Query 1:

```
(select avg(Price) from Products
where Product_Name='iPhone 12'
group by Product_Name)
union
(select avg(Price) from Products
where Product_Name = 'iPhone 13'
group by Product_Name);
```

```
mysql> (select avg(Price) from Products
    -> where Product_Name='iPhone 12'
    -> group by Product_Name)
    -> union
    -> (select avg(Price) from Products
    -> where Product_Name = 'iPhone 13'
    -> group by Product_Name)
    -> limit 15;
+--------------------+
| avg(Price)         |
+--------------------+
| 437.35714285714283 |
|  493.2105263157895 |
+--------------------+
2 rows in set (0.00 sec)
```

Then we use explain analyze command to measure the performance.

```
mysql> explain analyze (select avg(Price) from Products where Product_Name='iPhone 12' group by Product_Name) union (select avg(Price) from Products where P
roduct_Name = 'iPhone 13' group by P
roduct_Name) ;

+----------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                    |
+----------------------------------------------------------------------------------------------------------------------------+
| -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.001 rows=2 loops=1)
    -> Union materialize with deduplication  (actual time=2.042..2.043 rows=2 loops=1)
        -> Group aggregate: avg(Products.Price)  (actual time=1.584..1.584 rows=1 loops=1)
            -> Filter: (Products.Product_Name = 'iPhone 12')  (cost=101.50 rows=100) (actual time=1.287..1.557 rows=28 loops=1)
                -> Table scan on Products  (cost=101.50 rows=1000) (actual time=0.060..0.275 rows=1000 loops=1)
        -> Group aggregate: avg(Products.Price)  (actual time=0.287..0.287 rows=1 loops=1)
            -> Filter: (Products.Product_Name = 'iPhone 13')  (cost=101.50 rows=100) (actual time=0.031..0.280 rows=76 loops=1)
                -> Table scan on Products  (cost=101.50 rows=1000) (actual time=0.025..0.223 rows=1000 loops=1)
 |
```

Create an index price_idx on Product_Name attributes in table Products

```
mysql> create index price_idx on Products(Product_Name(10));
Query OK, 0 rows affected (0.10 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Use explain analyze command to measure the performance of index.

```
| -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.001 rows=2 loops=1)
    -> Union materialize with deduplication  (actual time=0.311..0.311 rows=2 loops=1)
        -> Group aggregate: avg(Products.Price)  (actual time=0.190..0.190 rows=1 loops=1)
            -> Filter: (Products.Product_Name = 'iPhone 12')  (cost=7.30 rows=28) (actual time=0.175..0.181 rows=28 loops=1)
                -> Index lookup on Products using price_idx (Product_Name='iPhone 12')  (cost=7.30 rows=28) (actual time=0.142..0.146 rows=28 loops=1)
        -> Group aggregate: avg(Products.Price)  (actual time=0.109..0.109 rows=1 loops=1)
            -> Filter: (Products.Product_Name = 'iPhone 13')  (cost=12.10 rows=76) (actual time=0.091..0.103 rows=76 loops=1)
                -> Index lookup on Products using price_idx (Product_Name='iPhone 13')  (cost=12.10 rows=76) (actual time=0.090..0.097 rows=76 loops=1)
 |
```

In line 2, the upper bound and lower bound of the time-cost of the Union is reduced from 2.042 to 0.311. In line 3, the actual time cost in Group aggregate is reduced from 1.584 to 0.190. In the filter step, the cost is reduced from 101 to 7.30, the scanned rows number is reduced from 100 to 28, and the index lookup is used when scanning Products in this step. The same goes for the next group aggregate step.

Therefore, using the index in this query shortens the time cost and increases the query performance.

Query 2:

select c.Product_ID, p.Product_Name from

Classification c join Products p

on c.Product_Id = p.Product_Id

where p.Product_ID in

(select Product_ID from Products where Price >700)

