

Tyler Lapiana - tj19096

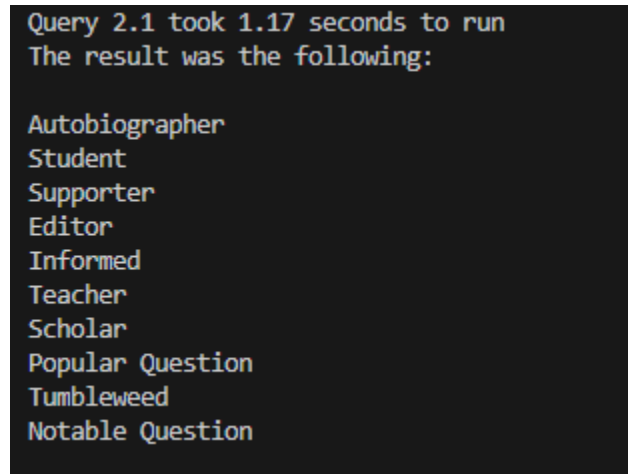
This is the report file for Assignment 2

In order to properly run the code, please read the comment block at the very top of the code file.

1.
 - a. The code for this question can all be found in the provided zipped code
2. Below is the list of SQL statements used to get the requested data as well as the timing of each one. The timings can also be seen via running the program. An image is also provided to show the output when I ran the code.

a. 2.1

- i. SELECT Name FROM Badges as b
- ii. JOIN Yousers as u ON b.UserId = u.id
- iii. WHERE b.Date BETWEEN u.CreationDate AND
 1. u.CreationDate + interval '1 year'
- iv. GROUP BY b.Name
- v. ORDER BY COUNT(b.Name) DESC
- vi. LIMIT 10;



```
Query 2.1 took 1.17 seconds to run
The result was the following:
```

```
Autobiographer
Student
Supporter
Editor
Informed
Teacher
Scholar
Popular Question
Tumbleweed
Notable Question
```

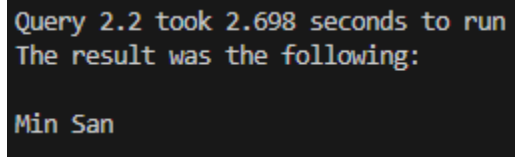
vii.

b. 2.2

- i. SELECT DisplayName FROM Yousers WHERE
- ii. Reputation > 1000 AND Yousers.Id IN (
 1. SELECT Id FROM Yousers
 2. EXCEPT

3. SELECT OwnerUserId FROM Posts

iii.);



Query 2.2 took 2.698 seconds to run
The result was the following:

Min San

iv.

c. 2.3

- i. SELECT u.DisplayName, u.Reputation FROM Users as u
- ii. JOIN Posts as p ON u.Id = p.OwnerUserId
- iii. JOIN PostTags as pt ON p.ParentId = pt.PostId
- iv. JOIN Tags as t ON pt.TagId = t.Id
- v. WHERE t.TagName = 'postgresql'
- vi. GROUP BY u.DisplayName, u.Reputation
- vii. HAVING COUNT(u.DisplayName) > 1;

```
Query 2.3 took 0.437 seconds to run
The result was the following:
```

```
A.B. - 89553
aneeshep - 30133
Anvesh - 581
Anwar - 76163
BDRSuite - 3136
Braiam - 67310
Brian.D.Myers - 610
Caesium - 15647
Carlos Dagorret - 634
Craig Ringer - 5224
Daniel Vérité - 1193
David Valenza - 111
Dennis Kaarsemaker - 6834
dessert - 39682
devav2 - 35976
Elder Geek - 35683
Evan Carroll - 7396
Federico Munerotto - 21
Félicien - 1163
Feriman - 198
Florian Diesch - 86509
frlan - 1030
Gino - 113
Gryu - 7449
guiverc - 29537
heemayl - 90993
hmayag - 2236
ish - 139186
James Henstridge - 40716
Jos - 28701
K. Darien Freeheart - 551
karel - 112036
klin - 131
Luís de Sousa - 13108
maletin - 148
Marc Vanhoomissen - 2204
matigo - 21151
Maythux - 83381
Michael Durrant - 10756
```

viii.

ix. (This continues, see program for full result)

d. 2.4

- i. SELECT u.DisplayName FROM Youusers as u
- ii. JOIN Comments as c ON u.Id = c.UserId
- iii. WHERE c.Score > 10 AND
- iv. c.CreationDate BETWEEN u.CreationDate AND
 1. u.CreationDate + interval '1 week'
- v. GROUP BY u.DisplayName;

```
Query 2.4 took 0.566 seconds to run
The result was the following: (only the first 100 rows)
```

```
64pi0r
Abdelouahab
Agargara
Aitch
Aleksandar Nikolic
alex
Alex
Alex White
alex.forencich
Alexander Oh
Alistair Buxton
amalloy
amphibient
AndiChin
AndreG
Andrew
Angel O'Sphere
AntoineG
Arthur B
ATX
Augustus Kling
ayckoster
barrypicker
Behrang
Ben Mordecai
Birla
bluesmoon
bobince
Brad Figg
Brian H
btk
C. M.
Caimen
Caltor
Charles Offenbacher
Charles Roper
Cheyne
Chip Castle
Chris Foster
```

vi.

