# DBMS LAB 3

**Ridhima Kohli**
**B19CSE071**

1. Consider the following partial table of an ordered library catalogue:

| Author_ID | Book_ID | Author_Name | Book |
|---|---|---|---|
| Da_001 | Da001_Sel | Damasio | Self Comes to Mind |
| To_015 | To015_Fel | Tolkien | Fellowship of the Rings_Lord of the Rings |
| Mi_009 | Mi009_Emo | Minsky | Emotion Machine |
| Mi_009 | Mi009_Soc | Minsky | Society of Mind |
| Ra_001 | Ra001_Pha | Ramachandran | Phantoms in the Brain |
| Ro_015 | Ro015_Fan | Rowling | Fantastic Beasts and Where to Find Them |
| Ro_015 | Ro015_Gob | Rowling | Goblet of Fire_Harry Potter |
| Ro_015 | Ro015_Pri | Rowling | Prisoner of Azkaban_Harry Potter |
| Sa_001 | Sa001_Wha | Safina | What Animals Think |
| Wo_015 | Wo015_Wod | Wodehouse | Wodehouse at the Wicket |

**a) For the given table: What are the potential candidate keys, and which one would you use as the primary key and why? (10 points)**

Potential candidate keys for given data entries are

- Book_ID
- Book_ID + Book
- Book

On extension if Book initials repeat then Book_ID+Book can be a good choice for candidate key
I would use generate a new primary key with the help of Book_ID and book incase Book_ID repeats , Book can help in distinguishing the full name of Books. → new primary key can be PK which is **Book_ID_code =** (Book_ID+Book+a number) similar to how authorid is made.
For this we need to keep track of number of books under same author with same initials
Example : Da001_Sel is book ID for **Self Comes to Mind** by Damasio.Now suppose Damasio has another book called **Sell your mind , still the Book_ID would remain same here : Da001_Sel**
So instead we can make a new Primary Key **Book_ID_code** which is Da001_Sel001 for **Self Comes to Mind** and Da001_Sel002 for **Sell your mind**
Thus it will help uniquely identify every entity even if the author name is same and book name has same initial 3 letters

**Note:**

**● It is known that most user queries involve the Author_Name and/or Book_Name**

**● Typing the first three characters <xxx> of a search value for either of the attributes, on the library exploration portal, is sufficient to retrieve a set of records that begin with <xxx>.**

For such search we can create two tables , one containing author information and other book information
Author table has an author id with initial 3 letters of author name while book table has book id with initial 3 letters of book name
These can help in searching
Though for simplicity here we have considered the given 1 table and performed analysis of hashing techniques on them

## New Primary key correctness

```
int main()
{

    string Book_ID_1 = "Da001_Sel001";
    string Book_ID_2="Da001_Sel002";

    comp(Book_ID_1,Book_ID_2);
    cout<<"Book_ID_code Da001_Sel001 in
binary form : -\n";
    ConverToBinary(Book_ID_1);
    cout<<"\n-------------\nBook_ID_code
Da001_Sel002 in binary form : -\n";
    ConverToBinary(Book_ID_2);
    return 0;
}
```

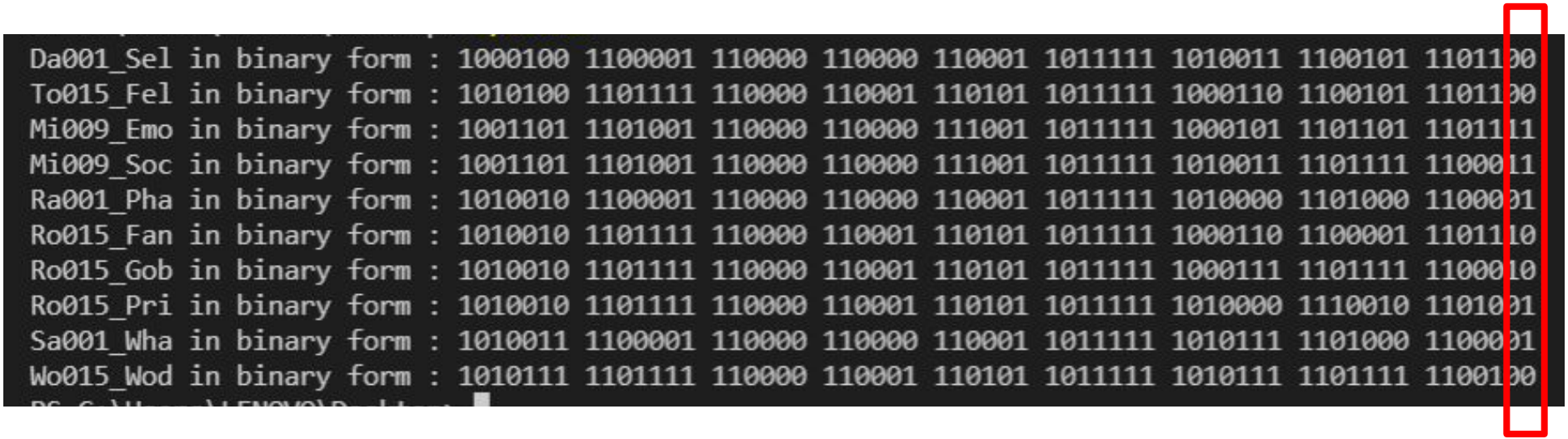
```
index 0 is equal
index 1 is equal
index 2 is equal
index 3 is equal
index 4 is equal
index 5 is equal
index 6 is equal
index 7 is equal
index 8 is equal
index 9 is equal
index 10 is equal
index 11 is not equal
Book_ID_code Da001_Sel001 in binary form : -
1000100 1100001 110000 110000 110001 1011111 1010011 1100101 1101100 110000 110000 110001
------------
Book_ID_code Da001_Sel002 in binary form : -
1000100 1100001 110000 110000 110001 1011111 1010011 1100101 1101100 110000 110000 110010
```

