哈尔滨工业大学

<<数据库系统>>

实验报告 2

(2022 年度春季学期)

姓名:	王艺丹
学号:	1190201303
学院:	计算机学院
教师:	程思瑶

实验 2

一、实验目的

掌握关系连接操作的实现算法，理解算法的 I/O 复杂性，使用高级语言实现重要的关系连接操作算法。

掌握关系数据库中查询优化的原理，理解查询优化算法在执行过程中的时间开销和空间开销，使用高级语言实现重要的查询优化算法。

二、实验环境

Win 10; MySQL; Python 3.8; Jupyter Notebook

三、实验过程及结果

参考已有的 ExtMem 程序库用 python 编写相关函数：

```
class Buffer:
    def __init__(self, blk_num: int = 8):
        self.io_num = 0   # 磁盘IO次数
        self.blk_num = blk_num   # 缓冲区中可以保存的块数目
        self.free_blk_num = self.blk_num   # 缓冲区中可用的块数目
        self.data_occupy = [False] * self.blk_num   # False表示未被占用
        self.data = [[]] * self.blk_num   # 缓存中按块放置的数据，数据为str类型

    def get_free_blk(self) -> int:
        for idx, flag in enumerate(self.data_occupy):
            if not flag:
                self.data_occupy[idx] = True
                self.free_blk_num -= 1
                return idx
        return -1


    def free_blk(self, index) -> bool:   # 释放缓冲区的一个磁盘块
        flag = self.data_occupy[index]
        if flag:
            self.free_blk_num += 1
            self.data_occupy[index] = False
        return flag

    def load_blk(self, addr: str) -> int:   # 加载磁盘块到缓冲区中，输入参数形如'./disk/relation/r15.blk'
        index = self.get_free_blk()
        if index != -1:
            with open(addr) as f:
                self.data_occupy[index] = True
                self.data[index] = f.read().split('\n')
                self.io_num += 1
        return index
```

```
    def write_blk(self, addr, index):    # 将缓冲区中数据写入磁盘块
        with open(addr, 'w') as f:
            self.io_num += 1
            self.free_blk_num += 1
            self.data_occupy[index] = False
            f.write('\n'.join(self.data[index]))
            return True

    def write_buffer(self, data_lst: list, addr):    # 将CPU处理后的数据暂存入缓冲区，再存入磁盘
        index = self.get_free_blk()
        if index != -1:
            self.data[index] = data_lst
            self.write_blk(addr, index)
        return index != -1
```

```
def drop_blk(addr: str) -> bool:    # 存在返回True，不存在返回False
    blk_path = disk_dir + addr + '.blk'
    blk_exists = os.path.exists(blk_path)
    if blk_exists:
        os.remove(blk_path)
    return blk_exists


def drop_blk_in_dir(file_dir: str):
    for file in os.listdir(file_dir):
        os.remove(file_dir + file)


def gene_data(): # 随机生成R与S
    drop_blk_in_dir(disk_dir)
    all_data, item = [([], set(), blk_num1 * tuple_num, 1, 40), ([], set(), blk_num2 * tuple_num, 20, 60)], None
    for data in all_data:
        for idx in range(data[2]):    # data[2]保存的是关系元组数目
            while True:
                item = (randint(data[3], data[4]), randint(1, 1000))    # data[3]和data[4]保存属性值域上下界
                if item not in data[1]:    # data[1]是一个集合set，用于生成唯一的元组
                    break
            data[0].append(item)    # data[0]用于保存最终结果
            data[1].add(item)
    return all_data[0][0], all_data[1][0]


def write_disk(r_lst: list, s_lst: list):
    all_data = [('r', blk_num1, r_lst), ('s', blk_num2, s_lst)]
    for data in all_data:    # 将关系实例写入模拟磁盘
        for idx in range(data[1]):
            with open('%s%s%d.blk' % (disk_dir, data[0], idx), 'w') as f:
                blk_data = ['%d %d' % item for item in data[2][idx * tuple_num:(idx + 1) * tuple_num]]
                f.write('\n'.join(blk_data))
```