vii. (This continues, see program for full result)

e. 2.5

- i. SELECT t.TagName, COUNT(t.TagName) FROM Tags as t
- ii. JOIN PostTags as pt ON t.Id = pt.TagId
- iii. JOIN Posts as p ON pt.PostId = p.Id
- iv. JOIN PostTags as pt_two ON p.Id = pt_two.PostId
- v. JOIN Tags as t_two ON pt_two.TagId = t_two.Id
- vi. WHERE t_two.TagName = 'postgresql' AND t.TagName != 'postgresql'

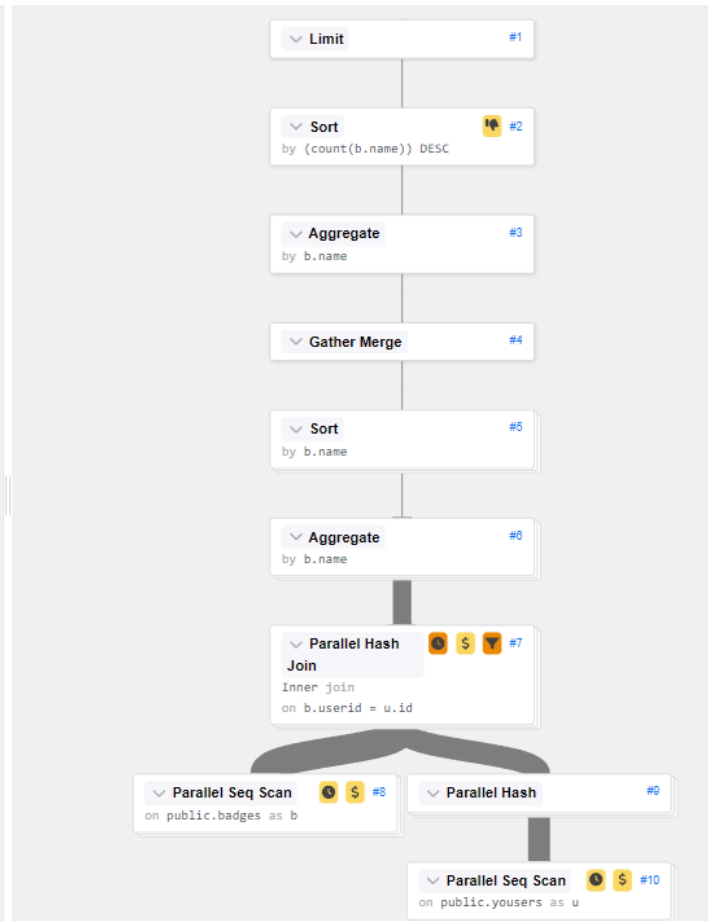
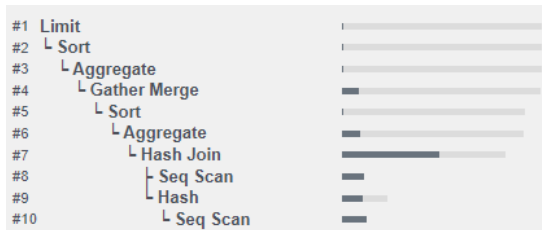
- vii. GROUP BY t.TagName
- viii. ORDER BY COUNT(t.TagName) DESC, t.TagName ASC
- ix. LIMIT 25;

```
Query 2.5 took 0.16 seconds to run
The result was the following:

apt - 110
server - 68
package-management - 53
14.04 - 52
database - 33
software-installation - 33
16.04 - 28
12.04 - 27
pgadmin - 27
permissions - 24
bash - 21
command-line - 20
20.04 - 19
networking - 19
upgrade - 19
dependencies - 17
18.04 - 15
dpkg - 15
php - 15
python - 14
apache2 - 12
ruby - 12
22.04 - 11
mysql - 11
systemd - 11
```

