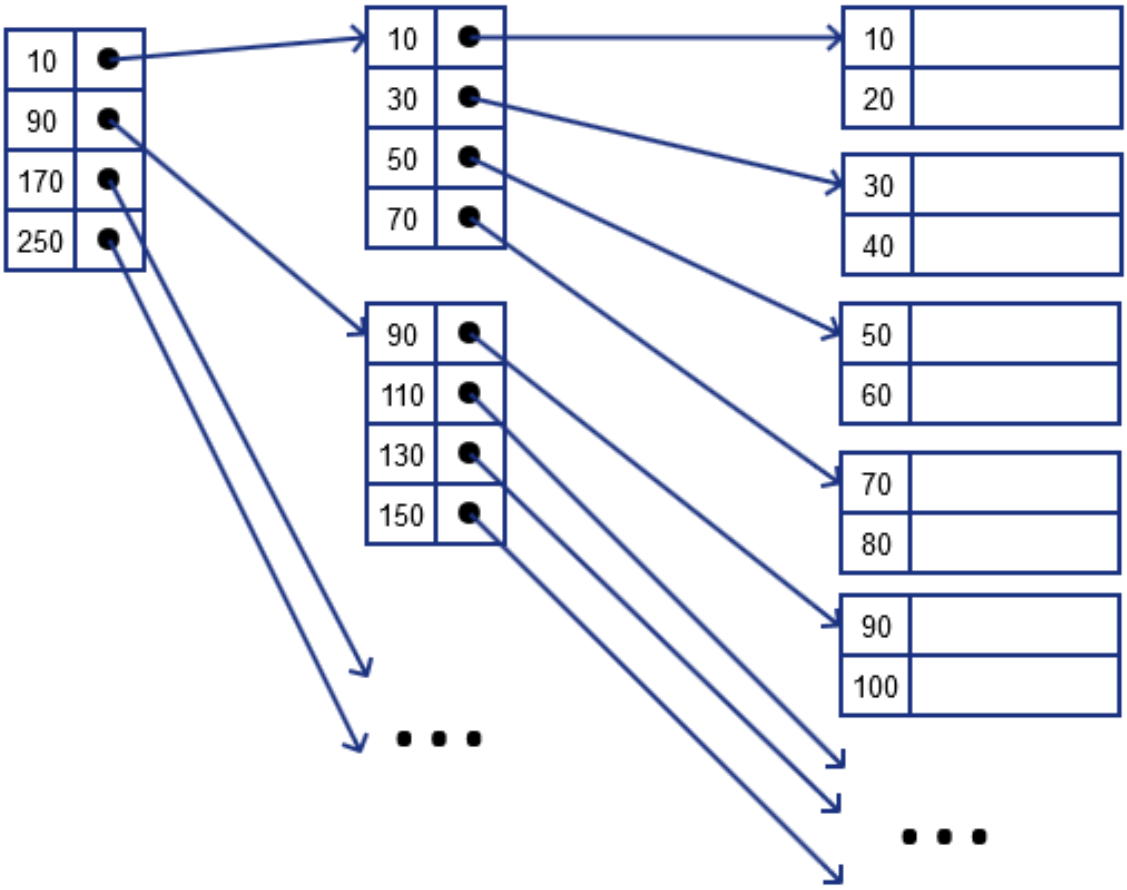
राष्ट्रीय प्रौद्योगिकी संस्थान वारंगल
National Institute of Technology Warangal
Department of Computer Science and Engineering

DBMS INDEXING

B+ TREE TO B TREE



Indexing in a Database Management System
B Tree and B+ Tree Structures
Open Source DBMS Software - MariaDB

CHINMAY JOSHI
197222 - SECTION B

DEEKSHITA TIRUMALA
197125 - SECTION A

NANDEPU INDIRA
197155 - SECTION A

Acknowledgement

Inspiration and motivation play a key role in the success of any venture. Successful completion of any project requires proper guidance and help. We express our gratitude to the following people who have made this possible.

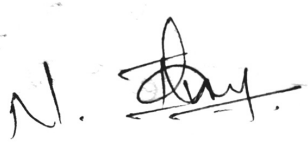
- We thank DBMS professors, **Dr. T Ramakrishnudu** and **Dr. RBV Subramanyam** for giving us the opportunity to delve deep into database concepts. Their lessons have been really helpful in learning the fundamental DBMS concepts like relations, E-R diagrams, Structured Query Language, functional dependencies, normalization, indexing, etc. in detail.
- We extend our thanks to the whole CSE Department for being so helpful and available whenever we required any kind of guidance. The DBMS Lab faculty has also trained and made us practice the Structured Query Language with the help of the periodic assignments on Google Classroom.
- The students of CSE batch (II-Year) have been very interactive and healthy class discussions are always a boon for students.
- We thank the NITW management, Dean - Academic and Director - NITW, for giving us this opportunity to discover this subject in the course, and enter new avenues in the field of database management.

We also thank our parents, elders and well-wishers for being there with us and giving us all kinds of technical and moral support.

Regards,

Chinmay Joshi
197222 (Section B)

Deekshita Tirumala
197125 (Section A)



Nandepu Indira
197155 (Section A)

Table of Contents

Topic	Page No.
Abstract	4
Indexing	5
An Overview of B Trees	8
An Overview of B+ Trees	11
Innobase Storage Structure of MariaDB Software	16
C++ Implementation of DBMS Indexing Essentials	20
Transforming B+ Tree Code To B Tree Code	36
Differences Between B Tree and B+ Tree Indexing	40
Conclusion	41

Abstract

A database index improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

In this project we will discuss the concept of Indexing, particularly regarding its two major structures, B Tree and B+ Tree. Then, we will analyse the source code of two of the most popular open source databases, MySQL and MariaDB. Later, referencing a simple database open source code, we will explore its indexing features and programs. We will modify the source code converting the index structure from B+ Tree to B tree and compare and contrast the performance of each.

A database index is a specialized data structure that allows us to locate information quickly.

It is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.

When we create an index on one or more columns, we store their values in a new structure. We also store pointers to rows the values come from. This organizes and sorts the information, but it doesn't change the information itself.

Indexes can be created using some database columns.

- The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

Indexing is defined based on its indexing attributes. Indexing can be of the following types −

· Primary Index − Primary index is defined on an ordered data file. The data file is ordered on a key field. The key field is generally the primary key of the relation.

· Secondary Index − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.

· Clustering Index − Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing is of two types −

- Dense Index
- Sparse Index

Dense Index

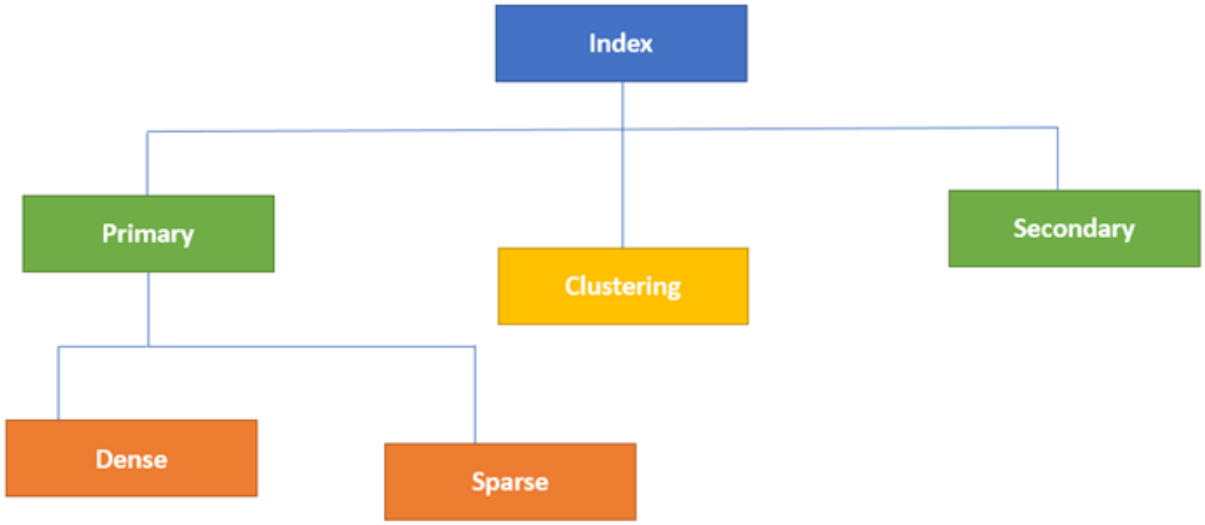
In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.

