# Data Engineering - CSL4030

Lab Assignment - 02

---

Irwindeep Singh
B22AI022

Sarthak Malviya
B22AI034

## 1 Hotel Management System

We have designed a hotel management system incorporating `MySQL` for database management. Further, we use `Python-Flask` app for backend.
The codebase used for the assignments along-with a `README.md` file can be found at: GitHub

## 2 SQL Queries using C and Python

### 2.1 Connecting to MySQL Connector

To access `SQL` database using `python`, a connection to the `SQL` server needs to be made with a client API. We have used the `MySQLdb`[1] python library and `<mysql/mysql.h>` header for C to establish the required connection and access the database corresponding to the hotel management database (Assignment-1).
Installation of the `MySQLdb` library can be done using the `python environment` installed in the system, for our case; we have used `python pip`[2] for the installation:

```
pip install mysqlclient
```

After successful installation, the following python script can be used to connect to the database `hotel_management` (the following programs assumes that the password for the `MySQl` server is stored in the path):

```python
import MySQLdb
import os

connection = MySQLdb.connect(
    host="localhost", user="root",
    password=os.getenv('MySQL'), database="hotel_management"
)
print("Connected to Database Successfully!")
```

Similarly, the installation and database connection for C header can be done as follows:

```
sudo pacman -S mysql-clients # Arch Linux
sudo apt-get install libmysqlclient-dev # Ubuntu/Kali Linux
```

Once installed, we can proceed with connecting to the database as follows:

```c
#include <stdio.h>
#include <stdlib.h>
#include <mysql/mysql.h>

MYSQL* connect_to_database(char *host, char *user, char *password, char *dbname){
    MYSQL* conn = mysql_init(NULL);
    mysql_real_connect(conn, host, user, password, dbname, 0, NULL, 0);
    return conn;
}

MYSQL_RES* execute_query(MYSQL* connection, char* query){
    mysql_query(connection, query);
    MYSQL_RES* res = mysql_store_result(connection);
    return res;
}

int main(){
    char* password = getenv("MySQL");
    MYSQL* connection = connect_to_database("localhost", "root", password, "hotel_management");
    return 0;
}
```

Furthermore, compiling the C program requires additional flags as follows:

```
gcc -o sql_connector sql_connector.c `mysql_config --cflags --libs`
./sql_connector
```

Now, the comming subsections contain the MySQL queries to be executed in both C and python, and the time taken for execution is to be compared. So, each subsection will only have MySQL query, without its implementation in C or python (in order to save space, the subsequent codes can be found in the github), relational algebra expression and the time comparasion.

## 2.2 Extracting Bookings in all months of 2023

To extract a list of all <room_number, type> who had bookings in all months in 2023, we have used rooms ⋈ bookings on room numbers for both the tables being same (to access the same room-data corresponding to the booking), where the YEAR(check_in) being 2023 as follows:

```sql
SELECT r.room_no, r.type
FROM rooms r
JOIN bookings b ON r.room_no = b.room_no
WHERE YEAR(b.check_in) = 2023
GROUP BY r.room_no, r.type
HAVING COUNT(DISTINCT MONTH(b.check_in)) = 12;
```

**Relational Algebra Expression:**

$$\pi_{\text{r.room\_no, r.type}} \left(\sigma_{\text{COUNT(DISTINCT MONTH(b.check\_in))}=12 \wedge YEAR(\text{b.check\_in})=2023} \right.$$
$$\left. (\rho_b(\text{rooms}) \bowtie_{\text{r.room\_no = b.room\_no}} \rho_r(\text{bookings}))\right)$$

**Time for Execution:**

- **C:** `0.000002 s`
- **Python:** `0.0016317367553710938 s`

## 2.3 Months with at-least one Deluxe Suite Booking

To print the names of all months in 2023 that had at least one 'deluxe suite' booking, we use rooms ⋈ bookings on room numbers being the same in both tables (to link bookings with room details), where the room type is 'deluxe suite' and YEAR(check_in) is 2023, as follows:

```sql
SELECT DISTINCT
    MONTHNAME(b.check_in) AS month_name
FROM rooms r
JOIN bookings b ON r.room_no = b.room_no
WHERE r.type = 'deluxe' AND YEAR(b.check_in) = 2023
ORDER BY MONTH(b.check_in)
```

