

Assumptions:

- There can be multiple authors with the same name.
- There can be multiple books with the same name.

Question 1:

a) Primary Key: **Book_id**

Reasons:

Functional dependencies

$$\text{Author_id} \rightarrow \text{Author_name}$$
$$\text{Book_id} \rightarrow \text{Book_name}, \text{Author_name}, \text{Author_id}$$
$$\text{Book_id}, \text{Author_name} \rightarrow \text{Author_id}$$

Thus, $\text{Book_id}^* = \{ \text{Book_name}, \text{Author_name}, \text{Author_id}, \text{Book_id} \}$
(closure)

\therefore Candidate keys = $\{ \text{Book_id} \}$

\uparrow
Primary key, Since only 1 candidate key exists.

Thus, since **Book_id** can determine a specific individual row, I am choosing it to be the Primary Key of the table.

b) Design hash function

```
//The below function hashes the book_id
int hashFunction(string key) { // Adding ASCII values of characters
in the book_id and then adding the last 4 characters (book_number)
to it.
    int hashCode = 0;
    for (int i = 0; i < key.length()-4; i++) {
        hashCode += key[i];
    }
    return abs(hashCode+stoi(key.substr(key.size() - 4)));
}
```

The function hashFunction hashes the specific Book_ID and then returns the integer value of the hash.

(This type of Hash can cause collisions, but since the dataset was small and no collisions are occurring, thus I am using this hash function.)

c) Comment on the provided codes for book_id and author_id. Do you think these are sufficiently effective?

- From my understanding, the Book_ID is created using the First four letters of Book_Name + '_' + Initials of Author Name + Some number assigned to the Author.

This is not sufficiently effective as we can have a clash for book_id for two different books. Eg: The Aesthetic Brain By Angan Chakraborty (10001), Aesthetic Body by Angan Chatterjee(10002).

Thus, Book_ID for these would be the same, i.e **Aest_AC_1000**.

The last 4 characters of Book_ID are the same since we are only considering the first 4 characters(assumption). We should increase the size of the number of characters we are considering in Book_ID. Since books of more than 10000 authors can be present in the library.

Book_ID should be different for different books. Thus, we should use something else such as an ISBN number for book_id.

- There is a similar case with Author_ID. Which is made from combining first two characters of First Name and Last Name of the Author and the unique id of the Author. We should consider more than 4 digit numbers for unique author numbers to avoid clashes.

Question 2:

- a) I have implemented Extendible hashing in the Extendible_hashing.cpp file.
- b) According to the hash function that I have used, the most effective bucket size is 4.

The output/directory structure after inserting all entries in the extendible hashing format is :

Output for bucket size 4,5:

Local depth of the bucket is : 2 (00)

1048 1752 1928 2388

Local depth of the bucket is : 2 (01)

821 1757 1761 2969

Local depth of the bucket is : 2 (10)

822 1746 1778 2062

Local depth of the bucket is : 2 (11)

2055 2379 2967

Output for bucket size=3 :

Local depth of the bucket is : 3

1048 1752 1928

Local depth of the bucket is : 4

821 1757 1761

Local depth of the bucket is : 3

1746 1778

Local depth of the bucket is : 2

2055 2379 2967

Local depth of the bucket is : 3

2388

Local depth of the bucket is : 3

822 2062

Local depth of the bucket is : 4

2969

As seen, there are many buckets with different local depths for bucket size=3, thus, we are not utilizing buckets to their full capacity.

At bucket_size=4 all the buckets have same local depth, and the data is stored in equal proportion, thus this bucket_size is optimal. Also, for bucket_size=5, too similar output is coming, thus extra space is getting wasted in each bucket.

Thus, best bucket_size is 4.

- c) I have implemented extendible hashing using Binary Search Tree in Extendible_hashing_tree.cpp file, but there are a few arrows in the code.

By using Tree instead of a linear list inside a bucket, we can further decrease the search time of a hash value to $O(\log n)$ instead of $O(n)$ (here n denotes the bucket size).

Question 3:

- a) We will choose global bucket order as 3, as according to our data, it is **equally spaced with respect to HashValue mod 3**.

Our hash values are as follows wrt primary key i.e Book_id:

821,822,1068,1746,1757,1752,1761,1778,1928,2062,2055,2388,2378,2969,2967.

