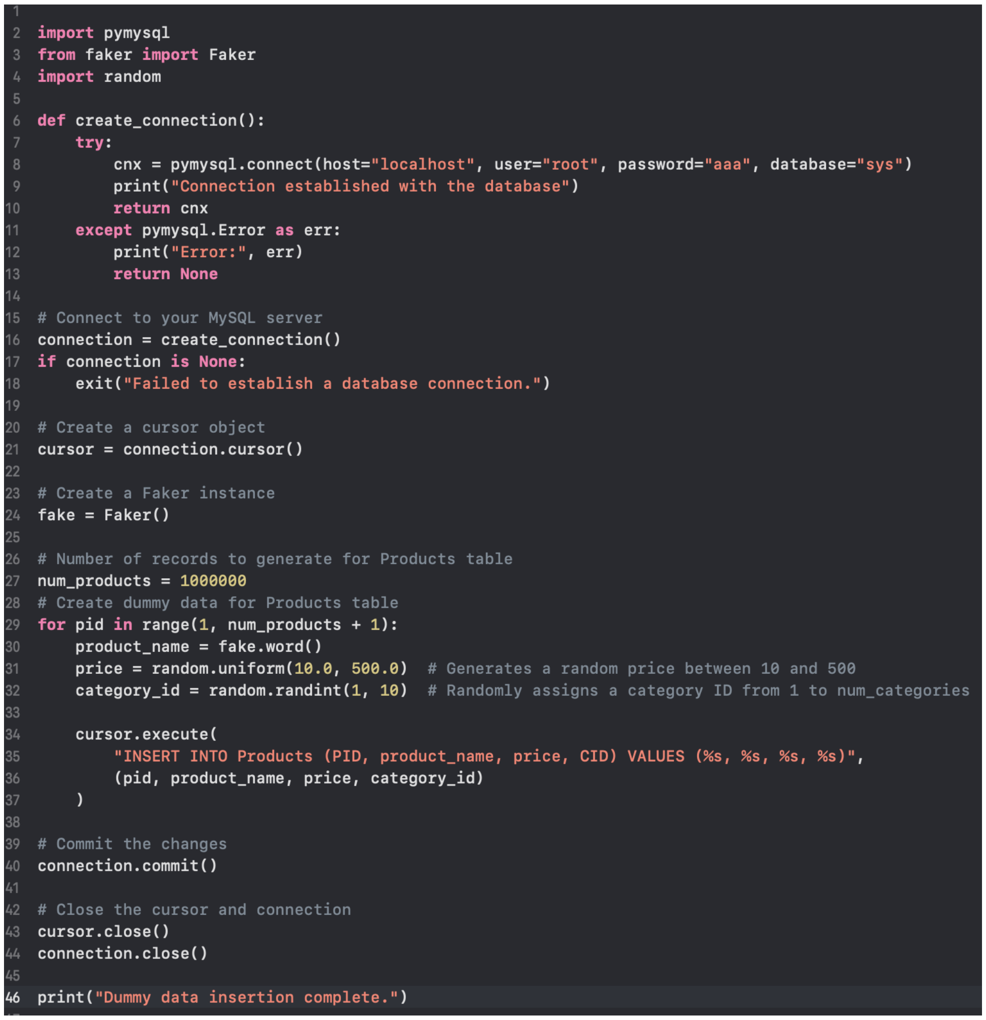
Cs 306 Project Phase 3

Muhammed Arda Akkan(29099), İdil Güler(29570), Halil İbrahim Umut Çolak(28879)

In the modern era, where databases burgeon with millions of data points, efficient navigation through this vast digital terrain is paramount. In Phase 3 of our project, we utilized Python through the PyCharm IDE to methodically add one million dummy records to our databases, initially without indexing. We meticulously measured query times in this state, then implemented indexing and re-measured. This comparative study highlighted the profound significance of indexing for operational efficiency in database management, revealing its critical role in expediting data retrieval in today's voluminous data environments.

