Lab Assignment 4 & 5

Team Number: **50**
Team Members: Soumik Roy (B20AI042)
Kulkarni Tanmay Shreevallabh (B20CS029)
Yash Bhargava (B20AI050)

Executing the Code files

First, we need to create the tables in our database using the file *create_table.cpp* which takes the data entries from the file *library_catalogue.txt.* Make sure to create a database named `dbms_lab4_5` and an user with name: `group50_dbms` password: `password`

```
Command to Compile: g++ create_table.cpp -o create_table.exe -lmysql
```

Now to work on this data, we need to extract it from the database to a .txt file (*output.txt*). For this, we have created the file *extract_data.cpp*

```
Command to Compile: g++ extract_data.cpp -o extract_data.exe -lmysql
```

Now, the main part of this lab is executed in the files with names starting with the relevant question number.

```
Command to Compile: g++ <filename>.cpp -o <filename>.exe
```

Question 1

a. The primary key chosen is ***book_id***, as it is unique and non-null for every entry in the current data. All the rows have a unique book id and hence it can be used to identify every entry uniquely.

b. We have used **Polynomial Rolling/Hashing** on the string book_id's to hash them into integers. This function works as follows :
Suppose we have a string S of length n.
Hashed value of **S =** $\sum_{i=0}^{n-1} ASCII\{S[i]\}^{i}$

```
Hashed book Ids :
Aest_AC_0103 -> 553780
Anim_CS_0319 -> 573282
Aunt_PW_1623 -> 515377
Deat_JR_1018 -> 414622
Emer_VR_2218 -> 433213
Emot_MM_1313 -> 384004
Fant_JR_1018 -> 417309
Gobl_JR_1018 -> 413652
Mind_LC_1203 -> 460581
Phan_VR_2218 -> 381875
Phil_JR_1018 -> 415066
Pris_JR_1018 -> 376230
Self_AD_0104 -> 471321
Soci_MM_1313 -> 378227
Wode_PW_1623 -> 482691
```

c. The codes that have been provided as book_id and author_id are effective for the amount of data provided to us, but in

general **given a larger amount of data, they would not be sufficiently effective**. This is so because:

author_id is : {1st 2 letters of first name}_{1st 2 letters of last name}_someNumber
book_id is : {1st 4 letters of book name}_{1st letters of author firstname & lastname}_{ number from author_id}

Here author IDs are sufficiently effective as even if 2 authors have 1st 2 letters of first and last name same, then the random number at the end can be different hence making the author_id unique for each author.

However in book_id, if 2 books written by same author have the same first 4 letters in their book name, then the book_id for these 2 books would become same, and even if we had the pair of book_id and author_id as the primary key, still the 2 books would have the same primary key (same book_id and same author) hence rendering the chosen primary keys useless.

For example,
Book1 : “Hello World” by John Doe -> author_id = jo_do_001 & book_id = hell_JD_001
Book2 : “Hello Universe” by John Doe -> author_id = jo_do_001 & book_id = hell_JD_001

=> As we can see both can not have any unique primary key combination.

Question 2

a. After extendible hashing with bucket size 4, resulting hash table obtained :

```
Extendible Hashing Table :-
00 : 553780 384004 413652
10 : 573282 414622 415066 376230
001 : 515377 471321
101 : 433213 417309 460581
11 : 381875 378227 482691
```

b. **Decreasing the bucket size** would **increase the number of overflows** occurring and hence we would require overflow-handling (bucket splitting) too often. However **slightly increasing the bucket size** would **improve the efficiency** as here comparatively **less amount of overflows** would occur and having a decently small bucket size would also ensure quick enough in-bucket search after the required bucket index has been found. Although it is important to keep in mind that **bucket size shouldn't be too large** as this would make **in-bucket searches slower**.

