Table DDL Commands:

```
CREATE TABLE Students (
        NetID VARCHAR(255) NOT NULL,
        FirstName VARCHAR(255),
        LastName VARCHAR(255),
        Major VARCHAR(100),
        Residence VARCHAR(100),
        PRIMARY KEY (NetID)
);
CREATE TABLE Major (
        MajorID INTEGER NOT NULL,
        NetID VARCHAR(255) NOT NULL,
        CurrentClasses VARCHAR(1000),
        PRIMARY KEY (MajorID, NetID)
);
CREATE TABLE Login (
        NetID VARCHAR(255) NOT NULL,
        FirstName VARCHAR(255),
        LastName VARCHAR(255),
        PRIMARY KEY (NetID)
);
CREATE TABLE Interests(
        InterestID INTEGER NOT NULL,
        NetID VARCHAR(255) NOT NULL,
        Interests VARCHAR(255),
        PRIMARY KEY (InterestID, NetID)
);
CREATE TABLE RSOMembers(
        RsoID INTEGER NOT NULL,
        NetID VARCHAR(255) NOT NULL,
```

```
    FirstName VARCHAR(255),
    LastName VARCHAR(255),
    PRIMARY KEY (RsoID, NetID)
);
```

Our Tables

For our project, we are using GCP to store our SQL data and tables. The following screenshots are taken directly from the output of the GCP console after running the appropriate commands.

show tables;

```
+------------------------+
| Tables_in_MatchandMeet |
+------------------------+
| Interests              |
| Login                  |
| Major                  |
| RSOMembers             |
| Students               |
+------------------------+
5 rows in set (0.01 sec)
```

We have five tables, Interests, Login, Major, RSOMembers, and Students. Currently, our tables all have greater than 1500 rows and are created with auto-generated data. Our data auto generation algorithm is a bit basic at this point in our project, but we hope to expand upon it in future stages of our project.

InterestID	NetID	Interests
0	aagee7	Powerlifting
0	aalvarez8	Powerlifting
0	aandrew7	Powerlifting
0	abild5	Powerlifting
0	abowens4	Powerlifting
0	abridges8	Powerlifting
0	abruns2	Powerlifting
0	abunch9	Powerlifting
0	abunnell0	Powerlifting
0	acameron3	Powerlifting
0	acasey8	Powerlifting
0	aclemente8	Powerlifting
0	acorey4	Powerlifting
0	acorona5	Powerlifting
0	acraig1	Powerlifting

15 rows in set (0.01 sec)

SELECT * FROM Interests LIMIT 15; This is an example of fifteen rows in our **Interests** table.

```
mysql> SELECT * From Login LIMIT 15;
```

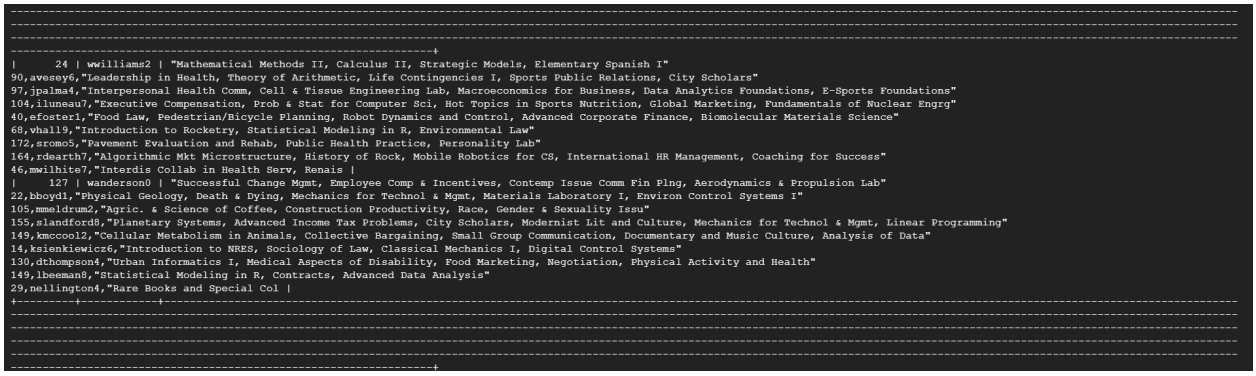
NetID	FirstName	LastName
|e7	Angela	Agee
|lcala0	Alice	Alcala
|alvarez8	Alvin	Alvarez
|ndrew7	Aileen	Andrew
|ias7	Amy	Arias
|ailey9	Adam	Bailey
|l4	Albert	Ball
|ballard6	Angela	Ballard
|rry3	Ada	Berry
|d5	Alvin	Bild
|abillings0	Alton	Billings
|gan9	Agnes	Bogan
|ooker7	Ann	Booker
|g0	Alicia	Borg
|orton5	Alfred	Borton