Step 1: Connecting to Mysql Server and adding dummy data -> ARDA AKKAN

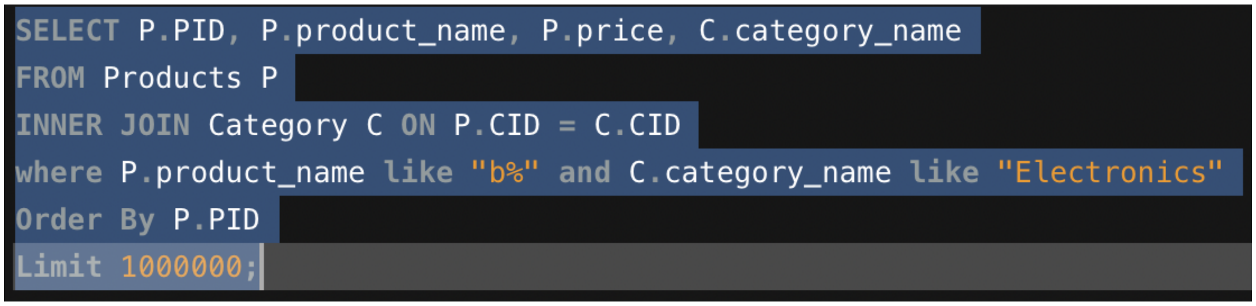
This screenshot illustrates how the python code which was revised from recitation codes used to connect Mysql server and add millions of dummy data into our tables. pymysql library enables us to manipulate our sequential databases by using python.



```python

import pymysql
from faker import Faker
import random

def create_connection():
    try:
        cnx = pymysql.connect(host="localhost", user="root", password="aaa", database="sys")
        print("Connection established with the database")
        return cnx
    except pymysql.Error as err:
        print("Error:", err)
        return None

# Connect to your MySQL server
connection = create_connection()
if connection is None:
    exit("Failed to establish a database connection.")

# Create a cursor object
cursor = connection.cursor()

# Create a Faker instance
fake = Faker()

# Number of records to generate for Products table
num_products = 1000000
# Create dummy data for Products table
for pid in range(1, num_products + 1):
    product_name = fake.word()
    price = random.uniform(10.0, 500.0)  # Generates a random price between 10 and 500
    category_id = random.randint(1, 10)  # Randomly assigns a category ID from 1 to num_categories

    cursor.execute(
        "INSERT INTO Products (PID, product_name, price, CID) VALUES (%s, %s, %s, %s)",
        (pid, product_name, price, category_id)
    )

# Commit the changes
connection.commit()

# Close the cursor and connection
cursor.close()
connection.close()

print("Dummy data insertion complete.")
```

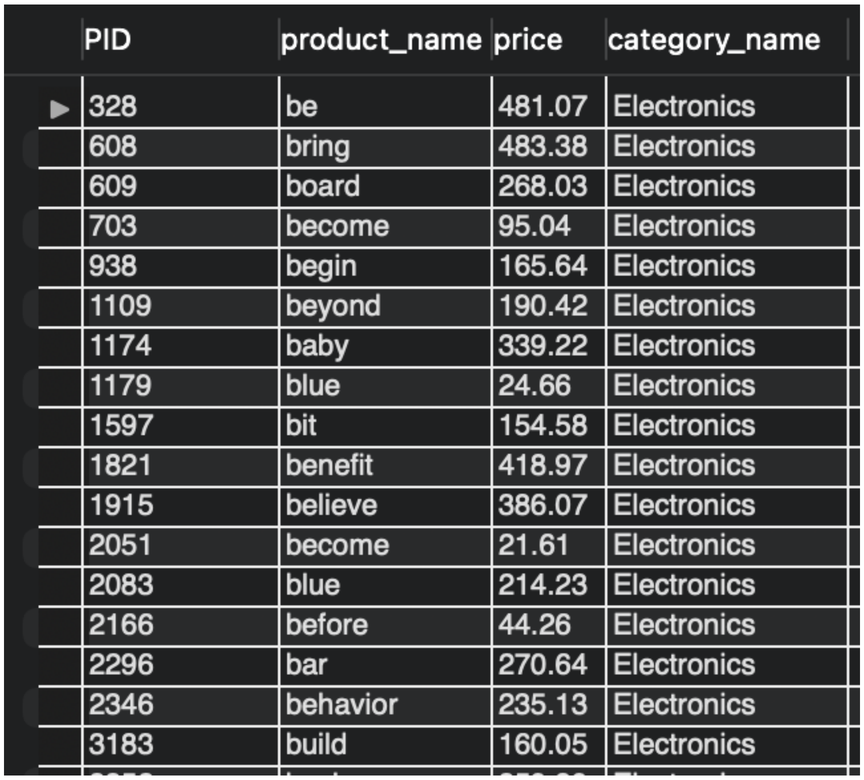
Step 2: Running sql query without using indexing-> ARDA AKKAN

In this query tables Category and Products are joined on attribute CID and then filtered by product name starting with letter b and category name Electronics.



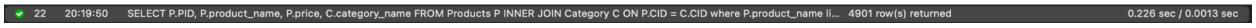
```
SELECT P.PID, P.product_name, P.price, C.category_name
FROM Products P
INNER JOIN Category C ON P.CID = C.CID
where P.product_name like "b%" and C.category_name like "Electronics"
Order By P.PID
Limit 1000000;
```

We are sure that data is inserted via the code runned on pycharm by looking at the result of the query..

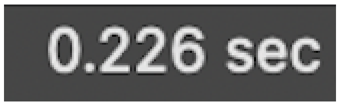


PID	product_name	price	category_name
328	be	481.07	Electronics
608	bring	483.38	Electronics
609	board	268.03	Electronics
703	become	95.04	Electronics
938	begin	165.64	Electronics
1109	beyond	190.42	Electronics
1174	baby	339.22	Electronics
1179	blue	24.66	Electronics
1597	bit	154.58	Electronics
1821	benefit	418.97	Electronics
1915	believe	386.07	Electronics
2051	become	21.61	Electronics
2083	blue	214.23	Electronics
2166	before	44.26	Electronics
2296	bar	270.64	Electronics
2346	behavior	235.13	Electronics
3183	build	160.05	Electronics

Total time of this sql query took approximately 0.226 seconds to execute.



