

Database Design (CS 411 Project Stage 3)

We started with creating the database (marketplace) schema using the following DDL commands:

1. CREATE TABLE Customer (
 ID INT Primary Key,
 Name VARCHAR(255),
 Password VARCHAR(30)
);
2. CREATE TABLE Orders(
 ID INT Primary Key,
 deliveryLocation VARCHAR(100)
);
3. CREATE TABLE Product(
 ID INT Primary Key,
 Name VARCHAR(255),
 Brand VARCHAR(255),
 Price INT,
 Details VARCHAR(255)
);
4. CREATE TABLE Seller(
 ID INT Primary Key,
 Name VARCHAR(255),
 Location VARCHAR(100)
);
5. CREATE TABLE DeliveryTime(
 iLocation VARCHAR(100),
 fLocation VARCHAR(100),
 time(days) INT,
 Primary Key (iLocation, fLocation)
);
6. CREATE TABLE Places(
 custId INT,
 orderId INT,
 FOREIGN KEY (custId) REFERENCES Customer(ID),
 FOREIGN KEY (orderId) REFERENCES Orders(ID),
 Primary Key (custId, orderId)
);
7. CREATE TABLE Includes(
 orderId INT,
 productId INT,
 quantity INT,
 FOREIGN KEY (productId) REFERENCES Product(ID),
 FOREIGN KEY (orderId) REFERENCES Orders(ID),
 Primary Key (orderId, productId)

);

```
8. CREATE TABLE Sells(  
    sellerId INT,  
    productId INT,  
    FOREIGN KEY (productId) REFERENCES Product(ID),  
    FOREIGN KEY (sellerId) REFERENCES Seller(ID),  
    Primary Key (sellerId, productId)  
);
```

```
9. CREATE TABLE Reviews(  
    custId INT,  
    productId INT,  
    date DATE,  
    text VARCHAR(255),  
    rating INT,  
    FOREIGN KEY (productId) REFERENCES Product(ID),  
    FOREIGN KEY (custId) REFERENCES Customer(ID),  
    Primary Key (custId, productId)  
);
```

Having created the schema, we filled in the data in the above tables adhering to constraints defined. Most of our tables contain around 1000 entries.

We performed the following queries on our tables:

1. The first query was to output the number of products being shipped from each location which used 'location' of Sellers, and count of products from 'Sells' tables. The query is as follows:

```
SELECT sr.Location, COUNT(s.productId) as numProducts  
FROM Seller sr JOIN Sells s ON sr.ID = s.sellerId  
GROUP BY sr.Location  
ORDER BY numProducts desc;
```

The results are shown below:

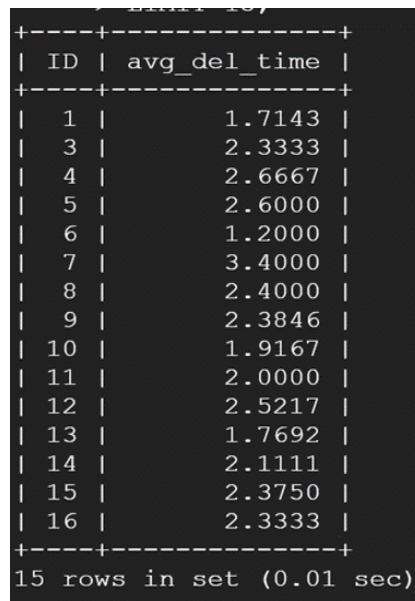
Location	numProducts
Las Vegas	110
Boston	107
Bloomington	91
Helena	87
Dallas	84
Milwaukee	66
Lansing	64
Seattle	64
Chicago	60
Kankakee	58
Portland	53
San Francisco	52
Austin	38
New York	33
Madison	27

15 rows in set (0.01 sec)

2. Another useful query was to find the average time for delivery of items in an order. The query is as follows:

```
SELECT o.ID, avg(d.`time(days)`) as avg_del_time
FROM Orders o JOIN Includes i ON o.ID = i.orderId JOIN Sells s ON i.productId =
s.productId JOIN
Seller se ON s.sellerId = se.ID JOIN DeliveryTime d
ON (se.Location = d.iLocation and o.delLocation = d.fLocation)
GROUP BY o.ID LIMIT 15;
```

Which gives the following result:



The screenshot shows a terminal window with a SQL query result. The result is a table with two columns: 'ID' and 'avg_del_time'. It contains 15 rows of data. At the bottom of the terminal output, it says '15 rows in set (0.01 sec)'.

ID	avg_del_time
1	1.7143
3	2.3333
4	2.6667
5	2.6000
6	1.2000
7	3.4000
8	2.4000
9	2.3846
10	1.9167
11	2.0000
12	2.5217
13	1.7692
14	2.1111
15	2.3750
16	2.3333

15 rows in set (0.01 sec)

Upon using EXPLAIN ANALYZE on the above two queries, the cost shown is really high in the second case and moderate in the first case. So, we tried to create index on location and productId for the first query, since location is a non-primary key attribute. However, we did not find any reasonable increase in query performance. We also created indices on Seller(ID) and Sells(sellerId) with no change in the cost of query as shown in the output of EXPLAIN ANALYZE.

Similarly, we also tried creating indices for the locations in different tables for the second query. In this case also, we could not find any speedup worth noting.

Therefore, we tried a simple query and tried to observe the speed up using index in this case. This is a simple query to select customers with first names starting with 'A' or last names starting with 'B':

```
EXPLAIN ANALYZE SELECT * FROM Customer
Where first_name LIKE 'A%' OR last_name LIKE 'B%';
```

The result is as follows:

```
mysql> EXPLAIN ANALYZE SELECT * FROM Customer
  -> Where first_name LIKE 'A%' OR last_name LIKE 'B%';
+-----+
| EXPLAIN |
+-----+
| -> Filter: ((Customer.first_name like 'A%') or (Customer.last_name like 'B%')) (cost=101.25 rows=210) (actual time=0.065..0.620 rows=179 loops=1)
  -> Table scan on Customer (cost=101.25 rows=1000) (actual time=0.059..0.413 rows=1000 loops=1)
|
```

We then created indices on first_name and last_name of the Customer table to see if the query performance improves. But, it did not improve. However, if we tried a single Boolean operator in the Where-clause (i.e., only first_name LIKE 'A%'), and repeated the same action, we find a reduction in cost from 101.25 as shown above to 35.36.

```
mysql> EXPLAIN ANALYZE SELECT * FROM Customer Where first_name LIKE 'A%';
+-----+
| EXPLAIN |
+-----+
| -> Index range scan on Customer using first_idx, with index condition: (Customer.first_name like 'A%') (cost=35.36 rows=78) (actual time=0.287..0.303 rows=78 loops=1)
|
+-----+
| row in set (0.00 sec) |
```

We concluded that with multiple Boolean operators in the Where-clause, a single index does not help in improving query performance. We will work on optimizing the query performance using indexes with our joins in the queries written above.