15 rows in set (0.01 sec)

SELECT * FROM Login LIMIT 15;This is an example of fifteen rows in our Login table, the formatting of the output in Google Cloud Platform is a bit weird which is why some of our NetIDs are getting cut off.

```
mysql> SELECT * FROM RSOMembers LIMIT 15;
+-------+--------------+-----------+-------------+
| RsoID | NetID        | FirstName | LastName    |
+-------+--------------+-----------+-------------+
|       | 0            | Betty     | Boyd        |
|     0 | aandrew7     | Aileen    | Andrew      |
|     0 | abooker7     | Ann       | Booker      |
|       | abunch9      | Amanda    | Bunch       |
|     0 | acunningham3 | Ann       | Cunningham  |
|     0 | aengler5     | Aaron     | Engler      |
|     0 | agaffney8    | Alicia    | Gaffney     |
|       | ahope8       | Antonina  | Hope        |
|       | amagee4      | Alvin     | Magee       |
|     0 | amccloud7    | Anita     | Mccloud     |
|     0 | amurphy3     | Alexander | Murphy      |
|     0 | aparker5     | Angel     | Parker      |
|       | aperez0      | Agnes     | Perez       |
|     0 | arobinson2   | Abe       | Robinson    |
|       | asears9      | Angel     | Sears       |
+-------+--------------+-----------+-------------+
15 rows in set (0.00 sec)
```

SELECT * FROM RSOMembers LIMIT 15; This is an example of our RSOMembers table, again the format of the query output in GCP is weird.



SELECT * FROM Majors LIMIT 15;This is a screenshot of our output in our Majors table where we select all the attributes in the table. For some reason, our CSV file doesn't get translated properly to a SQL relation. We are looking into why this is happening and we are trying to get this problem solved. **If a TA could take a look at our Major.csv file in our GitHub under /src/sql/CSVs, and give us feedback on why it's potentially not working in Google Cloud Platform that would be great.**

Three Tables with 1000+ Rows:

Students

```
mysql> SELECT COUNT(*) FROM Students;
+----------+
| COUNT(*) |
+----------+
|     2979 |
+----------+
1 row in set (0.01 sec)
```

Interests

```
mysql> SELECT COUNT(*) FROM Interests;
+----------+
| COUNT(*) |
+----------+
|     7659 |
+----------+
1 row in set (0.01 sec)
```

RSOMembers

```
mysql> SELECT COUNT(*) FROM RSOMembers;
+----------+
| COUNT(*) |
+----------+
|     2200 |
+----------+
1 row in set (0.01 sec)
```

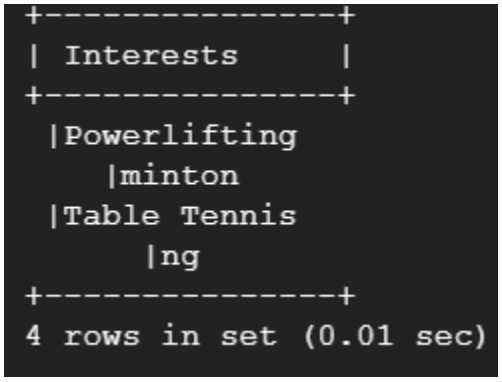
Login

```
mysql> SELECT COUNT(*) FROM Login;
+----------+
| COUNT(*) |
+----------+
|     2979 |
+----------+
1 row in set (0.01 sec)
```

Advanced Queries

Advanced Subquery 1

- SELECT DISTINCT i.Interests FROM Students s NATURAL JOIN Interests i WHERE i.InterestID IN (SELECT i2.InterestID FROM Interests i2 GROUP BY InterestID HAVING COUNT(InterestID) > 850) LIMIT 15;
    - This subquery finds the students who have an interest which is popular, that is to say, the interest has more than 850 people interested in it.