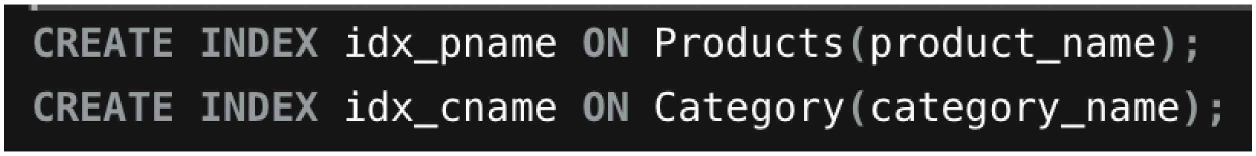
```
22   20:19:50   SELECT P.PID, P.product_name, P.price, C.category_name FROM Products P INNER JOIN Category C ON P.CID = C.CID where P.product_name li...   4901 row(s) returned   0.226 sec / 0.0013 sec
```



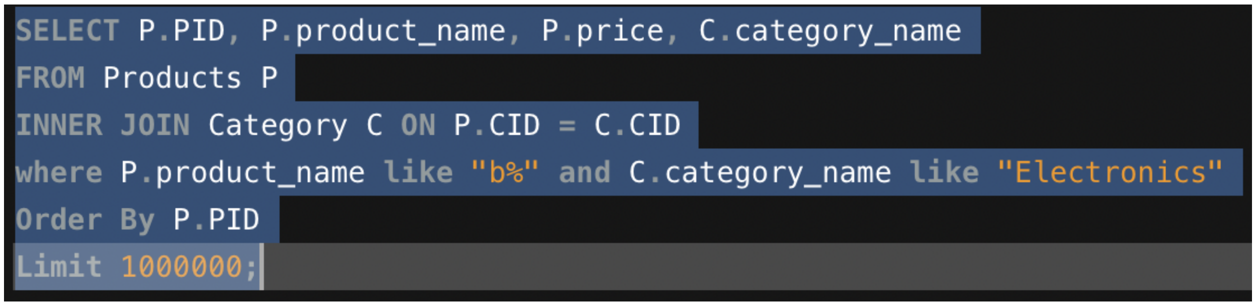
0.226 sec

Step 3: Running sql query with using indexing-> ARDA AKKAN

This time we will add indexes into both Products (on the attribute name of product) and Category (on the attribute name of Category) then execute exact same query to see results.

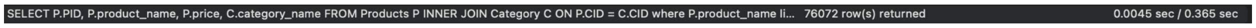


```sql
CREATE INDEX idx_pname ON Products(product_name);
CREATE INDEX idx_cname ON Category(category_name);
```

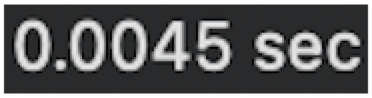


```sql
SELECT P.PID, P.product_name, P.price, C.category_name
FROM Products P
INNER JOIN Category C ON P.CID = C.CID
where P.product_name like "b%" and C.category_name like "Electronics"
Order By P.PID
Limit 1000000;
```

After executing exact same query, we see a significant decrease for the time taken for query.