Results obtained for different bucket sizes :

```
Bucket size 3 :-
Extendible Hashing Table :-
00 : 553780 384004 413652
010 : 573282 415066
110 : 414622 376230
001 : 515377 471321
101 : 433213 417309 460581
11 : 381875 378227 482691

Bucket size 4 :-
Extendible Hashing Table :-
00 : 553780 384004 413652
10 : 573282 414622 415066 376230
001 : 515377 471321
101 : 433213 417309 460581
11 : 381875 378227 482691

Bucket size 5 :-
Extendible Hashing Table :-
00 : 553780 384004 413652
10 : 573282 414622 415066 376230
01 : 515377 433213 417309 460581 471321
11 : 381875 378227 482691
```

As we can see less overflows/splits occurred for bucket size = 5, as compared to bucket sizes 3 and 4.

c. Used Set data structure instead of a linear list (vector) for the buckets.

Question 3

a. We used a global bucket order (n) = 5.

b. The results obtained after linear hashing :

```
Linear Hashing Table :
0 : 553780 376230
1 : 460581 415066 471321 482691
2 : 573282 515377 414622 413652 378227
3 : 433213
4 : 384004 417309
5 : 381875
6 :
7 :
8 :
9 :
```

c. The effect of changing global bucket order would depend on the distribution of mod values of the hashed book_id values i.e what are the mod-n (%n) values of the book_id values and how uniformly distributed they are. On experimenting with n = 4, 5, and 6 we can make some useful observations.

Results obtained for different Global bucket orders:

```
Bucket size = 4

Global Order 4 :-
Linear Hashing Table :
0 :
1 : 515377 433213 417309 460581 471321
2 : 573282 414622 415066 376230
3 : 381875 378227 482691
4 : 553780 384004 413652
5 :
6 :
7 :

Global Order 5 :-
Linear Hashing Table :
0 : 553780 376230
1 : 460581 415066 471321 482691
2 : 573282 515377 414622 413652 378227
3 : 433213
4 : 384004 417309
```

```
5 : 381875
6 :
7 :
8 :
9 :

Global Order 6 :-
Linear Hashing Table :
0 : 573282 413652 376230
1 : 515377 433213
2 :
3 : 417309 460581 471321 482691
4 : 553780 414622 384004 415066
5 : 381875 378227
```

As we can see, n = 6 gives the best results, i.e for the given data, the mod-n values of the book_id's are uniformly distributed in a way that no overflow occurs and hence no splitting is required. On the other hand, taking n = 4 and 5 both cause 1 overflow each, and hence splitting is needed. Even after splitting, we can see that the overflow was not resolved.

d. **Decreasing the bucket size** would **increase the number of overflows** occurring and hence we would require overflow-handling (bucket splitting) too often. However, **slightly increasing the bucket size** would **improve the efficiency** as here comparatively **less amount of overflows** would occur. Having a decently small bucket size would also ensure a quick enough in-bucket search after the required bucket index has been found. Although it is important to keep in mind that **bucket size shouldn't be too large** as this would make **in-bucket searches slower**.

Results obtained for different bucket sizes:

```
Global Order = 5

Bucket size 3 :-
Linear Hashing Table :
0 : 553780 376230
1 : 460581 471321 482691
2 : 573282 515377 414622 413652 378227
3 : 433213
4 : 384004 417309
5 : 381875
```

```
6 : 415066
7 :
8 :
9 :

Bucket size 4 :-
Linear Hashing Table :
0 : 553780 376230
1 : 460581 415066 471321 482691
2 : 573282 515377 414622 413652 378227
3 : 433213
4 : 384004 417309
5 : 381875
6 :
7 :
8 :
9 :

Bucket size 5 :-
Linear Hashing Table :
0 : 553780 381875 376230
1 : 460581 415066 471321 482691
2 : 573282 515377 414622 413652 378227
3 : 433213
4 : 384004 417309
```

As we can see no overflow occurred for bucket size = 5, however, overflows occurred for bucket sizes 3 and 4, resulting in splitting to be done.

e. Used Set data structure instead of a linear list (vector) for the buckets.

