

Python 实现 SQL 查询优化

一、实验目的

- 1、理解并掌握选择运算和连接运算的执行过程；
- 2、掌握不同选择运算操作，即表扫描和索引扫描，的适用条件；
- 3、掌握不同连接运算操作，即嵌套循环连接、归并连接和散列连接，的适用条件；
- 4、掌握不同选择运算和连接运算的代价估计过程。

二、实验平台

- 1、OS: Windows 10
- 2、DBMS: SQL Server 2008

三、实验语言

Python

四、实验内容

(一) 实验数据导入

将数据导入 SQL Server 中。

(二) 算法实现

1、选择运算

- A. 给定基于 dbcourse 数据库的选择语句；
- B. 根据上述选择语句，将 dbcourse 中的对应数据导出成为 txt 文件；
- C. 请以上述 txt 文件作为输入，通过编程的方式【编程语言不限】实现步骤 A 中的选择语句（包括两种选择运算操作-表扫描和索引扫描），并输出选择结果。

预处理

(1) 语句说明

- 支持子句 select,from,where
- 支持大写、小写
- 支持任意多的换行、空格、制表符，但要求必须以 ‘;’ 结束
- 支持 select 子句中有 distinct、*、或自定义属性名的输入，但不能在属性名前加[表名.]
- 支持 from 子句中有一个表
- 支持 where 子句中有多个查询条件，目前支持的条件类型为 >、<、=、<>、>=、<=

(2) 相关代码说明

- 支持任意行的输入，判断是否到‘；’来结束。注意递归调用

```
#由于python默认认为回车结束，因此在；结束前，不断递归式的读取输入，将其全部拼接在sql之后。  
#注意在读取非结束语句那一行时，要加上return，否则会因为t_sql是临时变量而无法将拼接后的结果返回到上一层，导致结果为None  
def enterSelect(sql):#输入select语句  
    t_sql=input()  
    #大写字母转为小写字母，注意条件引号中的大写字母不能变  
    new=[]  
    for s in t_sql:  
        new.append(s)  
    i=0  
    while i < len(new):  
        if new[i]==":":  
            for j in range(i+1,len(new)):  
                if new[j]==":":  
                    i=j+1  
                    break  
            new[i]=new[i].lower()  
        i+=1  
    t_sql=''.join(new)  
    if(t_sql.find(';') != -1):#存在‘；’  
        sql+=t_sql  
        #print(sql)  
        return sql  
    else:  
        sql+=t_sql  
    return (enterSelect(sql))#注意加上return，否则递归后返回值为None
```

图 2.1.1

- 解析输入的 SQL 语句

```
#将语句分解，存到字典中，key为select/from/where，value为相应子句后的值  
#stmtTag储存关键词标志，为['select','from','where','group by','order by',';']  
#通过找到下一个关键词来获得本关键词和下关键字之间的子句  
def getDictSql():  
    part_sql=sql  
    currentTag='select'  
    while True:  
        if currentTag == ';':  
            break  
        preTag=currentTag  
        part_sql=part_sql[part_sql.find(currentTag,0):]  
        currentTag=nextStmtTag(part_sql,currentTag)  
        child_sql=part_sql[part_sql.find(preTag,0)+len(preTag):part_sql.find(currentTag,0)]#获取两个关键词之间的句子  
        arow=child_sql.strip(",").split(',')  
        for i in range(0,len(arow)):  
            arow[i]=arow[i].strip()  
        dict_sql[preTag]=arow  
    return dict_sql
```

图 2.1.2

- 解析 select 子句

```
#解析select子句，主要需要处理*和distinct的情况
def processSelect():
    distinct=False
    if '*' in dict_sql['select']:#处理select中的*，将字典内容换成这个表中所有属性名
        dict_sql['select'].remove('*')
        for i in range(0,len(dict_data[table_name[0]])):
            dict_sql['select'].append(dict_data[table_name[0]][i])
    for j in range(0,len(dict_sql['select'])):
        if dict_sql['select'][j].find('distinct')!=-1:#处理distinct
            distinct=True
            dict_sql['select'][j]=(dict_sql['select'][j][dict_sql['select'][j].find('distinct')+8:]).strip()
    return dict_sql['select'],distinct
```

图 2.1.3

- 解析 from 子句

```
#处理from子句，主要需要在表名后面加上'.txt'，方便后面读取文件
def processFrom():
    from_sql=dict_sql['from']
    table_name=[]
    for i in range(0,len(from_sql)):
        table_name.append(from_sql[i]+'.txt')
    return table_name
```