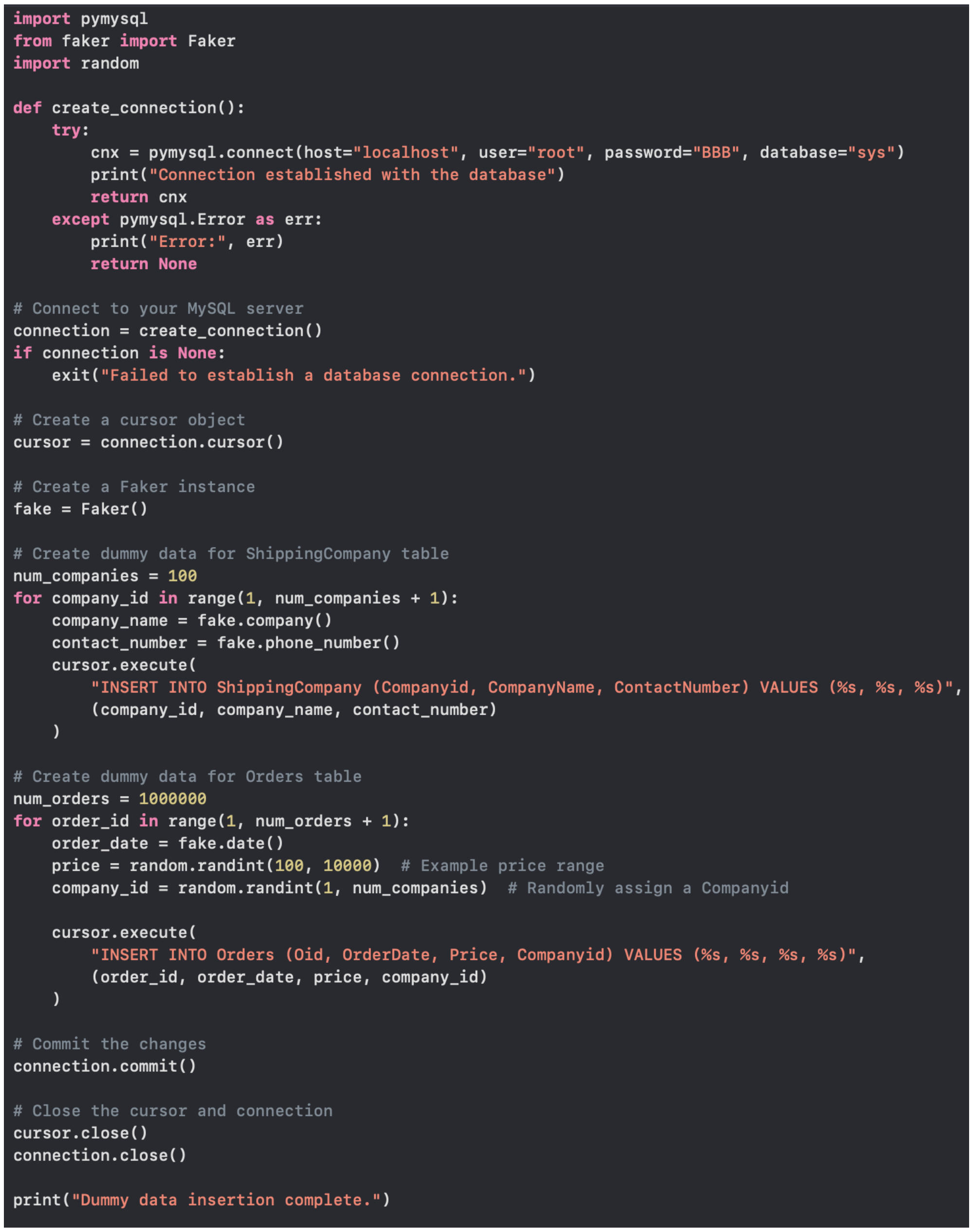
```
SELECT P.PID, P.product_name, P.price, C.category_name FROM Products P INNER JOIN Category C ON P.CID = C.CID where P.product_name li...   76072 row(s) returned   0.0045 sec / 0.365 sec
```



0.0045 sec

Step 1: Connecting to Mysql Server and adding dummy data -> IDIL GULER

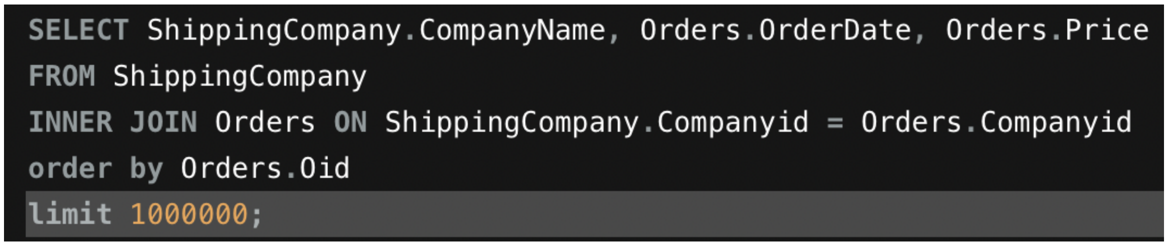
The screenshot depicts the modified Python script, adapted from example code, utilized to establish a connection with the MySQL server and insert millions of placeholder records into our tables. The pymysql library facilitates the management of our relational databases through Python.



```python
import pymysql
from faker import Faker
import random

def create_connection():
    try:
        cnx = pymysql.connect(host="localhost", user="root", password="BBB", database="sys")
        print("Connection established with the database")
        return cnx
    except pymysql.Error as err:
        print("Error:", err)
        return None

# Connect to your MySQL server
connection = create_connection()
if connection is None:
    exit("Failed to establish a database connection.")

# Create a cursor object
cursor = connection.cursor()

# Create a Faker instance
fake = Faker()

# Create dummy data for ShippingCompany table
num_companies = 100
for company_id in range(1, num_companies + 1):
    company_name = fake.company()
    contact_number = fake.phone_number()
    cursor.execute(
        "INSERT INTO ShippingCompany (Companyid, CompanyName, ContactNumber) VALUES (%s, %s, %s)",
        (company_id, company_name, contact_number)
    )

# Create dummy data for Orders table
num_orders = 1000000
for order_id in range(1, num_orders + 1):
    order_date = fake.date()
    price = random.randint(100, 10000)  # Example price range
    company_id = random.randint(1, num_companies)  # Randomly assign a Companyid

    cursor.execute(
        "INSERT INTO Orders (Oid, OrderDate, Price, Companyid) VALUES (%s, %s, %s, %s)",
        (order_id, order_date, price, company_id)
    )

# Commit the changes
connection.commit()

# Close the cursor and connection
cursor.close()
connection.close()

print("Dummy data insertion complete.")
```