Question 4

a. We used a number of higher positioned bits = 4. It can be seen that the hashed book_id values are very large 6-digit numbers, hence their binary forms will also be very large hence lower order buckets would mostly be empty. Moreover, we chose a bucket size of 4, and we observed that taking the highest 4 bits would distribute the keys in a uniform way such that no overflows occur and only 2(out of 16) of the buckets are full.

b. The results obtained after Distributed Hashing with n_bits = 4 & bucket_size = 4 :

```
Distributed Hashing Table
0000 :
0001 :
0010 :
0011 :
0100 :
0101 :
0110 :
0111 :
1000 : 553780 573282
1001 :
1010 :
1011 : 384004 381875 376230 378227
1100 : 414622 417309 413652 415066
1101 : 433213
1110 : 460581 471321 482691
1111 : 515377
```

c. On decreasing the value of n_bits order the values would collide more in the buckets leading to higher chances of overflow. On the other hand, we can see that only around the lower half of the buckets hold values (as the hashed numbers are very large - 6 digit). Hence increasing the n_bits value would lead to more and more buckets being left empty. We observed a good distribution at n_bits = 4 and hence conclude decreasing or increasing the value would lead to inefficiency.

Results obtained for different n_bit's:

```
Bucket size 4

n_bits = 3 :-
Distributed Hashing Table
000 :
001 :
010 :
011 :
100 : 553780 573282
101 : 384004 381875 376230 378227
110 : 414622 433213 417309 413652 415066
111 : 515377 460581 471321 482691

n_bits = 4 :-
```

```
Distributed Hashing Table
0000 :
0001 :
0010 :
0011 :
0100 :
0101 :
0110 :
0111 :
1000 : 553780 573282
1001 :
1010 :
1011 : 384004 381875 376230 378227
1100 : 414622 417309 413652 415066
1101 : 433213
1110 : 460581 471321 482691
1111 : 515377

n_bits = 5 :-
Distributed Hashing Table
00000 :
00001 :
00010 :
00011 :
00100 :
00101 :
00110 :
00111 :
01000 :
01001 :
01010 :
01011 :
01100 :
01101 :
01110 :
01111 :
10000 : 553780
10001 : 573282
10010 :
10011 :
```

```
10100 :
10101 :
10110 : 376230
10111 : 384004 381875 378227
11000 :
11001 : 414622 417309 413652 415066
11010 : 433213
11011 :
11100 : 460581 471321
11101 : 482691
11110 :
11111 : 515377
```

As we can see, n = 3 overflows the buckets and at n=6 there are too many buckets being left empty.

d. **Decreasing the bucket size** would **increase the number of overflows**. However, **slightly increasing the bucket size** would **improve the efficiency** as here comparatively **less amount of overflows** would occur and having a decently small bucket size would also ensure a quick enough in-bucket search after the required bucket index has been found. Although it is important to keep in mind that **bucket size shouldn't be too large** as this would make **in-bucket searches slower**.

Results obtained for different bucket sizes:

```
n_bits = 4

Bucket size 3
Distributed Hashing Table
0000 :
0001 :
0010 :
0011 :
0100 :
0101 :
0110 :
0111 :
1000 : 553780 573282
1001 :
1010 :
```

```
1011 : 384004 381875 376230 378227
1100 : 414622 417309 413652 415066
1101 : 433213
1110 : 460581 471321 482691
1111 : 515377

Bucket size 4
Distributed Hashing Table
0000 :
0001 :
0010 :
0011 :
0100 :
0101 :
0110 :
0111 :
1000 : 553780 573282
1001 :
1010 :
1011 : 384004 381875 376230 378227
1100 : 414622 417309 413652 415066
1101 : 433213
1110 : 460581 471321 482691
1111 : 515377

Bucket size 5
Distributed Hashing Table
0000 :
0001 :
0010 :
0011 :
0100 :
0101 :
0110 :
0111 :
1000 : 553780 573282
1001 :
1010 :
1011 : 384004 381875 376230 378227
1100 : 414622 417309 413652 415066
```

```
1101 : 433213
1110 : 460581 471321 482691
1111 : 515377
```

As we can see no overflow occurred for bucket size = 4 & 5, however overflows occurred for bucket sizes 3.

e. Used Set data structure instead of a linear list (vector) for the buckets.