图 2.1.4

· 解析 where 子句

```
#进一步处理where中的条件
#根据标志符号>,<,=,<>,>=,<=，将每个条件划分为三部分—左属性、操作符、右条件
def getTerms(terms):
    for i in range(0,len(term_name)):
        if term_name[i].find('<>')!=-1:#处理含<>的条件
            first_term=term_name[i][:term_name[i].find('<>')].strip()
            second_term=term_name[i][term_name[i].find('<>')+2:].strip()
            if(first_term=='id'):
                second_term=int(second_term)
            terms.loc[i]=[first_term,0,second_term]
            continue
        elif term_name[i].find('>')!=-1:#处理含>=的条件
            first_term=term_name[i][:term_name[i].find('>')].strip()
            second_term=term_name[i][term_name[i].find('>')+2:].strip()
            if(first_term=='id'):
                second_term=int(second_term)
            terms.loc[i]=[first_term,4,second_term]
            continue
        elif term_name[i].find('<')!=-1:#处理含<=的条件
            first_term=term_name[i][:term_name[i].find('<')].strip()
            second_term=term_name[i][term_name[i].find('<')+2:].strip()
            if(first_term=='id'):
                second_term=int(second_term)
            terms.loc[i]=[first_term,5,second_term]
            continue
        elif term_name[i].find('>')!=-1:#处理含>的条件
            first_term=term_name[i][:term_name[i].find('>')].strip()
            second_term=term_name[i][term_name[i].find('>')+1:].strip()
            if(first_term=='id'):
                second_term=int(second_term)
            terms.loc[i]=[first_term,1,second_term]
            continue
        elif term_name[i].find('<')!=-1:#处理含<的条件
            first_term=term_name[i][:term_name[i].find('<')].strip()
            second_term=term_name[i][term_name[i].find('<')+1:].strip()
            if(first_term=='id'):
                second_term=int(second_term)
            terms.loc[i]=[first_term,2,second_term]
            continue
        elif term_name[i].find('>')!=-1:#处理含=的条件
            first_term=term_name[i][:term_name[i].find('>')].strip()
            second_term=term_name[i][term_name[i].find('>')+1:].strip()
            if(first_term=='id'):
                second_term=int(second_term)
            terms.loc[i]=[first_term,3,second_term]
            continue
    return terms
```