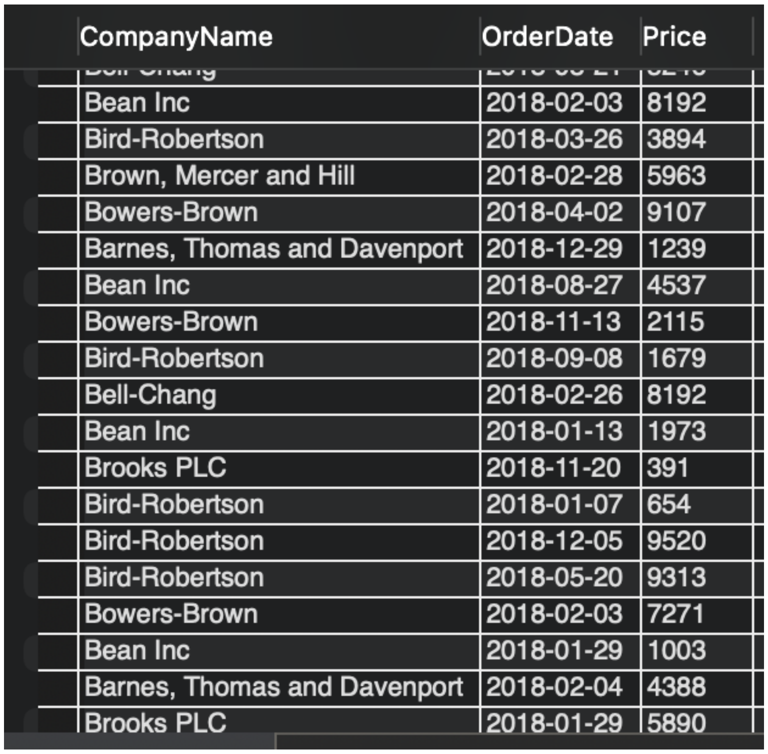
Step 2: Running sql query without using indexing-> IDIL GULER

The query performs an INNER JOIN between the ShippingCompany and Orders tables on the Companyid attribute. It selects the CompanyName from ShippingCompany table, and the OrderDate and Price from the Orders table. The results are ordered by the Oid of the Orders table. Furthermore, the query is limited to return a maximum of 1,000,000 records.



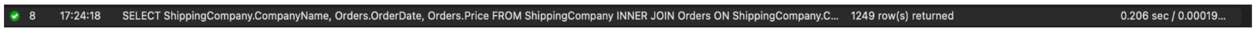
```
SELECT ShippingCompany.CompanyName, Orders.OrderDate, Orders.Price
FROM ShippingCompany
INNER JOIN Orders ON ShippingCompany.Companyid = Orders.Companyid
order by Orders.Oid
limit 1000000;
```

By examining the outcome of the query, we can confirm that data insertion was successfully executed through the code operated in PyCharm

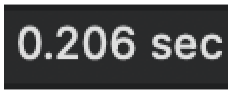


CompanyName	OrderDate	Price
Bean Inc	2018-02-03	8192
Bird-Robertson	2018-03-26	3894
Brown, Mercer and Hill	2018-02-28	5963
Bowers-Brown	2018-04-02	9107
Barnes, Thomas and Davenport	2018-12-29	1239
Bean Inc	2018-08-27	4537
Bowers-Brown	2018-11-13	2115
Bird-Robertson	2018-09-08	1679
Bell-Chang	2018-02-26	8192
Bean Inc	2018-01-13	1973
Brooks PLC	2018-11-20	391
Bird-Robertson	2018-01-07	654
Bird-Robertson	2018-12-05	9520
Bird-Robertson	2018-05-20	9313
Bowers-Brown	2018-02-03	7271
Bean Inc	2018-01-29	1003
Barnes, Thomas and Davenport	2018-02-04	4388
Brooks PLC	2018-01-29	5890

The execution of this SQL query was approximately completed in 0.206 seconds.



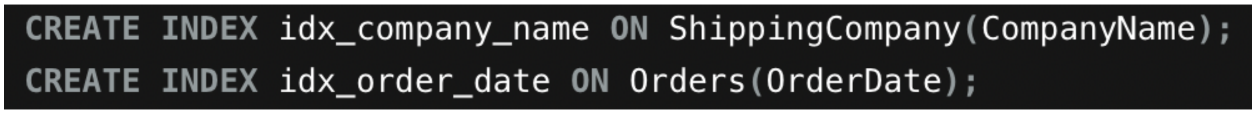
```
8    17:24:18    SELECT ShippingCompany.CompanyName, Orders.OrderDate, Orders.Price FROM ShippingCompany INNER JOIN Orders ON ShippingCompany.C...    1249 row(s) returned    0.206 sec / 0.00019...
```