Question 5

a. Across all the hashing mechanisms explored, the common observation is that **decreasing the bucket size** would **increase the number of overflows**. However, **slightly increasing the bucket size** would **improve the efficiency** as comparatively **less amount of overflows** would occur. Also, having a decently small bucket size would also ensure a quick enough in-bucket search after the required bucket index has been found. Although it is important to remember that **bucket size shouldn't be too large** as this would make **in-bucket searches slower**.

b. If we analyze the insertion time complexities in the 3 hashing techniques :
   i. **Extendible hashing** : we first find the number of least significant bits whose value is a key in the hash table {O($log_2(value)$)} and going to the index in the map data structure would take O($log_2(n_{buckets})$), then we insert the value in the corresponding bucket {O(1)}, then if overflow occurs, we split the bucket into 2 {O(bucket_size)}.

      Total time : O(($log_2(value)$ * $log_2(n_{buckets})$ + bucket_size)
   ii. **Linear hashing** : we first find the index value in the hash table where new_value needs to be inserted using new_value%mod_val {O(1)}, then we insert the value in the corresponding bucket {O(1)}, then if overflow occurs, we split the split index pointed bucket into 2 {O(bucket_size)}.

      Total time : O(bucket_size)
   iii. **Distributed Hashing** : we first find the index value in the hash table where new_value needs to be inserted by finding the highest n bits of the number {O($log_2(value)$)} and going to the index in the map data structure would take O(

$log_2(n_{buckets})$) time where $n_{buckets} = 2^n$, then we insert the value in the corresponding bucket {O(1)}.

Total time : O(($log_2(value)$ * n) (n here would be a very small number like 4 or 5 or in extreme cases something like 30(meaning 2^{30} sized hash table))

Hence order of time taken in insertion for the different hashing techniques: Extendible Hashing > Linear Hashing > Distributed Hashing

***Note : Inserting just the new book_id into the hash table takes a negligible amount of time and hence c++ is unable to find the exact amount of time taken to do so and hence returns 0 nanoseconds. Hence we inserted 15,000 new values into the 3 hash tables each and compared the times taken by the 3. Results obtained -*

```
new_book_id = "Find_KY_1111"
Hashed new_book_id = 555879

Time taken in inserting new_book_id's from 548396 to 563396 :

Extendible Hashing -> 97533 microseconds
Linear Hashing -> 16000 microseconds
Distributed Hashing -> 0 microseconds
```

As can be seen, our above proven order of time holds true in this case too.

c. If we analyze the searching time complexities in the 3 hashing techniques :
   i. **Extendible hashing** : we first find the number of least significant bits whose value is a key in the hash table {O($log_2(value)$)} and going to the index in the map data structure would take O($log_2(n_{buckets})$), then we search the value in the corresponding bucket {O(bucket_size)}.

      Total time : O(($log_2(value)$ * $log_2(n_{buckets})$ + bucket_size)
   ii. **Linear hashing** : we first find the index value in the hash table where required_value would be placed using value%mod_val {O(1)}, then we search the value in the corresponding bucket {O(bucket_size)}.

      Total time : O(bucket_size)

iii. **Distributed Hashing** : we first find the index value in the hash table where required_value would exist by finding the highest n bits of the number {O($log_2(value)$)} and going to the index in the map data structure would take O($log_2(n_{buckets})$) time where $n_{buckets} = 2^n$, then we search the value in the corresponding bucket {O(bucket_size)}. **this will be O(bucket_size) only if it is pre-ensured that the buckets aren't overflowing, as when overflow occurs then no handling would be done.

Total time :

Ideal : O($log_2(value) * n$ + bucket_size)

Worst : O($log_2(value) * n$ + number_of_total_elements)

Hence order of time taken in searching for the different hashing techniques:
Ideal : Extendible Hashing > Linear Hashing > Distributed Hashing
Worst : Distributed Hashing > Extendible Hashing > Linear Hashing

d. The amount of time to search several values in the hash table would take time in similar order as the time taken in searching a single value. And the data provided acts as an ideal scenario. Hence the order of time taken :

Extendible Hashing > Linear Hashing > Distributed Hashing

However if the data provided was sufficiently large, which would be the case in real live, then the order of time taken in searching would take :

Distributed Hashing > Extendible Hashing > Linear Hashing