图 2.1.5

a. 表扫描

代码详情请见附件中 table_scan.py

(1) 程序执行流程

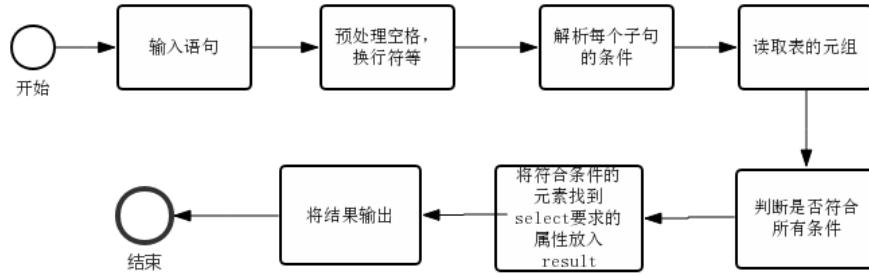


图 2.1.6

(2) 核心代码说明

- 进行表扫描，读入文件一行进行判断，然后判断该元组是否符合所有条件，找到符合条件的元组，并将结果存到 result 中。

```

#查询结果存到result中
result=pd.DataFrame(columns=([select_name[i] for i in range(0,len(select_name))]))#columns形式为列表！！！
cnt=0
t=0
while line:
    arow=line.strip().split("\t")
    arow[0]=int(arow[0])
    flag=0#判断是否符合所有条件，符合条件为flag=len(terms)，满足一项条件，flag就加一
    for i in range(0,len(terms)):#所有的条件储存在terms，其数据结构为DataFrame
        if terms['char'][i]==0:
            if arow[attr_to_int[table_name[0]][terms['first_term'][i]]] != terms['second_term'][i]:
                flag+=1
        elif terms['char'][i]==1:
            if arow[attr_to_int[table_name[0]][terms['first_term'][i]]] > terms['second_term'][i]:
                flag+=1
        elif terms['char'][i]==2:
            if arow[attr_to_int[table_name[0]][terms['first_term'][i]]] < terms['second_term'][i]:
                flag+=1
        elif terms['char'][i]==3:
            if arow[attr_to_int[table_name[0]][terms['first_term'][i]]] == terms['second_term'][i]:
                flag+=1
        elif terms['char'][i]==4:
            if arow[attr_to_int[table_name[0]][terms['first_term'][i]]] >= terms['second_term'][i]:
                flag+=1
        elif terms['char'][i]==5:
            if arow[attr_to_int[table_name[0]][terms['first_term'][i]]] <= terms['second_term'][i]:
                flag+=1
    #符合条件的结果存入result中
    if flag == len(terms):#当该元组满足所有要求时
        t_arow=[]
        for j in range(0,len(select_name)):
            t_arow.append(arow[attr_to_int[table_name[0]][select_name[j]]])
        result.loc[cnt]=[arow[attr_to_int[table_name[0]][select_name[j]]] for j in range(0,len(select_name))]
        cnt+=1
    line=file.readline()
file.close()
#结果写出到csv
if distinct==True:
    result=result.drop_duplicates()#进行distinct操作
    result.index=[k for k in range(0,len(result))]#去重后重新给行索引编号
result.to_csv("result_table_scan.csv", mode='w', encoding = "utf-8")

```

图 2.1.7

(3) 运行结果

测试用例 1

- 测试语句:

```
SELECT * FROM JOURNALS
```

```
WHERE ID<=500 AND ADDR='北京市' AND CLASS='A';
```

运行结果:

```
请输入查询语句:  
SELECT * FROM JOURNALS  
WHERE ID<=500 AND ADDR='北京市' AND CLASS='A';  
  
查询的表名为: ['journals.txt']  
查询的属性为: ['id', 'name', 'addr', 'class'] False  
查询的条件为:  
    first_term char second_term  
0      id      5      500  
1      addr     3      北京市  
2      class    3      A  
  
正在执行查询, 请稍后.....  
  
查询已成功执行, 结果请看result_table_scan.csv文件  
查询用时 1.42 s, 查询结果返回 65 行
```

图 2.1.8

- 结果查看:

A	B	C	D	E
	id	name	addr	class
0	1	语言文字	北京市	A
1	12	考古	北京市	A
2	18	当代电影	北京市	A
3	21	电影艺术	北京市	A
4	25	中国翻译	北京市	A
5	37	当代语言学	北京市	A
6	38	文物	北京市	A
7	41	哲学动态	北京市	A
8	42	方言	北京市	A
9	50	中国现代文	北京市	A
10	55	文艺研究	北京市	A
11	58	世界哲学	北京市	A
12	59	中央音乐学	北京市	A
13	65	民族语文	北京市	A
14	66	世界历史	北京市	A
15	67	自然辩证法	北京市	A
16	73	故宫博物院	北京市	A
17	74	中国音乐	北京市	A
18	94	文献	北京市	A
19	97	戏剧(中央)	北京市	A
20	101	中国典籍与	北京市	A
21	102	世界电影	北京市	A
22	109	世界美术	北京市	A
23	113	新文学史料	北京市	A
24	114	剧本	北京市	A
25	132	北京第二外	北京市	A
26	144	中国俄语教	北京市	A
27	155	中国藏学	北京市	A
28	158	北京电影学	北京市	A

图 2.1.9

测试用例 2

- 测试语句：

```
select distinct author,journal  
from papers  
where id <= 500;
```

- 运行结果：

请输入查询语句：
`select distinct author, journal
from papers
where id <= 500;`

查询的表名为： ['papers.txt']
查询的属性为： ['author', 'journal'] True
查询的条件为：
first_term char second_term
0 id 5 500

正在执行查询，请稍后.....