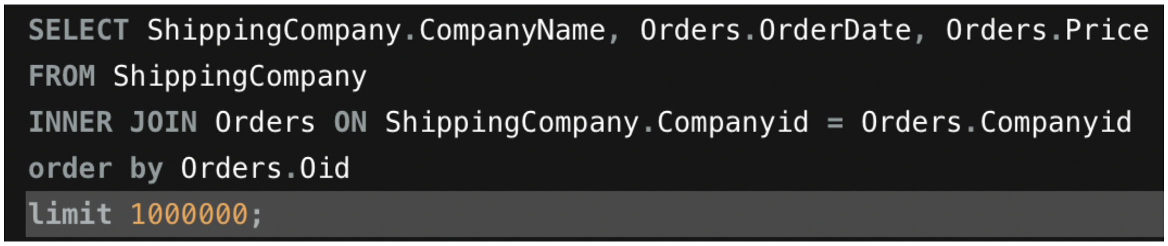
0.206 sec

Step 3: Running sql query with using indexing-> IDIL GULER

This time, we will create indexes on the CompanyName attribute in the ShippingCompany table and on the OrderDate attribute in the Orders table, and then rerun the exact same query to observe the results.

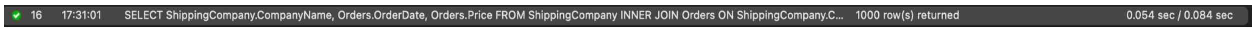


```sql
CREATE INDEX idx_company_name ON ShippingCompany(CompanyName);
CREATE INDEX idx_order_date ON Orders(OrderDate);
```

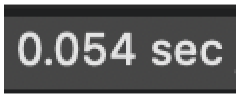


```sql
SELECT ShippingCompany.CompanyName, Orders.OrderDate, Orders.Price
FROM ShippingCompany
INNER JOIN Orders ON ShippingCompany.Companyid = Orders.Companyid
order by Orders.Oid
limit 1000000;
```

Upon running the identical query again, we notice a substantial reduction in the query's execution time.



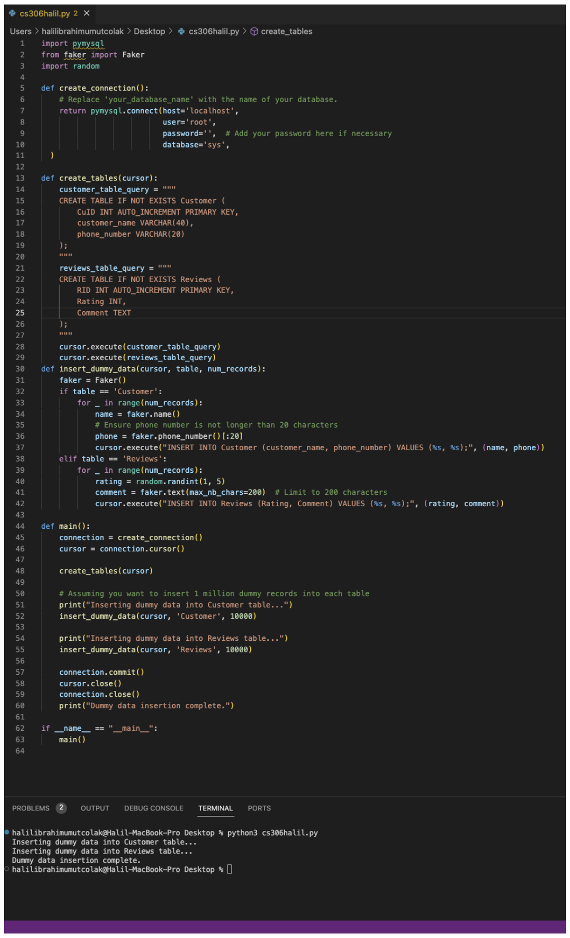
```
16    17:31:01    SELECT ShippingCompany.CompanyName, Orders.OrderDate, Orders.Price FROM ShippingCompany INNER JOIN Orders ON ShippingCompany.C...    1000 row(s) returned    0.054 sec / 0.084 sec
```



```
0.054 sec
```

Step 1: Connecting to Mysql Server and adding dummy data -> HALIL IBRAHIM UMUT COLAK

This Python script connects to a MySQL database using the pymysql library and creates two tables, Customer and Reviews. The Faker library is used to generate fake data, which is then inserted into these tables to populate them with dummy records. The insert_dummy_data function inserts the specified number of dummy records into either the Customer or Reviews table. Different faker methods are employed to ensure variety in the generated data, particularly for the comments in the Reviews table. The script commits the changes to the database and closes the connection once the data insertion is complete.