**Relational Algebra Expression:**

$$\pi_{\text{MONTHNAME(b.check\_in)}} \left(\sigma_{\text{r.type = 'deluxe'} \wedge YEAR(\text{b.check\_in})=2023} \right.$$
$$\left. (\rho_b(\text{rooms}) \bowtie_{\text{r.room\_no = b.room\_no}} \rho_r(\text{bookings}))\right)$$

**Time for Execution:**

- **C:** `0.000002 s`
- **Python:** `0.0019106864929199219 s`

## 2.4 Guests with 'Single Room' in 2022

To extract guest id's who have booked 'single_room' in 2022, we join the guests table with bookings table on `guest_id` to match guest information with their bookings. Then, we have joined rooms table on `room_no` to get room details for each booking.

```sql
SELECT g.guest_id
FROM guests g
JOIN bookings b ON g.guest_id = b.guest_id
JOIN rooms r ON b.room_no = r.room_no
WHERE r.type = 'single_room' AND YEAR(b.check_in) = 2022;
```

**Relational Algebra Expression:**

$$\pi_{\text{g.guest\_id}} \big(\sigma_{\text{r.type = 'single\_room'} \wedge \text{YEAR(b.check\_in)} = 2022} (\rho_g(\text{guests}) \bowtie_{\text{g.guest\_id = b.guest\_id}} (\rho_b(\text{bookings}) \bowtie_{b.room\_no=r.room\_no} \rho_r(\text{rooms})))$$

**Time for Execution:**

- **C:** `0.000002 s`
- **Python:** `0.0015079975128173828 s`

## 2.5 Guests Information with 'Single Room' in 2022

To extract guest information who have booked 'single_room' in 2022, we join the guests table with the bookings table on `guest_id` to match guest information with their bookings. Then, we join the rooms table on `room_no` to get room details for each booking.

```sql
SELECT g.lname, g.email, g.phone_no
FROM guests g
JOIN bookings b ON g.guest_id = b.guest_id
JOIN rooms r ON b.room_no = r.room_no
WHERE r.type = 'single_room' AND YEAR(b.check_in) = 2022;
```

**Relational Algebra Expression:**

$$\pi_{\text{g.lname, g.email, g.phone\_number, g.address}} \big(\sigma_{\text{r.type = 'single\_room'} \wedge \text{YEAR(b.check\_in)} = 2022} (\rho_g(\text{guests}) \bowtie_{\text{g.guest\_id = b.guest\_id}} (\rho_b(\text{bookings}) \bowtie_{\text{b.room\_no = r.room\_no}} \rho_r(\text{rooms})))$$

**Time for Execution:**

- **C:** `0.000002 s`
- **Python:** `0.0014922618865966797 s`

### 2.6 Guests with No Purchases

To derive a list of all guest profiles who have not made any purchases, we perform a *left join* between the guests table and the bookings table on `guest_id`. We then select guests who do not have any corresponding entries in the bookings table.

```sql
SELECT g.guest_id, g.name, g.email, g.phone_no
FROM guests g
LEFT JOIN bookings b ON g.guest_id = b.guest_id
WHERE b.guest_id IS NULL;
```

**Relational Algebra Expression:**

$$\pi_{\text{g.guest\_id, g.lname, g.email, g.phone\_no}} \left(\sigma_{\text{b.guest\_id IS NULL}} \left(\rho_g(\text{guests}) \bowtie_{\text{g.guest\_id = b.guest\_id}} (\text{bookings})\right)\right)$$

**Time for Execution:**

- **C:** `0.000002 s`
- **Python:** `0.0024569034576416016 s`

# 3 Optimized Relational Algebra Queries

- **Extracting Bookings in all months of 2023:**

$$\pi_{\text{r.room\_no, r.type}} \left(\sigma_{\text{YEAR(b.check\_in) = 2023}} \left(\rho_r(\text{rooms}) \bowtie_{\text{r.room\_no = b.room\_no}} \rho_b(\text{bookings})\right) \div \pi_{\text{m.month}} \left(\sigma_{\text{YEAR(b.check\_in) = 2023}} \left(\rho_b(\text{bookings})\right)\right)\right)$$