查询已成功执行，结果请看result_table_scan.csv文件
查询用时 1.24 s，查询结果返回 500 行

图 2.1.10

- 结果查看：

	A	B	C	D	E	F
1		author	journal			
2	0	余胜军	特区展望			
3	1	徐军军	社会保障问题研究			
4	2	杨梅英	中国国情国力			
5	3	高亚宾	中国高新区			
6	4	袁志彬	湖北民族学院学报(哲学社会科学版)			
7	5	黎永强	上海市经济学会学术年刊			
8	6	缪晓春	温州人			
9	7	周义哲	乡镇经济			
10	8	赵荣妹	云南政报			
11	9	周效英	林业经济			
12	10	邓世州	北方音乐			
13	11	廖翠兰	河北工程大学学报(社会科学版)			
14	12	张景山	河海大学学报(哲学社会科学版)			
15	13	丁笈	内蒙古金融研究			
16	14	杨力舟	中小学电教(下半月)			
17	15	田向红	理论研究			
18	16	金平伟	卓越理财			
19	17	刘灿	现代审计与经济			
20	18	顾益君	校园英语			
21	19	白术明	南通大学学报(社会科学版)			
22	20	孟宪新	北京交通大学学报(社会科学版)			
23	21	卢嵒	北京劳动保障职业学院学报			
24	22	薛艳茹	藏外佛教文献			
25	23	张柱荣	天津经济			
26	24	杨迁	中国矿业大学学报(社会科学版)			
27	25	李兴英	河北旅游职业学院学报			
28	26	刘青瑜	农业工程技术(农业产业化)			
29	27	蔡艳芬	中国注册会计师			
30	28	蒋舸	中国农垦			

图 2.1.11

测试用例 3

- 测试语句:

```
select* from authors  
where id <= 50 and age > 30 and sex='女';
```

- 运行结果:

```
请输入查询语句:  
select* from authors  
where id <= 50 and age > 30 and sex='女';  
  
查询的表名为: ['authors.txt']  
查询的属性为: ['id', 'name', 'sex', 'age'] False  
查询的条件为:  
    first_term    char    second_term  
0      id      5          50  
1      age      1          30  
2      sex      3          女  
  
正在执行查询, 请稍后.....  
  
查询已成功执行, 结果请看result_table_scan.csv文件  
查询用时 132.12 s, 查询结果返回 15 行
```

图 2.1.12

- 结果查看:

	A	B	C	D	E	F
1		id	name	sex	age	
2	0	6	阿碧	女	47	
3	1	7	阿彬	女	48	
4	2	8	阿滨	女	43	
5	3	10	阿炳	女	44	
6	4	11	阿波里奈尔	女	36	
7	5	12	阿不道列提	女	38	
8	6	13	阿不都	女	33	
9	7	14	阿不都艾尼	女	37	
10	8	15	阿不都卡	女	32	
11	9	19	阿不都热合	女	32	
12	10	23	阿不都肉苏	女	36	
13	11	31	阿不力孜	女	46	
14	12	40	阿布都克里	女	46	
15	13	42	阿布都拉	女	33	
16	14	50	阿布来提	女	33	
17						

图 2.1.13

b. 索引扫描

代码详情请见附件中 index_scan.py

(1) 程序执行流程

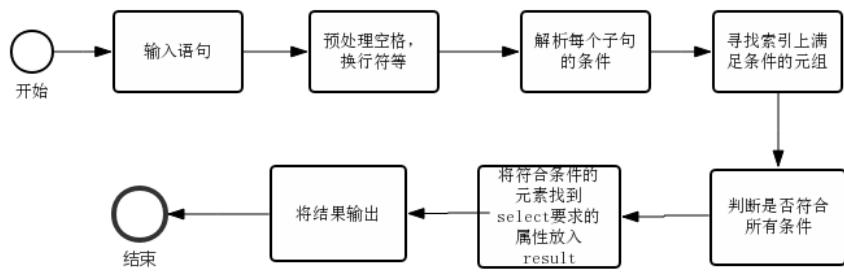


图 2.1.14

(2) 核心代码说明

【说明】由于时间有限，目前只对 authors 表上的 id 和 age 属性建立了 B+树索引。程序一进去就开始自动建立索引。这是 B+树索引的程序不足的地方。

由于代码过长，B+树建立索引的部分详情请看程序文件 index_scan.py。

(3) 运行结果

测试用例

为进行比较分析，该测试用例与表扫描部分的测试用例 3 相同。

- 测试语句：

```
select* from authors
where id <= 50 and age > 30 and sex='女';
```

- 运行结果：

```

建立索引中.....  

索引建立完成! 用时 38.36 s.  

请输入查询语句:  

select* from authors  

where id <= 50 and age > 30 and sex='女';  

查询的表名为: ['authors.txt']  

查询的属性为: ['id', 'name', 'sex', 'age']  False  

查询的条件为:  

first_term  char  second_term  

0          id      5          50  

1          age     1          30  

2          sex     3          女  

正在执行查询, 请稍后.....  

查询已成功执行, 结果请看result_index_scan.csv文件  

查询用时 0.00198 s, 查询结果返回 15 行

