IndexManager 设计报告

刘圣源

• IndexManager 类的说明：

IndexManager 利用 B+树的接口，完成索引的构造和对索引建造、删除、插入、查询等操作。

下面是 IndexManager 类的结构，此处只包含其成员变量，而不含成员函数。

```
class IndexManager{
private:
    typedef map<string,BPlusTree<int> *> intMap;
    typedef map<string,BPlusTree<string> *> stringMap;
    typedef map<string,BPlusTree<float> *> floatMap;

    API *api; // to call the functions of API

    intMap indexIntMap;
    stringMap indexStringMap;
    floatMap indexFloatMap;
    struct keyTmp{
        int intTmp;
        float floatTmp;
        string stringTmp;
        };
    struct keyTmp kt;
};
```

如上，IndexManager 类根据 Index 的类型的不同，包含了 3 类 B+树，即每类 B+树对应一种类型。每次 create 索引都会构造一个新的 B+树，我们通过一个指针指向这个 B+树，而不同索引的 B+树通过索引名进行区分。我们在这里利用了 C++的 map 容器模板类，map 可以在两个不同类型的变量之间形成联系，使得我们能够通过索引名找到这个索引对应的 B+树。

定义的新的结构 keyTmp 使得我们能够使用一个变量表示 3 种不同类型的变量。

- **B+树节点的结构（不包含成员函数）：**

```
template <typename KeyType>
class TreeNode{
public:
    size_t count;  // the count of keys
    TreeNode* parent;    //point to the parent node
    vector <KeyType> keys;
    vector <TreeNode*> childs;
    vector <offsetNumber> vals;

    TreeNode* nextLeafNode;
   // point to the next leaf node（if this node is leafnode）

    bool isLeaf; // the flag whether this node is a leaf node

private:
  int degree;
}
```

由于会需要存储不同类型的 B+树，因此使用了模板类。Vector 容器共 3 个，keys 用于存储具体数值,childs 存储指向下一层的指针，vals 用于指出索引中的该值具体处于 table 的哪一个 record 中。而 private 中的 degree 即 B+树的“degree”的概念。

- **B+树的结构（不包含成员函数）：**

```
template <typename KeyType>
class BPlusTree
{
private:
  typedef TreeNode<KeyType>* Node;
```

```
private:
  string fileName;
  Node root;  // the root node of the tree
  Node leafHead; // the head of the leaf node
  size_t keyCount;
  size_t level;
  size_t nodeCount;
  fileNode* file; // the filenode of this tree
  int keySize; // the size of key
  int degree;
}
```

可发现,BPlusTree 模板类中的成员变量大部分是对 B+树相关信息进行描述。(fileName 证明了是对每个索引都会有一个专门 B+树)

- 下面介绍 IndexManager 的公共接口：

•createIndex()函数，将创建一个新的 B+Tree，同时将新的 B+Tree 的指针和 filePath(索引名)共同放进 map 容器中，参数 type 决定了索引引用的值的类型，决定了将其放在哪个 map 容器内。

```
void createIndex(string filePath,int type);
```

•dropIndex()函数，createIndex()的反操作，调用B+Tree的析构函数，并将相应元素从map中删除。

```
void dropIndex(string filePath,int type);
```

•searchIndex()函数，通过filePath确定B+Tree,再通过key来找到所需要的节点，返回节点的相应的val()值，从而利用其查找到所需要的record信息。

```
offsetNumber searchIndex(string filePath,string key,int type);
```

• insertIndex()、deleteIndexByKey()通过B+Tree的相关接口，在B+Tree中增添节点。

```
void insertIndex(string filePath,string key,offsetNumber blockOffset,int type);
void deleteIndexByKey(string filePath,string key,int type);
```