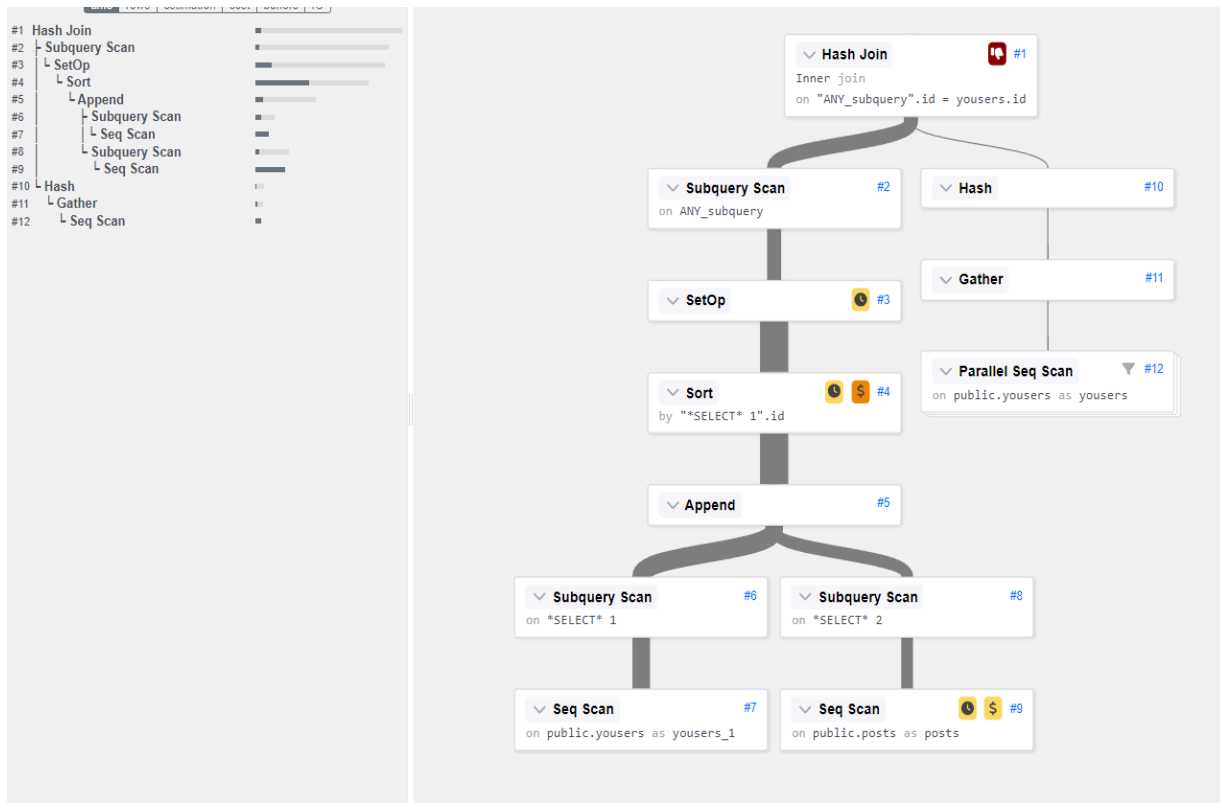
x.

- 3. For each query listed above, I will provide an image of the visualized “EXPLAIN” Query using the visualizer resource followed at my best attempt to explain what is going on in each query.
 - a. Query 2.1



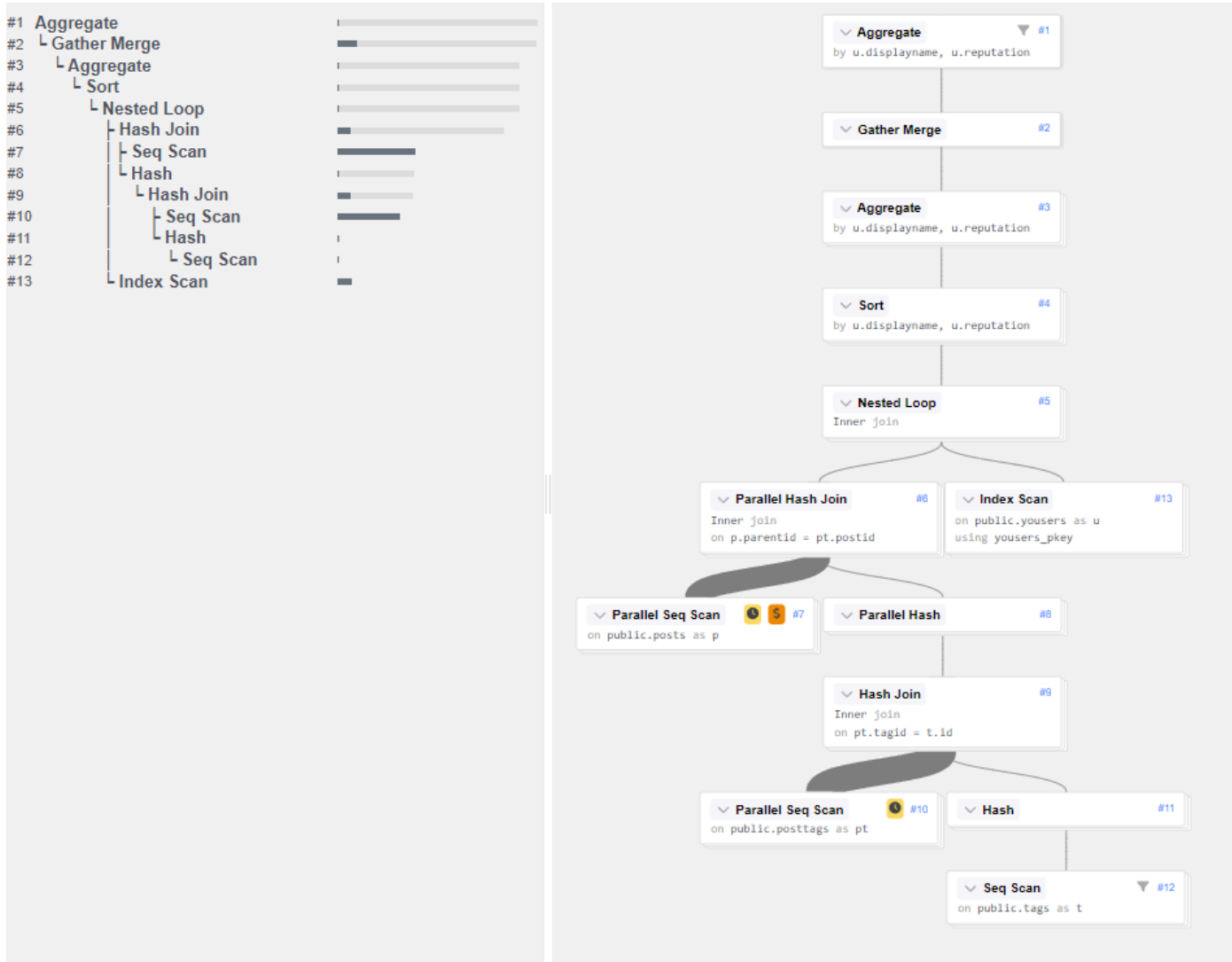
- i. Explanation:
- ii. The main thing this query needs to do is join the two tables together. It does this via a parallel hash join, which consists of a Parallel Seq. Scan and a Parallel Hash. The Seq. Scan sequentially scans through the Badges table to get the information it needs while the Parallel hash creates a hash table for the join to reference that gives the information about the Yousers table. It then joins the tables on the given information using the hash table. It then starts to do the aggregation, grouping by the badge name utilizing a hash table. It then sorts the data according to the badge name, which is used to gather the work of all the parallel workers together. It then finalizes the aggregation once the data is sorted, re-sorts the data by the count, and finally takes just the first 10 rows.