- **Months with at-least one Deluxe Suite Booking:**

$$\pi_{\text{m.month}} \left(\sigma_{\text{r.type = 'deluxe'}} \left(\rho_r(\text{rooms}) \bowtie_{\text{r.room\_no = b.room\_no}} \rho_b(\text{bookings})\right)\right)$$

- **Guests with 'Single Room' in 2022:**

$$\pi_{g.guest\_id} \left(\sigma_{r.type='single\_room' \wedge \text{YEAR}(b.check\_in)=2022} \left(\rho_g(\text{guests}) \bowtie_{\text{g.guest\_id = b.guest\_id}} \left(\rho_b(\text{bookings}) \bowtie_{\text{b.room\_no = r.room\_no}} \rho_r(\text{rooms})\right)\right)\right)$$

- **Guests Information with 'Single Room' in 2022:**

$$\pi_{g.guest\_lname, g.email, g.phone\_number, g.address} \left(\sigma_{r.type='single\_room' \wedge \text{YEAR}(b.check\_in)=2022} \left(\rho_g(\text{guests}) \bowtie_{\text{g.guest\_id = b.guest\_id}} \left(\rho_b(\text{bookings}) \bowtie_{\text{b.room\_no = r.room\_no}} \rho_r(\text{rooms})\right)\right)\right)$$

- **Guests with No Purchases:**

$$\pi_{\text{g.guest\_profile}} \left(\rho_g(\text{guests}) - \pi_{\text{g.guest\_id}} \left(\rho_g(\text{guests}) \bowtie_{\text{g.guest\_id = b.guest\_id}} \text{bookings}\right)\right)$$

# 4 Optimized SQL Queries

- **Extracting Bookings in all months of 2023:**

```sql
SELECT r.room_no, r.type
FROM rooms r
JOIN bookings b ON r.room_no = b.room_no
WHERE YEAR(b.check_in) = 2023
GROUP BY r.room_no, r.type
HAVING COUNT(DISTINCT MONTH(b.check_in)) = 12;
```

**Time for Execution:**

  - **C:** 0.000002 s
  - **Python:** 0.0015377998352050781 s

- **Months with at-least one Deluxe Suite Booking:**

```sql
SELECT DISTINCT MONTH(b.check_in) AS month
FROM bookings b
JOIN rooms r ON b.room_no = r.room_no
WHERE r.type = 'deluxe';
```

**Time for Execution:**

  - **C:** 0.000002 s
  - **Python:** 0.0020248889923095703 s

- **Guests with 'Single Room' in 2022:**

```sql
SELECT g.guest_id
FROM guests g
JOIN bookings b ON g.guest_id = b.guest_id
JOIN rooms r ON b.room_no = r.room_no
WHERE r.type = 'single_room' AND YEAR(b.check_in) = 2022;
```

**Time for Execution:**

  - **C:** 0.000002 s
  - **Python:** 0.0010666847229003906 s

- **Guests Information with 'Single Room' in 2022:**

```sql
SELECT g.lname, g.email, g.phone_no
FROM guests g
JOIN bookings b ON g.guest_id = b.guest_id
JOIN rooms r ON b.room_no = r.room_no
WHERE r.type = 'single_room' AND YEAR(b.check_in) = 2022;
```

**Time for Execution:**

  - **C:** `0.000002 s`
  - **Python:** `0.0017931461334228516 s`

- **Guests with No Purchases:**

```sql
SELECT g.guest_id, g.lname, g.email, g.phone_no
FROM guests g
LEFT JOIN bookings b ON g.guest_id = b.guest_id
WHERE b.guest_id IS NULL;
```

**Time for Execution:**

  - **C:** `0.000002 s`
  - **Python:** `0.001325368881225586 s`

# 5 Execution Times Comparasion

The asked comparasion is already done in the above sections.

# 6 Search Algorithm made by the SQL package

The SQL package utilized in database management systems like MySQL, incorporates indexing tools so as to speed up the search.

