ZHEJIANG UNIVERSITY
DATABASE SYSTEM, 2020 SUMMER

Index Manager 设计报告

Date:2020.6.25

邢书婷
3180106027
数字媒体技术
计算机科学技术学院

目录

一、 模块概述

MiniSQL 的整体设计要求对于表的主键自动建立 B+树索引，对于声明为 unique 的属性可以通过 SQL 语句由用户指定建立或删除索引。因此，所有的 B+树索引都是单属性单值的。

Index Manager 负责 B+树索引的实现，实现 B+树的创建（由索引的定义和创表时主键的定义引起）和删除（由索引的删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

二、 主要功能

创建 B+树：从 API 传入索引名，关键字位置，各自段类型数组，各字段名称数组，若语句执行成功，则建立一个空的 B+树，若创建成功，则生成一个.index 文件，若创建失败，则抛出异常

等值查找：根据所查找的索引名找到.index 文件，在根据关键字信息和键值返回拥有该键值的记录在表中的位置（块号和块中偏移），若语句执行成功，则返回位置信息，若失败则返回 FileAddress{0,0 }。

插入键值：从 API 传入新键值的信息和地址，若键值已存在，则插入失败，抛出异常；若键值不存在，则继续插入。

更新键值： 从 API 传入更新键值的信息和地址，删除关键字对应的信息，保存记录地址，在插入更新关键字的信息，和上一步保存的地址。

删除键值：在索引中删除某个键值，若该键值不存在则返回地址为 null。 若该节点存在，则删除节点的信息，返回节点的地址。

三、 对外提供的接口

1.根据 API 传进的索引名找到相应的索引文件

BTree(string idx_name);

2.建表时，根据 API 传进的索引名，主键位置信息，Unique 位置信息数组，各个字段类型数组，各个字段名称数组创建一个空的 B+树

```
BTree(const string idx_name, int KeyTypeIndex,
        int(&_UniqueKeyIndex)[RecordColumnCount],
        char(&_RecordTypeInfo)[RecordColumnCount],
        char(&_RecordColumnName)[RecordColumnCount / 4 * ColumnNameLength]);
```

3. 创建 Unique 索引时，根据 API 传进的索引名，Unique 位置信息，各个字段类型数组，各个字段名称数组创建一个空的 B+树

```
BTree(const string idx_name,
        int KeyTypeIndex,
       char(&_RecordTypeInfo)[RecordColumnCount],
       char(&_RecordColumnName)[RecordColumnCount / 4 * ColumnNameLength]);
```

4. 等值查找，在插入时检查主键和 Unique 字段是否重复

```
FileAddress Search(Key_Attr search_key);
```

5. 插入键值

```
bool Insert(Key_Attr k, FileAddress k_fd);
```

6. 更新键值

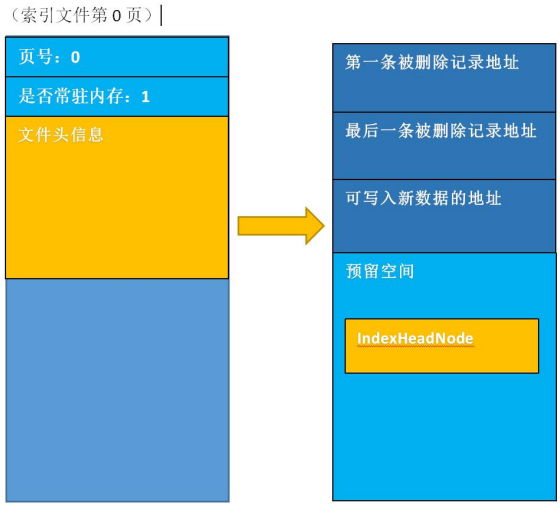
```
FileAddress UpdateKey(Key_Attr k, Key_Attr k_new);
```