To compare two values in Book_ID_code , we convert each letter of string to binary and compare the values as shown. comp() function is being used for comparison and strToBinary converts letters to binary form by converting their corresponding *decimal base ascii to binary*. Hence in this way , this newly introduced field can be used as primary key. Assumption made - each author can have at most 999 books with same initial letters which is quiet practical in real life

```
void ConverToBinary(string s)
{
    int n = s.length();
int v = 0;
    for (int i = 8; i <= n; i++)
    { int val = int(s[i]);
        v+=val;
        string bin = "";
        while (val > 0)
        {
            (val % 2)? bin.push_back('1') :
                    bin.push_back('0');
            val /= 2;
        }
        reverse(bin.begin(), bin.end());
        cout << bin << " ";
    }
    cout<<"\n";
    cout<<"ASCII value of "<<s<<" is "<<v<<endl;
    int m = v%3;
    int m2=v%6;
    cout<<"Hash function : h1 = key % 3 gives hashKey = " <<m<<endl;;
    cout<<"New Hash function : h2 = key % 6 gives hashKey = " <<m2<<endl;;
    cout<<s<<" : "<<m<<"  ---  "<<m2<<endl;
}
```

Function to convert and display the binary forms and values from hash function applies

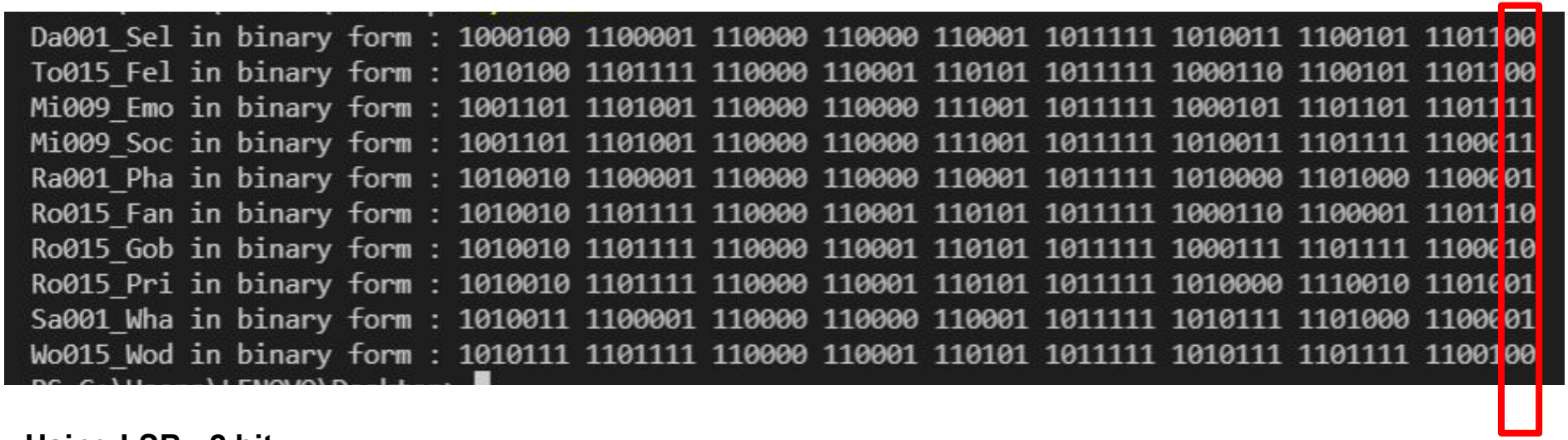
**b) How would you use extendible hashing with bucket size 4 to design an effective access strategy for the given scenario?**




**Using LSB - 1 bit**

| Global | Local , bucket size = 4 |
|---|---|
| 0 | Da001_Sel , To015_Fel ,Ro015_Fan , Ro015_Gob |
| 1 | Mi009_Emo , Mi009_Soc,Ra001_Pha,Ro015_Pri,**Sa001_Wha** |

Overflow




**Using LSB - 2 bit**

| Global | Local , bucket size = 4 |
|---|---|
| 00 | Da001_Sel , To015_Fel ,Wo015_Wod |
| 10 | Ro015_Fan , Ro015_Gob |
| 01 | Ra001_Pha,Ro015_Pri,**Sa001_Wha** |
| 11 | Mi009_Emo, Mi009_Soc |

Overflow adjusted

First we try to insert using the LSB and incase the LSB gives a slot that is full , we extend the storage

```
void insertExtend(vector<vector<int>> A , val ){
    int slot = getLSB(val);
    if(isNotFull(A[slot])){
        A[slot].push_back(val);

    }
    else{
        extend(A,val,slot);
    }
}
```

Some extra work we can do (This is not the answer to part b , just some way I tried to work out)

**Other methods ( apart from answer to ques b )we can try to use**
Method 1 : Use initials of author name as global directory
Method 2 : Use book name initials as global directory

If we could directly search for the whole binary number , then it would be faster

The following slides demonstrate how we can categorise the data

## Method 1

Since for the table in **given** question ,

- every row can be uniquely identified just with the Book_ID
- Making Book_ID_code won't make much difference since the IDs will have the same additional book number 001 as they are all unique

so let's solve the given problems with **Book_ID as primary key** for simplicity

b)

