MeSQL 个人报告

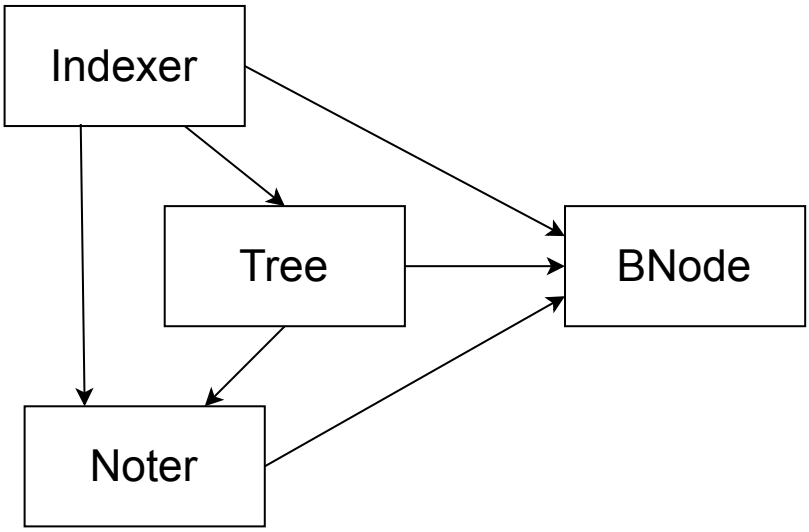
本报告详细叙述 `Indexer` (即 IndexManager) 的设计与实现。

我的 `Indexer` 实现基于 B+ 树数据结构，关于此数据结构的基础知识这里不重复。

约定

- 所有指向文件内地址、文件块的指针类型都是 `std::size_t` 。

组件概览



- `BNode` : B+ 树节点类，是 B+ 树节点在内存中的表现形式，它是这样定义的：

```
class BNode {
public:
    size_t seg; // seg == 0 means null node
    bool is_leaf;
    vector<size_t> P;
    vector<Literal> K;

    BNode(size_t _seg);
};
```

B+ 树中一个节点就是文件系统中的一个块。 `seg` 表示这个节点在文件中的块编号； `is_leaf` 表示本节点是否是叶子节点； `P` 是本节点包含的所有指针， `K` 是本节点的所有值，它们的关系是：

P_0	K_0	P_1	K_1	...	P_{m-2}	K_{m-2}	P_{m-1}

特别需要说明的是，虽然每个 B+ 树节点的最大指针数 N 是固定的，但为了高效和实现方便，我的 `P` 和 `K` 长度是动态的，但始终保证 `P.size() >= 1 && K.size() + 1 == P.size()` 。

- `Noter`：与 `BufferManager` 交互，是支持创建、删除、读取、写回 B+ 树节点的功能类，也是把内存中的 B+ 树节点转化为 `BNode` 的桥梁。基于前述基本功能，它还实现了直接读取 B+ 树的根节点块编号、修改根节点的功能。
- `Tree`：进行 B+ 树操作，维护 B+ 树数据结构的功能类，其唯一的成员变量是一个 `Noter` 的引用，即 `Tree` 本身不储存信息，它只是在 `Noter` 基础上操作索引文件中的 B+ 树的下层手柄。
- `Indexer`：Index 部分向外提供的接口，利用 `Tree` 和 `Noter` 实现了：

  1. 索引文件的初始化、索引文件删除
  2. 插入记录、删除记录
  3. 直接按块编号获取 B+ 树节点，用于上层组件遍历 B+ 树时
  4. Find 系列：找最左边的叶子、第一个大于给定值的叶子和位置、第一个大于等于给定值的叶子和位置，用于按索引查询时

  这里最复杂的组件是 `Tree` ，因为它承担了 B+ 树的所有维护操作；其他组件都是直接而简单的（`Indexer` 实际上就是一个包装，内部把大部分工作都转移给了 `Tree` 或 `Noter`）。因此下面详述 `Tree` 的实现。

Tree

`Tree` 的任务可以归结为三个——插入、删除、查找。我们从易到难介绍。

查找

找到最左边的叶子不用说了，直接从根开始一直往最左边走就行了。

找第一个大于给定值的和第一个大于等于给定值的类似，我们讨论前者，即找第一个大于给定值的叶子和位置。

设给定值为 `val` 。由于 B+ 树的儿子区间是左闭右开的，我们没办法直接找到第一个大于 `val` 的位置，所以采用分步策略，**先假装 `val` 这个值在 B+ 树中存在，找到这个值应该在的位置，然后在这个位置附近调整找到第一个大于 `val` 的位置**。其实 大于 和 大于等于 的差别只在于最后调整的过程。这样我们就得到了一个递归算法：

```
pair<BNode,size_t> Tree::_first_greater_pos(size_t x,const Literal &val) {
    // x: 当前点在索引文件中的块编号 ; val: 给定值
    BNode now = noter.get_bnode(x);
    if (now.is_leaf) { // 已到叶子
        // STL upper_bound 二分查找第一个大于 val 的位置
        size_t j = upper_bound(now.K.begin(),now.K.end(),val) - now.K.begin();
        // 在本叶子中直接返回
        if (j < now.K.size()) return pair<BNode,size_t>(now,j);
        // 如果没有下一个叶子, 那么目标位置不存在
        size_t y = now.P.back();
        if (y == 0) return pair<BNode,size_t>(BNode(0),0);
        BNode nxt = noter.get_bnode(y);
        assert(val < nxt.K.front());
        // 下一个叶子的开头必然是目标
        return pair<BNode,size_t>(nxt,0);
    } else { // 未到叶子
        // 找到 val 本身应该出现的儿子走进去
        size_t j = upper_bound(now.K.begin(),now.K.end(),val) - now.K.begin();
        return _first_greater_pos(now.P.at(j),val);
    }
}
```