Sparse Index

In sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.

Before we start describing index types, let's have a quick review of the most common node types:

- Root node – the topmost node in the tree structure
- Child nodes – nodes that another node (the parent node) points to
- Parent nodes – nodes that point to other nodes (child nodes)
- Leaf nodes – nodes that have no child nodes (located on the bottom of the tree structure)
- Internal nodes – all "non-leaf" nodes, including the root node
- External nodes – another name for leaf nodes



In most cases, we can find data faster using an index than by searching sequentially through the database. The exception is if we only have a few records in our database. If we expressed this in a formula, with *t* = *time*, then

t using index < t for *sequential search*

We can calculate these values, and the result of that calculation is the formula for algorithm complexity.

Indexes are very powerful. They allow us to perform search and sort operations much faster. But that speed comes with a price: creating an index will require disk space and could slow down other operations. If a table has one or more indexes, it will definitely slow the INSERT operation. This is because when a record is added to the table, it has to go in the right place. Therefore, the index will need to be adjusted, which takes time.

An index is automatically created for the primary key. In the MySQL InnoDB engine, an index is also created automatically for foreign keys.

B-Tree is a self-balancing search tree Data structure. We use B-Trees when huge amount of data that cannot fit in main memory needs to be accessed.The data is read from disk in the form of blocks. Usage of B-Trees reduces the number of disk accesses. Most of the tree operations (search, insert, delete) require O(h) disk accesses where h is the height of the tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. And B-Tree's node is equal to the disk block size in general. Due to its low height total disk accesses for most of the operations are reduced significantly compared to Sequential Search.

B-tree stores data such that each node contains keys in ascending order. Each of these keys has two references to another two child nodes. The left side child node keys are less than the current keys and the right side child node keys are more than the current keys. If a single node has “n” number of keys, then it can have maximum “n+1” child nodes.
Record pointers will be present at leaf nodes as well as on internal nodes. Whereas in B+ tree we will have data (record pointers) only at leaf level.

Root Node :
Children : It can have children/pointers between 2 and N inclusive.
Records : No. of Records between 1 to p-1 inclusive.
Internal Node :
Children : It can have children/pointers between ceil(n/2) and n inclusive.
Records : No. of Records between ceil(p/2)-1 to p-1 inclusive.
Leaf Node :
Records : No. of Records between ceil(p/2)-1 to p-1 keys inclusive.

- Order of B-Tree for a given relation in DB depends on the record size and is given by :

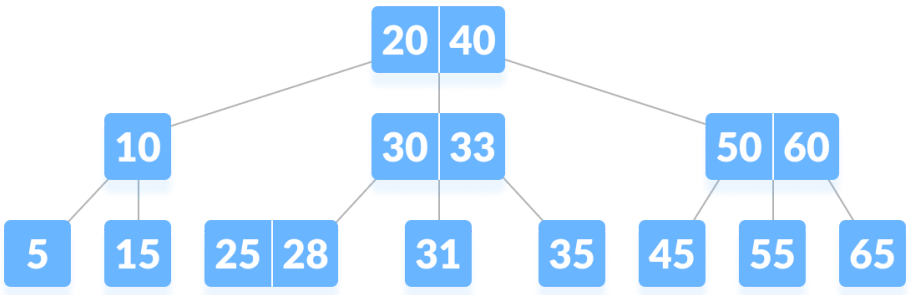
n*(Block pointer size) + (n-1)(Searchkey_size + record pointer size) <= Block Size

B-Tree Insertion :

- B-tree starts with a single root node (which is also a leaf node) at level 0.
- Once the root node is full with p – 1 records with search key values and when attempt to insert another entry in the tree,
  - The root node splits into two nodes at level 1.
  - Only the middle key value record is kept in the root node, and the rest of the records are split evenly between the two nodes in Level 1.
- When a non root node is full and a new entry is inserted into it,
  - Node is split into two nodes at the same level.
  - The middle entry is moved to the parent node along with two pointers to the new split nodes.
- If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.
- In this way every split in the Root node leads to a new level.

B-Tree Search:

- For retrieving a record from the table in DB, Consider the key of the record to be searched is k.
  - The search starts from the Root , Checks for presence of record in root node records. If present return the record
  - If not present Move till we get to a record with key value > k and then to to the left child pointer and repeat the same



B-Tree Deletion :

- If the key k of the record is in leaf node, delete the key k's record from node.
- If the key k of the record is in node is an internal node x
    - If the child y that precedes k in node x has at least t records, then find the predecessor k0 of k in the subtree rooted at y. Recursively delete k0's record, and replace k by k0's in x.
    - If y has fewer than t Record keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t record keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete the record with key k0, and replace k by k0's record in x.
    - Otherwise, if both y and z have only t-1 record keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 record keys. Then free z and recursively delete k from y.
- If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.
    - If x.c(i) has only t-1 records but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a record from x down into x.c(i), moving a record from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).
    - If x.c(i) and both of x.c(i)'s immediate siblings have t-1 records, merge x.c(i) with one sibling, which involves moving a record from x down into the new merged node to become the median record key for that node.
- This way if deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. And can reduce the number of tree levels.

A B+-tree of order m is a tree where each internal node contains up to m branches (children nodes) and thus store up to m-1 search key values -- in a BST, only one key value is needed since there are just two children nodes that an internal node can have. m is also known as the branching factor or the fanout of the tree.

1. The B+-tree stores records (or pointers to actual records) only at the leaf nodes, which are all found at the same level in the tree, so the tree is always height balanced.
2. All internal nodes, except the root, have between Ceiling(m/2) and m children.
3. The root is either a leaf or has at least two children.
4. Internal nodes store search key values, and are used only as placeholders to guide the search. The number of search key values in each internal node is one less than the number of its non-empty children, and these keys partition the keys in the children in the fashion of a search tree. The keys are stored in non-decreasing order (i.e. sorted in lexicographical order).
5. Depending on the size of a record as compared to the size of a key, a leaf node in a B+-tree of order m may store more or less than m records. Typically this is based on the size of a disk block, the size of a record pointer, etc. The leaf pages must store enough records to remain at least half full.
6. The leaf nodes of a B+-tree are linked together to form a linked list. This is done so that the records can be retrieved sequentially without accessing the B+-tree index. This also supports fast processing of range-search queries as will be described later.

To understand the B+-tree operations more clearly, assume, without loss of generality, that there is a table whose primary is a single attribute and that it has a B+-tree index organized on the PK attribute of the table.

Searching for records that satisfy a simple condition

To retrieve records, queries are written with conditions that describe the values that the desired records are to have. The most basic search on a table to retrieve a single record given its PK value K.

Search in a B+-tree is an alternating two-step process, beginning with the root node of the B+-tree. Say that the search is for the record with key value K -- there can only be one record because we assume that the index is built on the PK attribute of the table.