其中随机生成关系函数，利用循环生成指定范围内的随机数实现

1）实现关系选择算法：基于 ExtMem 程序库，使用高级语言实现关系选择算法，选出 R.A=40 或 S.C=60 的元组，并将结果存放在磁盘上

```
def linear_search(buffer: extmem.Buffer):  # 关系选择：线性搜索R.A=40, S.C=60；并将结果写入到磁盘中
    extmem.drop_blk_in_dir(select_dir)  # 删除文件夹下的所有模拟磁盘文件
    two_items, buffer.io_num, count, res = [('r', extmem.blk_num1, 40), ('s', extmem.blk_num2, 60)], 0, 0, []
    for item in two_items:
        for disk_idx in range(item[1]):  # item[1]表示关系占用的物理磁盘块数
            index = buffer.load_blk('%s%s%d.blk' % (disk_dir, item[0], disk_idx))  # 加载磁盘块内容到缓冲区中
            for data in buffer.data[index]:
                data0, data1 = data.split()
                if int(data0) == item[2]:
                    res.append(data)  # item[2]表示关系选择的结果
                    if len(res) == tuple_num:
                        buffer.write_buffer(res, '%s%s%d.blk' % (select_dir, item[0], count))
                        res, count = [], count + 1
            buffer.free_blk(0)
        if res:
            buffer.write_buffer(res, '%s%s%d.blk' % (select_dir, item[0], count))
```

实现思想：

线性顺序遍历所有的关系磁盘块；

将缓冲区的 1 块作为关系数据输入块，1 块作为输出块

磁盘 IO 为： B(r)+B(s)；即 16+32=48

由于将结果写入文件时额外产生了一次 io，所以显示为 49

```
linear_search(buffer)  # 关系选择，线性搜索
print('关系选择的磁盘IO次数为：%d' % buffer.io_num)
```

```
关系选择的磁盘IO次数为：49
```

2）实现关系投影算法：基于 ExtMem 程序库，使用高级语言实现关系投影算法，对关系 R 上的 A 属性进行投影，并将结果存放在磁盘上。

```
def relation_project(buffer: Buffer):  # 关系投影，对R的A属性进行投影，并进行去重
    extmem.drop_blk_in_dir(project_dir)  # 删除文件夹下的所有模拟磁盘文件
    buffer.io_num, res, count, = 0, [], 0  # 投影选择的结果
    all_res = set()  # 去重
    for disk_idx in range(blk_num1):
        index = buffer.load_blk('%sr%d.blk' % (disk_dir, disk_idx))  # 加载磁盘块内容到缓冲区中
        for data in buffer.data[index]:
            if data.split()[0] not in all_res:
                res.append(data.split()[0])
                all_res.add(data.split()[0])
                if len(res) == tuple_num * 2: # tuple_num 每个块最多保存的元组数目
                    buffer.write_buffer(res, '%sr%d.blk' % (project_dir, count))
                    res, count = [], count + 1
        buffer.free_blk(0)
    if res:
        buffer.write_buffer(res, '%sr%d.blk' % (project_dir, count))
```

主要思想：

顺序遍历所有的关系磁盘块；将缓冲区的 1 块作为关系数据输入块，B(buffer)-1 块作为输出块

对结果使用 set()进行去重(可将其视为一个哈希表)

磁盘 IO 为： B(r) ；即 16，将结果写入文件时额外产生了一次 io

```
relation_project(buffer)  # 关系投影
print('关系投影的磁盘IO次数为：%d' % buffer.io_num)
```

```
关系投影的磁盘IO次数为：17
```

3）实现 Nested-Loop Join (NLJ)、hash-join 和 sort-merge-join 算法：基于 ExtMem 程序库，使用高级语言实现以上三种 join 算法，对关系 R 和 S 计算 R.A 连接 S.C，并将结果存放在磁盘上。