We can see that, out of the 15 values,

8 values are divisible by 3, 4 are of type $3n+1$, 3 are of type $3n+2$.

Thus, when a linear hashing algorithm runs, there would be efficient utilization of buckets.

- b) I have written the code for linear hashing using bucket size=4 and global bucket order =3 in the linear_hashing.cpp file.

Output:

Creating Table

822 The Aesthetic Brain

821 Self Comes to Mind

1048 What Animals Think

1746 Deathly Hallows_Harry Potter

1757 Fantastic Beasts and Where to Find Them

1752 Goblet of Fire_Harry Potter

1761 Philosophers Stone_Harry Potter

1778 Prisoner of Azkaban_Harry Potter

1928 The Mind of a Bee

2062 Emotion Machine

2055 Society of Mind

2388 Aunts Aren't Gentlemen

2379 Wodehouse at the Wicket

2969 The Emerging Mind

2967 Phantoms in the Brain

Created Table Successfully!

Enter Bucket Size:

4

Enter Global Bucket Order:

3

Linear hashing

0 ->

1 ->

2 -> 821

0 -> 822

1 ->

2 -> 821

0 -> 822

1 -> 1048

2 -> 821

0 -> 822 1746

1 -> 1048

2 -> 821

0 -> 822 1746 1752

1 -> 1048

2 -> 821

0 -> 822 1746 1752

1 -> 1048

2 -> 821 1757

0 -> 822 1746 1752 1761

1 -> 1048

2 -> 821 1757

0 -> 822 1746 1752 1761

1 -> 1048

2 -> 821 1757 1778

Splitting Bucket: 0

0 -> 822 1746 1752

1 -> 1048

2 -> 821 1757 1778 1928

3 -> 1761

4 ->

5 ->

0 -> 822 1746 1752

1 -> 1048

2 -> 821 1757 1778 1928

3 -> 1761 2055

4 ->

5 ->

0 -> 822 1746 1752
1 -> 1048
2 -> 821 1757 1778 1928
3 -> 1761 2055 2379 2967
4 -> 2062
5 ->

0 -> 822 1746 1752 2388
1 -> 1048
2 -> 821 1757 1778 1928
3 -> 1761 2055 2379 2967
4 -> 2062
5 -> 2969

- c) To confirm that our choice of global bucket order was correct, I have attached outputs when different global bucket orders were chosen. (Keeping bucket size=4, fixed)

For Global Bucket Order 2: (Final directory structure)

0 -> 1048 1752 1928
1 -> 1761 2969
2 -> 822 1746 1778 2062
3 -> 2055 2379
4 -> 2388
5 -> 821 1757
6 ->
7 -> 2967

For Global Bucket Order 3: (Final Directory Structure)

0 -> 822 1746 1752 2388
1 -> 1048
2 -> 821 1757 1778 1928
3 -> 1761 2055 2379 2967
4 -> 2062
5 -> 2969

For Global Bucket Order 4: (Final Directory Structure)

0 -> 1048 1752 1928

1 -> 821 1757 1761 2969

2 -> 822 1746 1778 2062

3 -> 2055 2379

4 -> 2388

5 ->

6 ->

7 -> 2967

We can clearly see that the better way of utilizing the buckets is using global bucket order=3.

d) Keeping the global bucket order =3, I have experimented with different bucket sizes.

Output for bucket_size=2 :

0 -> 822 1746 1752 2388

1 ->

2 -> 821 1778 1928

3 -> 1761 2055 2379 2967

4 -> 1048

5 -> 1757 2969

6 ->

7 ->

8 ->

9 ->

10 -> 2062

11 ->

(High amount of space wastage.)

There are more than 2 elements in some buckets due to overflow.

Output for bucket_size=4 :

0 -> 822 1746 1752 2388

1 -> 1048

2 -> 821 1757 1778 1928

3 -> 1761 2055 2379 2967

4 -> 2062

5 -> 2969

Output for bucket_size=5 :