1. **B-Tree Indexing:** This is the default indexing method for most MySQL storage engines. This is the default indexing method for most MySQL storage engines. A B-tree, or Balanced tree, index is organized and joined at a parent node where each level below each parental node has its own sequence. This system method reduces the excessive read times to the hard disk that will reduce the number of read operations, insertion and deletion operations to be performed. When a query is processed, the structure of a B-tree permits from directly locating the relevant rows raised above the other by the kernel tree structure, thus eliminating hanging in the middle of relevant records on some other structures, especially range query equal searches.

2. **Hash Indexing:** This is carried out mainly with the MEMORY storage engine. A hash index is obtained by identifying primary objectives and repairing them with the appropriate tools. While this type of indexing is mostly effective for equality searches because data can be directly reached without any constraints, this method does not novelly sustain range queries as there are derived order relations of B-trees and is non-existent in index hash.

These indexing methods ensure that sql issues are performed in the least possible time span, particularly in reference to data access.

# 7 Algorithm potentially affects the Execution time

The search algorithms utilized in MySQL, such as B-Tree and Hash indexing, play a crucial role in determining the execution time of queries. Here's how they impact performance:

- **B-Tree Indexing:**
  - **Efficiency:** B-Tree indexes allow MySQL to quickly locate data by traversing a balanced hierarchical structure. Since the data is organized at different tree levels, MySQL can perform searches efficiently, reducing the time required to locate rows. B-Tree indexing performs especially well for range queries (e.g., BETWEEN, $>$, $<$), as the structure naturally supports ordered traversal.
  - **Execution Time Impact:** This indexing method significantly improves execution time for large datasets, reducing search complexity to $\mathcal{O}(\log n)$, where n is the number of rows. Without B-Tree indexing, MySQL would resort to a full table scan ($\mathcal{O}(n)$), which would be far slower for large tables.
- **Hash Indexing:**
  - **Efficiency:** Hash indexing is highly efficient for equality-based searches, as it computes a hash value and provides direct access to the relevant data. For example, queries with $=$ conditions benefit greatly from this method, as the lookup time is close to constant, $\mathcal{O}(1)$.
  - **Execution Time Impact:** For equality searches, hash indexes dramatically reduce query execution time, making data retrieval almost instantaneous. However, hash indexing is less efficient for range queries, as it doesn't store data in an ordered fashion. The system might return to less efficient methods like full table scans for such queries, leading to longer execution times.

By using B-Tree or Hash indexing, MySQL avoids the need for slower operations like table scans, drastically reducing query execution time. B-Tree indexing is optimal for both range and equality queries, while hash indexing excels in exact matches, ensuring efficient data retrieval for specific use cases.

# 8 Compilation / Interpretation strategy of C and Python

- **C Language:**
  - **Strategy:** C is a compiled language. Its code is first translated into machine code by a compiler before execution. This process involves several stages, including preprocessing, compilation, assembly, and linking.
  - **Preprocessing:** The preprocessor handles directives like `#include` and `#define`, modifying the source code accordingly.
  - **Compilation:** The compiler translates the preprocessed code into assembly language.
  - **Assembly:** The assembler converts the assembly code into machine code, producing an object file.
  - **Linking:** The linker combines object files and libraries into a single executable file.
  - **Execution:** The operating system runs the executable file directly.
- **Python Language:**
  - **Strategy:** Python is an interpreted language. Its code is executed line-by-line by an interpreter rather than being compiled into machine code beforehand.
  - **Bytecode Compilation:** When Python code is run, it is first compiled into intermediate bytecode (.pyc files) by the Python interpreter. This bytecode is a lower-level, platform-independent representation of the source code.
  - **Interpretation:** The Python Virtual Machine (PVM) interprets the bytecode, executing it on the fly. This process allows for dynamic execution and flexibility but can be slower than compiled languages.
  - **Execution:** The interpreter executes the bytecode, converting it into machine code at runtime.

In summary, C adopts a compilation strategy to translate source code to executable binary but is rather efficient in that it does not require separate compilation of the code. Python uses an interpretive approach wherein code typed in the form on source is converted into an intermediate form called bytecode for an interpreter to run. This promotes dynamic execution and development of the code.

# References


[1] The mysqlclient Contributors. *mysqlclient*. https://pypi.org/project/mysqlclient/. Accessed: 2024-09-17.

[2] Python Software Foundation. *pip: The Python Package Installer*. https://pypi.org/project/pip/. Accessed: 2024-09-17. 2024.