使用 SQL 语言完成如下查询：

- **NLJ：**

```
def nested_loop_join(buffer: extmem.Buffer): #基于块
    extmem.drop_blk_in_dir(nlj_dir)  # 删除文件夹下的所有模拟磁盘文件
    res, buffer.io_num, count = [], 0, 0
    for outer_idx in range(ceil(blk_num1 / (blk_num - 2))):  # 关系R做外层for循环内容
        start, end, outer_data = outer_idx * (blk_num - 2), min((outer_idx + 1) * (blk_num - 2), blk_num1), []
        outer_data = [buffer.data[buffer.load_blk('%sr%d.blk' % (disk_dir, idx))] for idx in range(start, end)]
        for inner_idx in range(extmem.blk_num2):  # 关系S做内层for循环内容
            inner_data = buffer.data[buffer.load_blk('%ss%d.blk' % (disk_dir, inner_idx))]
            for outer_lst in outer_data:  # 内存中执行连接操作
                for outer_item in outer_lst:
                    r_a, r_b = outer_item.split()
                    for inner_item in inner_data:
                        s_c, s_d = inner_item.split()
                        if r_a == s_c:
                            res.append('%s %s' % (outer_item, inner_item))
                            if len(res) == int(tuple_num / 2):
                                buffer.write_buffer(res, '%srs%d.blk' % (nlj_dir, count))
                                res, count = [], count + 1
            buffer.free_blk(len(outer_data))
        buffer.data_occupy = [False] * blk_num
    if res:
        buffer.write_buffer(res, '%srs%d.blk' % (nlj_dir, count))  # 将结果磁盘上的剩余数据写入磁盘
```

主要思想：

基于块的嵌套循环连接，**由于 R 块数小于 S，将 R 作为外关系**

缓冲区的 **B(buffer)-2 块作为 R 数据输入块，1 块做 S 数据输入块，1 块作为输出块**；缓冲区满了就立即输出

磁盘 IO 为 B(r)+B(r)*B(s)/(B(buffer)-2)

```
nested_loop_join(buffer)
print('nest-loop-join算法的磁盘IO次数为：%d' % buffer.io_num)
```

```
nest-loop-join算法的磁盘IO次数为：217
```

- **Hash-join**

代码较长，不做赘述，简述主要思想与 IO 分析

划分数据桶：划分为 B(buffer)-1 个数据桶，缓冲区的 B(buffer)-1 块用作数据桶暂存，1 块用作 数据输入

执行连接操作：循环 B(buffer)-1 次，对每个桶执行连接操作；将缓冲区中的 B(buffer-2)块作 为关系 R 的输入块，1 块作为关系 S 的输入快，1 块作为输出块

```
hash_join(buffer)
print('hash-join算法的磁盘IO次数为：%d' % buffer.io_num)
```

```
hash-join算法的磁盘IO次数为：148
```

- **SMJ：**

代码较长，不做赘述，简述主要思想与 IO 分析

由于(B(Buffer)-1)^2<16*32，所以选取两路归并;

先进行块内排序，再进行块间排序

块内排序：缓冲区 B(buffer)-1 块作为数据输入，1 块作为数据输出；故写入磁盘的结果中每 B(buffer)-1 块是有序的

块间排序：缓冲区 1 块作为整体有序数据输出块，其余块作为输入块，依次取有序磁盘块中的 1 块，不断选取其中最小的元组

连接操作：设置 2 个游标不断滑动，若从 R 中取到的数据较小，则滑动 R 的游标；若较大，则滑 动 S 的游标；否则，输出该结果，并生成 2 个临时游标，临时滑动关系 R 或 S 的数据直到两者数 据不相同。最终原游标分别滑动一个元素

```
sort_merge_join(buffer)
print('sort-merge-join算法的磁盘IO次数为：%d' % buffer.io_num)
```