```
mysql> select c.Product_ID, p.Product_Name from
    -> Classification c join Products p
    -> on c.Product_Id = p.Product_Id
    -> where p.Product_ID in
    -> (select Product_ID from Products where Price >700)
    -> limit 15;
+------------+--------------+
| Product_ID | Product_Name |
+------------+--------------+
|        872 | iPad Pro     |
|        747 | laptop       |
|        875 | iPad Pro     |
|        764 | laptop       |
|        889 | iPad Pro     |
|        922 | iPad Pro     |
|        864 | iPad Pro     |
|        748 | laptop       |
|        715 | laptop       |
|        881 | iPad Pro     |
|        798 | laptop       |
|        773 | laptop       |
|        785 | laptop       |
|        763 | laptop       |
|        800 | laptop       |
+------------+--------------+
15 rows in set (0.01 sec)
```

First, use explain analyze to measure the performance of this query.

```
| -> Nested loop inner join  (cost=334.81 rows=333) (actual time=0.333..0.602 rows=47 loops=1)
    -> Nested loop inner join  (cost=218.15 rows=333) (actual time=0.327..0.534 rows=47 loops=1)
        -> Filter: (Products.Price > 700)  (cost=101.50 rows=333) (actual time=0.281..0.401 rows=47 l
oops=1)
            -> Table scan on Products  (cost=101.50 rows=1000) (actual time=0.041..0.309 rows=1000 lo
ops=1)
        -> Index lookup on c using Product_ID (Product_ID=Products.Product_ID)  (cost=0.25 rows=1) (a
ctual time=0.002..0.002 rows=1 loops=47)
    -> Single-row index lookup on p using PRIMARY (Product_ID=Products.Product_ID)  (cost=0.25 rows=1
) (actual time=0.001..0.001 rows=1 loops=47)
```

Create index price_idx on table Classification's column Product_ID, because the Product_ID is a Foreign key and is often used when joining other tables. By creating an index on such an attribute, we could increase the query speed.

```
mysql> create index price_idx on Products(Price);
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Then we measure the performance of the same query:

```
--+
| -> Nested loop inner join  (cost=42.98 rows=47) (actual time=0.167..0.344 rows=47 loops=1)
    -> Nested loop inner join  (cost=26.53 rows=47) (actual time=0.155..0.257 rows=47 loops=1)
        -> Filter: (Products.Price > 700)  (cost=10.08 rows=47) (actual time=0.134..0.146 rows=47 loo
ps=1)
            -> Index range scan on Products using price_idx  (cost=10.08 rows=47) (actual time=0.077.
.0.084 rows=47 loops=1)
        -> Index lookup on c using Product_ID (Product_ID=Products.Product_ID)  (cost=0.25 rows=1) (a
ctual time=0.002..0.002 rows=1 loops=47)
    -> Single-row index lookup on p using PRIMARY (Product_ID=Products.Product_ID)  (cost=0.25 rows=1
) (actual time=0.002..0.002 rows=1 loops=47)
 |
```

In the first step, Nested loop inner join, the cost is reduced from 334.81 to 42.98, and the number of rows scanned is reduced from 333 to 47. But the actual time bounds are reduced from (0.319, 0.624) to (0.233, 0.472). In line 4, the scanned method changes from table scan to Index range scan, which reduces the time cost from 101.50 to 10.08. Therefore, this index improves the query performance significantly.

Query 3:

Select u.Name, u.Email, p.Product_Name

from Products p join Users u

on p.Seller_ID = u.Student_ID

where u.Department in

(select Department from Users

Group by Department

having count(Department)>20)