1. Perform a binary search on the search key values in the current node -- recall that the search key values in a node are sorted and that the search starts with the root of the tree. We want to find the key Ki such that $Ki \leq K < Ki+1$.
2. If the current node is an internal node, follow the proper branch associated with the key Ki by loading the disk page corresponding to the node and repeat the search process at that node.
3. If the current node is a leaf, then:
    a. If K=Ki, then the record exists in the table and we can return the record associated with Ki.
    b. Otherwise, K is not found among the search key values at the leaf, we report that there is no record in the table with the value K.

Inserting into a B+ tree

Insertion in a B+-tree is similar to inserting into other search trees, a new record is always inserted at one of the leaf nodes. The complexity added is that insertion could overflow a leaf node that is already full. When such overflow situations occur a brand new leaf node is added to the B+-tree at the same level as the other leaf nodes. The steps to insert into a B+-tree are:

1. Follow the path that is traversed as if a Search is being performed on the key of the new record to be inserted.
2. The leaf page L that is reached is the node where the new record is to be indexed.
3. If L is not full then an index entry is created that includes the search key value of the new row and a reference to where new row is in the data file. We are done; this is the easy case!
4. If L is full, then a new leaf node Lnew is introduced to the B+-tree as a right sibling of L. The keys in L along with the an index entry for the new record are distributed evenly among L and Lnew. Lnew is inserted in the linked list of leaf nodes just to the right of L. We must now link Lnew to the tree and since Lnew is to be a sibling of L, it will then be pointed to by the parent of L. The smallest key value of Lnew is copied and inserted into the parent of L -- which will also be the parent of Lnew. This entire step is known as commonly referred to as a split of a leaf node.
    a. If the parent P of L is full, then it is split in turn. However, this split of an internal node is a bit different. The search key values of P and the new inserted key must still be distributed evenly among P and the new page introduced as a sibling of P. In this split, however, the middle key is moved to the node above -- note, that unlike splitting a leaf node where the middle key is copied and inserted into the parent, when you split an internal node the middle key is removed from the node being split and inserted into the parent node. This splitting of nodes may continue upwards on the tree.
    b. When a key is added to a full root, then the root splits into two and the middle key is promoted to become the new root. This is the only way for a B+-tree to increase in height -- when split cascades the entire height of the tree from the leaf to the root.

When a key is added to a full root, then the root splits into two and the middle key is promoted to become the new root. This is the only way for a B+-tree to increase in height -- when split cascades the entire height of the tree from the leaf to the root.

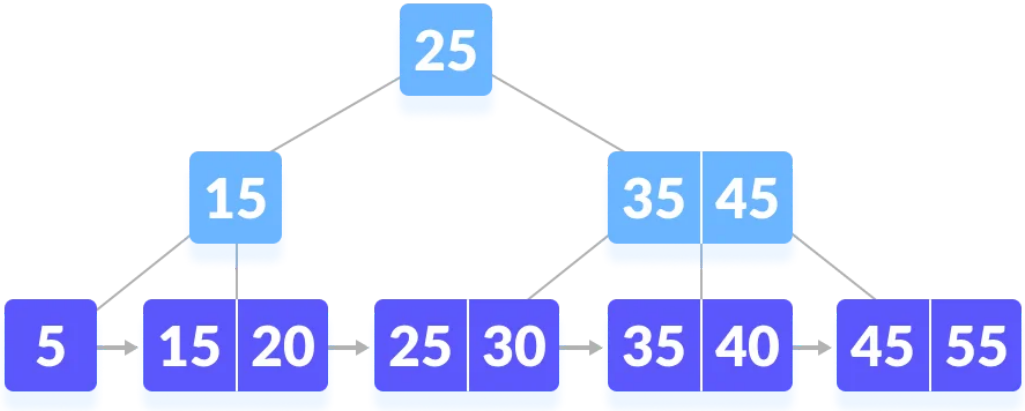
Deletion

Deletion from a B+-tree again needs to be sure to maintain the property that all nodes must be at least half full. The complexity added is that deletion could underflow a leaf node that has only the minimum number of entries allowed. When such underflow situations take place, adjacent sibling nodes are examined. Otherwise, if both adjacent sibling nodes are also at their minimum, then two of these nodes are merged into a single node. The steps to delete from a B+-tree are:

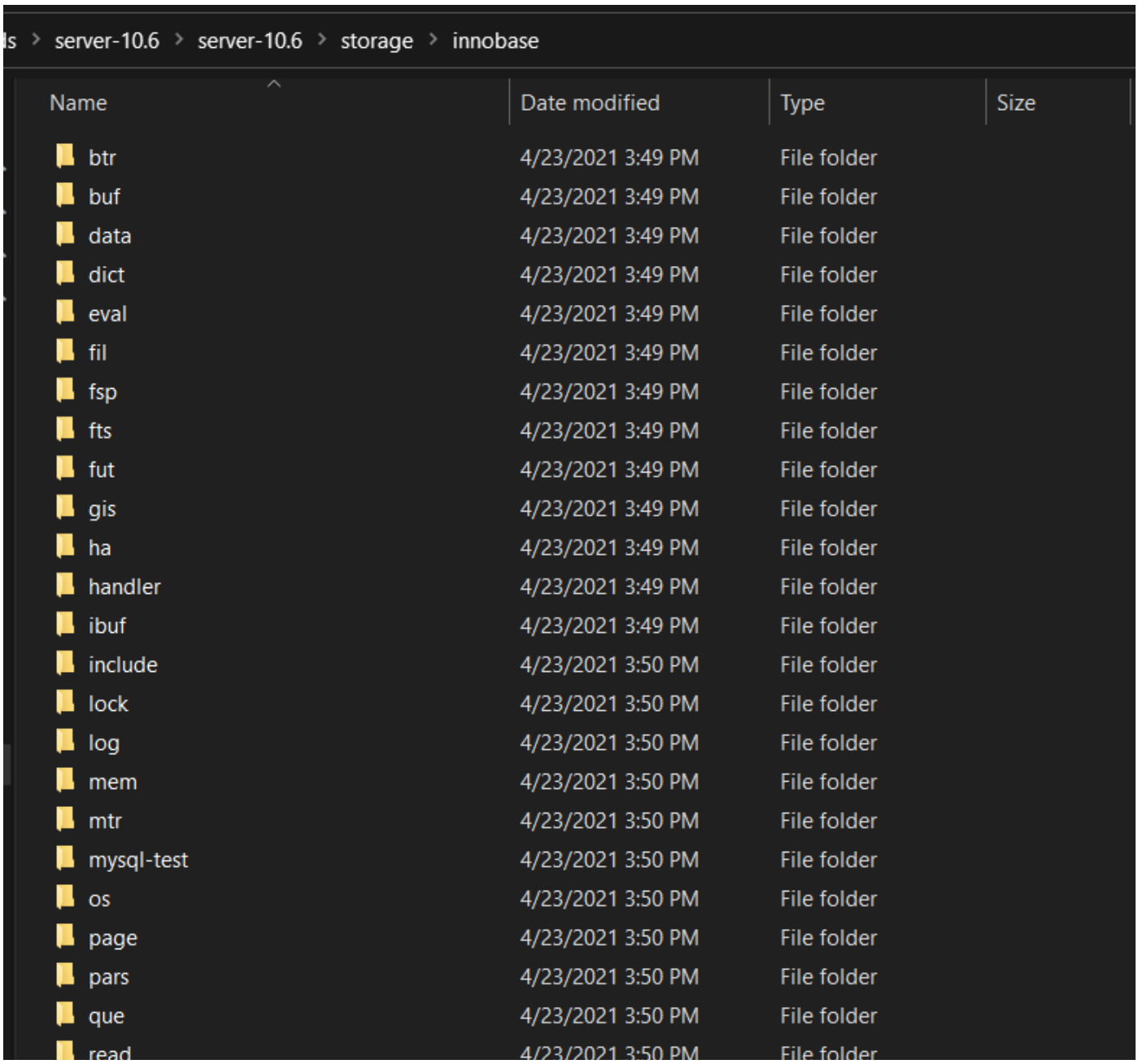
1. Perform the search process on the key of the record to be deleted. This search will end at a leaf L. If the leaf L contains more than the minimum number of elements (more than m/2 - 1), then the index entry for the record to be removed can be safely deleted from the leaf with no further action.
2. If the leaf contains the minimum number of entries, then the deleted entry is replaced with another entry that can take its place while maintaining the correct order. To find such entries, we inspect the two sibling leaf nodes Lleft and Lright adjacent to L -- at most one of these may not exist.
    a. If one of these leaf nodes has more than the minimum number of entries, then enough records are transferred from this sibling so that both nodes have the same number of records. This is a heuristic and is done to delay a future underflow as long as possible; otherwise, only one entry need be transferred. The placeholder key value of the parent node may need to be revised.
    b. If both Lleft and Lright have only the minimum number of entries, then L gives its records to one of its siblings and it is removed from the tree. The new leaf will contain no more than the maximum number of entries allowed. This merge process combines two subtrees of the parent, so the separating entry at the parent needs to be removed -- this may in turn cause the parent node to underflow; such an underflow is handled the same way that an underflow of a leaf node.
    c. If the last two children of the root merge together into one node, then this merged node becomes the new root and the tree loses a level.

