

---

# 哈尔滨工业大学

## <<数据库系统>>

### 实验报告三

(2023 年度春季学期)

姓名:	
学号:	
学院	
教师:	

## 实验三

### 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。

### 二、实验环境

Windows 操作系统、MinGW 编译器或 Microsoft Visual C++ 编译器。C、C++，JAVA 等语言均可。

### 三、实验过程及结果

#### (1) 关系连接算法的实现

##### ①数据生成

根据任务要求，关系 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]；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]。

利用 Math.random 方法生成 0 到 1 间的随机数，再进一步计算得到符合值域要求的数据。

```
1. dataR[i]=(int)(Math.random()*39+1)+" "+(int)(Math.random()*999+1);  
2. dataS[i]=(int)(Math.random()*40+20)+" "+(int)(Math.random()*999+1);
```

##### ②关系选择算法的实现

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。这部分任务的实现采用了线性查找的方法，即逐个遍历 R 和 S 中的元素，找出符合要求的元组存放到磁盘中。在每轮查找中，首先从缓冲区中读入 R 或 S 的数据，经过筛选得到符合要求的元组后，将元组存入缓冲区的特定的块中，在该块装满后再写入磁盘，若遍历结束后块中还剩余有内容，将这部分数据也写入。

实现过程如下：

首先通过 extmem 中的 loadBlock 方法将指定路径下的磁盘块数据读入缓冲

区，并返回数据存储位置的缓冲区块号。

```
1. String path="task1\\dir\\relation\\"+"R"+i+".blk";
2. int index=buffer.loadBlock(path);
```

逐个遍历该块中的数据，选出符合要求的元组添加到缓冲区负责写入磁盘的块中，若负责写入磁盘的块已满则及时将数据写入磁盘并释放该块；遍历结束后，将存储数据的块释放，读入下一轮数据。

```
1. if(TupleNumR==7){
2.     String writePath="task1\\dir\\select\\"R"+countR+".blk";
3.     buffer.writeBuffer(resR,writePath);
4.     resR=new String[7]
5.     TupleNumR=0;
6.     countR++;
7. }
```

## ②关系投影算法的实现

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。这部分任务的实现与上述线性查找的思路相同，将 R 中数据读入到缓冲区，逐个遍历 R 中的元组，选出其中的 A 属性，再将其写入到磁盘当中。

实现过程如下：

首先通过 extmem 中的 loadBlock 方法将指定路径下的磁盘块数据读入缓冲区，并返回数据存储位置的缓冲区块号。

```
3. String path="task1\\dir\\relation\\"+"R"+i+".blk";
4. int index=buffer.loadBlock(path);
```

通过 split 方法将元组拆分从而得到 R 中的 A 属性，再将数据存入到缓冲区负责写入磁盘的块中。

```
1. String data1=data.split(" ")[0];
2. resR[TupleNumR]=data1;
3. TupleNumR++;
```

## ③Nested-Loop Join (NLJ)算法的实现

基于 ExtMem 程序库，对关系 R 和 S 计算 R.A 连接 S.C，并将结果存放在磁盘上。NLJ 算法是通过一次一行循环地从第一张表中读取行，在这行数据中取到关联字段，根据关联字段在另一张表里取出满足条件的行，然后取出两张表的结果合集。由于缓冲区的大小为 8 个块，因此在该任务中，选用 6 个块用于读取 R 中的数据，1 个块用于读取 S 中的数据，1 个块用于将数据写入磁盘。

实现过程如下：

首先读取 R 中的数据，一次读取 6 个块的内容。

```
1. int start=i*6;
```

```

2. int end=Math.min((i+1)*6,16);
3. String[][] dataR=new String[end-start][];
4. for(int j=start;j<end;j++){
5.     String path="task1\\dir\\relation\\"+"R"+j+".blk";
6.     String[] data=buffer.data[buffer.loadBlock(path)];
7.     dataR[j-start]=data;
8. }

```

然后每轮读取 S 中的一个块到缓冲区中，遍历缓冲区中 R 的数据，判断 S.C 是否其中某个 R.A 相同，将符合要求的数据添加到负责写入磁盘的缓冲区块中，并及时进行写入。

```

1. for(String itemR:R){
2.     if(itemR==null){continue;}
3.     String R_a=itemR.split(" ")[0];
4.     for(String itemS:dataS){
5.         if(itemS==null){continue;}
6.         String S_c=itemS.split(" ")[0];
7.         if(Objects.equals(R_a, S_c)){

```

#### ④hash-join 算法的实现

Hash-join 的大致流程为使用两个表中较小的表利用 Join Key 在内存中建立散列表，然后扫描较大的表并探测散列表，找出与 Hash 表匹配的行。由于缓冲区的大小为 8 个块，因此选用了 6 个块作为哈希桶，1 个块用于从磁盘中读入 R 或 S 的数据，剩余的 1 个块用于将哈希结果写入磁盘。

