loops=1000)
                                -> Single-row index lookup on pr using PRIMARY (userName=po.userName)  (cost=0.81 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
```

Get User's Post History:

- **Without indexes:**

```
| -> Sort: po.`time` DESC  (actual time=1.004..1.004 rows=1 loops=1)
    -> Table scan on <temporary>  (cost=2.50..2.50 rows=0) (actual time=0.001..0.001 rows=1 loops=1)
        -> Temporary table with deduplication  (cost=208.25..208.25 rows=0) (actual time=0.995..0.995 rows=1 loops=1)
            -> Inner hash join (no condition)  (cost=205.75 rows=0) (actual time=0.666..0.967 rows=1 loops=1)
                -> Filter: (po2.userName = 'Peppa145')  (cost=65.74 rows=2) (actual time=0.103..0.404 rows=1 loops=1)
                    -> Table scan on po2  (cost=65.74 rows=1000) (actual time=0.026..0.251 rows=1000 loops=1)
                -> Hash
                    -> Nested loop inner join  (cost=101.80 rows=2) (actual time=0.176..0.553 rows=1 loops=1)
                        -> Filter: (po.userName = 'Peppa145')  (cost=101.25 rows=2) (actual time=0.160..0.536 rows=1 loops=1)
                            -> Table scan on po  (cost=101.25 rows=1000) (actual time=0.060..0.376 rows=1000 loops=1)
                        -> Single-row index lookup on s using PRIMARY (songId=po.songId)  (cost=0.31 rows=1) (actual time=0.015..0.015 rows=1 loops=1)
 |
```

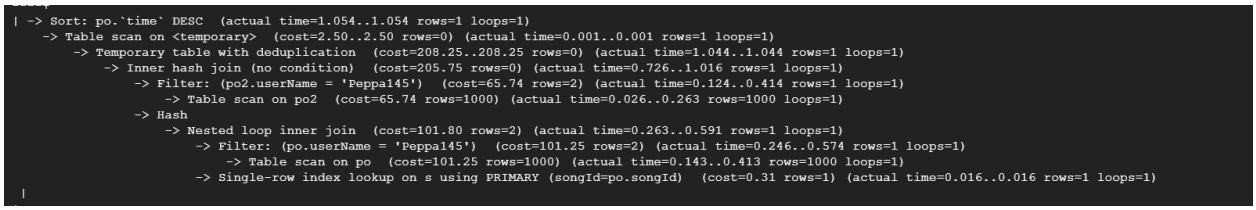
- **With index on Post(time):**
    - As seen above, the required time to run the query actually increases from 1.004 to 1.006, while the cost stays the same for the table scan, temporary table and inner hash. The actual time for the temporary table increases from 0.995 to 1.057, and the inner hash increases from 0.666 to 1.039. As you can see, these differences are not very big as well. Since there is not a big difference, we can conclude that indexing time does not help speed our query. This is because the rows we retrieve from the query are out of order with or without the index on time. This means that the ORDER BY operation will run regardless of the index or not. Thus, it does not improve our algorithm runtime.

```
| -> Sort: po.`time` DESC  (actual time=1.066..1.066 rows=0 loops=1)
    -> Table scan on <temporary>  (cost=2.50..2.50 rows=0) (actual time=0.001..0.001 rows=0 loops=1)
        -> Temporary table with deduplication  (cost=208.25..208.25 rows=0) (actual time=1.057..1.057 rows=0 loops=1)
            -> Inner hash join (no condition)  (cost=205.75 rows=0) (actual time=1.039..1.039 rows=0 loops=1)
                -> Filter: (po2.userName = 'Peppa51')  (cost=65.74 rows=2) (never executed)
                    -> Table scan on po2  (cost=65.74 rows=1000) (never executed)
                -> Hash
                    -> Nested loop inner join  (cost=101.80 rows=2) (actual time=0.994..0.994 rows=0 loops=1)
                        -> Filter: (po.userName = 'Peppa51')  (cost=101.25 rows=2) (actual time=0.994..0.994 rows=0 loops=1)
                            -> Table scan on po  (cost=101.25 rows=1000) (actual time=0.547..0.821 rows=1000 loops=1)
                        -> Single-row index lookup on s using PRIMARY (songId=po.songId)  (cost=0.31 rows=1) (never executed)
 |
```

- **With index on Song(title):**

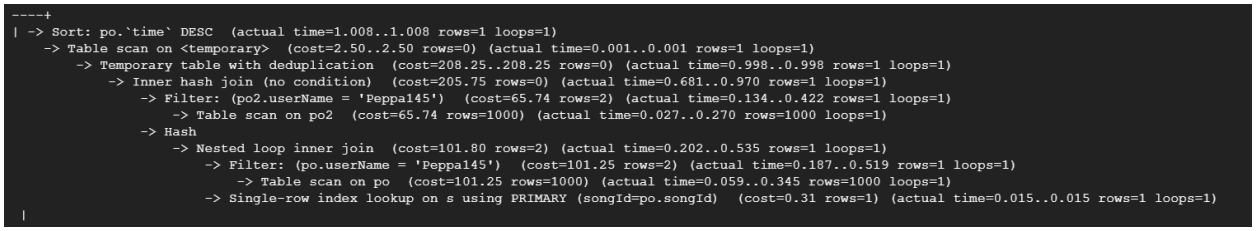
In the nested loop inner join hash, there was a slight improvement. There were a few other improvements in actual time; however, they were all small decreases, meaning again, there was not a huge improvement. This is most likely for the same reason as the other query. We do not

use time other than for selecting so ordering time by an index does not help improve our system.



- **With index on Song(artist):**

Again, we can see that there were no huge improvements when indexing Song(artist). However, we can see that the times did slightly increase. The sorting time increased from 1.004 to 1.008, and the hash nested loop inner join went from 0.176 to 0.026. Although, once again it is not a large difference. Thus, we can conclude that indexing artist from Song does not help improve our system.



Conclusion on Indexes:

As seen in the analysis above, our queries' runtimes weren't decreased as a result of applying indexes. This is because in our queries, the primary keys and foreign keys were already indexed by default in MySQL Workshop. Hence, when we applied indexes to the other attributes that were not primary or foreign keys, we didn't notice a significant difference. If we had more data for Post, it could be expected that after applying indexes on the GROUP BY attribute for time in the GetUserPosts that the runtime would decrease.

Additionally, when we created our data, we randomized the user to each Post. Thus, our data didn't have enough entries for a specific user to show any decrease in runtime. However, it could be assumed that with more data, a decrease in the runtime could be observed as a result of indexing. It is also possible that there is no difference in our data because while generating our data the usernames were already in order. In a real life scenario, the data wouldn't be ordered, hence a greater improvement could be observed.

Furthermore, it is also possible that indexes are not improving our time much due to the fact that we only have 1000 entries max in each table. Sorting a table is O(n * log(n)) generally with the worst case being O(n) time. Meaning we are usually sorting ~3000 entries and in the worst case scenario we may be sorting 1,000,000 rows. Both operations are computationally minimal with modern computers. It is possible that we may need tens of thousands of rows to see significant differences in time performance.