只要从根开始调用这个函数就能得到结果啦。

插入

插入比删除简单，我认为主要原因在于回溯方向与节点更新方向相同。

经过一番思考，我发现任何一层的插入流程可以简化为：

1. 不管其它的，直接插入
2. 如果此点过大，则分裂，并向上返回新点子树的最小值和新点的地址

也就是插入过程和分裂过程可以分离来考虑，这大大简化了插入的实现。

叶子和内点稍有些不同，其分裂需要分开写。

我实现了 `split_leaf` 和 `split_inner` 两个函数，它们会检测传入的点是否需要分裂，如果不需要则返回一个空值，否则执行分裂并返回父节点应添加的 key 与 pointer 。最后 insert 操作实现为

```
ins_rev Tree::_insert(size_t x,const Literal &val,size_t pos) {
    assert(x);
    BNode now = noter.get_bnode(x);
    if (now.is_leaf) {
        size_t j = upper_bound(now.K.begin(),now.K.end(),val) - now.K.begin();
        // 先直接插入
        now.K.insert(now.K.begin() + j,val);
        now.P.insert(now.P.begin() + j,pos);
        return split_leaf(now); // 内部考虑本点的分裂
    } else {
        size_t j = upper_bound(now.K.begin(),now.K.end(),val) - now.K.begin();
        ins_rev res = _insert(now.P.at(j),val,pos);
        if (res.second) { // 如果下层分裂则本点需要插入
            now.K.insert(now.K.begin() + j,res.first);
            now.P.insert(now.P.begin() + j + 1,res.second);
            return split_inner(now); // 内部考虑本点的分裂
        }
        return ins_rev(Literal(),0); // 本点没有插入,返回空值表示上层也无需插入
    }
}
```

分裂是容易实现的，只要把当前点的 `P` 和 `K` 分成两半就行了。

最后如果根节点需要分裂，则创建新根节点并修改文件记录的根节点地址。

删除

删除比插入难，我认为原因是回溯过程与节点更新方向冲突。回溯是从下往上的，而发生节点合并、重分配时不仅需要从下往上移动数据，还需要从上往下移动数据。

我的解决办法是：不像插入那样在当前节点考虑自己的分裂，删除时我们在父节点考虑当前节点的调整，这样一来就理顺了回溯与信息传递的关系，整个实现就容易了。

这里最重要的函数是 `adjust_two(now,one,two)` ，其中 `one` 和 `two` 一定是同一个父亲 `now` 的相邻儿子节点。我们这个函数内我们无需知道究竟哪个点不满足 B+ 树限制 (这是 `adjust_two` 的调用者考虑的问题) ，只需执行调整这两个点的操作。

当然叶子和内点情况有所不同，要分开考虑。调整的时候有两种情况：

1. 两个节点合起来满足要求，则合并两点为一点。
2. 两个节点合起来超过了限制，则两点间平均分配 `K` 或 `P` 。

这里截取调整两个非叶子节点的代码展示：

```
void Tree::adjust_two(BNode &now,size_t fir,size_t sec) {
    // fir 和 sec 是 now 的儿子序号
    // 即 now.P[fir] 和 now.P[sec] 就是要调整的两个儿子的地址
    assert(fir + 1 == sec);
    size_t fir_node = now.P.at(fir);
    size_t sec_node = now.P.at(sec);
    BNode one = noter.get_bnode(fir_node);
    BNode two = noter.get_bnode(sec_node);
    if (one.is_leaf) {
        调整叶子;
    } else {
        // 从上向下传递信息,如此实现理顺关系
        one.K.push_back(now.K.at(fir));
        // 先把右侧点的信息全部塞进左侧点
        for (size_t v:two.P) one.P.push_back(v);
        for (const Literal &v:two.K) one.K.push_back(v);
        size_t all = one.P.size();
        if (all <= noter.N) { // 合成一个点满足要求
            // 删除右侧点
            now.K.erase(now.K.begin() + fir);
            now.P.erase(now.P.begin() + sec);
            noter.del_bnode(two);
            noter.write_bnode(one);
            return;
        } else {
            // 均匀重分配
            size_t rig = all / 2;
            size_t lef = all - rig;
            two.K.resize(rig - 1);
            two.P.resize(rig);
            copy(one.K.begin() + lef,one.K.end(),two.K.begin());
            copy(one.P.begin() + lef,one.P.end(),two.P.begin());
            now.K.at(fir) = one.K.at(lef-1); // 从下向上传递信息
            one.P.erase(one.P.begin() + lef,one.P.end());
            one.K.erase(one.K.begin() + lef - 1,one.K.end());
            noter.write_bnode(one);
            noter.write_bnode(two);
            return;
        }
    }
}
```

API

最终 `Indexer` 向外提供了这些 API :

```
void insert_record(const vector<Literal> &tup,size_t pos);
void delete_record(const vector<Literal> &tup,size_t pos);
pair<BNode,size_t> first_leaf_start();
pair<BNode,size_t> first_greater_pos(const Literal &val);
pair<BNode,size_t> first_greater_equal_pos(const Literal &val);
BNode get_bnode(size_t seg);
void init_index();
void remove_index();
```

十分清晰。