7. 删除键值

```
FileAddress Delete(Key_Attr k);
```

四、 设计思路

每份文件的文件头上定义有预留空间，大小为 512，里面储存着索引的文件信息头 IndexHeadNode，包含着 B+树的根节点地址，最左端叶节点地址，用来建树的关键字段位置(例如建表时对主键建立索引，记录的则是主键在整个表的字段中的位置)，表中字段类型信息数组，表中字段名称信息数组，如果是建立主键的 B+树，还会调用记录的表中 Unique 的位置，该表已建索引的名称，该表已建索引字段的位置。当我们需要调用索引时，会通过索引名到 BufferManager 中找到索引的文件信息头，在进行下一步操作。



```
class IndexHeadNode
{
public:
    FileAddress    root;                                    // the address of the root
    FileAddress    MostLeftNode;                            // the address of the most left node
    int         KeyTypeIndex;                              // 关键字字段的位置  即第几个
    char        RecordTypeInfo[RecordColumnCount];         // 记录字段类型信息
    char        RecordColumnName[RecordColumnCount / 4 * ColumnNameLength];
    int         UniqueKeyIndex[RecordColumnCount];                      //记录Unique位置

    char        RecordIndexName[RecordColumnCount / 4 * ColumnNameLength]; //记录索引名
    int         RecordIndexInfo[RecordColumnCount];              //记录索引位置
};
```

同时我们还定义了 B+树的节点类 BTNode，有 ROOT,INNER,LEAF 三种类型，每个 BTNode 类中包含了该节点类型，节点中储存的关键字数目，关键字信息数组，关键字地址数组，假如该节点是叶节点还会用到指向下一个节点地址的指针 next。

```
// define B+tree Node
enum class NodeType { ROOT, INNER, LEAF };
class BTNode
{
public:
    NodeType node_type;                          // 节点类型
    int count_valid_key;                         // 该节点中储存的关键字数目

    Key_Attr key[MaxKeyCount];                    // array of keys
    FileAddress children[MaxChildCount];         // if the node is not a leaf node, children store the
    ¦                                            // otherwise it store record address;

    FileAddress next;                             // if leaf node
    void PrintSelf();
};
```

