哈尔滨工业大学

<<数据库系统>>

实验报告二

(2023 年度秋季学期)

姓名：	陈俊乐
学号：	2021113694
学院：	未来技术学院
教师：	李东博

实验二 关系连接与查询优化算法设计与实现

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

Windows10，jdk20，gcc， gdb,

本实验在关系连接实现时候使用 C 语言，用 extmem,c，extmem.h 所给的代码，在实现语法分析器和查询优化方法实现的时候使用的是 Java 语言。

三、实验过程及结果

3.1 数据的生成与存储

1. **数据生成与属性：**
   - 关系 R 包含属性 A 和 B，每个属性是一个整数（int 型，4 个字节）。
     - A 的值域在[1, 40]之间。
     - B 的值域在[1, 1000]之间。
   - 关系 S 包含属性 C 和 D，同样是整数型。
     - C 的值域在[20, 60]之间。
     - D 的值域在[1, 1000]之间。
2. **元组数量和数据块结构（一个数据块大小为 64 字节）：**
   - 关系 R 有 112 个元组（每个元组 8 字节），分布在 16 个数据块中（每个块 7 个元组，共需 56 字节，最后八个字节包含下一块的地址）。
   - 关系 S 有 224 个元组（每个元组 8 字节），分布在 32 个数据块中（每个块 7 个元组，共需 56 字节，最后八个字节包含下一块的地址）。
3. **随机数据生成：**
   - 使用 **rand()**函数生成每个属性的值。通过取模运算来确保值在指定的范围内。
4. **数据块的创建和写入：**
   - 使用 **getNewBlockInBuffer** 来创建一个新的数据块。

- 数据块中，每个元组的数据按顺序写入，占据前 56 个字节。
- 每个数据块的最后 4 个字节用于存储下一个数据块的地址。

5. **数据块的链接：**
   - 每个块的最后 4 个字节包含下一个数据块的地址。
   - 对于关系 R，下一个块的地址从 0 开始，最后一个块存储-1 作为结束标记。
   - 对于关系 S，下一个块的地址从 16 开始，同样最后一个块存储-1。
6. **数据块的存储：**
   - 使用 **writeBlockToDisk** 函数将每个数据块写入磁盘。
   - 关系 R 的数据块地址从 0 开始，总共 16 块
   - 关系 S 的数据块地址从 16 开始，总共 32 块。

3.2 关系选择算法的实现方法

选择算法的核心是根据特定的选择条件（选择的数字）来处理数据，在实现过程中，遍历所有元组，找到符合筛选条件的元组，打印出来，并存储到磁盘中。

1. **参数与初始化：**
   - Buffer buf：表示用于管理内存中数据块的缓冲区。
   - int addr：指定开始搜索的数据块的地址。ADDRR 和 ADDRS 分别代表关系 R 和 S 的开始地址，通过 addr 来指定关系并进行选择。
   - int select：要筛选的特定属性值。
   - int index：指定要筛选的属性，0 和 1 分别代表第一个和第二个属性，如 R 关系里面 A 表示第一个属性，B 表示第二个属性
2. **筛选条件打印：**
   - 根据 addr 和 index 确定在关系 R 还是 S 中选择哪个属性，并打印相应的信息。
3. **处理数据块：**
   - 使用循环遍历关系中的所有数据块。
   - 使用 readBlockFromDisk 函数从磁盘读取当前数据块。
   - 为存放筛选结果申请一个新的数据块 outBlk。
4. **元组筛选与结果存储：**
   - 在每个数据块中，遍历每个元组。
   - 使用 memcpy 读取元组中相应属性的值。
   - 如果这个值等于 select，则将此元组复制到 outBlk 中，并在控制台打印此元组。
5. **写入结果块：**
   - 将 outBlk 写入磁盘，地址从 500 开始递增。
6. **链式结构处理：**
   - 在每个数据块的末尾读取下一个数据块的地址。
   - 如果下一个地址不是-1，则继续读取下一个数据块。
7. **资源释放：**
   - 处理完每个数据块后，释放该数据块。
8. **函数调用示例：**
   - selectOperation(buf, ADDRR, 40, 0); 选择关系 R 中 A 属性值等于 40 的所有元组。