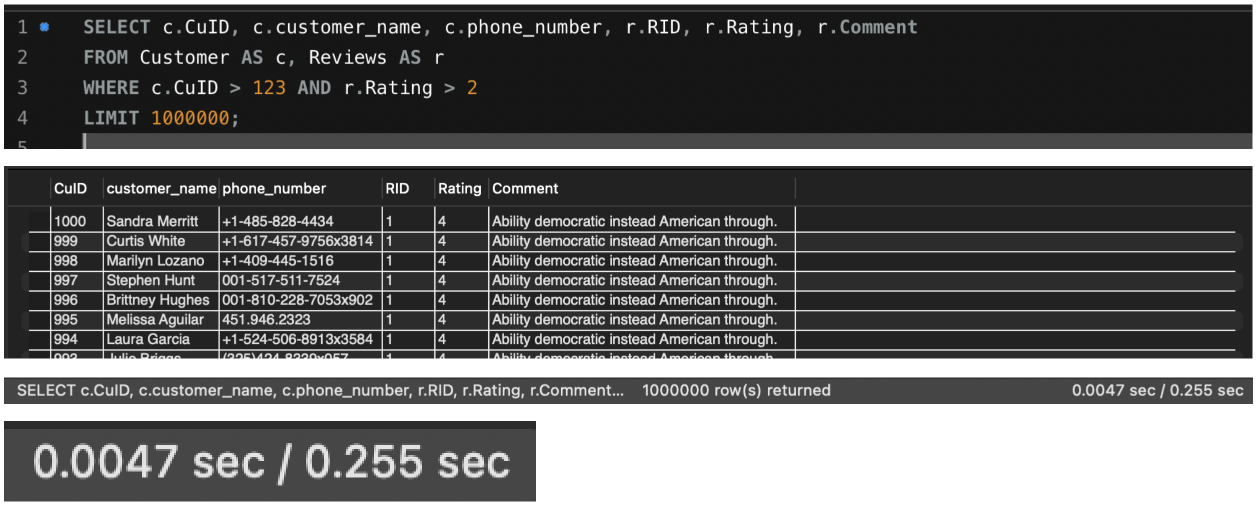

cs306halil.py 2
Users > halilibrahimumutcolak > Desktop > cs306halil.py > create_tables

```python
import pymysql
from faker import Faker
import random

def create_connection():
    # Replace 'your_database_name' with the name of your database.
    return pymysql.connect(host='localhost',
                           user='root',
                           password='',  # Add your password here if necessary
                           database='sys',
    )

def create_tables(cursor):
    customer_table_query = """
    CREATE TABLE IF NOT EXISTS Customer (
        CuID INT AUTO_INCREMENT PRIMARY KEY,
        customer_name VARCHAR(40),
        phone_number VARCHAR(20)
    );
    """
    reviews_table_query = """
    CREATE TABLE IF NOT EXISTS Reviews (
        RID INT AUTO_INCREMENT PRIMARY KEY,
        Rating INT,
        Comment TEXT
    );
    """
    cursor.execute(customer_table_query)
    cursor.execute(reviews_table_query)
def insert_dummy_data(cursor, table, num_records):
    faker = Faker()
    if table == 'Customer':
        for _ in range(num_records):
            name = faker.name()
            # Ensure phone number is not longer than 20 characters
            phone = faker.phone_number()[:20]
            cursor.execute("INSERT INTO Customer (customer_name, phone_number) VALUES (%s, %s);", (name, phone))
    elif table == 'Reviews':
        for _ in range(num_records):
            rating = random.randint(1, 5)
            comment = faker.text(max_nb_chars=200)  # Limit to 200 characters
            cursor.execute("INSERT INTO Reviews (Rating, Comment) VALUES (%s, %s);", (rating, comment))

def main():
    connection = create_connection()
    cursor = connection.cursor()

    create_tables(cursor)

    # Assuming you want to insert 1 million dummy records into each table
    print("Inserting dummy data into Customer table...")
    insert_dummy_data(cursor, 'Customer', 10000)

    print("Inserting dummy data into Reviews table...")
    insert_dummy_data(cursor, 'Reviews', 10000)

    connection.commit()
    cursor.close()
    connection.close()
    print("Dummy data insertion complete.")

if __name__ == "__main__":
    main()

```

```
halilibrahimumutcolak@Halil-MacBook-Pro Desktop % python3 cs306halil.py
Inserting dummy data into Customer table...
Inserting dummy data into Reviews table...
Dummy data insertion complete.
halilibrahimumutcolak@Halil-MacBook-Pro Desktop % 
```

Step 2: Running sql query without using indexing-> HALIL IBRAHIM UMUT COLAK

The SQL query displayed joins two tables, Customer and Reviews, without specifying a join condition, which implies a cross join resulting in a Cartesian product of the two tables. The query filters the combined dataset to only include customers with an ID greater than 123 and reviews with a Rating greater than a specified value. It limits the result set to the first 1,000,000 rows that match these criteria, which could potentially lead to performance issues due to the large volume of data being processed.



```sql
SELECT c.CuID, c.customer_name, c.phone_number, r.RID, r.Rating, r.Comment
FROM Customer AS c, Reviews AS r
WHERE c.CuID > 123 AND r.Rating > 2
LIMIT 1000000;
```