Book_ID as primary key

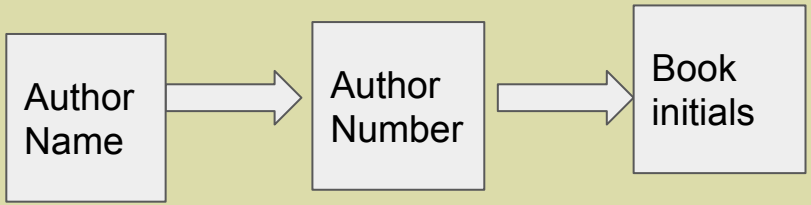
| Global Directory |
|---|
| Da |
| To |
| Mi |
| Ra |
| Ro |
| Sa |
| Wo |
| . |
| . |
| . |
| . |

| Local Directory |
|---|
| 001_ |
| 002_ |
| 003_ |
| 004_ |
| . |
| . |
| . |

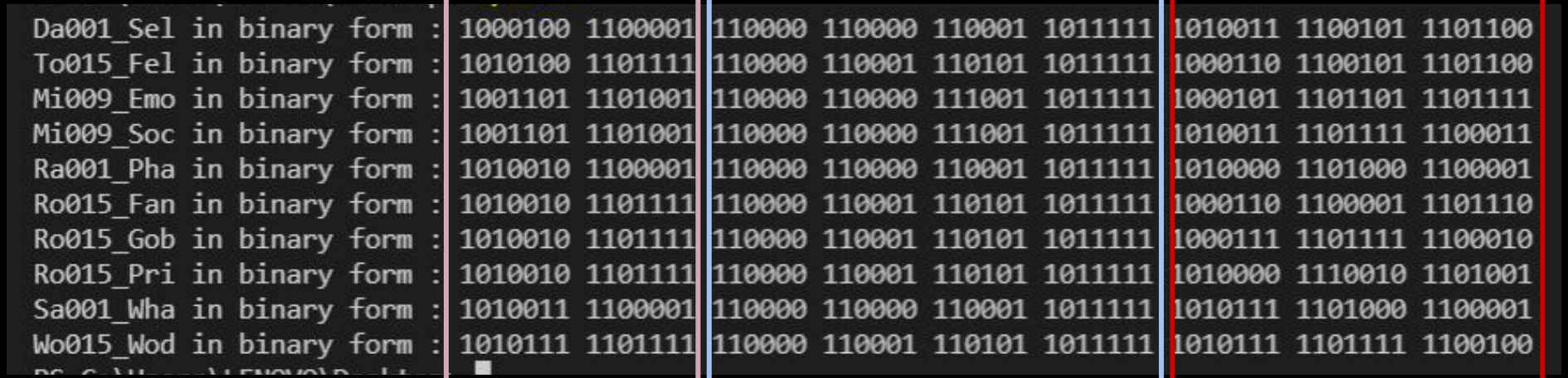
Bucket Size 4

| | | | |
|---|---|---|---|
| Da001_Sel | Ra001_Pha | Sa001_Wha | |

The binary representations can be compared after being combined accordingly.
Suppose we want to find Da001_Sel
We will look at the author name initials' binary representation , then we go further and check the author's number bytes in binary representation , followed by book name



Book_ID as primary key




Global Directory

Local Directory

Bucket

Book_ID_code as primary key

## Method 2

**Now suppose we use the new introduced primary key Book_ID_code**

**b)**

**Book_ID as primary key**

| Global Directory |
|---|
| Sel |
| Pha |
| Wha |
| . |
| . |

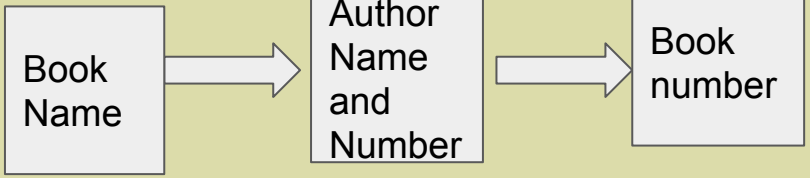
| Local Directory |
|---|
| Da001 |
| Da003 |
| Wha002 |
| . |
| . |
| . |
| . |

Bucket Size 4

| | | | |
|---|---|---|---|
| Da001_Sel001 | Da001_Sel002 | | |

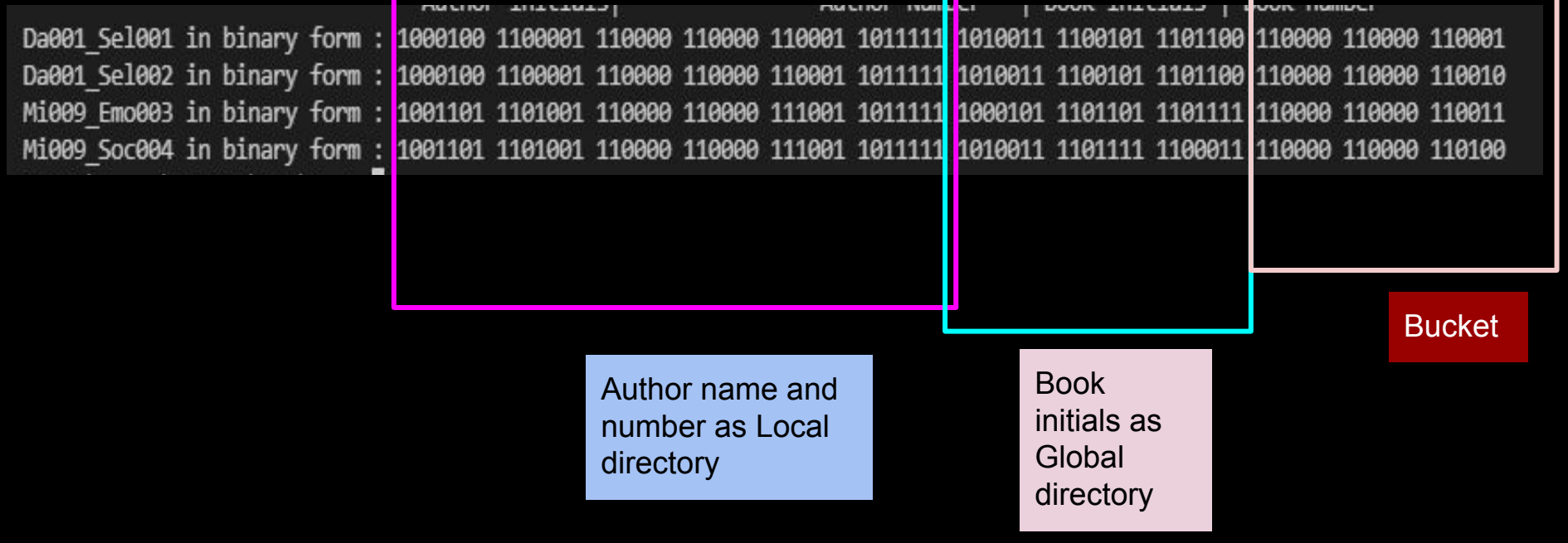
The binary representations can be compared after being combined accordingly.
Suppose we want to find Da001_Sel001