b. Query 2.2



- i. Explanation:
- ii. This Query primarily has to work on the sub-query to complete the work. The database decided that a hash join was the fastest way to determine if the Id was in the subquery, but first, it had to get the subquery table. To do this, it first sequentially scans both Yousers and Posts in order to get the two tables to then do the "EXCEPT" Set operation on. Once the set operation is done, it joins that information to the information it had already got from Yousers using a parallel Seq. Scan. This encompassed the reputation condition as part of the scan. It had previously hashed the Yousers information to efficiently join the two tables together

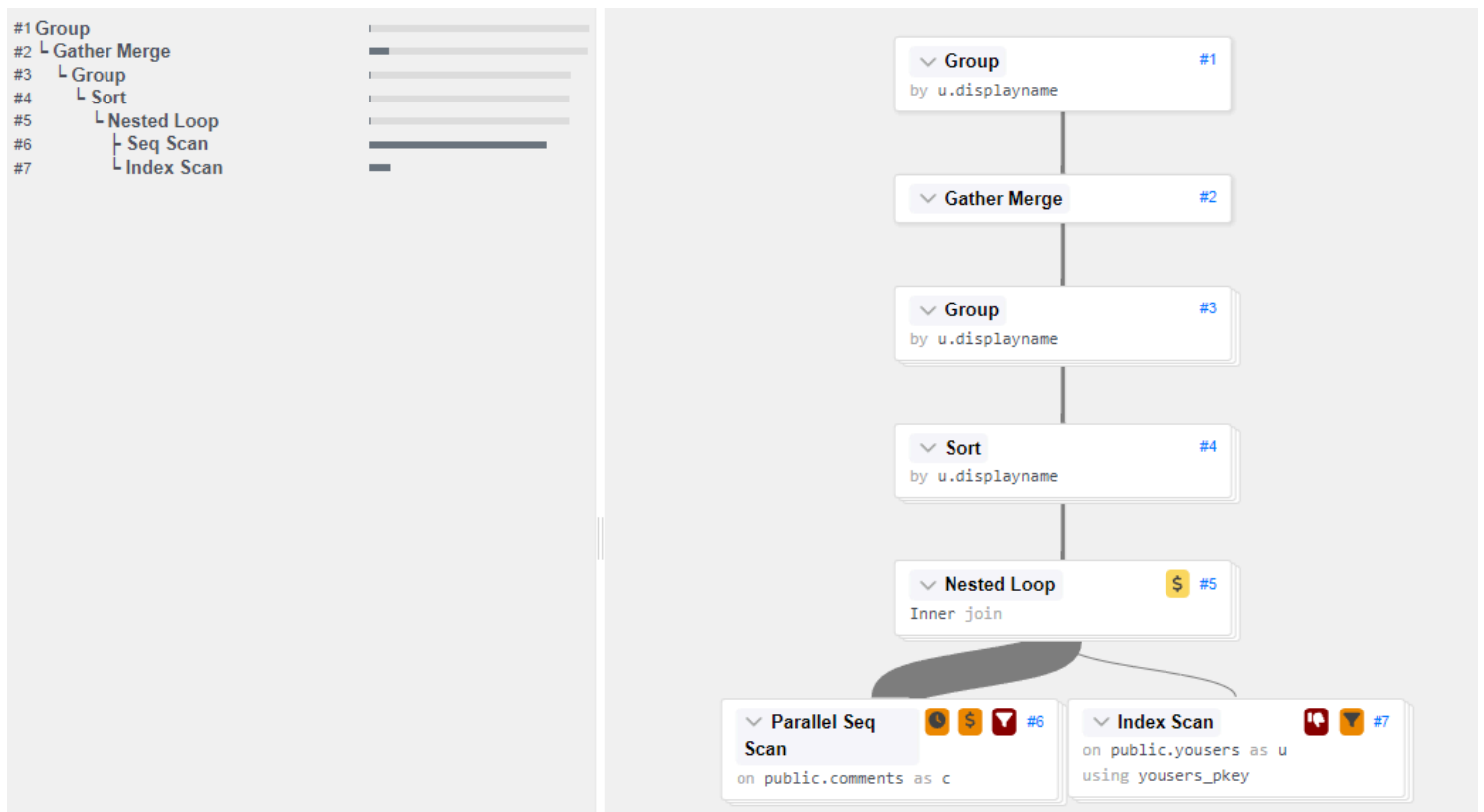
c. Query 2.3



- Explanation:
- The First big thing this program tries to do is a nested loop, which for each record in the first set, attempts to find a match in the second set. The second set for this nested loop is the result of an index scan on the Youusers table, which returned all necessary user info we needed. It used an index scan because there is always an index on the primary key and there was no selection of columns at this point yet. The First set of the nested loop ultimately was a set of ownerId's, but to get there it needed to do a parallel hash join. The thing this parallel hash join scanned through was posts and matched it with the hash values resulting from another hash join. This sub-hash join scanned through post tags to

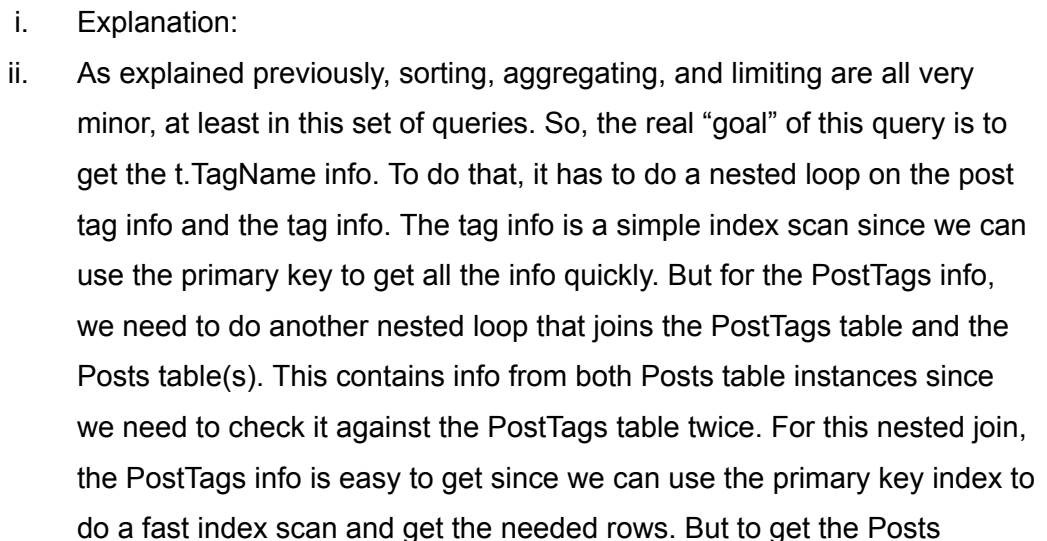