limit 15;

```
mysql> Select u.Name, u.Email, p.Product_Name
    -> from Products p join Users u
    -> on p.Seller_ID = u.Student_ID
    -> where u.Department in
    -> (select Department from Users
    -> Group by Department
    -> having count(Department)>20)
    -> limit 15;
+----------------+----------------------------------+----------------------+
| Name           | Email                            | Product_Name         |
+----------------+----------------------------------+----------------------+
| Alyssa Cook    | ultrices.sit@hotmail.edu         | iPhone 12            |
| Lunea Curry    | ipsum.leo@protonmail.com         | iPhone 12            |
| Lunea Curry    | ipsum.leo@protonmail.com         | iPhone 11            |
| Lunea Curry    | ipsum.leo@protonmail.com         | Surviving The Chasm  |
| Aline Phelps   | vulputate.mauris.sagittis@aol.org | iPad Pro            |
| Giacomo Carey  | massa@yahoo.ca                   | coat                 |
| Giacomo Carey  | massa@yahoo.ca                   | coat                 |
| Cheyenne Woods | donec.luctus.aliquet@hotmail.net | iPad Air             |
| Carson Hoover  | nec@hotmail.net                  | shoes                |
| Carson Hoover  | nec@hotmail.net                  | iPad Pro             |
| Tamara Morrow  | laoreet.posuere@aol.com          | Depths Of The River  |
| Tamara Morrow  | laoreet.posuere@aol.com          | paints               |
| Whoopi Francis | donec.consectetuer@protonmail.net | iPhone 12           |
| Whoopi Francis | donec.consectetuer@protonmail.net | Parrot Of A Woman   |
| Whoopi Francis | donec.consectetuer@protonmail.net | shoes               |
+----------------+----------------------------------+----------------------+
15 rows in set (0.01 sec)
```

Then we analyze the performance using the explain analyze command.

```
| -> Nested loop inner join  (cost=660.58 rows=1584) (actual time=0.928..6.357 rows=1000 loops=1)
    -> Filter: <in_optimizer>(u.Department,u.Department in (select #2))  (cost=106.23 rows=1001) (actual time=0.882..1.769 ro
ws=1001 loops=1)
        -> Table scan on u  (cost=106.23 rows=1001) (actual time=0.048..0.567 rows=1001 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Filter: (count(Users.Department) > 20)  (actual time=0.794..0.797 rows=7 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.002 rows=7 loops=1)
                    -> Aggregate using temporary table  (actual time=0.791..0.793 rows=7 loops=1)
                        -> Table scan on Users  (cost=106.23 rows=1001) (actual time=0.018..0.257 rows=1001 loops=1)
    -> Index lookup on p using Seller_ID (Seller_ID=u.Student_ID)  (cost=0.40 rows=2) (actual time=0.004..0.004 rows=1 loops=
1001)
 |
```

Considering that the user's email should be unique, we add a unique index name_index on the Email attribute of table Users. Then we analyze the performance after adding the index:

```
mysql> create index email_idx on Users(Email);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze Select u.Name, u.Email, p.Product_Name from Products p join Users u  on p.Seller_ID = u.Student_ID whe
re u.Department in (select Department from Users group by Department having count(Department)>20);
```

```
---------------------------------------------------+
| -> Nested loop inner join  (cost=660.58 rows=1584) (actual time=0.708..3.898 rows=1000 loops=1)
    -> Filter: <in_optimizer>(u.Department,u.Department in (select #2))  (cost=106.23 rows=1001) (actual time=0.692..1.206 ro
ws=1001 loops=1)
        -> Table scan on u  (cost=106.23 rows=1001) (actual time=0.032..0.317 rows=1001 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Filter: (count(Users.Department) > 20)  (actual time=0.647..0.648 rows=7 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.001 rows=7 loops=1)
                    -> Aggregate using temporary table  (actual time=0.645..0.646 rows=7 loops=1)
                        -> Table scan on Users  (cost=106.23 rows=1001) (actual time=0.017..0.242 rows=1001 loops=1)
    -> Index lookup on p using Seller_ID (Seller_ID=u.Student_ID)  (cost=0.40 rows=2) (actual time=0.002..0.002 rows=1 loops=
1001)
 |
```

There are only some slight reductions at the actual time' s upper and lower bounds. And there is no significant reduction in total time cost in each step. This is because the unique index mainly serves to constrained data uniqueness rather than to improve the query speed.