Explain Analyze

- This is the explain analyze analysis command we used
- EXPLAIN ANALYZE SELECT DISTINCT i.Interests FROM Students s NATURAL JOIN Interests i WHERE i.InterestID IN (SELECT i2.InterestID FROM Interests i2 GROUP BY InterestID HAVING COUNT(InterestID) > 850);

Before Indexing on InterestID

- This is the explain analyze analysis before we used indexing, and running it a few times resulted in varying times of 0.01 and 0.02 seconds. Our data only has 9 different interests, so searching for the different interests would yield such low lookup times.

```
| -> Table scan on <temporary>  (cost=0.01..98.24 rows=7659) (actual time=0.002..0.002 rows=4 loops=1)
    -> Temporary table with deduplication  (cost=4220.71..4318.94 rows=7659) (actual time=12.882..12.883 rows=4 loops=1)
        -> Nested loop inner join  (cost=3454.80 rows=7659) (actual time=2.899..11.471 rows=3470 loops=1)
            -> Filter: <in_optimizer>(i.InterestID,i.InterestID in (select #2))  (cost=774.15 rows=7659) (actual time=2.790..5.564 rows=3470 loops=1)
                -> Table scan on i  (cost=774.15 rows=7659) (actual time=0.084..2.159 rows=7659 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((i.InterestID = `<materialized_subquery>`.InterestID))  (actual time=0.003..0.003 rows=0 loops=10)
                        -> Limit: 1 row(s)  (actual time=0.002..0.002 rows=0 loops=10)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (InterestID=i.InterestID)  (actual time=0.002..0.002 rows=0 loops=10)
                                -> Materialize with deduplication  (cost=2305.95..2305.95 rows=7659) (actual time=2.609..2.609 rows=4 loops=1)
                                    -> Filter: (count(i2.InterestID) > 850)  (cost=1540.05 rows=7659) (actual time=0.558..2.570 rows=4 loops=1)
                                        -> Group aggregate: count(i2.InterestID)  (cost=1540.05 rows=7659) (actual time=0.554..2.565 rows=9 loops=1)
                                            -> Index scan on i2 using PRIMARY  (cost=774.15 rows=7659) (actual time=0.051..2.042 rows=7659 loops=1)
            -> Limit: 1 row(s)  (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=3470)
                -> Single-row index lookup on s using PRIMARY (NetID=i.NetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3470)
```

After Indexing on InterestID

- This is the explain analyze analysis after we used indexing. For this query, we only wanted to extrapolate the interests themselves that had # of students > 850, so the only thing we could index would be InterestID. After applying indexing on InterestID, the time difference in the lookup was practically negligible. Running the SELECT query multiple times also gave various times of 0.02 and 0.01, showing that the indexing had no noticeable effect on our lookup. As a result, this advanced query is not advanced enough to show a considerable difference in runtime when using indexing, so in the future, we would definitely try to make a more advanced query to show a potential optimization.