```

图 2.1.15

- 结果查看：

	A	B	C	D
1	id	name	sex	age
2	6	阿碧	女	47
3	7	阿彬	女	48
4	8	阿滨	女	43
5	10	阿炳	女	44
6	11	阿波里奈尔	女	36
7	12	阿不道列提	女	38
8	13	阿不都	女	33
9	14	阿不都艾尼	女	37
10	15	阿不都卡	女	32
11	19	阿不都热合曼	女	32
12	23	阿不都肉苏力	女	36
13	31	阿不力孜	女	46
14	40	阿布都克里木	女	46
15	42	阿布都拉	女	33
16	50	阿布来提	女	33
17				

图 2.1.16

综合对比

表 2.1 两种扫描对比

扫描类型	表扫描	索引扫描
执行时间	132.12s	0.00198s
返回行数	15	15

结果分析：可以明显看到，对于同一个查询，索引扫描效率要高于表扫描。

2、连接运算

- A. 给定基于 dbcourse 数据库的连接语句；
- B. 根据上述连接语句，将 dbcourse 中的对应数据导出成为 txt 文件；
- C. 请以上述 txt 文件作为输入，通过编程的方式【编程语言不限】实现步骤 A 中的连接语句（包括三种连接运算操作-嵌套循环连接、归并连接和散列连接），并输出连接结果。

预处理

(1) 语句说明

- 支持子句 select,from,where
- 支持大写、小写
- 支持任意多的换行、空格、制表符，但要求必须以 ‘;’ 结束
- 支持 select 子句中有 distinct、*、或自定义属性名的输入，属性名前可以加[表名.]，对于能够区分的属性名也可以不加
- 支持 from 子句中有两个表
- 支持 where 子句中有多个查询条件，目前支持的条件类型为 >、<、=、<>、>=、<=，条件中的属性可以加[表名.]，对于能够区分的属性名也可以不加。

(2) 相关代码说明

enterSelect(sql)、getDictSql()、processWhere()函数请见选择运算的说明。

- SQL 语句的标志词

```
#SQL子语句标志词
stmtTag = ['select', 'from', 'where', 'group by', 'order by', ';' ]
```

图 2.2.1

- 模拟数据字典，表名+属性的列表

```
#模拟数据字典
dict_data={'authors.txt':['id','name','sex','age'],
           'journals.txt':['id','name','addr','class'],
           'papers.txt':['id','title','author','journal','year','keyword','org']}
```

图 2.2.2

- 表名对应的属性名转为 int 值，方便后续在元组（实际存储为列表）找到对应的值

```
#属性名转换成数字
attr_to_int={'authors.txt':{'id':0,'name':1,'sex':2,'age':3},
             'journals.txt':{'id':0,'name':1,'addr':2,'class':3},
             'papers.txt':{'id':0,'title':1,'author':2,'journal':3,'year':4,'keyword':5,'org':6}}
```

图 2.2.3

- 处理 from 子句

```
#处理from子句，主要需要在表名后面加上'.txt'，方便后面读取文件
def processFrom():
    from_sql=dict_sql['from']
    table_name=[]
    for i in range(0,len(from_sql)):
        table_name.append(from_sql[i]+'.txt')
    table_name.sort()#little trick 表顺序确定方便后面处理
    return table_name
```

图 2.2.4

- 处理 select 子句，注意处理*时需要处理所有的表名

```
#解析select子句，主要需要处理*和distinct的情况
def processSelect():
    distinct=False
    if '*' in dict_sql['select']:#处理select中的*，将字典内容换成这个表中所有属性名
        dict_sql['select'].remove('*')
        for tt in range(0,len(table_name)):#需要遍历所有的表名
            for i in range(0,len(dict_data[table_name[tt]])):#遍历表的所有属性
                dict_sql['select'].append(dict_data[table_name[tt]][i])
    for j in range(0,len(dict_sql['select'])):
        if dict_sql['select'][j].find('distinct')!=-1:#处理distinct
            distinct=True
            dict_sql['select'][j]=(dict_sql['select'][j][dict_sql['select'][j].find('distinct')+8:]).strip()
    return dict_sql['select'],distinct
```

图 2.2.5

为了后续方便，将 select 中的属性与其对应的表名存到 DataFrame 中。

```
#因为select中的属性可能来自不同的表，因此要对select中的属性加上表名，方便后面处理
#结果返回属性+所在的表名
def getSelects(selects,table_name):
    for i in range(0,len(select_name)):
        if select_name[i].find('.')!=-1:
            table=select_name[i][:select_name[i].find('.').strip()+'.']
            attr=select_name[i][select_name[i].find('.')+1:].strip()
        else:
            attr=select_name[i]
            for j in range(0,len(table_name)):
                if select_name[i] in dict_data[table_name[j]]:
                    table=table_name[j]
    selects.loc[i]=[attr,table]
return selects
```