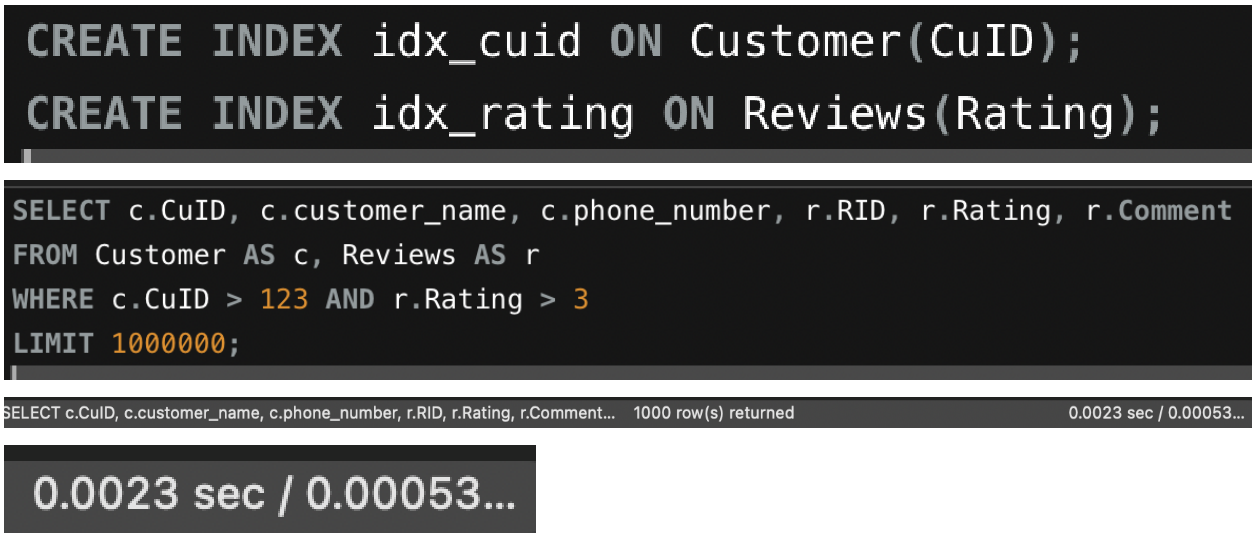
CuID	customer_name	phone_number	RID	Rating	Comment
1000	Sandra Merritt	+1-485-828-4434	1	4	Ability democratic instead American through.
999	Curtis White	+1-617-457-9756x3814	1	4	Ability democratic instead American through.
998	Marilyn Lozano	+1-409-445-1516	1	4	Ability democratic instead American through.
997	Stephen Hunt	001-517-511-7524	1	4	Ability democratic instead American through.
996	Brittney Hughes	001-810-228-7053x902	1	4	Ability democratic instead American through.
995	Melissa Aguilar	451.946.2323	1	4	Ability democratic instead American through.
994	Laura Garcia	+1-524-506-8913x3584	1	4	Ability democratic instead American through.
993	Julia Briggs	(325)424-8339x057	1	4	Ability democratic instead American through.

```
SELECT c.CuID, c.customer_name, c.phone_number, r.RID, r.Rating, r.Comment...    1000000 row(s) returned    0.0047 sec / 0.255 sec
```

0.0047 sec / 0.255 sec

Step 3: Running sql query with using indexing->HALIL IBRAHIM UMUT COLAK

The image showcases SQL commands for creating indexes on the CuID column of the Customer table and the Rating column of the Reviews table. Following the index creation, a SQL query is run which selects data from both tables where the Customer ID is greater than 123 and the Rating is above 3, with a limit set to return up to 1,000,000 rows. The timing results shown indicate that the query execution time has been significantly reduced due to the indexing, demonstrating how indexes can optimize query performance by allowing the database to more efficiently locate and retrieve the relevant rows.



```sql
CREATE INDEX idx_cuid ON Customer(CuID);
CREATE INDEX idx_rating ON Reviews(Rating);
```

```sql
SELECT c.CuID, c.customer_name, c.phone_number, r.RID, r.Rating, r.Comment
FROM Customer AS c, Reviews AS r
WHERE c.CuID > 123 AND r.Rating > 3
LIMIT 1000000;
```

```
SELECT c.CuID, c.customer_name, c.phone_number, r.RID, r.Rating, r.Comment...    1000 row(s) returned    0.0023 sec / 0.00053...
```

0.0023 sec / 0.00053...

Step 4: Outcome

The outcome of our comparative analysis between indexed and non-indexed query times has been enlightening. Indexing dramatically improved our database query efficiency, slashing retrieval times and showcasing the compelling advantage of this optimization technique. This efficiency gain is especially critical in data-intensive environments, validating indexing as an indispensable tool in modern database management.

Our analytical endeavor has provided a robust testament to the potency of indexing. The stark contrast in query execution times pre and post-indexing affirms that a well-indexed database is not just a luxury but a necessity, particularly when dealing with large-scale data sets. The empirical data harvested from our benchmarking process offers an unambiguous narrative: strategic indexing is a transformative approach that can significantly elevate the performance of database systems, thereby streamlining the workflow and enhancing the overall user experience.