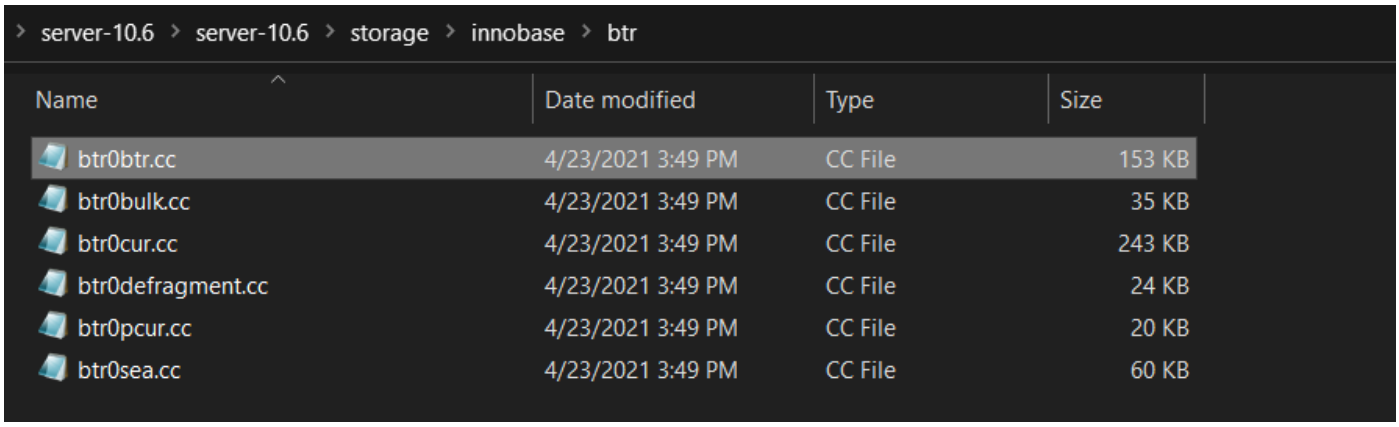
Range Search

It is common to request records from a table that fall in a specified range. For example, we may want to retrieve all employees making salaries between $50,000 and $60,000. B+-trees are exceptionally good for range queries. Once the first record in the range has been found using the search algorithm described above, the rest of the records in the range can be found by sequential processing the remaining records in the leaf node, and then continuing down (actually right of the current leaf node) the linked list of leaf nodes as far as necessary. Once a record is found that has a search key value above the upper bound of the requested range, then the search completes. Note that to use a B+-tree index to retrieve the employees described above, the B+-tree index must exist that has been organized using the salary attribute of the employees table.



We choose the btr0btr.cc file in the btr folder of the innobase structure in MariaDB.





The leaf pages of the B-tree are at level 0. The parent of a page at level L has level L+1. (The level of the root page is equal to the tree height.) The B-tree lock (index->lock) is the parent of the root page and has a level = tree height + 1.

Node pointers

Leaf pages of a B-tree contain the index records stored in the tree. On levels n > 0 we store 'node pointers' to pages on level n - 1. For each page there is exactly one node pointer stored: thus the our tree is an ordinary B-tree, not a B-link tree.

A node pointer contains a prefix P of an index record. The prefix is long enough so that it determines an index record uniquely. The file page number of the child page is added as the last field. To the child page we can store node pointers or index records which are >= P in the alphabetical order, but < P1 if there is a next node pointer on the level, and P1 is its prefix.

If a node pointer with a prefix P points to a non-leaf child, then the leftmost record in the child must have the same prefix P. If it points to a leaf node, the child is not required to contain any record with a prefix equal to P. The leaf case is decided this way to allow arbitrary deletions in a leaf node without touching upper levels of the tree.

We have predefined a special minimum record which we define as the smallest record in any alphabetical order. A minimum record is denoted by setting a bit in the record header. A minimum record acts as the prefix of a node pointer which points to a leftmost node on any level of the tree.

File page allocation

In the root node of a B-tree there are two file segment headers. The leaf pages of a tree are allocated from one file segment, to make them consecutive on disk if possible. From the other file segment we allocate pages for the non-leaf levels of the tree.

Inserts a data tuple to a tree on a non-leaf level. It is assumed that mtr holds an x-latch on the tree.

```
void btr_insert_on_non_leaf_level
(ulint  flags, dict_index_t*  index, ulint level, dtuple_t* tuple,  mtr_t* mtr)
{
  big_rec_t* dummy_big_rec;
  btr_cur_t cursor;
  dberr_t err;
  rec_t*  rec;
  mem_heap_t* heap = NULL;
  rec_offs  offsets_[REC_OFFS_NORMAL_SIZE];
  rec_offs* offsets = offsets_;
  rec_offs_init(offsets_);
  rtr_info_t rtr_info;

  ut_ad(level > 0);

  if (!dict_index_is_spatial(index))
  {
    dberr_t err = btr_cur_search_to_nth_level(index, level, tuple, PAGE_CUR_LE,
    BTR_CONT_MODIFY_TREE, &cursor, 0, mtr);
    if (err != DB_SUCCESS) {
      ib::warn() << " Error code: " << err
                 << " btr_page_get_father_node_ptr_func "
                 << " level: " << level
                 << " table: " << index->table->name
                 << " index: " << index->name;
    }
  } else {
    /* For spatial index, initialize structures to track
    its parents etc. */
    rtr_init_rtr_info(&rtr_info, false, &cursor, index, false);

    rtr_info_update_btr(&cursor, &rtr_info);

    btr_cur_search_to_nth_level(index, level, tuple,
                PAGE_CUR_RTREE_INSERT,
                BTR_CONT_MODIFY_TREE,
                &cursor, 0, mtr);
  }

  ut_ad(cursor.flag == BTR_CUR_BINARY);

  err = btr_cur_optimistic_insert(
    flags
    | BTR_NO_LOCKING_FLAG
    | BTR_KEEP_SYS_FLAG
    | BTR_NO_UNDO_LOG_FLAG,
```

```
    &cursor, &offsets, &heap, tuple, &rec, &dummy_big_rec, 0, NULL, mtr);
  if (err == DB_FAIL) {
    err = btr_cur_pessimistic_insert(flags
                                     | BTR_NO_LOCKING_FLAG
                                     | BTR_KEEP_SYS_FLAG
                                     | BTR_NO_UNDO_LOG_FLAG,
                                     &cursor, &offsets, &heap,
                                     tuple, &rec,
                                     &dummy_big_rec, 0, NULL, mtr);
    ut_a(err == DB_SUCCESS);
  }
  if (heap != NULL) {
    mem_heap_free(heap);
  }
  if (dict_index_is_spatial(index)) {
    ut_ad(cursor.rtr_info);
    rtr_clean_rtr_info(&rtr_info, true);
  }
}
```