After Indexing on NetID in Interests

```
| -> Table scan on <temporary>  (cost=0.01..98.24 rows=7659) (actual time=0.002..0.002 rows=4 loops=1)
    -> Temporary table with deduplication  (cost=4220.71..4318.94 rows=7659) (actual time=13.598..13.599 rows=4 loops=1)
        -> Nested loop inner join  (cost=3454.80 rows=7659) (actual time=2.509..12.246 rows=3470 loops=1)
            -> Filter: <in_optimizer>(i.InterestID,i.InterestID in (select #2))  (cost=774.15 rows=7659) (actual time=2.463..5.310 rows=3470 loops=1)
                -> Table scan on i  (cost=774.15 rows=7659) (actual time=0.136..2.254 rows=7659 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((i.InterestID = `<materialized_subquery>`.InterestID))  (actual time=0.002..0.002 rows=0 loops=10)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=10)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (InterestID=i.InterestID)  (actual time=0.001..0.001 rows=0 loops=10)
                                -> Materialize with deduplication  (cost=2305.95..2305.95 rows=7659) (actual time=2.338..2.338 rows=4 loops=1)
                                    -> Filter: (count(i2.InterestID) > 850)  (cost=1540.05 rows=7659) (actual time=0.292..2.302 rows=4 loops=1)
                                        -> Group aggregate: count(i2.InterestID)  (cost=1540.05 rows=7659) (actual time=0.290..2.299 rows=9 loops=1)
                                            -> Index scan on i2 using PRIMARY  (cost=774.15 rows=7659) (actual time=0.038..1.856 rows=7659 loops=1)
            -> Limit: 1 row(s)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3470)
                -> Single-row index lookup on s using PRIMARY (NetID=i.NetID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3470)
```

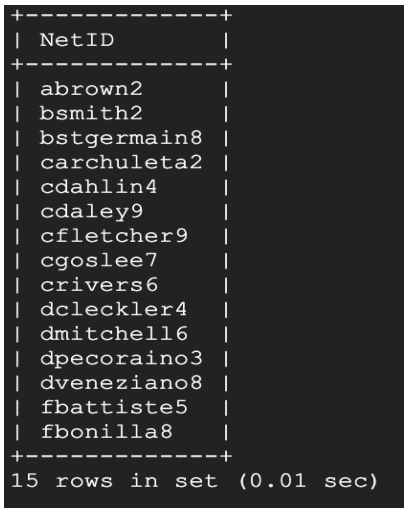
After Indexing on NetID in Students

```
| -> Table scan on <temporary>  (cost=0.01..98.24 rows=7659) (actual time=0.002..0.002 rows=4 loops=1)
    -> Temporary table with deduplication  (cost=4220.71..4318.94 rows=7659) (actual time=13.287..13.288 rows=4 loops=1)
        -> Nested loop inner join  (cost=3454.80 rows=7659) (actual time=2.335..11.895 rows=3470 loops=1)
            -> Filter: <in_optimizer>(i.InterestID,i.InterestID in (select #2))  (cost=774.15 rows=7659) (actual time=2.312..5.514 rows=3470 loops=1)
                -> Table scan on i  (cost=774.15 rows=7659) (actual time=0.061..2.556 rows=7659 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((i.InterestID = `<materialized_subquery>`.InterestID))  (actual time=0.002..0.002 rows=0 loops=10)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=10)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (InterestID=i.InterestID)  (actual time=0.001..0.001 rows=0 loops=10)
                                -> Materialize with deduplication  (cost=2305.95..2305.95 rows=7659) (actual time=2.267..2.267 rows=4 loops=1)
                                    -> Filter: (count(i2.InterestID) > 850)  (cost=1540.05 rows=7659) (actual time=0.297..2.235 rows=4 loops=1)
                                        -> Group aggregate: count(i2.InterestID)  (cost=1540.05 rows=7659) (actual time=0.296..2.232 rows=9 loops=1)
                                            -> Index scan on i2 using PRIMARY  (cost=774.15 rows=7659) (actual time=0.050..1.808 rows=7659 loops=1)
            -> Limit: 1 row(s)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3470)
                -> Single-row index lookup on s using PRIMARY (NetID=i.NetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3470)
```

- The following two indexes on the NetID in different tables yielded the same results as the first index, giving varying runtimes of 0.02 and 0.01. As we are not using either NetID in the advanced query, this makes sense as it should not have an effect on the resulting runtime of the advanced query.

