SYNERGY STRIKERS

Project Summary Report

Project Name: DBMS Using file based B-TREE implementation

Project Code:

Date Started: 25-October-2018

- *Main Goals: To develop a basic Database management system using B-tree, which supports SQL Operations like insert, search and update.*

Assigned to: Synergy Strikers
Team Members : MIT2018001, MIT2018002, MIT2018075

- *Learnings :Trying to write structure pointers on file and read them, leading to segmentation fault when we restart the program, fixed the issue by logically assigning block number to every node.*
- *Strtok function for tokenizing the string (for parsing our SQL Statements).*
- *Basics Shell scripting to trigger the program automatically and feed the input to program directly.*
- *Used ANSI colour codes to display colourful text in printf statements.*

Deadline: 14-November-2018

DBMS USING FILE BASED B-TREE IMPLEMENTATION

Abstract : To Implement a Database Management System using Btree on disk based on the algorithm provided in Introduction to Algorithms by CLRS

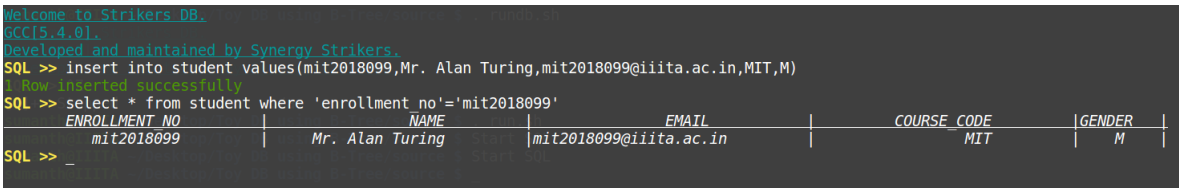
To run the program :

1. Open terminal in *"source"* folder, then run *". run.sh"*.
2. Type ***"Start SQL"*** (case sensitive).
3. Now we can go type our SQL commands in the prompt.

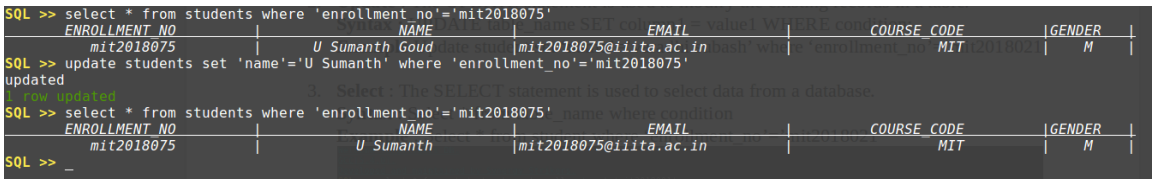
Properties supported :

We have tried to create a SQL replica which does basic operations on the student database.

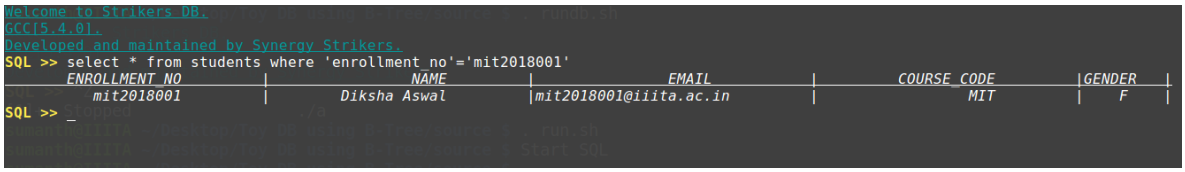
1. **Insert** : The INSERT INTO statement is used to insert new records in a table.
   Syntax : insert into table_name values(value1, value2, value3, ...);
   Example : insert into student values(mit2018099, Mr. Alan Turing, mit2018099@iiita.ac.in, MIT,M);



2. **Update** : The UPDATE statement is used to modify the existing records in a table.
   Syntax : UPDATE table_name SET column1 = value1 WHERE condition;
   Example:Update student set 'name'='varadhi subash' where 'enrollment_no'='mit2018021'



3. **Select** : The SELECT statement is used to select data from a database.
   Syntax : Select * from table_name where condition
   Example : Select * from student where 'enrollment_no'='mit2018021'



Why B-Tree over any other Data Structure :

B-Trees are a popular index data structure, coming in many variations and used in many databases, including MySQL, InnoDB and PostgreSQL. B-Tree is a generalization of the Binary Search Tree, in which more than two pointers are allowed per node. B-Trees are self-balancing, so there's no rotation step required during insertion and deletion, only merges and splits. The reason to use them for indexes, where lookup time is important, is their logarithmic lookup time guarantee.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed. B-tree algorithms keep only a

constant number of pages in main memory at any time; thus, the size of main memory does not limit the size of B-trees that can be handled.