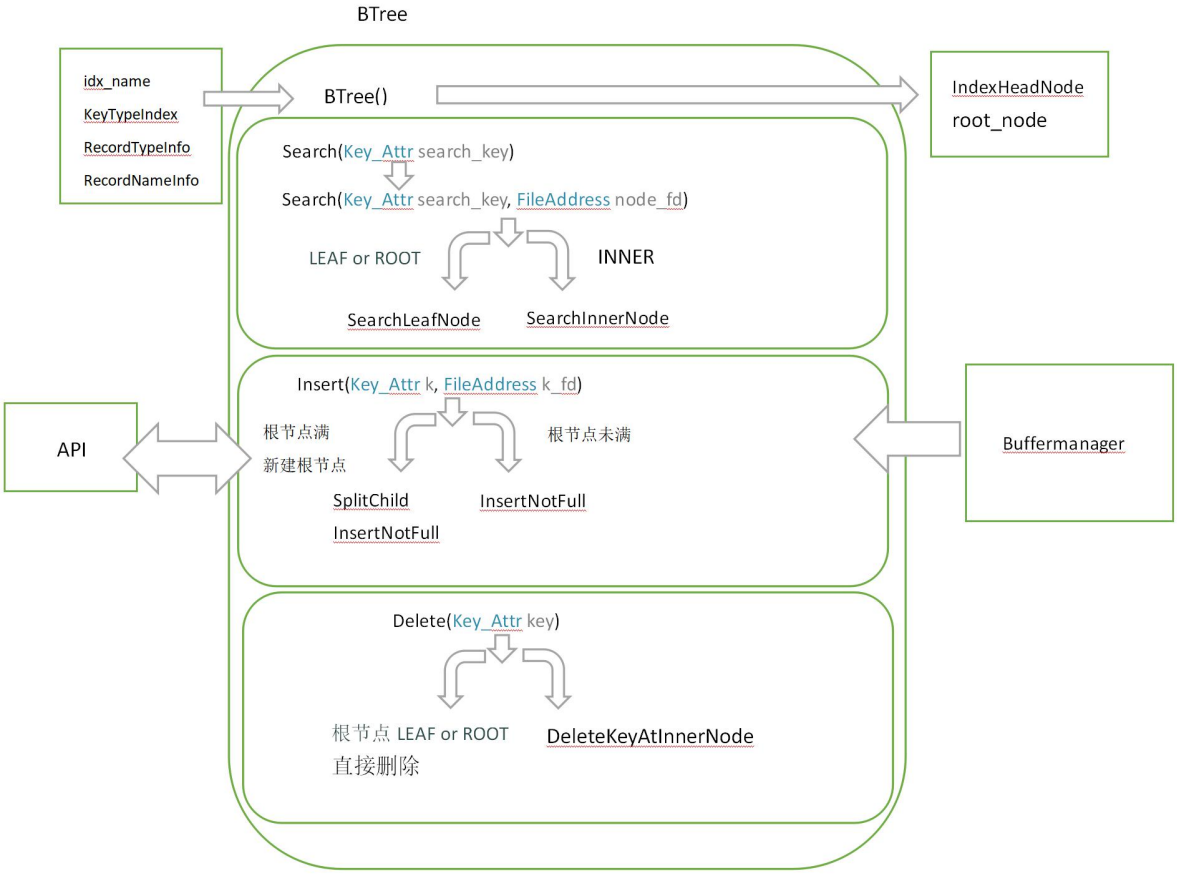
另一方面我们还建立一棵只提供方法的 BTree 类，里面包含了索引文件信息头，通过信息头找到树的根节点等信息，在根据 API 里进行的命令来进行具体的插入，删除，更新等操作。

```
class BTree
{
public:
    // 参数：索引文件名称， 关键字类型， 记录各个类型信息数组， 记录各个字段名称信息数组
    BTree(string idx_name);
    BTree(const string idx_name, int KeyTypeIndex, int(&_UniqueKeyIndex)[RecordColumnCount], char(&_RecordTypeInfo)[RecordColumnCount]
    BTree(const string idx_name, int KeyTypeIndex, char(&_RecordTypeInfo)[RecordColumnCount], char(&_RecordColumnName)[RecordColumnCou
    ~BTree() { }
    FileAddress Search(Key_Attr search_key);                          // 查找关键字是否已经存在
    bool Insert(Key_Attr k, FileAddress k_fd);                        // 插入关键字k
    FileAddress UpdateKey(Key_Attr k, Key_Attr k_new);                 // 返回关键字对应的记录地址
    FileAddress Delete(Key_Attr k);                                   // 返回该关键字记录在数据文件中的地址
    void PrintBTreeStruct();                                        // 层序打印所有结点信息
    void PrintAllLeafNode();
    IndexHeadNode *GetPtrIndexHeadNode();
    BTNode *FileAddrToMemPtr(FileAddress node_fd);                  // 文件地址转换为内存指针
    //void InsertIndexName();

private:
    FileAddress DeleteKeyAtInnerNode(FileAddress x, int i, Key_Attr key);          // x的下标为i的结点为叶子结点
    FileAddress DeleteKeyAtLeafNode(FileAddress x, int i, Key_Attr key);           // x的下标为i的结点为叶子结点
    void InsertNotFull(FileAddress x, Key_Attr k, FileAddress k_fd);               // 在一个非满结点 x, 插入关键字 k, k的数据地址
    void SplitChild(FileAddress x, int i, FileAddress y);                         // 分裂x的孩子结点x.children[i] , y
    FileAddress Search(Key_Attr search_key, FileAddress node_fd);                  // 判断关键字是否存在
    FileAddress SearchInnerNode(Key_Attr search_key, FileAddress node_fd);         // 在内部节点查找
    FileAddress SearchLeafNode(Key_Attr search_key, FileAddress node_fd);          // 在叶子结点查找


private:
    FILE* file_id;
    string str_idx_name;
    IndexHeadNode idx_head;
};
```