We will look at the book name initials' binary representation in global directory , then we go further and check the author's name and number bytes in binary representation , followed by book number




Da001_Sel001 :**Self Comes to Mind**

Da001_Sel002:**Sell your mind**


Da001_Sel001 in binary form : 1000100 1100001 110000 110000 110001 1011111 1010011 1100101 1101100 110000 110000 110001
Da001_Sel002 in binary form : 1000100 1100001 110000 110000 110001 1011111 1010011 1100101 1101100 110000 110000 110010
Mi009_Emo003 in binary form : 1001101 1101001 110000 110000 111001 1011111 1000101 1101101 1101111 110000 110000 110011
Mi009_Soc004 in binary form : 1001101 1101001 110000 110000 111001 1011111 1010011 1101111 1100011 110000 110000 110100
Author name and number as Local directory
Book initials as Global directory
Bucket

**c) Do you think a larger bucket size would be more effective? Experiment with at least one smaller and at least one larger bucket size to justify your claim.**

Let us suppose we have a large bucket size , say 10

- Now in this case , the number of collisions will reduce
- For the given data there would hardly be any collision
- But it will also lead to wastage of space incase more data is not added

```
Da001_Sel in binary form : 1000100 1100001 110000 110000 110001 1011111 1010011 1100101 1101100
To015_Fel in binary form : 1010100 1101111 110000 110001 110101 1011111 1000110 1100101 1101100
Mi009_Emo in binary form : 1001101 1101001 110000 110000 111001 1011111 1000101 1101101 1101111
Mi009_Soc in binary form : 1001101 1101001 110000 110000 111001 1011111 1010011 1101111 1100011
Ra001_Pha in binary form : 1010010 1100001 110000 110000 110001 1011111 1010000 1101000 1100001
Ro015_Fan in binary form : 1010010 1101111 110000 110001 110101 1011111 1000110 1100001 1101110
Ro015_Gob in binary form : 1010010 1101111 110000 110001 110101 1011111 1000111 1101111 1100010
Ro015_Pri in binary form : 1010010 1101111 110000 110001 110101 1011111 1010000 1110010 1101001
Sa001_Wha in binary form : 1010011 1100001 110000 110000 110001 1011111 1010111 1101000 1100001
Wo015_Wod in binary form : 1010111 1101111 110000 110001 110101 1011111 1010111 1101111 1100100
```

| Global | Local , bucket size = 10 |
|---|---|
| 0 | Da001_Sel , To015_Fel ,Ro015_Fan , Ro015_Gob |
| 1 | Mi009_Emo , Mi009_Soc,Ra001_Pha,Ro015_Pri,Sa001_Wha |

**c) Do you think a larger bucket size would be more effective? Experiment with at least one smaller and at least one larger bucket size to justify your claim.**

However as we saw in **part b** , for **smaller bucket size 4** , the collisions occur before and hence are more frequent ( since collision occurs when the bucket is full , so having small bucket will give more collisions )
Now to resolve this , we split.
Hence we can conclude that
If we need to prioritize less memory usage we should prefer lower bucket size as it would reduce chances of wasting memory
If we need to reduce the number of splittings , then we should try to consider higher bucket size

**Using LSB - 1 bit**

Overflow

| Global | Local , bucket size = 4 |
|---|---|
| 0 | Da001_Sel , To015_Fel ,Ro015_Fan , Ro015_Gob |
| 1 | Mi009_Emo , Mi009_Soc,Ra001_Pha,Ro015_Pri,**Sa001_Wha** |

**Using 2 bit**

| Global | Local , bucket size = 4 |
|---|---|
| 00 | Da001_Sel , To015_Fel ,Wo015_Wod |
| 10 | Ro015_Fan , Ro015_Gob |
| 01 | Ra001_Pha,Ro015_Pri,**Sa001_Wha** |
| 11 | Mi009_Emo, Mi009_Soc |

**d) How would you use linear hashing with a bucket size of 4 to design an effective strategy for the given scenario?**

For bucket size 4 , we need at least 3 slots ( because total data entered currently is 10 ) to prevent overflow

Hash function used :

- Convert primary key to its decimal equivalent - key
- Calculate the v = key % 3
- Enter the record into slot number v

```
Da001_Sel : 0
To015_Fel : 0
Mi009_Emo : 0
Mi009_Soc : 0
Ra001_Pha : 1
Ro015_Fan : 2
Ro015_Gob : 2
Ro015_Pri : 0
Sa001_Wha : 1
Wo015_Wod : 1
```