get the info that would be used to compare it to the hashed values of Tags. Once Tags and PostTags were joined, it could then hash those values to join once more on the posts table. This finally left us with a set of OwnerUserId's from Posts and the information from the Yousers table. Once it did the nested loop, ultimately resulting in just the display name and reputation, the rest of the work was trivial. It sorted the data, and did the aggregate, completing the query

d. Query 2.4



- Explanation:
- This query requires one join, in this case using a nested loop that ultimately results in just the display name. However, it first must get the two sets to use in the nested loop. The Index scan is a quick scan of the Yousers table since all we need from that table is the id. On the other hand, the parallel sequential scan of the Comments table takes a while. This is due to needing information on the UserId in order to do the join as well as the score and creation date in order to satisfy the where conditions. However, once these sets are run through the nested loop, the

e. Query 2.5



table(s) info, we need to do yet another nested loop joining the original Posts table's info with the info from the second instance. The rows from the first instance are easy to get since we can use the primary key index to get that information. But for the second instance of the Posts table, we will have to do a hash join to get the data. This hash join joins the second instance of the PostTags table with the hashed information from the second Tags table instance. The hashed information is easy to get since we just need the id from the tags and the tags table is relatively small. However, since we are doing a hash join checking for a specific value of tag, we must sequentially scan from the second instance of the PostTags table to use the hashed info against. This is where the majority of the time running this query is done.