Advanced Subquery 2

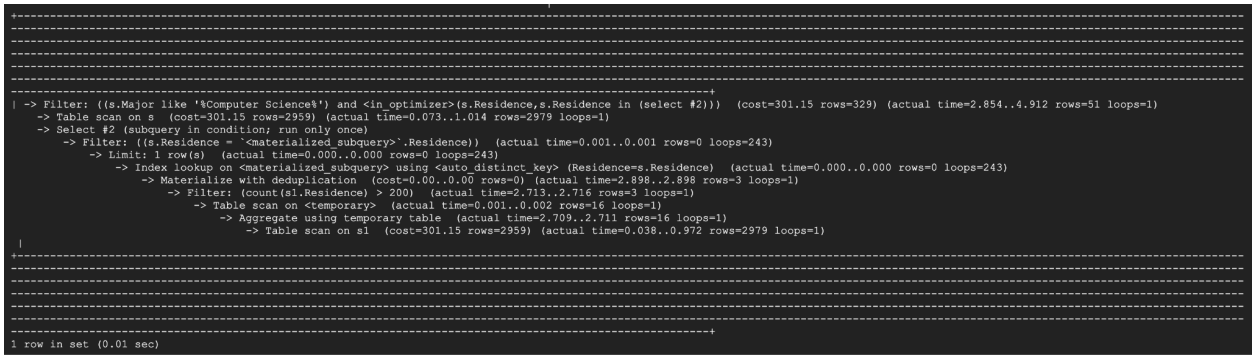
- SELECT s.NetID FROM Students s WHERE s.Major LIKE "%Computer Science%" AND s.Residence IN (SELECT Residence FROM Students s1 GROUP BY Residence HAVING COUNT(Residence) > 200) LIMIT 1



```
+-------------+
| NetID       |
+-------------+
| abrown2     |
| bsmith2     |
| bstgermain8 |
| carchuleta2 |
| cdahlin4    |
| cdaley9     |
| cfletcher9  |
| cgoslee7    |
| crivers6    |
| dcleckler4  |
| dmitchell6  |
| dpecoraino3 |
| dveneziano8 |
| fbattiste5  |
| fbonilla8   |
+-------------+
15 rows in set (0.01 sec)
```

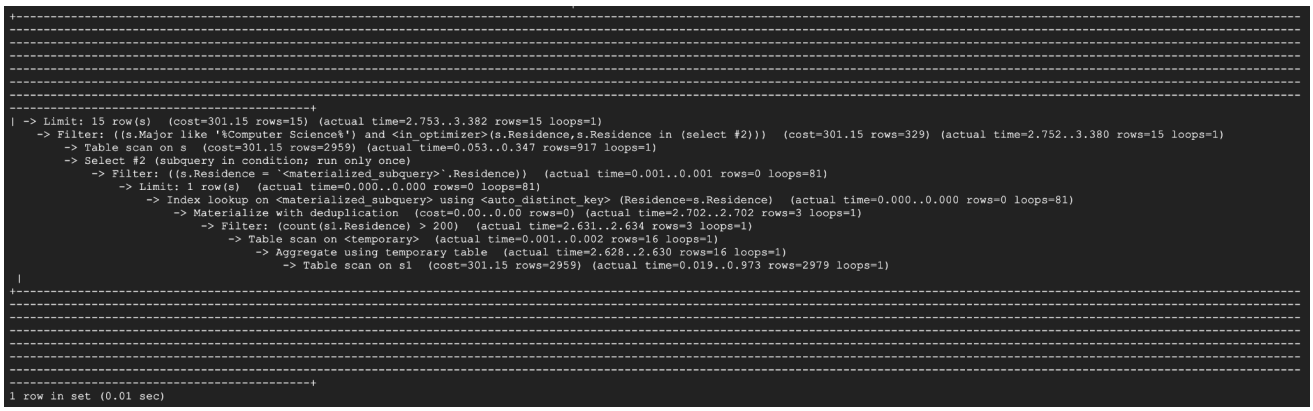
In this query, we are using subquery and group by SQL concepts. The purpose of this query is to obtain all of the Computer Science majors that live in a dorm that houses at least 200 people. The Subquery in this query returns the residence halls where there are at least 200 people. We are using GROUP BY within this subquery in order to count all residences in the same living location. From this subquery, our main query's WHERE clause has an AND between our subquery and students whose major is Computer Science. What this is accomplishing is our SELECT clause will only return students of major Computer Science AND live in a dorm of at least 200 people.