五、 整体架构



六、 关键函数和代码（伪代码）

1. 建表时初始化主键 B+树(建索引时和这个大体一样，只是少了 Unique 字段信息的传入)

```
BTree::BTree(const string idx_name, int KeyTypeIndex, int(&_UniqueKeyIndex)[RecordColumnCount],
    char(&_RecordTypeInfo)[RecordColumnCount],
    char(&_RecordColumnName)[RecordColumnCount / 4 * ColumnNameLength])//索引文件名，关键字序号，类型，名称
    :str_idx_name(idx_name)
{
    auto &buffer = GetGlobalBuffer();
    auto pMemFile = buffer[str_idx_name.c_str()];

    // 如果索引文件不存在则创建
    if (!pMemFile)
    {
        // 创建索引文件
        buffer.CreateFile(str_idx_name.c_str());
        pMemFile = buffer[str_idx_name.c_str()];
        // 初始化索引文件，创建一个根结点
        BTNode root_node;
        assert(sizeof(BTNode) < (PageSize - sizeof(PageHead)));
        root_node.node_type = NodeType::ROOT;
        root_node.count_valid_key = 0;
        root_node.next = FileAddress{ 0,0 };
        FileAddress root_node_fd = buffer[str_idx_name.c_str()]->AddRecord(&root_node, sizeof(root_node));
        // 初始化其他索引文件头信息
        idx_head.root = root_node_fd;
        idx_head.MostLeftNode = root_node_fd;
        idx_head.KeyTypeIndex = KeyTypeIndex;

        memcpy(idx_head.UniqueKeyIndex, _UniqueKeyIndex, RecordColumnCount);
        memcpy(idx_head.RecordTypeInfo, _RecordTypeInfo, RecordColumnCount);
        memcpy(idx_head.RecordColumnName, _RecordColumnName, RecordColumnCount / 4 * ColumnNameLength);
        // 将结点的地址写入文件头的预留空间区
        memcpy(buffer[str_idx_name.c_str()]->GetFileFirstPage()->GetFileHead()->Reserve, &idx_head, sizeof(idx_head));
    }
    file_id = pMemFile->FilePointer;
}
```

2. 进行查找操作时的函数

```
FileAddress BTree::Search(Key_Attr search_key)
{
    auto pMemPage = GetGlobalClock()->GetMemAddr(file_id, 0);
    auto pfilefd = (FileAddress*)pMemPage->GetFileHead()->Reserve;  // 找到根结点的地址
    return Search(search_key, *pfilefd);
}

FileAddress BTree::Search(Key_Attr search_key, FileAddress node_fd)
{
    BTNode* pNode = FileAddrToMemPtr(node_fd);

    if (pNode->node_type == NodeType::LEAF || pNode->node_type == NodeType::ROOT)
    {
        return SearchLeafNode(search_key, node_fd);
    }
    else
    {
        return SearchInnerNode(search_key, node_fd);
    }
}

FileAddress BTree::SearchLeafNode(Key_Attr search_key, FileAddress node_fd)
{

    BTNode* pNode = FileAddrToMemPtr(node_fd);
    for (int i = 0; i < pNode->count_valid_key; i++)
    {
        if (pNode->key[i] == search_key)
        {
            return pNode->children[i];
        }
    }
    return FileAddress{ 0,0 };
}

FileAddress BTree::SearchInnerNode(Key_Attr search_key, FileAddress node_fd)
{
    FileAddress fd_res{ 0,0 };

    BTNode* pNode = FileAddrToMemPtr(node_fd);
    for (int i = pNode->count_valid_key - 1; i >= 0; i--)
    {
        if (pNode->key[i] <= search_key)
        {
            fd_res = pNode->children[i];
            break;
        }
    }

    if (fd_res == FileAddress{ 0,0 })
    {
        return fd_res;
    }
    else
    {
        BTNode* pNextNode = FileAddrToMemPtr(fd_res);
        if (pNextNode->node_type == NodeType::LEAF)
            return SearchLeafNode(search_key, fd_res);
        else
            return SearchInnerNode(search_key, fd_res);
    }
    //return fd_res;
}
```