- selectOperation(buf, ADDRS, 60, 0); 选择关系 S 中 C 属性值等于 60 的所有元组。

3.3 关系投影算法的实现方法

基本思想

关系投影算法主要用于从一个关系中提取（投影）指定的属性列。它逐个遍历关系中的所有元组，提取感兴趣的属性，去掉重复值，并将结果存储在新的数据块中。

代码实现方法

1. 初始化：
- 分配一个新的块 **outBlk** 来存储投影结果。
- 创建一个数组 **projected** 来存储提取出的属性值。

2. 遍历关系：
- 算法遍历指定关系（例如关系 R）的所有数据块。
- 对于每个数据块，从磁盘读取该块，并获取下一个数据块的地址。

3. 提取指定属性：
- 在每个数据块中，算法遍历每个元组。
- 对于每个元组，算法提取指定的属性（按照实验要求，投影 R.A 属性）。
- 同时检查前面存储的数是否与当前属性值相同，如果相同的话，就不存储和打印，因为投影操作需要去掉重复值
- 提取的值被存储在 **projected** 数组中。

4. 处理投影结果：
- 生成的投影结果（即 **projected** 数组中的值）被写入到新分配的数据块 **outBlk** 中。
- 完成后，这个数据块被写入磁盘。

5. 打印投影结果：
- 算法打印出投影后的属性值，以展示投影操作的结果。

6. 资源管理：
- 在处理完每个数据块后，释放该数据块以避免内存泄漏。

3.4 Nested-Loop Join 算法的实现方法

基本思想：

1.遍历：嵌套循环连接通过遍历一个关系（称为外关系）的每个元组，并对每个元组执行一个内部循环来遍历另一个关系（称为内关系），从而找到符合连接条件的元组对。
2.连接条件：连接条件通常是两个关系中的属性之间的等值比较，例如，R.A = S.C。

代码中的实现方法：

1. 初始化输出块：创建一个新的数据块来存储连接结果。
2. 外层循环（关系 R）：

- 遍历关系 R 的每个数据块。
- 从磁盘读取当前数据块，并获取下一个数据块的地址。

3. 内层循环（关系 S）：
   - 对于关系 R 的每个数据块，遍历关系 S 的每个数据块。
   - 同样从磁盘读取数据块，并获取下一个数据块的地址。
4. 检查连接条件：
   - 对于关系 R 中当前数据块的每个元组，检查与关系 S 中当前数据块的每个元组的连接条件（代码中是 R.A = S.C）。
5. 输出匹配的元组对：
   - 若找到符合连接条件的元组对，则将这些元组的数据复制到输出块中。
   - 当输出块满时，将其写入磁盘，并分配新的块继续存储。
6. 循环控制：
   - 继续外层循环，处理关系 R 的下一个数据块，同时重置内层循环以重新遍历关系 S。
7. 资源管理：
   - 在处理完每个数据块后释放内存，以避免内存泄漏。

3.5 Hash Join 算法的实现方法

基本思想：

Hash Join 算法通过使用哈希函数将一个关系的元组分配到哈希桶中，然后通过对另一个关系的元组应用相同的哈希函数来快速定位可能的匹配元组。这种方法减少了需要比较的元组数量，特别是当两个关系的大小相差很大时，可以大幅提高连接操作的效率。不过，该算法的性能依赖于哈希函数的质量和哈希桶的大小，不当的选择可能导致性能下降。

代码实现方法

1. 初始化哈希桶和输出块：

- 创建哈希桶用于存储来自一个关系（例如关系 R）的元组。
- 分配一个新的块 **outBlk** 来存储 Join 结果。

2. 填充哈希桶：

- 遍历关系 R 的所有数据块。
- 对于每个元组，使用哈希函数（基于元组的连接属性）计算其哈希值。
- 根据哈希值将元组存储在对应的哈希桶中。

3. 处理第二个关系（S）：

- 遍历关系 S 的所有数据块。
- 对于关系 S 中的每个元组，也使用相同的哈希函数计算哈希值。

4. 在哈希桶中查找匹配元组：

- 对于关系 S 中的每个元组，根据其哈希值访问对应的哈希桶。
- 比较桶中的每个元组与当前处理的关系 S 的元组，以检查是否满足连接条件（实验要求是 R.A = S.C）。

5. 记录匹配的元组对：

- 如果找到匹配的元组对（即满足连接条件的元组），则将这些元组的数据复制到输出块 **outBlk** 中。
- 当输出块填满时，将其写入磁盘，并分配新的块继续存储。