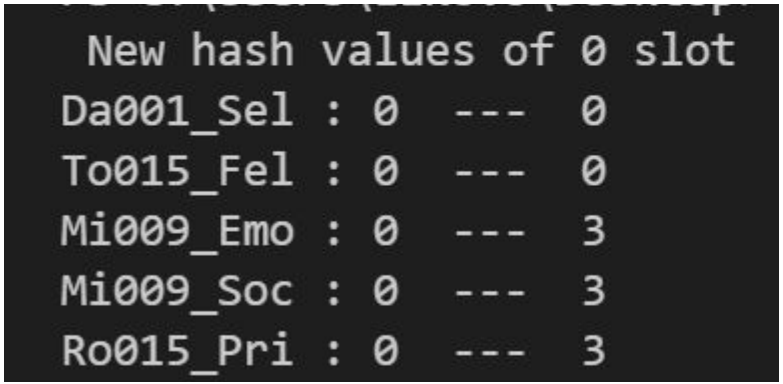
| Index | Bucket Size = 4 | | | |
|---|---|---|---|---|
| 0 | Da001_Sel | To015_Fel | Mi009_Emo | Mi009_Soc |
| 1 | Ra001_Pha | Sa001_Wha | Wo015_Wod | |
| 2 | Ro015_Fan | Ro015_Gob | | |

**Ro015_Pri**

**Overflow**

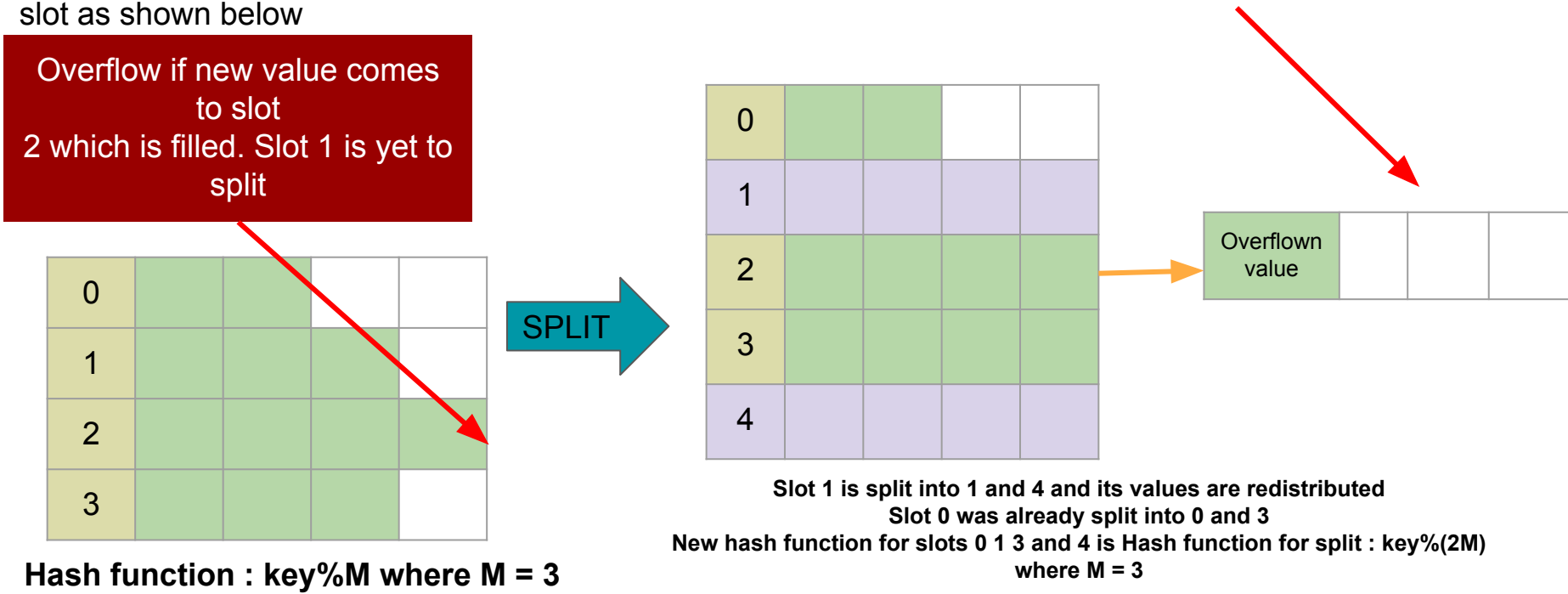
Since overflow has occurred , we split 0 into 0 and M (here 3) with new hash function key%2M = key%6

| Index | Bucket Size = 4 | | | |
|---|---|---|---|---|
| 0 | Da001_Sel | To015_Fel | | |
| 1 | Ra001_Pha | Sa001_Wha | Wo015_Wod | |
| 2 | Ro015_Fan | Ro015_Gob | | |
| 3 | Mi009_Emo | Mi009_Soc | **Ro015_Pri** | |



Now suppose on more data entries an overflow occurs due to the hash function producing the output slot number which is already filled in some slot other than 0
Then we will split slot 1 into 1 and M+1 (here 4) and rehash its values according to new hash function(key%6) and so on..

Incase the overflow value is different from the one split , it is stored in the **overflow bucket** in the same slot as shown below