3. 进行插入操作的函数

```
bool BTree::Insert(Key_Attr k, FileAddress k_fd)
{
    // 如果该关键字已经存在则插入失败
    try
    {
        auto key_fd = Search(k);
        if (key_fd != FileAddress{ 0,0 })
            throw Error("(KEY_INSERT_FAILED) Key Word Insert Failed! The record that to inset has been excisted!");
        }
    catch (const Error &error)
    {
        DispatchError(error);
        std::cout << std::endl;
        return false;
    }
    // 得到根结点的fd
    FileAddress root_fd = *(FileAddress*)GetGlobalBuffer()[str_idx_name.c_str()]->GetFileFirstPage()->GetFileHead()->Reserve;
    auto proot = FileAddrToMemPtr(root_fd);
    if (proot->count_valid_key == MaxKeyCount)
    {
        // 创建新的结点 s ，作为根结点
        BTNode s;
        s.node_type = NodeType::INNER;  // 只有初始化才使用 NodeType::ROOT
        s.count_valid_key = 1;
        s.key[0] = proot->key[0];
        s.children[0] = root_fd;
        FileAddress s_fd = GetGlobalBuffer()[str_idx_name.c_str()]->AddRecord(&s, sizeof(BTNode));
```

```
        // 将新的根节点文件地址写入
        *(FileAddress*)GetGlobalBuffer()[str_idx_name.c_str()]->GetFileFirstPage()->GetFileHead()->Reserve = s_fd;
        GetGlobalBuffer()[str_idx_name.c_str()]->GetFileFirstPage()->IsModified = true;

        //将旧的根结点设置为叶子结点
        auto pOldRoot = FileAddrToMemPtr(root_fd);
        if (pOldRoot->node_type == NodeType::ROOT)
            pOldRoot->node_type = NodeType::LEAF;

        // 先分裂再插入
        SplitChild(s_fd, 0, s.children[0]);
        InsertNotFull(s_fd, k, k_fd);
    }
    else
    {
        InsertNotFull(root_fd, k, k_fd);
    }
    return true;
}

// 将x下标为i的孩子满结点分裂
void BTree::SplitChild(FileAddress x, int i, FileAddress y)
{
    auto pMemPageX = GetGlobalClock()->GetMemAddr(file_id, x.FilePageID);
    auto pMemPageY = GetGlobalClock()->GetMemAddr(file_id, y.FilePageID);
    pMemPageX->IsModified = true;
    pMemPageY->IsModified = true;

    BTNode*px = FileAddrToMemPtr(x);
    BTNode*py = FileAddrToMemPtr(y);
    BTNode z;          // 分裂出来的新结点
    FileAddress z_fd;    // 新结点的文件内地址

    z.node_type = py->node_type;
    z.count_valid_key = MaxKeyCount / 2;

    // 将y结点的一般数据转移到新结点
    for (int k = MaxKeyCount / 2; k < MaxKeyCount; k++)
    {
        z.key[k - MaxKeyCount / 2] = py->key[k];
        z.children[k - MaxKeyCount / 2] = py->children[k];
    }
    py->count_valid_key = MaxKeyCount / 2;

    // 在y的父节点x上添加新创建的子结点 z
    int j;
    for (j = px->count_valid_key - 1; j > i; j--)
    {
        px->key[j + 1] = px->key[j];
        px->children[j + 1] = px->children[j];
    }

    j++; // after j++, j should be i+1;
    px->key[j] = z.key[0];

    if (py->node_type == NodeType::LEAF)
    {
        z.next = py->next;
        z_fd = GetGlobalBuffer()[str_idx_name.c_str()]->AddRecord(&z, sizeof(z));
        py->next = z_fd;
    }
    else
        z_fd = GetGlobalBuffer()[str_idx_name.c_str()]->AddRecord(&z, sizeof(z));

    px->children[j] = z_fd;
    px->count_valid_key++;
}

// 在一个非满结点 x, 插入关键字 k, k的数据地址为 k_fd
void BTree::InsertNotFull(FileAddress x, Key_Attr k, FileAddress k_fd)
{
    auto px = FileAddrToMemPtr(x);
    int i = px->count_valid_key - 1;

    // 如果该结点是叶子结点, 直接插入
    if (px->node_type == NodeType::LEAF || px->node_type == NodeType::ROOT)
    {
        while (i >= 0 && k < px->key[i])
        {
            px->key[i + 1] = px->key[i];
            px->children[i + 1] = px->children[i];
            i--;
        }
        px->key[i + 1] = k;
        px->children[i + 1] = k_fd;
        px->count_valid_key += 1;
    }
    else
    {
        while (i >= 0 && k < px->key[i])  i = i - 1;

        // 如果插入的值比内节点的值还小
        if (i < 0) {
            i = 0;
            px->key[i] = k;
        }
        if (i < 0) return;
        //assert(i >= 0);

        FileAddress ci = px->children[i];
        auto pci = FileAddrToMemPtr(ci);
        if (pci->count_valid_key == MaxKeyCount)
        {
            SplitChild(x, i, ci);
            if (k >= px->key[i + 1])
                i += 1;
        }
        InsertNotFull(px->children[i], k, k_fd);
    }
}
```