6. 打印和存储结果：

- 打印出符合连接条件的元组对。
- 将结果写入磁盘以进行持久化存储。

7. 资源管理：

- 在处理完每个数据块后释放内存，以避免内存泄漏。

3.6 sort-merge-join 算法的实现方法

基本思想

1. **排序**：首先对两个参与连接的关系进行排序，基于要连接的属性（例如，R.A 和 S.C）。
2. **合并**：然后通过合并步骤来寻找和生成匹配的元组对。由于关系已经排序，算法可以高效地同步遍历两个关系来寻找匹配的元组。

代码中的实现方法

1. **排序两个关系：**
   - 为关系 R 和 S 中的元组创建两个数组 **arrR** 和 **arrS**。
   - 使用 **sortRelation** 函数分别对这两个
2. **数组中的元组进行排序：**排序依据是要连接的属性（例如，R.A 和 S.C）。
3. **初始化输出块：**
   - 创建一个新的块 **outBlk** 用于存储 join 结果。
4. **同步遍历两个排序数组：**
   - 同时遍历两个排序后的数组 **arrR** 和 **arrS**。
   - 使用两个指针（或索引）**i** 和 **j** 分别指向两个数组的当前元组。
5. **查找匹配的元组对：**
   - 如果 **arrR[i]** 和 **arrS[j]** 的连接属性相等（例如，R.A = S.C），则处理所有具有相同连接属性值的元组对，并将匹配的元组对复制到输出块中。
   - 如果 **arrR[i]** 的连接属性小于 **arrS[j]** 的连接属性，增加 **i**，反之增加 **j**。
6. **处理输出块：**
   - 当输出块填满时，将其写入磁盘，并分配新的块继续存储后续的 join 结果。
7. **打印和存储结果：**
   - 打印出匹配的元组对并将它们存储到磁盘。
8. **资源管理：**
   - 在处理完成后释放分配的资源，包括排序后的数组和数据块。

3.7 语法分析器的实现方法

这个分析器的工作方式是一个典型的递归下降解析器，它根据表达式的结构递归地构建一个操作树。

代码实现方法

1. **递归下降解析**: 递归下降解析是一种编译原理中常用的技术，用于解析具有层次结构的表达式。这种方法通过递归地调用解析函数来匹配输入字符串的结构，并构造

出相应的内部表示（在这个例子中是操作树）。

2. **表达式处理**:
    - **trim()**: 首先去除表达式两边的空格，确保处理的是紧凑的字符串。
    - **startsWith**: 检查表达式的类型（SELECT, PROJECTION, JOIN, 或 NODE）。
    - **substring**: 提取出操作符的参数。
3. **操作类型**:
    - **SELECT**: 处理选择操作（**SELECT[条件](子表达式)**）。从中提取条件和子表达式，然后递归解析子表达式。
    - **PROJECTION**: 处理投影操作（**PROJECTION[条件](子表达式)**）。同样提取条件和子表达式，进行递归解析。
    - **JOIN**: 处理连接操作（**左表达式 JOIN 右表达式**）。将表达式分为两部分，分别递归解析左右子表达式。
    - **NODE**: 如果表达式不是以上类型，则视为单个节点。
4. **构建操作树**:
    - 每一步解析都会创建一个 **RelationalAlgebraOperator** 对象，代表当前操作。
    - 这个对象包含操作类型、条件（如果有）、以及子操作对象（如果有）。
    - 对于 JOIN 操作，会有两个子操作对象。
5. **递归结束条件**:
    - 当表达式不包含任何复杂操作符（如 SELECT, PROJECTION, JOIN），递归将结束。

3.8 查询优化方法

查询优化使用的方法是选择下推，它的主要目的是尽早应用选择操作（即过滤条件），从而减少后续操作需要处理的数据量。在关系代数查询中，这通常意味着将选择操作从查询树的较高位置移动到更接近数据源的位置。