1. B+ Tree Code

```
#include "BPtree.h"
class Btreenode
{
private:
  bool leaf;
  std::vector <int> pointers;
  std::vector <int> keys;
  int next_node;
public:
  Btreenode();
  Btreenode(bool makeleaf)
  {
    leaf = makeleaf;
    next_node = -1;
  }
  bool isleaf()
  {
    return leaf;
  }
  void set_leaf(bool val)
  {
    leaf = val;
  }
  int num_keys()
  {
    return keys.size();
  }
  int num_pointers()
  {
    return pointers.size();
  }
  int get_key(int i)
  {
    return keys[i - 1];
  }
  int get_pointer(int i)
  {
    return pointers[i - 1];
  }
  void set_next(int x)
  {
    next_node = x;
  }
  int get_next()
  {
    return next_node;
  }
```

```
void push_key(int val)
{
  keys.push_back(val);
}
void push_pointer(int val)
{
  pointers.push_back(val);
}
void clear_data()
{
  keys.clear();
  pointers.clear();
}
int* get_keys_add()
{
  return &keys[0];
}
int* get_point_add()
{
  return &pointers[0];
}
~Btreenode()
{
  clear_data();
}
/* Function that performs binary search and returns the
correct child. For Internal Nodes */
/*
  std::lower_bound()
  Returns an iterator pointing to the first element in
  the range [first,last) which does not compare less than val.
  If all the element in the range compare less than val, the
  function returns last.
*/
int get_next_key(int search_key)
{
  return (std::lower_bound(keys.begin(), keys.end(),
  search_key) - keys.begin());
}
/* Function that performs binary search and returns
boolean result */
int search_key(int search_key)
{
  return std::binary_search(keys.begin(), keys.end(), search_key);
}
bool full()
{
  if(leaf)
  {
    if (pointers.size() < BPTREE_MAX_KEYS_PER_NODE - 1)
      return false;
    else
      return true;
  }
  if(pointers.size() < BPTREE_MAX_KEYS_PER_NODE)
    return false;
  else
    return true;
}
```

```
/* Function that inserts a new record in a node which
is not yet full */
void insert_key(int key, int point)
{
  //get the position in vector keys to insert
  //new key so that keys[] remains sorted;
  int pos = get_next_key(key);
  keys.insert(keys.begin() + pos, key);
  //insert position of key inside pointers;
  if(leaf)
    pointers.insert(pointers.begin() + pos, point);
  else
    pointers.insert(pointers.begin() + pos + 1, point);
}
/* Function that copies first 'n' records from a given
node to the current node */
void copy_first(Btreenode & node, int n)
{
  keys.clear();
  pointers.clear();
  for(int i = 1; i < n; i++)
  {
    keys.push_back(node.get_key(i));
    pointers.push_back(node.get_pointer(i));
  }
  pointers.push_back(node.get_pointer(n));
  if(leaf)
  {
    keys.push_back(node.get_key(n));
  }
}
/* Function that copies records from a given node to
current node. It ignores first 'n' records */
void copy_last(Btreenode & node, int n)
{
  keys.clear();
  pointers.clear();
  int lim = node.num_pointers();
  for (int i = n + 1; i <= lim; i++)
  {
    pointers.push_back(node.get_pointer(i));
  }
  lim = node.num_keys();
  for (int i = n + 1; i <= lim; i++)
  {
    keys.push_back(node.get_key(i));
  }
}
// Overloading write operator to write to a file
/* write to file in format
  (is_leaf_or_not,keys_size,for(0 to key_size)
  <keys_Values>,pointer_size,for(0to pointers_size)
  <pointer_values>, next_node);
  according to sample program
```

```
    tree0.dat will store data in following format
    (1 8 1 3 5 6 7 8 9 12 8 2 4 1 7 3 0 6 5 -1);
  */
  friend std::ofstream & operator<<(std::ofstream & os, const Btreenode & en)
  {
    // is leaf or not;
    os << en.leaf << " ";
    // keys size
    os << (int) en.keys.size() << " ";
    for(unsigned int i = 0; i < en.keys.size(); i++)
    {
      os << en.keys[i] << " ";
    }
    os << (int) en.pointers.size() << " ";
    // pointers are the indices of the stored elements in vector a;
    for(unsigned int i = 0; i < en.pointers.size(); i++)
    {
      os << en.pointers[i] << " ";
    }
    os << en.next_node;
    return os;
  }
  //Overloading read operator to read from a fil
  friend std::ifstream & operator>>(std::ifstream & is, Btreenode & en)
  {
    // read all the data stored in file <descrbed in << operator overloading>
    int ts;
    is >> en.leaf;
    is >> ts;
    en.keys.resize(ts);
    for (int i = 0; i < ts; i++)
    {
      is >> en.keys[i];
    }
    is >> ts;
    en.pointers.resize(ts);
    for(int i = 0; i < ts; i++)
    {
      is >> en.pointers[i];
    }
    is >> en.next_node;
    return is;
  }
  //Overloading Assignment Operator
  Btreenode & operator=(const Btreenode & n)
  {
    if (this != &n)
    {
      leaf = n.leaf;
      keys.assign(n.keys.begin(), n.keys.end());
      pointers.assign(n.pointers.begin(), n.pointers.end());
    }
    return *this;
  }
};
/* Function that writes a node 'n' to a given
filenum(like pointer) */
```

```
void BPtree :: write_node(int filenum, Btreenode n)
{
  char *str;
  str = (char *) malloc(sizeof(char) * BPTREE_MAX_FILE_PATH_SIZE);
  // if filenum=0; open tree0.dat;
  sprintf(str, "table/%s/tree/tree%d.dat", tablename,filenum);
  std::ofstream out_file(str, std::ofstream::binary |  std::ofstream::out | std::ofstream::trunc);
  free(str);
  // write data to out_file root = n()
  out_file << n;
  out_file.close();
}
/* Function that updates Meta-Data of a table after
inserting a record */
void BPtree :: update_meta_data()
{
  char *str;
  str = (char *) malloc(sizeof(char) * BPTREE_MAX_FILE_PATH_SIZE);
  sprintf(str, "table/%s/tree/meta_tree.dat", tablename);
  std::ofstream out_file(str, std::ofstream::out | std::ofstream::trunc
  | std::ofstream::binary );
  out_file << files_till_now << " " << root_num;
  out_file.close();
  free(str);
}
/* Constructor function that initializes tree with existing
meta-data or creates new tree and writes meta-data */
BPtree :: BPtree(char table_name[])
{
  strcpy(tablename,table_name);
  char *str;
  str = (char *) malloc(sizeof(char) * BPTREE_MAX_FILE_PATH_SIZE);
  sprintf(str, "mkdir -p table/%s/tree", tablename);
  // cout<<"str"<<str<<endl;
  system(str);
  sprintf(str, "table/%s/tree/meta_tree.dat", tablename);
  std::ifstream in_file(str, std::ifstream::in | std::ifstream::binary);
  if (!in_file)
  {
    /* If no meta data for table's BTree is found i.e.,
    It is new table. Then Create new meta data */
    std::ofstream out_file(str, std::ofstream::binary |
    std::ofstream::out | std::ofstream::trunc);
    if (!out_file)
    {
      printf("Critical Error : Unable to Write on Disk !!");
      printf("\nAborting ... ... \n");
      abort();
    }
    files_till_now = root_num = 0;
    out_file.write((char *) (&files_till_now), sizeof(files_till_now));
    out_file.write((char *) (&root_num), sizeof(root_num));
    out_file.close();
    //initialize with root node = leaf node;
    Btreenode root(true);
    //set next_node=-1; as it is root;
    root.set_next(-1);
```

```
    //write node data to outfile;
    write_node(0, root);
  }
  //if file already created read the previously stored data;
  else
  {
    /* Read old Meta Data */
    in_file >> files_till_now >> root_num;
    in_file.close();
  }
  free(str);
}
/* Function that reads a node 'n' from a given
filenum(like pointer) */
void BPtree :: read_node(int filenum, Btreenode & n)
{
  char *str;
  str = (char *) malloc(sizeof(char) * BPTREE_MAX_FILE_PATH_SIZE);
  sprintf(str, "table/%s/tree/tree%d.dat", tablename, filenum);
  std::ifstream in_file(str, std::ifstream::in | std::ifstream::binary);
  free(str);
  in_file >> n;
  in_file.close();
}
/* Function that traverses the BPtree and returns the leaf node
where 'primary_key' record should exist, if it exist at all */
Btreenode BPtree::search_leaf(int primary_key)
{
  Btreenode n(true);
  int q, curr_node = root_num;
  read_node(curr_node, n);
  //Traversing the Tree from root till leaf
  while (!n.isleaf())
  {
    q = n.num_pointers();
    if (primary_key <= n.get_key(1))
    {
      //set curr_node =pointers[0];
      curr_node = n.get_pointer(1);
    }
    else if (primary_key > n.get_key(q - 1))
    {
      //set curr_node = pointers[q-1];
      curr_node = n.get_pointer(q);
    }
    else
    {
      //find the correct position of key to be stored;
      curr_node = n.get_pointer(n.get_next_key(primary_key) + 1);
    }
    read_node(curr_node, n);
  }
  return n;
}
/* A function that returns the record number of a tuple
with indexed column = 'primary_key' */
```

```
int BPtree::get_record(int primary_key)
{
  clock_t start=clock();
  Btreenode n = search_leaf(primary_key);
  int pos = n.get_next_key(primary_key) + 1;
  clock_t stop=clock();
  double elapsed=(double)(stop-start)*1000.0/CLOCKS_PER_SEC;
  printf("\nTime elapsed for search is %f ms\n",elapsed);
  if (pos <= n.num_keys() && n.get_key(pos) == primary_key)
  {
    return n.get_pointer(pos);
  }
  else
  {
    return BPTREE_SEARCH_NOT_FOUND;
  }
}
/* A function the inserts a (key, record_num) pair in the
B+ Tree */
// key is first column of database either can be int or varchar;
int BPtree::insert_record(int primary_key, int record_num)
{
  //printf("pri %d\n record_num %d",primary_key,record_num);
  // Btreenode n= leaf=true, next_node =-1;
  Btreenode n(true);
  int q, j, prop_n, prop_k, prop_new, curr_node = root_num;
  bool finish = false;
  std::stack < std::pair<int, Btreenode> > S;
  // read all the data of node stored in file tree%d.data (%d==curr_node=root_num=file_no);
  // now n contains all the previously stored data;
  read_node(curr_node, n);
  // Traverse the tree till we get the leaf node;
  while (!n.isleaf())
  {
    S.push(make_pair(curr_node,n));
    // Storing address in case of split
    // num_pointers==function that returns size of pointers vector from the block file;
    q = n.num_pointers();
    if (primary_key <= n.get_key(1))
    {
      curr_node = n.get_pointer(1);
    }
    else if (primary_key > n.get_key(q - 1))
    {
      curr_node = n.get_pointer(q);
    }
    else
    {
      curr_node = n.get_pointer(n.get_next_key(primary_key) + 1);
    }
    // get all the data of node n from file tree%d.dat(%d==curr_node);
    read_node(curr_node, n);
  }
```

```
// Here n is Leaf Node
// if key already exists then return ERROR
if (n.search_key(primary_key))
{
  return BPTREE_INSERT_ERROR_EXIST;
}
/* if n is not full, insert key and pointer to node
write back node to file, update meta_data and return */
if (!n.full())
{
  n.insert_key(primary_key, record_num);
  write_node(curr_node, n);
  update_meta_data();
  return BPTREE_INSERT_SUCCESS;
}
//if node n is full, then split;
Btreenode temp(true), new_node(true);
temp = n;
temp.insert_key(primary_key, record_num);
j = ceil((BPTREE_MAX_KEYS_PER_NODE + 1.0) / 2.0);
// if max_key_per_node = 4, j = 3
// copy the first j values of temp to node n
// and remaining to new_node
n.copy_first(temp, j);
//now one file is increased to store the new node;
files_till_now++;
// next pointer of new_node will be next pointer of n
// and next pointer of n will be newly created node new_node
new_node.set_next(n.get_next());
n.set_next(files_till_now);
new_node.copy_last(temp, j);
/*
  prop_k is key to be inserted into root
  and prop_new and prop_n are pointers to
  be attached to this root value
*/
prop_k = temp.get_key(j); // key to be moved to new root
prop_new = files_till_now;
prop_n = curr_node;
// write back the two new nodes created
write_node(files_till_now, new_node);
write_node(curr_node, n);
temp.clear_data();
new_node.clear_data();
/* Keep repeating until we reach root
or find an empty internal node */
while (!finish)
{
  if (S.size() == 0)
  {
    /*Last element splitted was root
    so create new root and assign meta_data */
    Btreenode nn(false);
    // insert key to new root
```

```
      nn.push_key(prop_k);
      // insert two pointer associated to this key
      // for left and right
      nn.push_pointer(prop_n);
      nn.push_pointer(prop_new);
      files_till_now++;
      write_node(files_till_now, nn);
      root_num = files_till_now;
      finish = true;
    }
    else
    {
      std::pair<int, Btreenode> p = S.top();
      curr_node = p.first;
      n = p.second;
      S.pop();
      // read_node(curr_node, n);
      if (!n.full())
      {
        n.insert_key(prop_k, prop_new);
        write_node(curr_node, n);
        finish = true;
      }
      else
      {
        /* Split is propogating towards top */
        temp = n;
        temp.insert_key(prop_k, prop_new);
        j = floor((BPTREE_MAX_KEYS_PER_NODE + 1.0) / 2.0);
        n.copy_first(temp, j);
        files_till_now++;
        new_node.set_leaf(false);
        new_node.copy_last(temp, j);
        write_node(files_till_now, new_node);
        write_node(curr_node, n);
        prop_k = temp.get_key(j);
        prop_new = files_till_now;
        prop_n = curr_node;
      }
    }
  }
  update_meta_data();
  return BPTREE_INSERT_SUCCESS;
}
```