4. 进行删除时的函数

```cpp
FileAddress BTree::Delete(Key_Attr key)
{
    auto search_res = Search(key);
    if (search_res.OffSet == 0)
        return FileAddress{ 0,0 };

    // 得到根结点的fd
    FileAddress root_fd = *(FileAddress*)GetGlobalBuffer()[str_idx_name.c_str()]->GetFileFirstPage()->GetFileHead()->Reserve;
    auto proot = FileAddrToMemPtr(root_fd);


    // 根节点为ROOT 或者 LEAF 直接删除
    if (proot->node_type == NodeType::ROOT || proot->node_type == NodeType::LEAF)
    {
        // 直接删除
        int j = proot->count_valid_key - 1;
        while (proot->key[j] != key)j--;
        //if (j < 0) return;
        assert(j >= 0);
        FileAddress fd_res = proot->children[j];
        for (j++; j < proot->count_valid_key; j++)
        {
            proot->key[j - 1] = proot->key[j];
            proot->children[j - 1] = proot->children[j];
        }
        proot->count_valid_key--;
        return fd_res;
    }

    int i = proot->count_valid_key - 1;
    while (proot->key[i] > key)i--;

    auto fd_delete = DeleteKeyAtInnerNode(root_fd, i, key);

    if (proot->count_valid_key == 1)
    {
        // 将新的根节点文件地址写入
        *(FileAddress*)GetGlobalBuffer()[str_idx_name.c_str()]->GetFileFirstPage()->GetFileHead()->Reserve = proot->children[0];
        GetGlobalBuffer()[str_idx_name.c_str()]->GetFileFirstPage()->IsModified = true;

        GetGlobalBuffer()[str_idx_name.c_str()]->DeleteRecord(&root_fd, sizeof(BTNode));
    }

    return fd_delete;
}


FileAddress BTree::DeleteKeyAtInnerNode(FileAddress x, int i, Key_Attr key)
{
    auto px = FileAddrToMemPtr(x);
    auto py = FileAddrToMemPtr(px->children[i]);
    FileAddress fd_res;
    if (py->node_type == NodeType::LEAF)
    {
        fd_res = DeleteKeyAtLeafNode(x, i, key);
    }
    else
    {
        int j = py->count_valid_key - 1;
        while (py->key[j] > key)j--;
        //assert(j >= 0);
        fd_res = DeleteKeyAtInnerNode(px->children[i], j, key);
    }

    // 判断删除后的结点个数
    if (py->count_valid_key >= MaxKeyCount / 2)
        return fd_res;

    // 如果删除后的关键字个数不满足B+树的规定，向兄弟结点借用key

    // 如果右兄弟存在且有富余关键字
    if ((i <= px->count_valid_key - 2) && (FileAddrToMemPtr(px->children[i + 1])->count_valid_key > MaxKeyCount / 2))
    {
        auto RBrother = FileAddrToMemPtr(px->children[i + 1]);
        // 借来的关键字
        auto key_bro = RBrother->key[0];
        auto fd_bro = RBrother->children[0];


        // 更新右兄弟的索引结点
        px->key[i + 1] = RBrother->key[1];
        // 跟新右兄弟结点
        for (int j = 1; j <= RBrother->count_valid_key - 1; j++)
        {
            RBrother->key[j - 1] = RBrother->key[j];