**Hash function : key%M where M = 3**

**Slot 1 is split into 1 and 4 and its values are redistributed**
**Slot 0 was already split into 0 and 3**
**New hash function for slots 0 1 3 and 4 is Hash function for split : key%(2M) where M = 3**

```
ASCII value of Da001_Sel is 108
Hash function : h1 = key % 3 gives hashKey = 0 key name : Da001_Selkey id :1
0 --> 1
1 -->
2 -->
ASCII value of To015_Fel is 108
Hash function : h1 = key % 3 gives hashKey = 0 key name : To015_Felkey id :2
0 --> 1 2
1 -->
2 -->
ASCII value of Mi009_Emo is 111
Hash function : h1 = key % 3 gives hashKey = 0 key name : Mi009_Emokey id :3
0 --> 1 2 3
1 -->
2 -->
ASCII value of Mi009_Soc is 99
Hash function : h1 = key % 3 gives hashKey = 0 key name : Mi009_Sockey id :4
0 --> 1 2 3 4
1 -->
2 -->
ASCII value of Ra001_Pha is 97
Hash function : h1 = key % 3 gives hashKey = 1 key name : Ra001_Phakey id :5
0 --> 1 2 3 4
1 --> 5
2 -->
ASCII value of Ro015_Fan is 110
Hash function : h1 = key % 3 gives hashKey = 2 key name : Ro015_Fankey id :6
0 --> 1 2 3 4
1 --> 5
2 --> 6
ASCII value of Ro015_Gob is 98
Hash function : h1 = key % 3 gives hashKey = 2 key name : Ro015_Gobkey id :7
0 --> 1 2 3 4
1 --> 5
2 --> 6 7
ASCII value of Ro015_Pri is 105
Overflow Alert : slot 0 's bucket is full
Overflow happens , Split slot 0 into slots 0 -- 3
New Hash function : h2 = key % 4 gives hashKey = 3for slot 0 and 3
0 --> 1 2
1 --> 5
2 --> 6 7
3 --> 3 4 8
ASCII value of Sa001_Wha is 97
```

## Implementation of linear hashing

An array of structs was made where each element represents a slot and each slot has various entities like id ( taken as 1 , 2 , 3 ..etc for simplicity of displaying) , bucket , overflow bucket , each of size 4 , etc as shown in the next slide

Overflow adjusted

```
#include <iostream>
#include <bits/stdc++.h>
#include <stdlib.h>
using namespace std;

struct Slot{
    int slotNo; //slot number
    int bucketFilled; // number of bucket components filled in original bucket
    int bucket[4]; //original bucket
    int asc[4]; //stores ascii value of original-bucket-inserted BookID which can be later
extracted while rehashing
    int overflowFill; //stores count of overflow bucket components
    int ovasc[4]; //stores ascii value of overflow-bucket-inserted BookID which can be later
extracted while rehashing
    int overflowBucket[4]; //overflow bucket
    int hashFunction; //stores hash function applicable for slot
};
struct Slot *Storage;
```

```
int toSplit = 0;
int extended=3;
int over=0;
void hash(string s , int x)
{   int n = s.length();
    int v = 0;
    for (int i = 8; i <= n; i++)
    {
        int val = int(s[i]);
        v+=val;
    }
    cout<<"ASCII value of "<<s<<" is "<<v<<endl;
    int m = v%3; //first hash function
    if(Storage[m].bucketFilled!=4){
        cout<<"Hash function : h1 = key % 3 gives hashKey = "<<m<<" key name : "<<s<<"key id
:"<<x<<endl;
        Storage[m].bucket[Storage[m].bucketFilled] = x;
        Storage[m].asc[Storage[m].bucketFilled] = v;
        Storage[m].bucketFilled++;}
```

```
  else{ //overflow
      over++;
      cout<<"Overflow Alert : slot " <<m<<" 's bucket is full \n";
        cout<<"Overflow happens , Split slot " <<toSplit<<" into slots "<<toSplit<<" -- "<<extended<<endl;
        int m2=v%6;
        cout<<"New Hash function : h2 = key % 4 gives hashKey = "  <<m2<< "for slot "<<toSplit<<" and "<<extended<<endl;
        vector<vector< int> > newMapping(2);
        Storage[toSplit].hashFunction=2;
        Storage[extended].hashFunction=2;
            //new storage location is m2
            //rehash toSplit into toSplit and extended
            for(int SplitBind=0;SplitBind<4;SplitBind++){
              int vasc = Storage[toSplit].asc[SplitBind];
              int newSlot = vasc%6;
              if(newSlot==toSplit)
                newMapping[0].push_back(Storage[toSplit].bucket[SplitBind]);
                else
                 newMapping[1].push_back(Storage[toSplit].bucket[SplitBind]);
           Storage[toSplit].bucket[SplitBind]=-1;
              Storage[toSplit].bucketFilled--;            }
if(m==toSplit){ //if overflow is in slot to be split take new hash function
               if(m2==toSplit)
                newMapping[0].push_back(x);
                else
                 newMapping[1].push_back(x);}
else{ //put in overflow bucket
    Storage[m].overflowBucket[Storage[m].overflowFill] = x;
        Storage[m].overflowFill++;}
```

```
        for(int r=0;r<newMapping[0].size();r++){
            Storage[toSplit].bucket[r] = newMapping[0][r];
        }

         for(int r=0;r<newMapping[1].size();r++){
            Storage[extended].bucket[r] = newMapping[1][r];
        }

        // rehash

        toSplit++;
        extended++;
    }
    for(int st=0;st<3+over;st++){
        cout<<st<<" --> ";
        for(int b=0;b<4;b++){
 if(Storage[st].bucket[b]!=-1)
    cout<<Storage[st].bucket[b]<<" ";
}cout<<endl;}}
```

```
int main(){
Storage = (struct
Slot*)malloc(sizeof(struct Slot)*6);
for(int i=0;i<6;i++){
    Storage[i].slotNo=i;
     Storage[i].bucketFilled=0;
      Storage[i].overflowFill=0;
       Storage[i].hashFunction=1;
       for(int j=0;j<4;j++){
Storage[i].bucket[j]=-1;

Storage[i].overflowBucket[j]=-1;
       }}
    string Book_ID_1 ="Da001_Sel";
    string Book_ID_2="To015_Fel";
    string Book_ID_3="Mi009_Emo"; //
    string Book_ID_4="Mi009_Soc";
    string Book_ID_5="Ra001_Pha";
    string Book_ID_6="Ro015_Fan";
```

```
string Book_ID_7="Ro015_Gob";
   string Book_ID_8="Ro015_Pri";
   string Book_ID_9="Sa001_Wha"; //
   string Book_ID_10="Wo015_Wod"; //
   string b = "Sa001_Voy"; //
   string c = "Ro015_Phi";
   hash(Book_ID_1,1);
   hash(Book_ID_2,2);
   hash(Book_ID_3,3);
   hash(Book_ID_4,4);
   hash(Book_ID_5,5);
   hash(Book_ID_6,6);
   hash(Book_ID_7,7);
   hash(Book_ID_8,8);
   hash(Book_ID_9,9);
   hash(Book_ID_10,10);
   hash(b,11);
    hash(c,12);
   return 0;}
```