实现方法

1. **递归优化子节点**:
    - 方法首先对当前节点的左子节点 **(node.left)** 应用优化（**optimize(node.left)**）。这保证了在进行选择下推之前，所有子节点都已经被优化过。
2. **处理 JOIN 操作**:
    - 如果当前节点的左子节点是一个 JOIN 操作（**node.left.type == OperatorType.JOIN**），则尝试将选择条件下推到 JOIN 的左右子节点。
    - 选择条件（**node.condition**）被分解为多个条件，通常是基于逻辑与（**&**）操作符。
    - 对于每个分解后的条件，通过 **pushDownSELECT** 方法分别下推到 JOIN 的左（**node.left.left**）和右（**node.left.right**）子节点。
    - 选择下推完成后，原来的 SELECT 节点被移除，返回优化后的 JOIN 节点。
3. **处理 PROJECTION 操作**:
    - 如果当前节点的左子节点是一个 PROJECTION 操作（**node.left.type == OperatorType.PROJECTION**），则将选择条件下推到 PROJECTION 的子

节点。

- 创建一个新的 SELECT 节点，其子节点是原 PROJECTION 节点的子节点，条件是当前 SELECT 节点的条件。
- 这意味着 SELECT 操作被移动到更接近数据源的位置。

3.9 生成的查询执行树和优化后的查询执行树

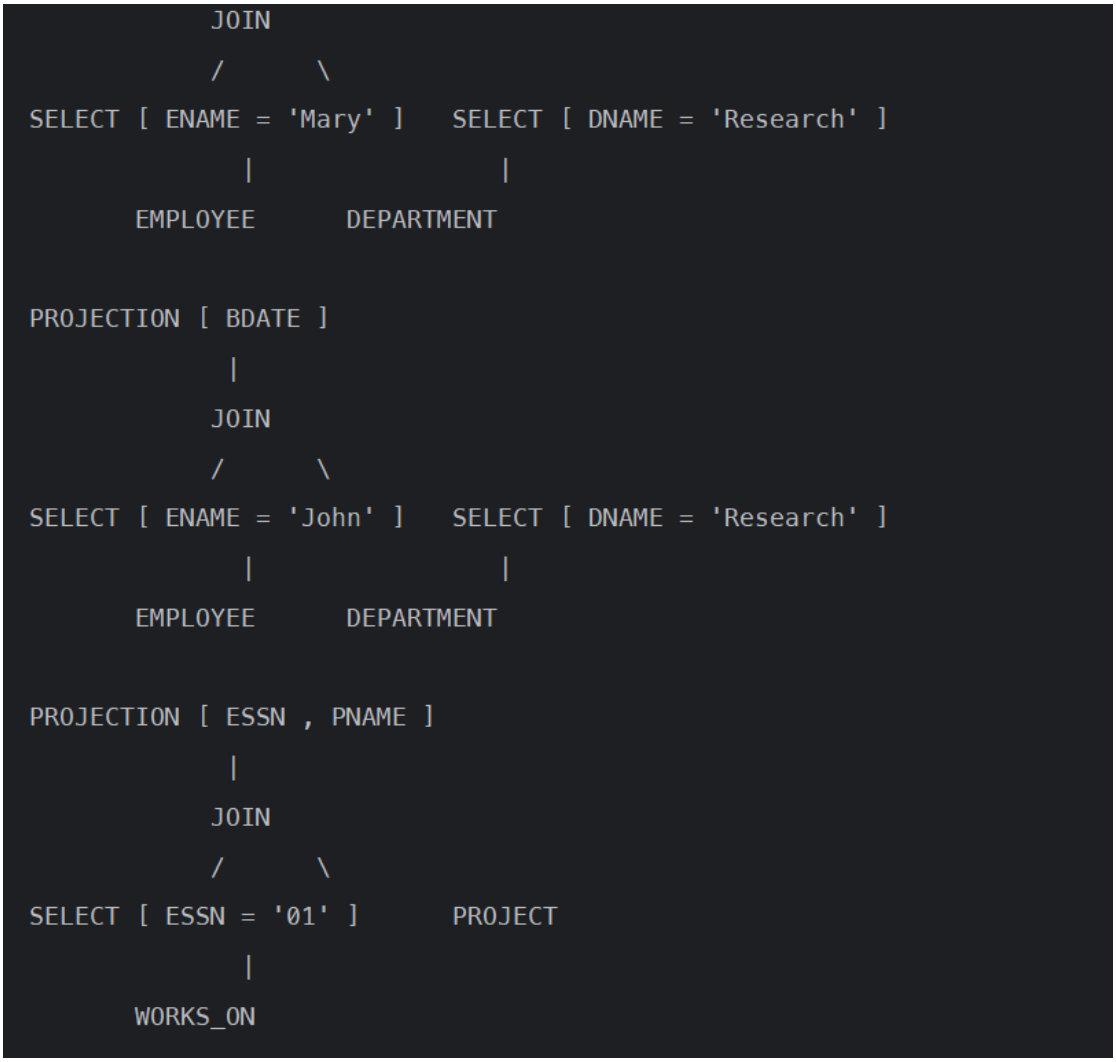
生成的查询执行树

```
SELECT [ ENAME = 'Mary' & DNAME = 'Research' ]
             |
            JOIN
            /      \
       EMPLOYEE      DEPARTMENT

PROJECTION [ BDATE ]
             |
SELECT [ ENAME = 'John' & DNAME = 'Research' ]
             |
            JOIN
            /      \
       EMPLOYEE      DEPARTMENT

SELECT [ ESSN = '01' ]
             |
PROJECTION [ ESSN , PNAME ]
             |
            JOIN
            /      \
       WORKS_ON      PROJECT
```