Before Indexing



```
| -> Filter: ((s.Major like '%Computer Science%') and <in_optimizer>(s.Residence,s.Residence in (select #2)))  (cost=301.15 rows=329) (actual time=2.854..4.912 rows=51 loops=1)
    -> Table scan on s  (cost=301.15 rows=2959) (actual time=0.073..1.014 rows=2979 loops=1)
    -> Select #2 (subquery in condition; run only once)
        -> Filter: ((s.Residence = `<materialized_subquery>`.Residence))  (actual time=0.001..0.001 rows=0 loops=243)
            -> Limit: 1 row(s)  (actual time=0.000..0.000 rows=0 loops=243)
                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (Residence=s.Residence)  (actual time=0.000..0.000 rows=0 loops=243)
                    -> Materialize with deduplication  (cost=0.00..0.00 rows=0) (actual time=2.898..2.898 rows=3 loops=1)
                        -> Filter: (count(s1.Residence) > 200)  (actual time=2.713..2.716 rows=3 loops=1)
                            -> Table scan on <temporary>  (actual time=0.001..0.002 rows=16 loops=1)
                                -> Aggregate using temporary table  (actual time=2.709..2.711 rows=16 loops=1)
                                    -> Table scan on s1  (cost=301.15 rows=2959) (actual time=0.038..0.972 rows=2979 loops=1)
 |
1 row in set (0.01 sec)
```

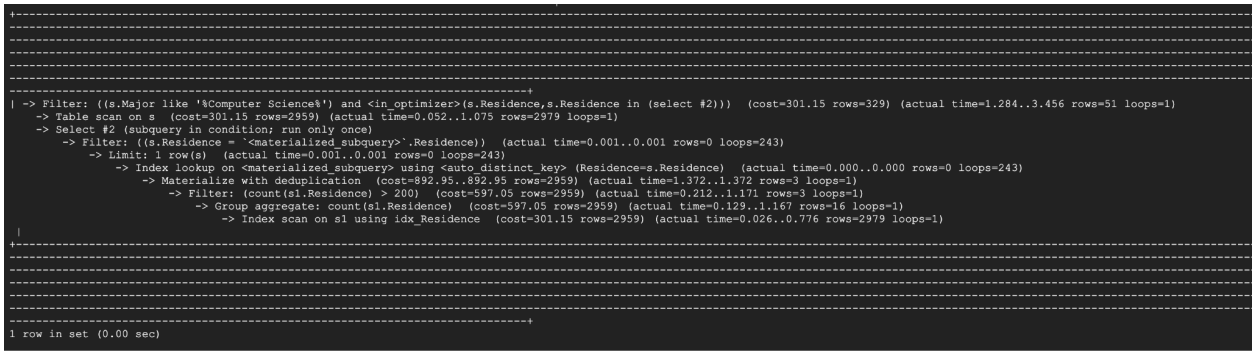
After Indexing on NetId



```
| -> Limit: 15 row(s)  (cost=301.15 rows=15) (actual time=2.753..3.382 rows=15 loops=1)
    -> Filter: ((s.Major like '%Computer Science%') and <in_optimizer>(s.Residence,s.Residence in (select #2)))  (cost=301.15 rows=329) (actual time=2.752..3.380 rows=15 loops=1)
        -> Table scan on s  (cost=301.15 rows=2959) (actual time=0.053..0.347 rows=917 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Filter: ((s.Residence = `<materialized_subquery>`.Residence))  (actual time=0.001..0.001 rows=0 loops=81)
                -> Limit: 1 row(s)  (actual time=0.000..0.000 rows=0 loops=81)
                    -> Index lookup on <materialized_subquery> using <auto_distinct_key> (Residence=s.Residence)  (actual time=0.000..0.000 rows=0 loops=81)
                        -> Materialize with deduplication  (cost=0.00..0.00 rows=0) (actual time=2.702..2.702 rows=3 loops=1)
                            -> Filter: (count(s1.Residence) > 200)  (actual time=2.631..2.634 rows=3 loops=1)
                                -> Table scan on <temporary>  (actual time=0.001..0.002 rows=16 loops=1)
                                    -> Aggregate using temporary table  (actual time=2.628..2.630 rows=16 loops=1)
                                        -> Table scan on s1  (cost=301.15 rows=2959) (actual time=0.019..0.973 rows=2979 loops=1)
 |
1 row in set (0.01 sec)
```