2. Insert.cpp

```
#include "insert.h"
#include "file_handler.h"
#include "BPtree.h"
#include "aes.h"

int search_table(char tab_name[])
{
  /* Check if new table already exists in table list or not
    use grep to search table_name string inside table_list
    -F -> --fixed-strings ;intepret pattern as a list of fixed strings
    -x -> --line-regexp ;select only those matches that exactly match the whole line
    -q -> quite, --silent ;write anything to standard output,
    exit immediately with zero status if any match is found */
  char str[MAX_NAME+1];
  strcpy(str,"grep -Fxq ");
  strcat(str,tab_name);
  strcat(str," ./table/table_list");
  int x = system(str);
  if(x == 0) return 1;
  else return 0;
  return 0;
}
void insert_command(char tname[], void *data[], int total)
{
  table *temp;
  int ret;
  BPtree obj(tname);

  //open meta data
  FILE *fp = open_file(tname, const_cast<char*>("r"));
  temp = (table*)malloc(sizeof(table));
  fread(temp, sizeof(table), 1, fp);

  //insert into table and write to btree file nodes
  ret = obj.insert_record(*((int *)data[0]), temp->rec_count);
  if(ret == 2)
  {
    std::cout << "\nkey already exists\n";
    std::cout << "\nexiting...\n";
    return ;
  }

  //if no error occurred during insertion of key
  //update the meta data;
  fp = open_file(tname, const_cast<char*>("w+"));
  int file_num = temp->rec_count;
  temp->rec_count = temp->rec_count + 1;
  temp->data_size = total;
  fwrite(temp, sizeof(table), 1, fp);
  fclose(fp);

  //update the particular entry of inserted data to file;
  char *str;
  str=(char *)malloc(sizeof(char)*MAX_PATH);
  sprintf(str, "table/%s/file%d.dat", tname, file_num);
```

```
    //std::cout<<str<<endl;
    FILE *fpr = fopen(str, "w+");
    int x;
    char y[MAX_NAME];
    for(int j = 0; j < temp->count; j++)
    {
      if(temp->col[j].type == INT)
      {
        x = *(int *)data[j];
        fwrite(&x, sizeof(int), 1, fpr);
      }
      else if(temp->col[j].type == VARCHAR)
      {
        strcpy(y, (char *)data[j]);
        fwrite(y, sizeof(char)*MAX_NAME, 1, fpr);
      }
    }
    fclose(fpr);
    free(str);
    free(temp);
}
void insert()
{
    char *tab;
    tab = (char*)malloc(sizeof(char)*MAX_PATH+1);
    std::cout << "enter table name: ";
    std::cin >> tab;
    int check = search_table(tab);
    if(check == 0)
    {
      printf("\nTable \" %s \" don't exist in database\n",tab);
      return ;
    }
    else
    {
      std::cout << "\nTable exists, enter data\n\n";
      table inp1;
      int count;

      //read column details from file;
      FILE *fp = open_file(tab, const_cast<char*>("r"));
      int i = 0;
      while(fread(&inp1, sizeof(table), 1, fp))
      {
        printf("-------------------------------------\n");
        std::cout << "\ninsert the following details ::\n";
        printf("\n-------------------------------------\n");
        count = inp1.count;
        for(i = 0; i < inp1.count; i++)
        {
          std::cout << inp1.col[i].col_name << "(" << inp1.col[i].type << "),size:" << inp1.col[i].size;
          std::cout << "\t";
        }
      }
      printf("\n-------------------------------------\n");
```

```
//enter data;
char var[MAX_NAME+1];
void * data[MAX_ATTR];
//void *data1[MAX_ATTR];

//input data for the table of desired datatype 1.int 2.varchar;
int size = 0;
int total = 0;
for(int i = 0; i < count; i++)
{
  if(inp1.col[i].type == INT)
  {
    data[i] = (int*) malloc(sizeof(int));
    total += sizeof(int);
    std::string inp_int;
    std::cin >> inp_int;
    if(inp_int.length() > (unsigned)inp1.col[i].size)
    {
      printf("\nwrong input, size <= %d\nexiting...\n",inp1.col[i].size);
      return;
    }
    else
    {
      //verify if entered input is integer and not a string;
      int num = 0;
      int factor_10 = 1;
      for(int j = inp_int.length()-1; j >= 0; j--)
      {
        if(inp_int[j] < 48 || inp_int[j] > 57)
        {
          printf("\nwrong input, input should be integer\nexiting...\n");
      return;
    }
    else
    {
      //verify if entered input is integer and not a string;
      int num = 0;
      int factor_10 = 1;
      for(int j = inp_int.length()-1; j >= 0; j--)
      {
        if(inp_int[j] < 48 || inp_int[j] > 57)
        {
          printf("\nwrong input, input should be integer\nexiting...\n");
          return;
        }
        else
        {
          num += (inp_int[j] - 48) * factor_10;
          factor_10 = factor_10 * 10;
        }
      }
      *((int*)data[i]) = num;
    }
    size++;
  }
```

```
        else if(inp1.col[i].type == VARCHAR)
        {
          //cout<<"inside varchar\n";
          data[i] = malloc(sizeof(char) * (MAX_NAME + 1));
          int flag = 1;
          while(flag)
          {
            std::cin >> var;
            total += sizeof(char) * (MAX_NAME + 1);
            if(strlen(var) > (unsigned int)inp1.col[i].size)
            {
              std::cout << "\nERROR\nEntered size of string is greater than specified \n";
            }
            else flag = 0;
          }
          strcpy((char*)(data[i]), var);
          //cout<<(char *)data[1]<<endl;
          size++;
        }
      }
      insert_command(tab, data, total);
    }
    free(tab);
}
```