We model disk operations in our pseudocode as follows. Let ***x*** be a pointer to an object. If the object is currently in the computer's main memory, then we can refer to the attributes of the object as usual: ***x.key***, for example. If the object referred to by ***x*** resides on disk, however, then we must perform the operation DISK-READ(***x***) to read object ***x*** into main memory before we can refer to its attributes. (We assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses; it is a "no-operation.") Similarly, the operation DISK-WRITE(***x***) is used to save any changes that have been made to the attributes of object ***x***. That is, the typical pattern for working with an object is as follows.

1. ***x*** = a pointer to some object.
2. DISK-READ(***x***)
3. operations that access and/or modify the attributes of ***x***.
4. DISK-WRITE(***x***) // omitted if no attributes of x were changed
5. other operations that access but do not modify attributes of ***x***.

The system can keep only a limited number of pages in main memory at any one time. We shall assume that the system flushes from main memory pages no longer in use; our B-tree algorithms will ignore this issue.

Since in most systems the running time of a B-tree algorithm depends primarily on the number of DISK-READ and DISK-WRITE operations it performs, we typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk page, and this size limits the number of children a B-tree node can have.

In most modern computers the size of diskpage = 4096Bytes (4Kb).
In our implementation of Btree for student database, Size of each student record = 70Bytes.
So floor of (4096 divided by 70) keys per node will give me the optimal performance i.e (58 keys)

For us to have 58 keys, our 2T – 1 should be equal to 58

- ➔ 2T – 1= 58
- ➔ 2T = 59
- ➔ T = ceil(29.5)
- ➔ T = 30.

So ideally when ***T*** is ***30***, we will get the optimal performace i.e disk reads and writes will be minimal.

The goal of the btree is to minimize the number of disk accesses. If the file system cluster size of 4k, then the ideal size for the nodes is 4k. So taking T as 30 will give us the ideal performace according to our analysis.

Approach :

Search & Update : The following steps were performed to implement the search operation. *Update* is a combination of search and disk write.
Source : Introduction to Algorithms, Third Edition – CLRS.

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3       i = i + 1
4  if i ≤ x.n and k == x.key_i
5       return (x, i)
6  elseif x.leaf
7       return NIL
8  else DISK-READ(x.c_i)
9       return B-TREE-SEARCH(x.c_i, k)
```

Insertion : The following steps were performed to implement the Insertion operation.
Source : Introduction to Algorithms, Third Edition – CLRS.

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t − 1
 3       s = ALLOCATE-NODE()
 4       T.root = s
 5       s.leaf = FALSE
 6       s.n = 0
 7       s.c_1 = r
 8       B-TREE-SPLIT-CHILD(s, 1)
 9       B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3       while i ≥ 1 and k < x.key_i
 4            x.key_{i+1} = x.key_i
 5            i = i − 1
 6       x.key_{i+1} = k
 7       x.n = x.n + 1
 8       DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10            i = i − 1
11       i = i + 1
12       DISK-READ(x.c_i)
13       if x.c_i.n == 2t − 1
14            B-TREE-SPLIT-CHILD(x, i)
15            if k > x.key_i
16                 i = i + 1
17       B-TREE-INSERT-NONFULL(x.c_i, k)
```

```
B-TREE-SPLIT-CHILD(x, i)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1
 6       z.key_j = y.key_{j+t}
 7  if not y.leaf
 8       for j = 1 to t
 9            z.c_j = y.c_{j+t}
10  y.n = t − 1
11  for j = x.n + 1 downto i + 1
12       x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15       x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

Learnings :

1. Trying to write structure pointers on file and read them, leading to segmentatation fault when we restart the program, fixed the issue by logically assigning block number to every node.
2. ***Strtok*** function for tokenizing the string (for parsing our SQL Statements).
3. Basics Shell scripting to trigger the program automatically and feed the input to program directly.
   - Command : ***cat testcases.txt | ./a*** .
4. Used ANSI Color codes to display colorful text in printf statements.

References :

1. Concepts of Btree : Intoduction to Algortihms, 3rd edition by CLRS.
2. Fixing bugs related to code and shell scipts : https://stackoverflow.com/
3. Writing and reading from file : https://www.tutorialspoint.com/cprogramming/c_file_io.htm
4. Printing text colorfully using printf statements : https://www.linuxjournal.com/article/8603