            RBrother->children[j - 1] = RBrother->children[j];
        }
        RBrother->count_valid_key -= 1;

        // 更新本叶子结点
        py->key[py->count_valid_key] = key_bro;
        py->children[py->count_valid_key] = fd_bro;
        py->count_valid_key += 1;
```

```
        return fd_res;
    }

    // 如果左兄弟存在且有富余关键字
    if (i > 0 && FileAddrToMemPtr(px->children[i - 1])->count_valid_key > MaxKeyCount / 2)
    {
        auto LBrother = FileAddrToMemPtr(px->children[i - 1]);
        // 借来的关键字
        auto key_bro = LBrother->key[LBrother->count_valid_key - 1];
        auto fd_bro = LBrother->children[LBrother->count_valid_key - 1];

        // 更新左兄弟结点
        LBrother->count_valid_key -= 1;

        // 更新本结点
        px->key[i] = key_bro;
        for (int j = py->count_valid_key - 1; j >= 0; j--)
        {
            py->key[j + 1] = py->key[j];
            py->children[j + 1] = py->children[j];
        }
        py->key[0] = key_bro;
        py->children[0] = fd_bro;

        py->count_valid_key += 1;

        return fd_res;
    }


    // 若兄弟结点中没有富余的key,则当前结点和兄弟结点合并成一个新的叶子结点，并删除父结点中的key

    // 若右兄弟存在将其合并
    if (i < px->count_valid_key - 1)
    {
        auto RBrother = FileAddrToMemPtr(px->children[i + 1]);
        for (int j = 0; j < RBrother->count_valid_key; j++)
        {
            py->key[py->count_valid_key] = RBrother->key[j];
            py->children[py->count_valid_key] = RBrother->children[j];
            py->count_valid_key++;
        }

        // 更新next
        py->next = RBrother->next;
        // 删除右结点
        GetGlobalBuffer()[str_idx_name.c_str()]->DeleteRecord(&px->children[i + 1], sizeof(BTNode));
        // 更新父节点索引
        for (int j = i + 2; j < px->count_valid_key; j++)
        {
            px->key[j - 1] = px->key[j];
            px->children[j - 1] = px->children[j];
        }
        px->count_valid_key--;
    }
    else
    {// 将左结点合并
        auto LBrother = FileAddrToMemPtr(px->children[i - 1]);
        for (int j = 0; j < py->count_valid_key; j++)
        {
            LBrother->key[LBrother->count_valid_key] = py->key[j];
            LBrother->children[LBrother->count_valid_key] = py->children[j];
            LBrother->count_valid_key++;
        }

        // 更新next
        LBrother->next = py->next;

        // 删除本结点
        GetGlobalBuffer()[str_idx_name.c_str()]->DeleteRecord(&px->children[i], sizeof(BTNode));
        // 更新父节点索引
        for (int j = i + 1; j < px->count_valid_key; j++)
        {
            px->key[j - 1] = px->key[j];
            px->children[j - 1] = px->children[j];
        }
        px->count_valid_key--;
    }
    return fd_res;
}

```