4. Relational Algebra

a. 4.1

- i. $\Pi_{\text{Yousers.DisplayName}} ($
 1. $\text{Comments} \bowtie_{\text{Yousers.Id} = \text{Comments.UserId}} ($
 - a. $\Pi_{\text{Yousers.Id}, \text{Yousers.DisplayName}} (\text{Yousers})$
 - b. -
 - c. $\Pi_{\text{Yousers.Id}, \text{Yousers.DisplayName}} ($
 - i. $\text{Yousers} \bowtie_{\text{Yousers.Id} = \text{Posts.OwnerUserId}} \text{Posts}$
 - d. $)$
 2. $)$
- ii. $)$

b. 4.2

- i. $\Pi_{\text{Yousers.DisplayName}} (\text{Yousers})$
- ii. -
- iii. $\Pi_{\text{Yousers.DisplayName}} ($
 1. $\sigma_{\text{Tags.TagName} \neq \text{"postgresql"}} ($
 - a. $\text{Tags} \bowtie_{\text{Tags.Id} = \text{PostTags.TagId}} ($
 - i. $\text{PostTags} \bowtie_{\text{PostTags.PostId} = \text{Post.Id}} ($
 1. $\text{Posts} \bowtie_{\text{Posts.OwnerUserId} = \text{Yousers.Id}} \text{Yousers}$
 - ii. $)$
 - b. $)$
 2. $)$

iv.)

c. 4.3

i. $\Pi_{\text{Yousers.DisplayName}}$ (

1. $\sigma_{\text{Yousers.DisplayName LIKE \%John\%}}$ (

a. $\sigma_{\text{Comments.CreationDate > date('2016-12-31') \wedge \text{Comments.CreationDate < date('2018-01-01')}} ($

i. $\text{Yousers} \bowtie_{\text{Yousers.Id = Comments.UserId}} \text{Comments}$

b.)

2.)

ii.)

d. 4.4

i. $\Pi_{\text{Posts.Title}}$ (

1. $\sigma_{\text{Posts.Score > 1000}}$ (

a. $\sigma_{\text{Posts.CreationDate < (Yousers.CreationDate + interval '1 year' + interval '1 day')}} ($

i. $\text{Posts} \bowtie_{\text{Posts.OwnerUserId = Yousers.Id}} \text{Yousers}$

b.)

2.)

ii.)

e. 4.5

i. $\Pi_{\text{Yousers.Id, Yousers.DisplayName}}$ (

1. $\sigma_{\text{Tags_one.TagName = "postgres" \wedge \text{Tags_two.TagName = "mysql"}}$ (

a. $\rho_{\text{Tags_one}}(\text{Tags}) \bowtie_{\text{Tags_one.Id = PostTags.TagId}}$ (

i. $\rho_{\text{Tags_two}}(\text{Tags}) \bowtie_{\text{Tags_two.Id = PostTags.TagId}}$ (

1. $\text{PostTags} \bowtie_{\text{PostTags.PostId = Posts.Id}}$ (

a. $\text{Posts} \bowtie_{\text{Posts.OwnerUserId = Yousers.Id}} \text{Yousers}$

2.)

ii.)

b.)

2.)

ii.)

5.

- a. The indexes that I created are listed under the “createIndexes” function of my code. Below is where I documented my decisions in choosing what indexes to make.

- b. The first thing I looked at when making the indexes was making an index for every column that is used in a WHERE clause that is not already indexed via the primary key. This would speed up any execution using that where clause since it would no longer have to search through all of them. Next, I created indexes for anything that I joined on that I didn't have already. This was supported by my execution plans from question 3 where a lot of extra time was spent on joins not part of a primary key. I then did a final check through the execution plans to check for any values that might be slowing the queries down.
- c. Below are the resulting sped-up query execution times after indexing as well as a brief explanation as to why they were sped up
 - i. Query 2.1

```
Query 2.1 took 0.77 seconds to run
The result was the following:

Autobiographer
Student
Supporter
Editor
Informed
Teacher
Scholar
Popular Question
Tumbleweed
Notable Question
```

- 1.
- 2. Looking at the execution plan for query 2.1, we can see that the majority of the time is spent on the join of Youusers.Id on Badges.UserId. Since Youusers.Id is indexed already as a primary key, the index on Badges.UserId is able to speed the query up a bit.