**e) Do you think a larger bucket size would be more effective? Experiment with at least 1 smaller and at least one larger bucket size to justify your claim.**

Just like the extendible hashing technique , here too larger bucket size will **reduce collisions** and hence the **frequency of using the second hash function will reduce**. However it can also cause **wastage of space** incase a slot isn't getting enough values to store.

Let's take example of larger bucket size 20
In this the number of slots required can be 1 only , and the hash function to be used can be taken as key%1 which will always direct us to slot 0

Now , if there are only 10 values , they all will be easily put into slot 0
However the remaining bucket parts will be empty and thus it will waste space of 10

But in smaller bucket size 4 , as we saw , the **space wastage was less**

## f) For a new record: < Sa_001, Sa001_Voy, Safina, Voyage of the Turtle >

## a. Trace the insertion into the extendible hash bucket of Q.a

```
PS C:\Users\LENOVO\Desktop> ./a.exe
Sa001_Voy in binary form : 1010011 1100001 110000 110000 110001 1011111 1010110 1101111 1111001
ASCII value of Sa001_Voy is 738
Hash function : h1 = key % 3 gives hashKey = 0
New Hash function : h2 = key % 6 gives hashKey = 0
```

For extendible hashing , we had made the table as shown

| Global | Local , bucket size = 4 |
|---|---|
| 00 | Da001_Sel , To015_Fel ,Wo015_Wod |
| 10 | Ro015_Fan , Ro015_Gob |
| 01 | Ra001_Pha,Ro015_Pri,Sa001_Wha , **Sa001_Voy** |
| 11 | Mi009_Emo, Mi009_Soc |

**No overflow**

## b. Trace the insertion into the linear hash bucket of Q.c


We had the given data , since hash function produces slot number 1 , which is not full , it means we can insert it without any overflow

| Index | Bucket Size = 4 | | | |
|---|---|---|---|---|
| 0 | Da001_Sel | To015_Fel | | |
| 1 | Ra001_Pha | Sa001_Wha | Wo015_Wod | Sa001_Voy |
| 2 | Ro015_Fan | Ro015_Gob | | |
| 3 | Mi009_Emo | Mi009_Soc | | |

## c. Write a short 150-200 words note on your observations between the above two insertions and subsequent time of retrieval for the record

**Insertion** ( Considering only the insertions made in part a and b as mentioned)
In extendible hashing we are using the LSBs to find out position of element. So the insertion would require conversion of key's last character into binary form. This would require access to last letter and convert it into binary which can be done in O(1) time. After that to check value of LSBs , we need constant time again. Now to insert , since there was no overflow , we just found the empty bucket slot in position number 01 and inserted it , which would require checking through the bucket slots . Here we went through 3 full bucket slots and found the 4th one empty.

In linear hashing method used here, we first added the ascii values of all characters which would take O(n) time and then applied the hash function which resulted in slot 1. Since slot 1 still has empty bucket slot , so no overflow happened.

**Retrieval**
To retrieve the record **Sa001_Voy** , in extendible hashing we first convert the last character into binary , check LSB and go to the required slot 01. Then we iterate through all buckets to check where the value is present. This can be done in O(n). Since bucket size is small so n = 4 and hence it can be assumed to be constant time.
In Linear hashing , we first added the ascii values of all characters which would take O(n) time and then applied the hash function which results in required slot x , where we iterate through bucket to find the value. Hence , it would take O(n) time where n = length of Book_ID

**g) For the record <Ro_015, Ro015_Phi, Rowling, Philosopher's Stone_Harry Potter>**
**a. Trace the insertion into the extendible hash bucket of Q.e**


For extendible hashing , we had made the table as shown

| Global | Local , bucket size = 4 |
|---|---|
| 00 | Da001_Sel , To015_Fel ,Wo015_Wod |
| 10 | Ro015_Fan , Ro015_Gob |
| 01 | Ra001_Pha,Ro015_Pri,Sa001_Wha , Sa001_Voy |
| 11 | Mi009_Emo, Mi009_Soc |

**Overflow**

So we need to split the extendible hashing bucket

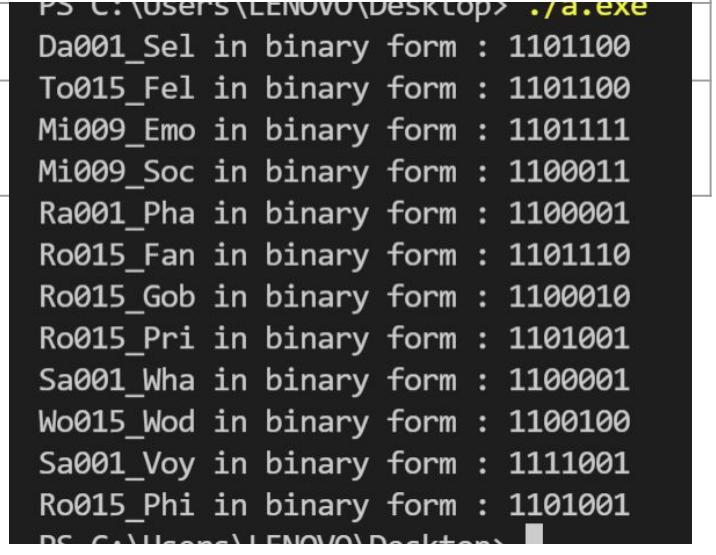
| Global | Local , bucket size = 4 |
|---|---|
| 00 | Da001_Sel , To015_Fel ,Wo015_Wod |
| 10 | Ro015_Fan , Ro015_Gob |
| 01 | Ra001_Pha,Ro015_Pri,Sa001_Wha , Sa001_Voy , **Ro015_Phi** |
| 11 | Mi009_Emo, Mi009_Soc |