优化的查询执行树



3.10 最关键代码

1.数据生成 (generateData 函数)

```
// 生成关系R的数据,包含16 * 7 = 112个元组
// R具有两个属性A和B，其中A和B的属性值均为int型（4个字节）。A的值域为[1, 40]，B的值域为[1, 1000]。
for (int i = 0; i < 16; i++)
{
    blk = getNewBlockInBuffer(&buf);
    for (int j = 0; j < 7; j++)
    {
        int a = rand() % 40 + 1;
        int b = rand() % 1000 + 1;
        memcpy(blk + j * 8, &a, 4);
        memcpy(blk + j * 8 + 4, &b, 4);
    }

    int nextBlkAddr_R = (i == 15) ? -1 : i+1;

    memcpy(blk + 56,&nextBlkAddr_R,sizeof(int)); //写入下一块的地址
    writeBlockToDisk(blk,i,&buf); //R块地址从0开始


}
```

```
//生成关系S的数据,包含32 * 7 = 224个元组
//关系S具有两个属性C和D，其中C和D的属性值均为int型（4个字节）。C的值域为[20, 60]，D的值域为[1, 1000]。
for (int i = 0; i < 32; i++)
{
    blk = getNewBlockInBuffer(&buf);
    for (int j = 0; j < 7; j++)
    {
        int c = rand() % 41 + 20;
        int d = rand() % 1000 + 1;
        memcpy(blk + j * 8, &c, 4);
        memcpy(blk + j * 8 + 4, &d, 4);
    }
    int nextBlkAddr_S = (i == 31) ? -1 : i+17;
    memcpy(blk+56,&nextBlkAddr_S,sizeof(int)); //写入下一块的地址
    writeBlockToDisk(blk,i+16,&buf); //S块地址从16开始

}
```

生成关系 R 的数据,包含 16 * 7 = 112 个元组，R 具有两个属性 A 和 B，其中 A 和 B 的属性值均为 int 型（4 个字节）。A 的值域为[1, 40]，B 的值域为[1, 1000]。

生成关系 S 的数据,包含 32 * 7 = 224 个元组，关系 S 具有两个属性 C 和 D，其中 C 和 D 的属性值均为 int 型（4 个字节）。C 的值域为[20, 60]，D 的值域为[1, 1000]。

2.Hash Join 中桶的填充和连接

```
// 填充R关系的哈希桶
do {
    blkR = readBlockFromDisk(addrR, buf);
    for (int i = 0; i < 7; i++) {
        int a = *(int *)(blkR + 8 * i);
        int b = *(int *)(blkR + 8 * i + 4);
        int hashValue = hashFunction(a);
        //当桶装得下元组的时候，就把元组放进去
        if (bucketCountR[hashValue] < TUPLES_PER_BLOCK) {
            bucketR[hashValue][bucketCountR[hashValue]].first = a;
            bucketR[hashValue][bucketCountR[hashValue]].second = b;
            bucketCountR[hashValue]++;
        }
    }
    addrR = *(int *)(blkR + 56);
    freeBlockInBuffer(blkR, buf);
} while (addrR != -1);
```