3. Search.cpp

```
#include "search.h"
#include "file_handler.h"
#include "BPtree.h"

void search()
{
  //Search for particular table or any specific entry inside table
  cout << "1.search table\n2.search particular entry in table\n\n";
  int ch;
  cin >> ch;
  char *tab;
  tab = (char*)malloc(sizeof(char)*MAX_NAME);
  if(ch==1)
  {
    cout<<"enter table name: ";
    cin >> tab;
    //check if table exists
    int check=search_table(tab);
    if(check==1) printf("\n%s exists!!!\n",tab);
    else printf("\n%s doesn't exist in table entry\n\n",tab);
  }
```

```
else if(ch==2)
{
  cout<<"enter table name: ";
  cin>>tab;
  int check=search_table(tab);
  if(check==1)
  {
    printf("%s exists!!!\n\nEnter key to search\n\n",tab);
    //open %s meta data file and display column details;
    table inp1;
    FILE *fp = open_file(tab, const_cast<char*>("r"));
    int i=0;
    while(fread(&inp1,sizeof(table),1,fp))
    {
      for(i=0;i<inp1.count;i++)
      {
        cout<<inp1.col[i].col_name<<"("<<inp1.col[i].type<<"),size:"<<inp1.col[i].size;
        cout<<"\t";
      }
    }
    int pri_int;
    int c;
    char d[MAX_NAME];
    char pri_char[MAX_NAME];
    BPtree mytree(tab);
    int ret=0;
    if(inp1.col[0].type==INT)
    {
      cout<<"\nenter key[col 0] to search\n";
      cin>>pri_int;
      ret = mytree.get_record(pri_int);
      if (ret == BPTREE_SEARCH_NOT_FOUND)
      {
        printf("\nSearching (%d) -> NOT FOUND !!", pri_int);
      }
      else
      {
        //printf("\nSearch (%d) exists -> record num = %d", pri_int, ret);
        printf("\n %d exists \n\n",pri_int);
        //print the details of the particular row;
        FILE *fpz;
        char *str1;
        printf("\n-------------------------------------\n");
        str1 = (char*)malloc(sizeof(char)*MAX_PATH);
        sprintf(str1,"table/%s/file%d.dat",tab,ret);
        fpz = fopen(str1,"r");
```

```
            for(int j=0;j<inp1.count;j++)
            {
              if(inp1.col[j].type==INT)
              {
                fread(&c,1,sizeof(int),fpz);
                cout<<c<<"\t";
              }
              else if(inp1.col[j].type==VARCHAR)
              {
                fread(d,1,sizeof(char)*MAX_NAME,fpz);
                cout<<d<<"\t";
              }
            }
            printf("\n---------------------------------\n");
            fclose(fpz);
            free(str1);
          }
        }
        else if(inp1.col[0].type==VARCHAR)
        {
          cout<<"\nenter  key[col 0] to search\n";
          cin>>pri_char;
          void *arr[MAX_NAME];
          arr[0]=(char*)malloc(sizeof(char)*MAX_NAME);
          arr[0]=pri_char;
          ret=mytree.get_record(*(int*)arr[0]);
          if (ret == BPTREE_SEARCH_NOT_FOUND)
          {
            printf("\nSearching (%s) -> NOT FOUND !!", pri_char);
          }
          else
          {
            // printf("\nSearching (%s) -> record num = %d", pri_char, ret);
            printf("\n%s exists\n\n",pri_char);
            //print the details of the particular row;
            FILE *fpz;
            char *str1;
            str1=(char*)malloc(sizeof(char)*MAX_PATH);
            sprintf(str1,"table/%s/file%d.dat",tab,ret);
            fpz=fopen(str1,"r");
            printf("\n---------------------------------\n");
            for(int j=0;j<inp1.count;j++)
            {
              if(inp1.col[j].type==INT)
              {
                fread(&c,1,sizeof(int),fpz);
                cout<<c<<"\t";
              }
```

```
            else if(inp1.col[j].type==VARCHAR)
            {
              fread(d,1,sizeof(char)*MAX_NAME,fpz);
              cout<<d<<"\t";
            }
          }
          printf("\n--------------------------------------\n");
          fclose(fpz);
          free(str1);
        }
      }
    }
    else
    {
      printf("%s doesn't exists!!!\n\n",tab);
    }
  }
  else
  {
    cout<<"wrong input!!!\n\n";
    return;
  }
  free(tab);
}
```