**Problem of overflow is still not resolved on taking 3 LSB**

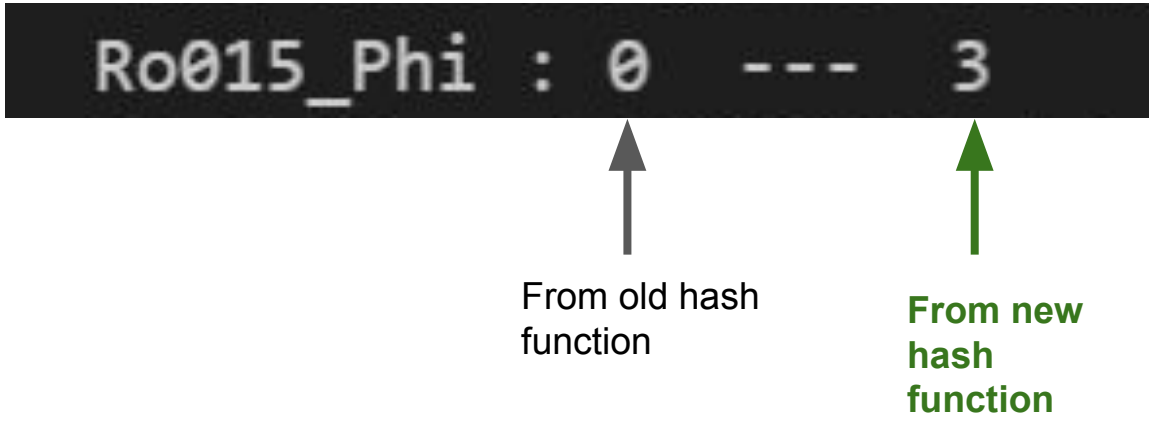
| Global | Local , bucket size = 4 |
|---|---|
| 000 , 100 | Da001_Sel , To015_Fel ,Wo015_Wod |
| 010 , 110 | Ro015_Fan , Ro015_Gob |
| **001** | **Ra001_Pha**,**Ro015_Pri,Sa001_Wha , Sa001_Voy** , **Ro015_Phi** |
| 101 | |
| 111,011 | Mi009_Emo, Mi009_Soc |

| Global | Local , bucket size = 4 |
| --- | --- |
| 0000 , 0100 , 1000,1100 | Da001_Sel , To015_Fel ,Wo015_Wod |
| 0010 , 0110,1010 , 1110 | Ro015_Fan , Ro015_Gob |
| **0001** | **Ra001_Pha,Sa001_Wha** |
| **1001** | **Ro015_Pri , Sa001_Voy, Ro015_Phi** |
| **0101,1101** | |
| 0111,0011,1111,1011 | Mi009_Emo, Mi009_Soc |

LSB 4 bit



## g)b. Trace the insertion into the linear hash bucket of Q.e




| Index | Bucket Size = 4 | | | |
|---|---|---|---|---|
| 0 | Da001_Sel | To015_Fel | | |
| 1 | Ra001_Pha | Sa001_Wha | Wo015_Wod | Sa001_Voy |
| 2 | Ro015_Fan | Ro015_Gob | | |
| 3 | Mi009_Emo | Mi009_Soc | **Ro015_Phi** | |

**c. Write a short 150-200 words note on your observations between the above two insertions and subsequent time of retrieval for the record**

**Insertion** ( Considering only the insertions made in part a and b as mentioned)
In extendible hashing we are using the LSBs to find out position of element. So the insertion would require conversion of key's last character into binary form. This would require access to last letter and convert it into binary which can be done in O(1) time. After that to check value of LSBs , we need constant time again. Now to insert , since there was overflow , we adjusted the hashing to 3 bits , still the overflow was not removed so we adjusted it to 4. Hence there were 2 overflows

In linear hashing method used here, we first added the ascii values of all characters which would take O(n) time and then applied the hash function which resulted in slot 0. Since slot 0 was not full , so it was inserted

**Retrieval**
To retrieve the record , in extendible hashing we first convert the last character into binary , check LSB and go to the required slot 01. Then we iterate through all buckets to check where the value is present. This can be done in O(n). Since bucket size is small so n = 4 and hence it can be assumed to be constant time.
In Linear hashing , we first added the ascii values of all characters which would take O(n) time and then applied the hash function which results in required slot x , where we iterate through bucket to find the value. Hence , it would take O(n) time where n = length of Book_ID

**h) Can you think of a better way to encode the author_id and book_id - with respect to the format followed in the given table? Why do you think your code is better?**

We can create new primary key

- **Book_ID+Book_numbe**r :

  Example : like Da001_Sel001 and Da001_Sel002 for two different books with same initials by same author Da001

This will help in handling cases when two different books with same author , and same initial letters in their names are entered , they can be identified distinctively

However it can cause more collisions as IDs ending with a number , say 001 might be many. So instead we can rearrange the ID as Da001_001Sel , which will have these benefits

- Identify record uniquely
- Using last 2-3 LSBs we can store and search using book_name initials
- Using initial MSBs we can store and search using author_name initials

Using this type of primary key is more beneficial if we consider such corner cases