首先填充 R 关系的桶，通过 Hash 值存在对应的桶里面，同时每个桶最大只能容纳 TUPLES_PER_BLOCK（这里是 7 个元组），填充完看以后在连接 R.A 和 S.C 的时候就可以通过 hash(S.C)找到对应的桶，然后遍历桶里的元组，满足连接条件就连接。

```
// hash join
do {
    blkS = readBlockFromDisk(addrS, buf);
    for (int i = 0; i < 7; i++) {
        int c = *(int *)(blkS + 8 * i);
        int d = *(int *)(blkS + 8 * i + 4);
        int hashValue = hashFunction(c);
        for (int  k = 0; k < bucketCountR[hashValue]; k++) {
            int a = bucketR[hashValue][k].first;
            //满足R.A = S.C
            if (a == c)
            {
                int b = bucketR[hashValue][k].second;
                printf("|%4d|%4d|%4d|%4d|\n", a, b, c, d);
                memcpy(outBlk + numTuple * 16, &a, 4); // 写入R.A
                memcpy(outBlk + numTuple * 16 + 4, &b, 4); // 写入R.B
                memcpy(outBlk + numTuple * 16 + 8, &c, 4); // 写入S.C
                memcpy(outBlk + numTuple * 16 + 12, &d, 4); // 写入S.D
                numTuple++;
                if (numTuple == 4) {
                    writeBlockToDisk(outBlk, outAddr++, buf);
                    numTuple = 0;
                    outBlk = getNewBlockInBuffer(buf);
                }
            }

        }
        }

    addrS = *(int *)(blkS + 56);
    freeBlockInBuffer(blkS, buf);
} while (addrS != -1);
```

3.merge-sort-join 算法

一开始先对关系分成多个段，对每个段使用 qsort 排序，以便后续加速连接。排序方法是按照元组的第一个元素进行排序，因为本实验要求的是对 R.A 和 S.C 进行连接，所以我进行了对 R 中 A 和 S 中 C 的元素按照大小进行排序。

```
int compareTuples(const void *a, const void *b) {
    Tuple *tupleA = (Tuple *)a;
    Tuple *tupleB = (Tuple *)b;
    if (tupleA->first < tupleB->first) return -1;
    if (tupleA->first > tupleB->first) return 1;
    return 0;
}
//*args: buf,addr,arr,len
//*addr:关系的起始地址
//*arr:存放关系的数组
//*len:关系的个数
//*排序从addr开始的关系，关系的个数为len。排序结果存在arr数组中
void sortRelation(Buffer *buf, int addr, Tuple * arr,int len) {
    unsigned char *blk;
    int i = 0;
    do {
        blk = readBlockFromDisk(addr, buf);
        for (int j = 0; j < 7; j++)
        {
            arr[i].first = *(int *)(blk + j * 8);
            arr[i].second = *(int *)(blk + j * 8 + 4);
            i++;
        }
        addr = *(int *)(blk + 56);
        freeBlockInBuffer(blk, buf);
    } while (addr != -1 && i < len);
    qsort(arr, len, sizeof(Tuple), compareTuples);


}
```

4 查询分解

```
class RelationalAlgebraParser {
    public RelationalAlgebraOperator parse(String expression) {
        expression = expression.trim();//去除空格
        //*递归地分解 */
        if (expression.startsWith("SELECT")) {
            int start = expression.indexOf("[") + 1;
            int end = expression.indexOf("]");
            String condition = expression.substring(start, end).trim();
            String subExpr = expression.substring(end + 1).trim().replace("(", "").replace(")", "");
            return new RelationalAlgebraOperator(OperatorType.SELECT, condition, parse(subExpr), null);
        } else if (expression.startsWith("PROJECTION")) {
            int start = expression.indexOf("[") + 1;
            int end = expression.indexOf("]");
            String condition = expression.substring(start, end).trim();
            String subExpr = expression.substring(end + 1).trim().replace("(", "").replace(")", "");
            return new RelationalAlgebraOperator(OperatorType.PROJECTION, condition, parse(subExpr), null);
        } else if (expression.contains("JOIN")) {
            int joinIndex = expression.indexOf("JOIN");
            String leftExpr = expression.substring(0, joinIndex).trim();
            String rightExpr = expression.substring(joinIndex + 4).trim();
            return new RelationalAlgebraOperator(OperatorType.JOIN, null, parse(leftExpr), parse(rightExpr));
        }
        return new RelationalAlgebraOperator(OperatorType.NODE, expression, null, null);
    }
}
```