- ii. Query 2.2

```
Query 2.2 took 1.293 seconds to run
The result was the following:

Min San
```

- 1.
- 2. Looking at the execution plan that we had for this query, the subquery(s) took the majority of the time. Putting an index on Posts.OwnerUserId alleviated some of this, but since we are also

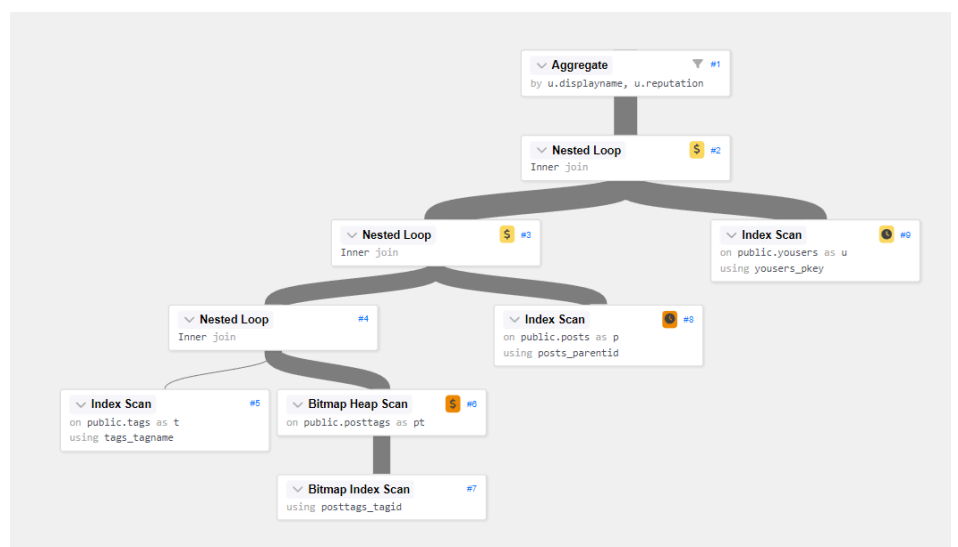
using Youusers.Reputation in the WHERE clause, the index on that column most likely also sped the query up

iii. Query 2.3

```
Query 2.3 took 0.268 seconds to run
The result was the following:

A.B. - 89553
aneeshep - 30133
Anvesh - 581
Anwar - 76163
BDRSuite - 3136
Braiam - 67310
Brian.D.Myers - 610
Caesium - 15647
Carlos Dagorret - 634
Craig Ringer - 5224
```

- 1.
2. (This is not the full result, check code for all values)
3. Looking at the old execution plan for query 2.3, we can see 2 clear slow-down spots, the two joins where we join on a value that is not a primary id. This is not ideal. But, if we look below at the new execution plan after indexing, it's a bit different. It may appear that there are more sections that are slow, but all of the scans are now index scans, which in general are much quicker. Thus, our indexes definitely sped this query up.



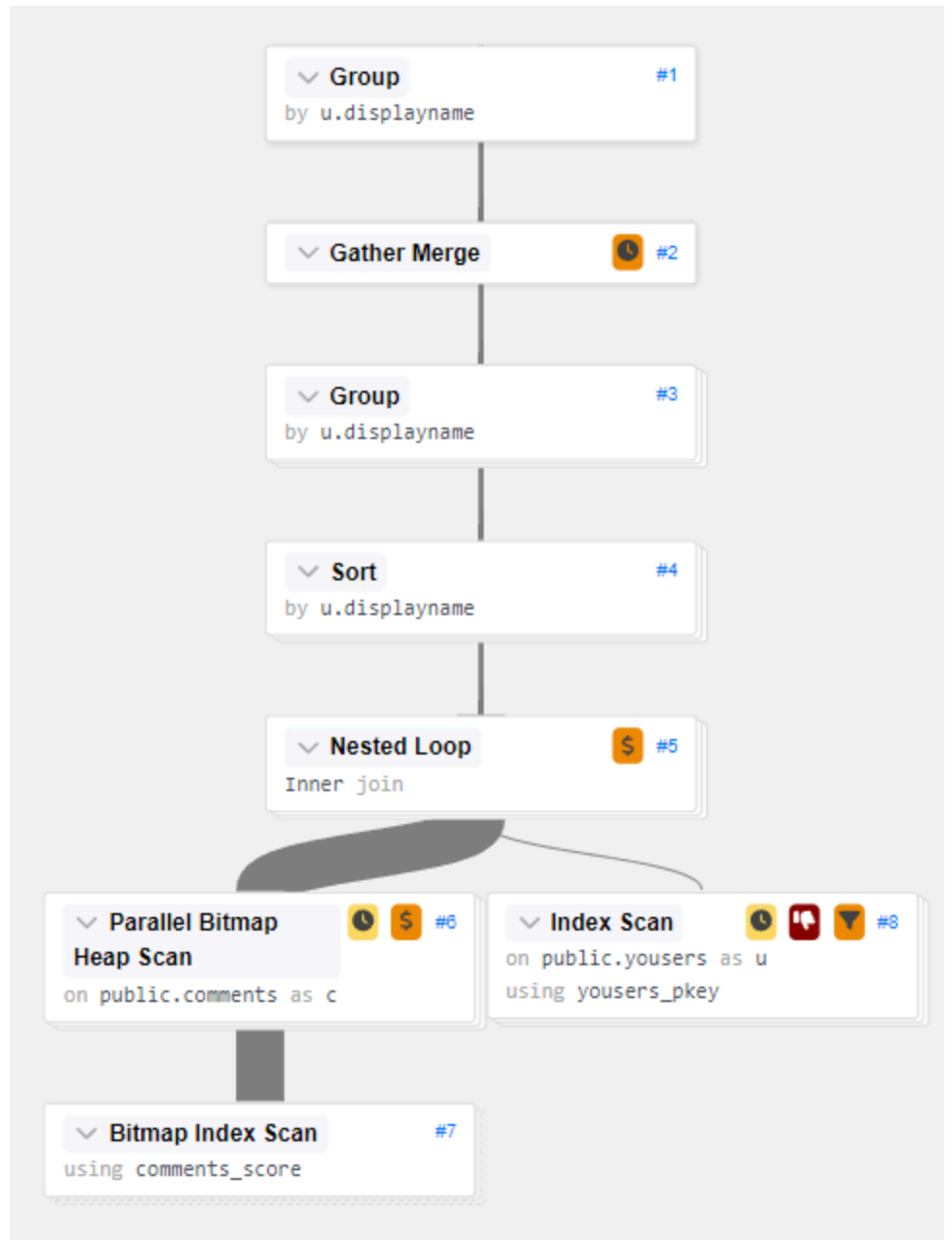
4.