实现过程如下：

首先将缓存区中的数据读入到缓冲区，根据哈希值将其存入到相应的哈希桶中。在哈希函数的选择上，采用了将 R.A 或 S.C 的值对 6（哈希桶数量）取模作为哈希值。在桶装满后，将结果写入到磁盘中，对于同一个编号桶中先后写入到磁盘的数据，额外加一个编号以区分，例如 R12 表示是编号为 1 的哈希桶所写入到磁盘中的编号为 2 的结果。

```

1. int hashIndex=Integer.parseInt(item.split(" ")[0])%hash_num;
2. hashBlock[hashIndex][countR[hashIndex]]=item;
3. countR[hashIndex]++;

```

在将数据装入哈希桶并写入磁盘后，开始进行连接操作。依次取出 R 和 S 相同哈希编号的哈希桶中的内容(由于缓冲区的大小只有 8 块，因此相同哈希编号桶的 R 和 S 的数量最多只能有 7 个，如果无法满足该条件，则需要调整哈希函数，使得数据能够较为均匀的分布在各个桶中)，并选出符合条件的数据存储缓冲区并将结果写入磁盘。

```

1. for(String rData:rBufferData){
2.     for(String sData:tempData){
3.         if(rData==null||sData==null){continue;}

```

```

4.      String R_a=rData.split(" ")[0];
5.      String S_c=sData.split(" ")[0];
6.      if(Objects.equals(R_a,S_c))

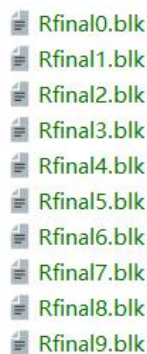
```

### ⑤sort-merge-join 算法的实现

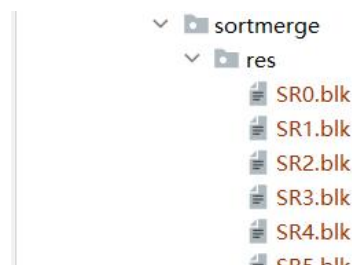
排序合并连接是嵌套循环连接的变种。如果两个数据集还没有排序，那么数据库会先对它们进行排序，这就是所谓的 sort join 操作。对于数据集里的每一行，数据库会从上一次匹配到数据的位置开始探查第二个数据集，这一步就是 Merge join 操作。因此该算法的实现主要分为两个步骤，排序与合并。

实现过程如下：

首先实现的是排序任务，这里采用了归并排序的方法。由缓冲区每次读入 7 个块大小的数据，对这 7 个块的数据进行排序之后，通过缓冲区用于写入磁盘的块将结果写入磁盘。经过这轮操作后，就得到了每 7 个块排序过后的数据，然后进行下一轮排序操作。关系 R 举例，经过第一轮排序后，现 Rfinal0-Rfinal6, Rfinal7-Rfinal13, Rfinal14-Rfinal15 分别为有序数据。



在第二轮排序中，每次取出三个块，第一次取出 R0, R7, R14，对这三个块的数据由小到大依次添加到缓冲区中负责写入磁盘的块，若 R0 的数据用尽，则补充以 R1 的数据；若 R7 的数据已写完，则补充以 R8 的数据…此轮完成后，便得到了完全有序的元组数据。



### (2) 查询优化算法的设计

在该部分任务中，选取了以下三条查询语句：

```

SELECT [ ENAME = 'Mary' & DNAME = 'Research' ] ( EMPLOYEE JOIN
DEPARTMENT )

```

```

PROJECTION [ BDATE ] ( SELECT [ ENAME = 'John' & DNAME = '

```

Research' ] ( EMPLOYEE JOIN DEPARTMENT ) )

SELECT [ ESSN = '01' ] ( PROJECTION [ ESSN, PNAME ] ( WORKS\_ON JOIN PROJECT ) )

### ①语法树结构

节点包括三个属性:

op: 存储该节点的操作关键字(类型为 String)

info: 存储该节点操作的内容(类型为 String)

child: 存储子节点(类型为 List<TreeNode>)

节点包括以下方法:

TreeNode(): 初始化

TreeNode(String info): 初始化

TreeNode(String op, String info): 初始化

toString():输出节点操作关键字及内容

### ②语法分析器

语法分析器需要能够识别上述关系代数语句, 并且对其进行解析, 生成对应的查询执行树。首先将查询语句以空格为分界得到 tokens, 对 tokens 进行遍历, 根据不同的关键字需要进行相应操作。

实现过程如下:

Case1: 若关键字为 SELECT 或 PROJECTION, 将该关键字设为节点的 op, 将语句后续[]中的内容作为 info 添加到节点当中, 例如 SELECT [ ENAME = ' Mary' & DNAME = ' Research' ] ( EMPLOYEE JOIN DEPARTMENT ) 语句, 在识别到关键字 SELECT 后, 构建节点并将节点的 op 设置为 SELECT, 将该节点的 info 设置为 ENAME = ' Mary' & DNAME = ' Research' 。

Case2: 若关键字为 JOIN, 将该关键字设为节点的 op, 将该 token 两侧的 token 作为节点添加到该节点的孩子节点中, 例如 EMPLOYEE JOIN DEPARTMENT, 在识别到 JOIN 关键字后, 将 EMPLOYEE 和 DEPARTMENT 加入到该节点的孩子节点中。

Case3: 若关键字为 (, 建立节点将语句括号中的内容添加到该节点的 child 当中, 例如语句 ( EMPLOYEE JOIN DEPARTMENT ), 识别到括号后, 将以语句 EMPLOYEE JOIN DEPARTMENT 再次建立树并将该树添加到节点的孩子节点当中。

### ③语法树结构的优化

根据课程所学内容, 对于选择和投影操作, 可让其尽早执行, 特别的对于同时存在连接与选择的语句, 先执行选择操作然后进行连接可以让执行代价明显降低。

因此在之前建立的语法树的基础上, 将树的根节点作为参数输入到优化函数

中，该函数主要执行以下操作：

Case1: 若节点的关键字为 SELECT，则将该节点更替为其孩子节点的优化结果，同时将选择的内容作为 info 加入到节点中，这个步骤的目的是将 SELECT 的内容提前到 JOIN 操作之前，即优先执行 SELECT 操作。

Case2: 若节点的关键字为 PROJECTION，则将该节点的孩子节点设置为孩子节点优化后的结果。

Case3: 若节点的关键字为 JOIN，则根据当前节点的信息(其中就包括了 SELECT 的内容)建立新的节点，并将新建立的节点加入到当前节点的孩子节点中。

```

if(node.op.equals("SELECT")){
    node=optimize(node.child.get(0),node.info.split( regex: "&"));
}
else if(node.op.equals("PROJECTION")){
    node.child.set(0,optimize(node.child.get(0),infoList));
}
else if(node.op.equals("JOIN")){
    String op="SELECT";

    TreeNode node0=new TreeNode(op,infoList[0]);
    node0.child.add(node.child.get(0));
    node.child.set(0,node0);
    if(infoList.length>1){
        TreeNode node1=new TreeNode(op,infoList[1]);
        node1.child.add(node.child.get(1));
        node.child.set(1,node1);
    }
}
return node;

```

对于所选的三条语句，优化的内容分别为：

1. SELECT [ ENAME = 'Mary' & DNAME = 'Research' ] ( EMPLOYEE JOIN DEPARTMENT )

该语句的原始树结构为：

```

SELECT ENAME = 'Mary' & DNAME = 'Research'
    JOIN
        EMPLOYEE
        DEPARTMENT

```

先将 EMPLOYEE 与 DEPARTMENT 两张表连接后，再根据条件进行 SELECT，经过优化后，先执行 SELECT 操作，再进行 JOIN，这样语句执行的代价显然更小。

```

JOIN
    SELECT ENAME = 'Mary'
        EMPLOYEE
    SELECT DNAME = 'Research'
        DEPARTMENT

```

2. PROJECTION [ BDATE ] ( SELECT [ ENAME = 'John' & DNAME = 'Research' ] ( EMPLOYEE JOIN DEPARTMENT ) )

该语句的原始树结构为：

```
PROJECTION BDATE
  SELECT ENAME = 'John' & DNAME = ' Research'
    JOIN
      EMPLOYEE
      DEPARTMENT
```

优化前语句的执行流程未先连接，再选择，最后再进行投影。经过优化后，先进行选择操作，然后进行连接，执行代价更小。

```
PROJECTION BDATE
  JOIN
    SELECT ENAME = 'John'
      EMPLOYEE
    SELECT DNAME = ' Research'
      DEPARTMENT
```

3. SELECT [ ESSN = '01' ] ( PROJECTION [ ESSN, PNAME ] ( WORKS\_ON JOIN PROJECT ) )

该语句的原始树结构为：

```
SELECT ESSN = '01'
  PROJECTION ESSN, PNAME
    JOIN
      WORKS_ON
      PROJECT
```

同样的，在优化前该语句的执行流程为先将 WORKS\_ON 与 PROJECT 表连接，然后再执行投影操作，最后才进行选择操作、优化后语句先执行选择操作，然后再进行连接与投影，执行代价更小。

```
PROJECTION ESSN, PNAME
  JOIN
    SELECT ESSN = '01'
      WORKS_ON
    PROJECT
```

#### 四、实验心得

对数据库的存储方式有了更深入的理解，对查询语句的优化在实践中得到了新的认识。