0 -> 822 1746 1752 2388

1 -> 1048 2062

2 -> 821 1757 1778 1928

3 -> 1761 2055 2379 2967

4 ->

5 -> 2969

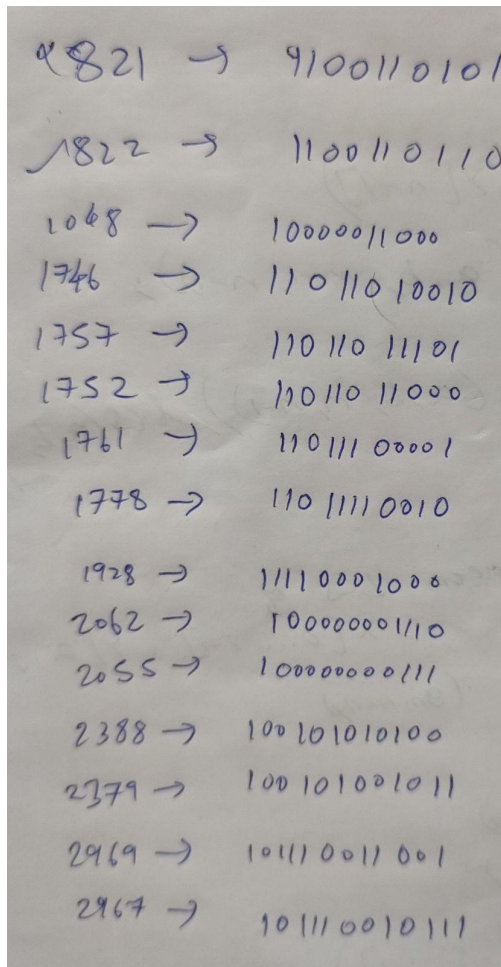
We can see that not much difference is observed in bucket size 4 and 5 thus keeping bucket size =4 will be the most optimal, as when bucket size is 5, storage space is getting wasted.

- e) I have tried using bst instead of a vector in the linear_hashing_tree.cpp file. This is giving a similar result to the normal linear list implementation of a bucket, **but the search time is coming out to be less, as in bst search takes $O(\log n)$ instead of $O(n)$**

Question 4:

- a) For the distributed hash tree, to get the best value of 'n' (number of higher positioned bits) I converted each of the hash values to binary so that I could get a rough idea of the ways in which the first 2-3 MSB's of the hash values were distributed.

Below is the picture of the distribution:



| | |
|--------|--------------|
| 1821 → | 9100110101 |
| 1822 → | 1100110110 |
| 1068 → | 10000011000 |
| 1746 → | 11011010010 |
| 1757 → | 11011011101 |
| 1752 → | 11011011000 |
| 1761 → | 11011100001 |
| 1778 → | 1101110010 |
| 1928 → | 11110001000 |
| 2062 → | 100000001110 |
| 2055 → | 10000000011 |
| 2388 → | 100101010100 |
| 2379 → | 100101001011 |
| 2969 → | 101110011001 |
| 2967 → | 101110010111 |

Seeing the pic, I think that if we use `bucket_size=4`, the value of `n='2'` would be sufficient to have an equal distribution of the data and utilize the buckets effectively.

- b) The final output after implementing distributed hashing is as follows:

(Distributed_hashing.cpp file)

Local depth of the bucket is : 1

822 1048 1746 1752

Local depth of the bucket is : 2

1761 2062 2379 2388

Local depth of the bucket is : 2

1778 1928 2055

Local depth of the bucket is : 3

821 1757 2967 2969

- c) The outputs for different global n(number of msb's we are considering)

Are as follows:

Output for n=1:

Local depth of the bucket is : 1

822 1048 1746 1752

Local depth of the bucket is : 2

1761 2062 2379 2388

Local depth of the bucket is : 2

1778 1928 2055

Local depth of the bucket is : 3

821 1757 2967 2969

Output for n=2:

Local depth of the bucket is : 1

822 1048 1746 1752

Local depth of the bucket is : 2

1761 2062 2379 2388

Local depth of the bucket is : 2

1778 1928 2055

Local depth of the bucket is : 3

821 1757 2967 2969

Output for n=3:

Local depth of the bucket is : 1

822

Local depth of the bucket is : 2

1761 2062 2379 2388

Local depth of the bucket is : 2

1778 1048

Local depth of the bucket is : 3

821 1757 2967 2969

Local depth of the bucket is : 3

1928 2055

Local depth of the bucket is : 3

1746 1752

Local depth of the bucket is : 0

Local depth of the bucket is : 0

As we can see, for n=1,2 the output is the same, since only 1 msb was not sufficient to differentiate the data. Thus according to the distributed hashing algorithm, we consider 2 msb's.