iv. Query 2.4

```
Query 2.4 took 0.066 seconds to run  
The result was the following: (only the first 100 rows)
```

```
64pi0r  
Abdelouahab  
Agargara  
Aitch  
Aleksandar Nikolic  
alex  
Alex  
Alex White  
alex.forencich  
Alexander Oh  
Alistair Buxton
```

- 1.
2. (This is not the full result, check code for all values)
3. Similarly to query 2.3, query 2.4 had 1 visible place where the slowdown occurred. And like 2.3, 2.4 also has a new-looking execution plan, just not as different. As shown below, the sequential scan is now an index scan for comments, generally speeding up the query thanks to the indexes on different columns in the comment table.



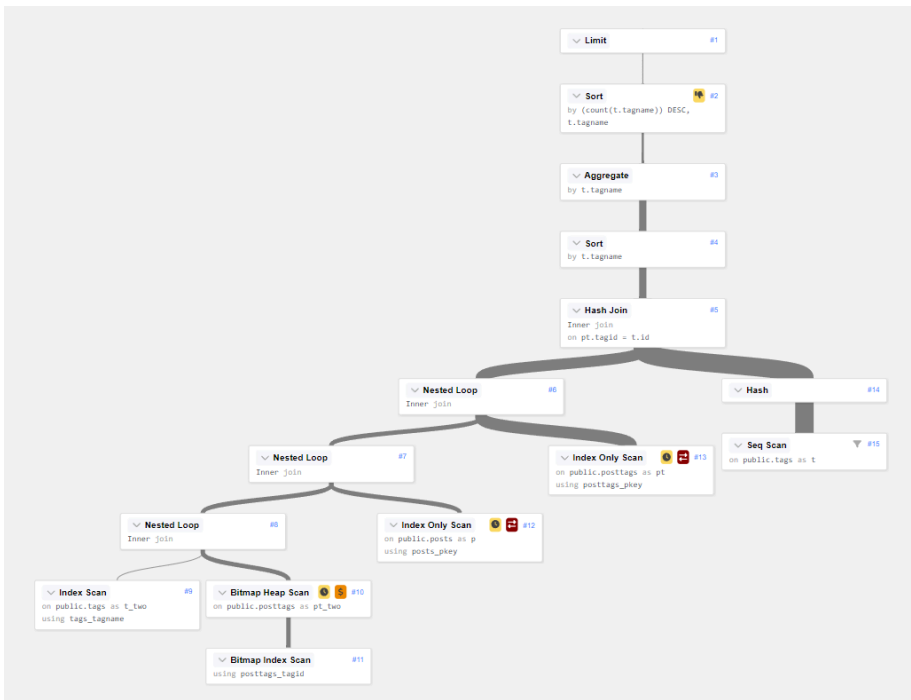
4.

v. Query 2.5


```
Query 2.5 took 0.02 seconds to run  
The result was the following:
```

```
apt - 110  
server - 68  
package-management - 53  
14.04 - 52  
database - 33  
software-installation - 33  
16.04 - 28  
12.04 - 27  
pgadmin - 27  
permissions - 24  
bash - 21  
command-line - 20  
20.04 - 19  
networking - 19  
upgrade - 19  
dependencies - 17  
18.04 - 15  
dpkg - 15  
php - 15  
python - 14  
apache2 - 12  
ruby - 12  
22.04 - 11  
mysql - 11  
systemd - 11
```

- 1.
2. Just like the two previous queries, query 2.5 had an obvious spot where the query was taking the most time. The new execution plan below shows that this is no longer the case and the time is slightly more spread out across different operations. This means that the index on PostTags worked well. This index was special as it was the only index I made that was a part of the primary key. But, since the primary key for PostTags is both columns, the index on just the TagId was important in speeding up the query.



3.