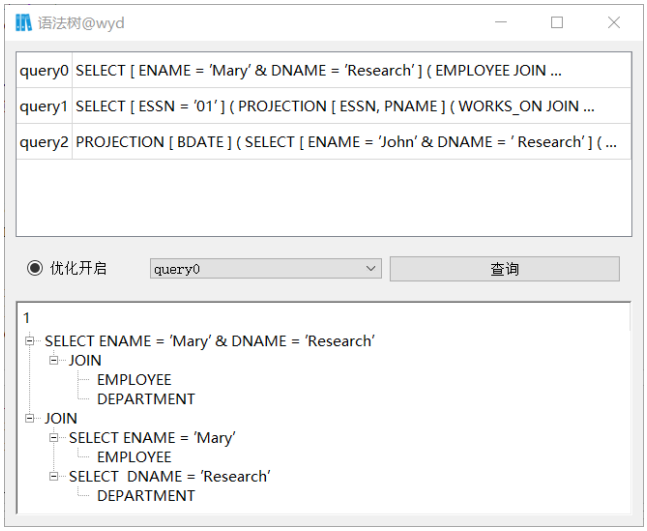
```
sort-merge-join算法的磁盘IO次数为：422
```

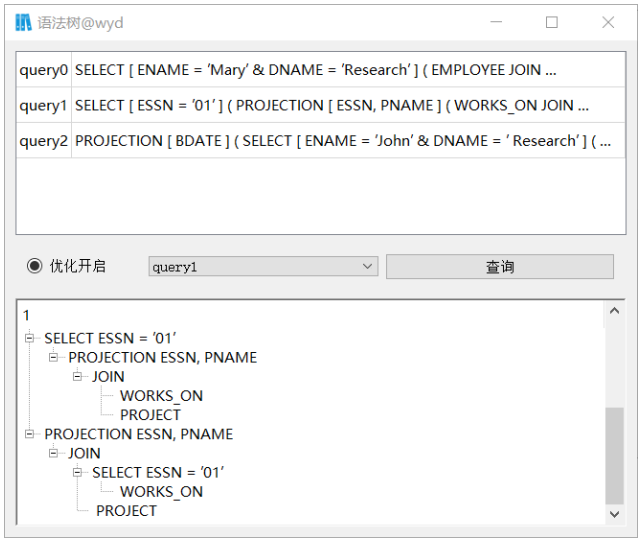
查询优化

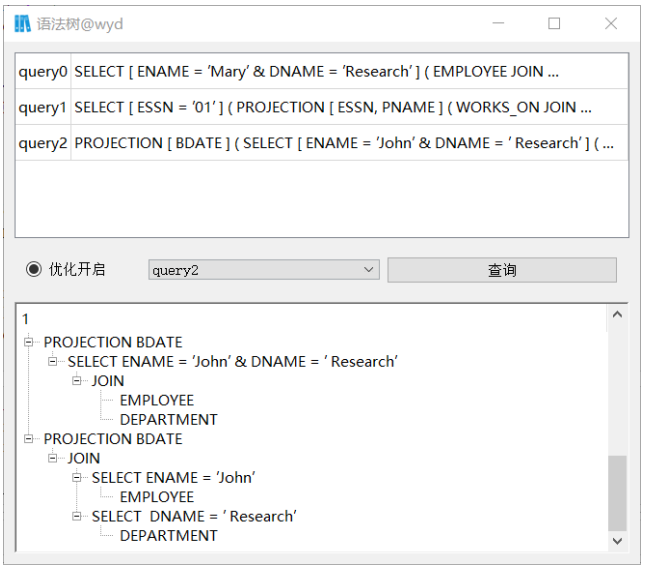
主要思想：

将 select 选择操作和投影操作尽可能地移向查询树的叶节点

各查询语句优化前后对比：







四、实验心得

- 掌握了各连接操作的具体执行过程与实现
- 理解了 IO 复杂性
- 代码书写过程中进一步认识到哈希表/缓冲区内存大小需要进行考虑
- 理解了查询优化算法在执行过程中的时间开销和空间开销