Also, for n=2 and n=3, we can see that the algorithm works better for n=2 , and memory is getting wasted for n=3. Since a total of 8 buckets are being used, and for n=2 only 4 buckets are being used.

d) Keeping n=2, I have experimented with different bucket sizes.

For bucket size=3:

The output had many different buckets at different local depths. And a lot of memory was getting wasted.

For bucket_size = 4 the output is already mentioned above.

For bucket size=5, the output was similar to bucket size=4. But there is no need of using more bucket size to get a similar result, thus, **the best value of bucket_size for our particular data and hash function is 4.**

e) I have experimented with trees instead of linear lists inside of buckets. Thus reducing time complexity.

Question 5:

a) The output in **extendible hashing**:

Inserting new entry according to question 5

2739 Finding Muchness

Successfully inserted: 2739

Inserted successfully

Local depth of the bucket is : 2

1048 1752 1928 2388

Local depth of the bucket is : 2

821 1757 1761 2969

Local depth of the bucket is : 2

822 1746 1778 2062

Local depth of the bucket is : 2

2055 2379 2967 2739

Time taken to insert (in microseconds):

1994

Linear Hashing:

Inserting new entry according to question 5

2739 Finding Muchness

Inserted successfully

0 -> 822 1746 1752 2388

1 -> 1048

2 -> 821 1757 1778 1928

3 -> 1761 2055 2379 2967 2739

4 -> 2062

5 -> 2969

Time taken to insert (in microseconds):

3986

Distributed Hashing:

Time taken to insert (in microseconds):

11961

B-Tree:

Inserting new entry according to question 5
2739 Finding Muchness Inserted successfully

Time taken to insert (in microseconds):
652

As we can see, the time order of inserting a new node is as follows:
B-Tree<Extendible Hashing< Linear Hashing< Distributed Hashing

B-Tree took the lowest time as it takes only $O(\log n)$ time, and it does not convert the hashValue to the binary form, and only does 2 or 3 comparisons and the job is done.

Then extendible hashing took the lowest time as after hashing the book_id, it has to directly point to the bucket number which is given by the binary representation's last 2 significant bits. Whereas, in LinearHashing, we are directly inserting the number in the bucket, but it takes extra time to accommodate the overflow condition, and, in Distributed Hashing, it has a similar implementation to Extendible hashing, but considers the MSB's of a number instead of LSB's and as it takes extra time to compute the MSB's, thus this is taking the highest time.

Note: B-Tree should take the highest time if the dataset was huge, as Hashing algorithms take $O(1)$ approximately to insert a single row in the dataset. But since our dataset is small, this is not the case.

b) The output in extendible hashing:

Searching new entry according to question 5
Found

Time taken to search (in microseconds):
2

Linear Hashing:

Searching new entry according to question 5

Found

Time taken to search (in microseconds):

4

Distributed Hashing:

Searching new entry according to question 5

Found

Time taken to search (in microseconds):

7

B-Tree:

Searching new entry according to question 5

Found

Time taken to search (in microseconds):

0

As we can see, the time order of searching for a row is as follows:

B-Tree<Extendible Hashing< Linear Hashing< Distributed Hashing.

This result is similar to the above question. And the explanation is also the same. Extendible hashing took the lowest time in all the 3 hashing mechanisms, as in Linear Hashing, we search in a particular bucket, and then we traverse in the bucket, to search for the particular hash value.

Extendible hashing took less time than linear hashing since $n \bmod 4$ takes less time than $n \bmod 6$. Also distributed hashing takes the highest time, as calculating bucket number using MSB's takes more time than calculating them using LSB's.

c) The output in **extendible hashing**:

Searching new entry according to question 5

Found all

Time taken to search (in microseconds):

26

Linear Hashing:

Searching new entry according to question 5

Found all

Time taken to search (in microseconds):

68

Distributed Hashing:

Searching new entry according to question 5

Found all

Time taken to search (in microseconds):

100

B-Tree:

Searching new entry according to question 5

Found all

Time taken to search (in microseconds):

2

As we can see, the time order of searching for a row is as follows:

B-Tree<Extendible Hashing< Linear Hashing< Distributed Hashing.

Here also, we need to search for 2 entries, while traversing in the data. Thus we have the same type of output.

EXTRA IMPLEMENTATION: I have implemented B-tree to store the database according to the hashed values.