The changes made in the B+ Tree code insert_record( ) and search_leaf( ) functions have been marked in red colour.

```
int BPtree::insert_record(int primary_key, int record_num){
    Btreenode n(true);
    int q, j, prop_n, prop_k, prop_new, curr_node = root_num;
    bool finish = false;
    std::stack < std::pair<int, Btreenode> > S;
    read_node(curr_node, n);

    // Traverse the tree till we get the leaf node;
    while (!n.isleaf()){
        S.push(make_pair(curr_node,n));
        q = n.num_pointers();
    // if key already exists then return ERROR
     if (n.search_key(primary_key)) {
         return BPTREE_INSERT_ERROR_EXIST;
     }
        if (primary_key <= n.get_key(1)){
            curr_node = n.get_pointer(1);
        }else if (primary_key > n.get_key(q - 1)){
            curr_node = n.get_pointer(q);
        }else{
            curr_node = n.get_pointer(n.get_next_key(primary_key) + 1);
        }
        // get all the data of node n from file tree%d.dat(%d==curr_node);
        read_node(curr_node, n);
    }

    // Here n is Leaf Node
    // if key already exists then return ERROR
    if (n.search_key(primary_key)) {
        return BPTREE_INSERT_ERROR_EXIST;
    }

    /*
      if n is not full, insert key and pointer to node
      write back node to file, update meta_data and return
    */
    if (!n.full()){
        n.insert_key(primary_key, record_num);
        write_node(curr_node, n);
        update_meta_data();
        return BPTREE_INSERT_SUCCESS;
    }
```

```
//if node n is full, then split;
  Btreenode temp(true), new_node(true);

  temp = n;
  temp.insert_key(primary_key, record_num);
  j = ceil((BPTREE_MAX_KEYS_PER_NODE + 1.0) / 2.0);
  // if max_key_per_node = 4, j = 3
  // copy the first j-1 values of temp to node n
  // and remaining to new_node
  n.copy_first(temp, j-1);
  //now one file is increased to store the new node;
  files_till_now++;
  // next pointer of new_node will be next pointer of n
  // and next pointer of n will be newly created node new_node
  //new_node.set_next(n.get_next());
  //n.set_next(files_till_now);** these two are removed and commented as btree doesn't
  // have linked leaves
  new_node.copy_last(temp, j);

  /*
    prop_k is key to be inserted into root
    and prop_new and prop_n are pointers to
    be attached to this root value
  */
  prop_k = temp.get_key(j); // key to be moved to new root
  prop_new = files_till_now;
  prop_n = curr_node;

  // write back the two new nodes created
  write_node(files_till_now, new_node);
  write_node(curr_node, n);
  temp.clear_data();
  new_node.clear_data();

  /* Keep repeating until we reach root
     or find an empty internal node */
  while (!finish){
     if (S.size() == 0){
        /*Last element splitted was root so create new root and assign meta_data */
        Btreenode nn(false);
        // insert key to new root
        nn.push_key(prop_k);
        // insert two pointer associated to this key for left and right
        nn.push_pointer(prop_n); nn.push_pointer(prop_new);
        files_till_now++;
```

```
            write_node(files_till_now, nn);
            root_num = files_till_now;
            finish = true;
        }else{
            std::pair<int, Btreenode> p = S.top();
            curr_node = p.first;
            n = p.second;
            S.pop();
            // read_node(curr_node, n);
            if (!n.full()){
                n.insert_key(prop_k, prop_new);
                write_node(curr_node, n);
                finish = true;
            }else{
                /* Split is propogating towards top */
                temp = n;
                temp.insert_key(prop_k, prop_new);
                j = floor((BPTREE_MAX_KEYS_PER_NODE + 1.0) / 2.0);
                n.copy_first(temp, j-1);**//all half nodes except jth node stay
               // (no redundancy of node)
                files_till_now++;
                new_node.set_leaf(false);
                new_node.copy_last(temp, j);
                write_node(files_till_now, new_node);
                write_node(curr_node, n);
                prop_k = temp.get_key(j);
                prop_new = files_till_now;
                prop_n = curr_node;
            }
        }
    }
    update_meta_data(); return BPTREE_INSERT_SUCCESS;
}

Btreenode BPtree::search_leaf(int primary_key)
{
    Btreenode n(true);
    int q, curr_node = root_num;
    read_node(curr_node, n);

    //Traversing the Tree from root till leaf
    while (!n.isleaf()){
    if (n.search_key(primary_key)) {
        return n;
     }
```

```
q = n.num_pointers();
      if (primary_key < n.get_key(1)){
         curr_node = n.get_pointer(1);
      }
      else if (primary_key > n.get_key(q - 1)){
         curr_node = n.get_pointer(q);
      }else{
         curr_node = n.get_pointer(n.get_next_key(primary_key) + 1);
      }
      read_node(curr_node, n);
   }
    if (n.search_key(primary_key)) {
       return n;
    }
   return null;
}
```

Differences Between B Tree and B+ Tree Indexing

S.No.	B Tree	B+ Tree
1	All internal and leaf nodes have data record pointers. In the B tree, all the keys and records are stored in both internal as well as leaf nodes.	Only leaf nodes have data record pointers.In the B+ tree, keys are the indexes stored in the internal nodes and records are stored in the leaf nodes.
2	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and more accurate.
3	There are no duplicates of keys.	Duplicates of keys are maintained and all nodes are present at leaf.
4	Insertion takes more time and it is unpredictable.	Insertion is easier and the results are predictable.
5	Deletion of an internal node is very complex and the tree undergoes a lot of transformations.	Deletion of any node is easy because all nodes are available at the leaf.
6	Leaf nodes are not stored as a structural linked list. There is no sequential access.	Leaf nodes are stored as a structural linked list. The leaf nodes are linked to each other to provide sequential access.
7	Deletion of internal nodes is very slow and a time-consuming process as we need to consider the child of the deleted key as well.	Deletion in B+ tree is very fast since all records are stored in leaf nodes so we do not have to consider the child of the node.
8	In B Tree, more splitting operations are performed due to which height increases faster compared to width.	B+ tree has more width as compared to height.

Conclusion

Succeeding the implementation of Indexing in Database Management with two approaches of B-Tree and an improvised version B+-Tree, it can be concluded that there is a considerable change in implementation times of both.

Though B+-Trees take up little more space in storing the record key values redundantly and having pointers to siblings in the bottom level/leaves, it reduces the search and access times of records. On the other hand multiple records along with pointers being stored in the internal nodes of B-Trees adds up the additional task of getting the starting index of key attributes in a record.

So, we conclude that B+-Tree supersedes B-tree in speeds of retrieval and alterations and thus most of the present Database Management Systems are biased towards B+ Tree for Index implementation.

References:

- https://dev.mysql.com/downloads/mysql/
- https://mariadb.org/download/
- https://github.com/msdeep14/DeepDataBase
- javatpoint.com
- geeksforgeeks.org
- guru99.com