CREATE INDEX idx_NetId ON Students(NetId). After we indexed on NetId, we didn't see much of a performance boost since our query returns a relatively small amount of data. As a matter of fact, the exact same commands get run after we run the same EXPLAIN ANALYZE output, so our indexing is doing nothing.

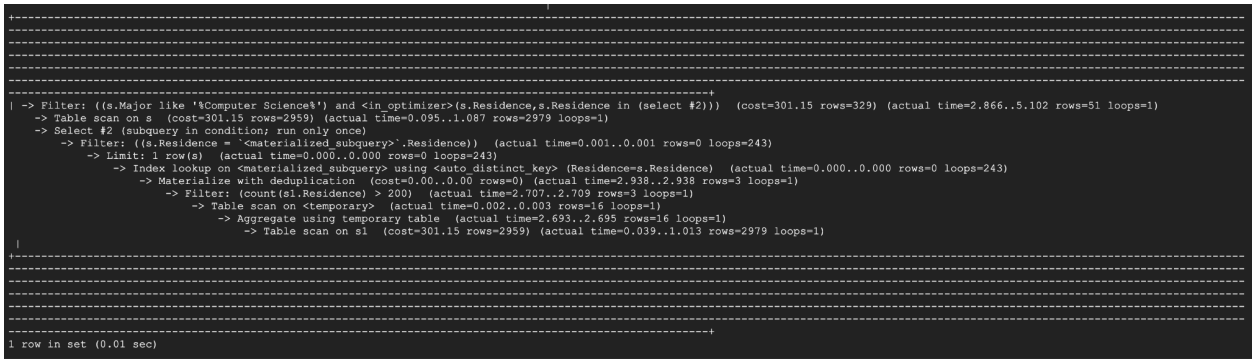
After Indexing on Residence



```
| -> Filter: ((s.Major like '%Computer Science%') and <in_optimizer>(s.Residence,s.Residence in (select #2)))  (cost=301.15 rows=329) (actual time=1.284..3.456 rows=51 loops=1)
    -> Table scan on s  (cost=301.15 rows=2959) (actual time=0.052..1.075 rows=2979 loops=1)
    -> Select #2 (subquery in condition; run only once)
        -> Filter: ((s.Residence = `<materialized_subquery>`.Residence))  (actual time=0.001..0.001 rows=0 loops=243)
            -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=243)
                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (Residence=s.Residence)  (actual time=0.000..0.000 rows=0 loops=243)
                    -> Materialize with deduplication  (cost=892.95..892.95 rows=2959) (actual time=1.372..1.372 rows=3 loops=1)
                        -> Filter: (count(s1.Residence) > 200)  (cost=597.05 rows=2959) (actual time=0.212..1.171 rows=3 loops=1)
                            -> Group aggregate: count(s1.Residence)  (cost=597.05 rows=2959) (actual time=0.129..1.167 rows=16 loops=1)
                                -> Index scan on s1 using idx_Residence  (cost=301.15 rows=2959) (actual time=0.026..0.776 rows=2979 loops=1)
 |
1 row in set (0.00 sec)
```

CREATE INDEX idx_Residence ON Students(Residence). After we indexed on Residence, we see a decent speed up since there are fewer steps in our EXPLAIN ANALYZE. There are a lot fewer rows in the original scan, which explains the significant speed up we see when we index with Residence.

After Indexing On Major



CREATE INDEX idx_Major ON Students(Residence)

So looking at our Analysis, we noticed that there is visible no speedup when creating an index for Major. This is likely because our data is not significant enough, causing minimal speedup in our query. If we happened to have a larger data set that is more diverse, we believe that the speedup would be more prominent. To understand this loss, we are going to create a larger pool of data, and we will retest this hypothesis in the future.