1. **递归下降解析**：递归下降解析是一种编译原理中常用的技术，用于解析具有层次结构的表达式。这种方法通过递归地调用解析函数来匹配输入字符串的结构，并构造出相应的内部表示（在这个例子中是操作树）。
2. **表达式处理**：
   - **trim()**：首先去除表达式两边的空格，确保处理的是紧凑的字符串。
   - **startsWith**：检查表达式的类型（SELECT, PROJECTION, JOIN, 或 NODE）。
   - **substring**：提取出操作符的参数（例如条件、子表达式）。
3. **操作类型**：
   - **SELECT**：处理选择操作（**SELECT[条件](子表达式)**）。从中提取条件和子表达式，然后递归解析子表达式。
   - **PROJECTION**：处理投影操作（**PROJECTION[条件](子表达式)**）。同样提取条件和子表达式，进行递归解析。
   - **JOIN**：处理连接操作（**左表达式 JOIN 右表达式**）。将表达式分为两部分，分别递归解析左右子表达式。
   - **NODE**：如果表达式不是以上类型，则视为单个节点。
4. **构建操作树**：
   - 每一步解析都会创建一个 **RelationalAlgebraOperator** 对象，代表当前操作。
   - 这个对象包含操作类型、条件（如果有）、以及子操作对象（如果有）。
   - 对于 JOIN 操作，会有两个子操作对象。
5. **递归结束条件**：
   - 当表达式不包含任何复杂操作符（如 SELECT, PROJECTION, JOIN），递归将结束。

5.选择下推

```
private RelationalAlgebraOperator pushDownSELECT(RelationalAlgebraOperator node, String condition) {
    //TODO:提取属性名字，也就是等于号左边的字符串    TODO:提取属性名字，也就是等于号左边的字符串
    String field = extractNameFromCondition(condition);
    if (node == null) {
        return null;
    }
    // 递归地处理 PROJECTION 节点
    if (node.type == OperatorType.PROJECTION) {
        node.left = pushDownSELECT(node.left, condition);
        return node;
    }
    // 检查节点关联的关系是否包含该字段
    if (node.type == OperatorType.NODE && isInProperties(field, node.condition)) {
        // 为合适的分支创建新的 SELECT 节点
        return new RelationalAlgebraOperator(OperatorType.SELECT, condition, node, null);
    }

    return node;
}
```

1. **提取属性名**：
   - 方法开始时，通过调用 **extractNameFromCondition** 函数来从选择条件（**condition**）中提取属性名称。

2. **递归处理**:
    - 如果传入的节点（**node**）为空，函数直接返回 **null**。这是递归的基本终止条件。
    - 如果节点类型是 **PROJECTION**，则对其左子节点递归调用 **pushDownSELECT** 方法。这意味着选择操作尝试继续向下推到查询树的更深层次。
3. **检查并创建新的 SELECT 节点**:
    - 如果当前节点是普通的节点（**OperatorType.NODE**）并且关联的关系包含了选择条件中的字段（通过 **isInProperties** 方法检查），那么会在这个节点上创建一个新的 SELECT 节点。
    - 新创建的 SELECT 节点将包含原始的选择条件（**condition**）和当前的节点作为其子节点。
4. **返回处理后的节点**:
    - 如果上述条件不满足，函数将返回原始节点，表示无需进一步下推选择操作。

四、实验心得

遇到问题：

实验代码所提供的 extmem.c 中在 windows 下读取块会出现问题，原因是在 Windows 系统中，文本模式和二进制模式的处理方式不同。特别是，文本模式会对特定字符进行特殊处理：

- 自动将写入文件的 **\n** (换行) 转换为 **\r\n** (回车换行)。
- 读取文件时，将 **\r\n** 转换回 **\n**。
- 遇到 EOF（文件结束符）可能会提前结束读写操作。

解决问题：

因此在读和写上改成以二进制形式读和写即可，即通过在 fopen 的模式字符串中加上 'b' 来实现。

心得体会：

本次实验中，我深入探究了关系数据库的核心操作：选择、投影和连接。通过实现和测试这些操作，我不仅加深了对关系代数理论的理解，还学习了如何在实践中应用这些理论。此外，我还接触了查询树和查询优化树的概念，这对于理解数据库查询执行的内部机制非常有益。

这次实验不仅提高了我对关系数据库操作的理解，还让我深刻领会到了查询优化的重要性。通过动手实践，我学会了如何构建更高效的数据库查询，这对我的学习和未来的工作都大有裨益。这次实验经历将成为我在数据库领域学习旅程中的一个宝贵财富。