图 2.2.6

- 处理条件，主要需要处理表名，左属性对应的表名存到 table1 中，右属性对应的表名存到 table2 中，这样在后续判断时能够到原表中找到对应元组。

```

def getTerms(terms,table_name):
    for i in range(0,len(term_name)):
        if term_name[i].find('<>')!=-1:
            first_term=term_name[i][:term_name[i].find('<>')].strip()
            second_term=term_name[i][term_name[i].find('<>')+2:].strip()
            c=0
        elif term_name[i].find('>=')!=-1:
            first_term=term_name[i][:term_name[i].find('>=')].strip()
            second_term=term_name[i][term_name[i].find('>=')+2:].strip()
            c=4
        elif term_name[i].find('<=')!=-1:
            first_term=term_name[i][:term_name[i].find('<=')].strip()
            second_term=term_name[i][term_name[i].find('<=')+2:].strip()
            c=5
        elif term_name[i].find('>')!=-1:
            first_term=term_name[i][:term_name[i].find('>')].strip()
            second_term=term_name[i][term_name[i].find('>')+1:].strip()
            c=1
        elif term_name[i].find('<')!=-1:
            first_term=term_name[i][:term_name[i].find('<')].strip()
            second_term=term_name[i][term_name[i].find('<')+1:].strip()
            c=2
        elif term_name[i].find('=')!=-1:
            first_term=term_name[i][:term_name[i].find('=')].strip()
            second_term=term_name[i][term_name[i].find('=')+1:].strip()
            c=3
        #认为连接的条件为等值连接，且显式地表示出来
        #当操作符为=时且两端属性所属的表不是一个表时，认为这个是连接的属性
        if c!=3:
            if first_term.find('.') != -1:#有表名限制
                table1=first_term[:first_term.find('.')).strip()+'txt'
                first_term=first_term[first_term.find('.')+1:]
            else:
                for j in range(0,len(table_name)):
                    if first_term in dict_data[table_name[j]]:
                        table1=table_name[j]
            table2=table1
        #对于其他属性，一般认为操作符两端的属性或常量都属于同一张表
        else:
            #寻找表名
            if first_term.find('.') != -1:#有表名限制
                table1=first_term[:first_term.find('.')).strip()+'txt'
                first_term=first_term[first_term.find('.')+1:]
                table2=table1
                if second_term.find('.')!=-1:
                    table2=second_term[:second_term.find('.')).strip()+'txt'
                    second_term=second_term[second_term.find('.')+1:]
                else:
                    for k in range(0,len(table_name)):
                        if second_term in dict_data[table_name[k]]:
                            table2=table_name[k]
            else:#无表名限制
                for j in range(0,len(table_name)):
                    if first_term in dict_data[table_name[j]]:
                        table1=table_name[j]
                table2=table1
                if second_term.find('.')!=-1:
                    table2=second_term[:second_term.find('.')).strip()+'txt'
                    second_term=second_term[second_term.find('.')+1:]
                else:
                    for k in range(0,len(table_name)):
                        if second_term in dict_data[table_name[k]]:
                            table2=table_name[k]
        if first_term=='id':
            second_term=int(second_term)
        #print("i: ",i," first_term: ",first_term, " second_term: ",second_term)
        terms.loc[i]=[first_term,c,second_term,table1,table2]
    return terms

```

图 2.2.7

a. 嵌套循环连接

代码详情请见附件中 loop_join.py

(1) 程序执行流程

(2) 核心代码说明

一次读取两个表中的一个元组，模拟内存中只能放下两个表中各一个元组的情况。

进行条件的判断：若 terms 中 table1 和 table2 的表一样，则进行如表扫描方式的判断即可；若两个属性对应的表不一样，则需要将两个表中对应属性的值拿出来比较，若满足条件则 flag+=1. 最后满足所有条件的两个表中 select 语句要求的属性的值存入 result 中。

```

#读取文件，连接两个表
file0=open(table_name[0],encoding='UTF-8-sig')
file1=open(table_name[1],encoding='UTF-8-sig')
line0=file0.readline()
line1=file1.readline()
l0=line0
l1=line1
dict_line={table_name[0]:line0,table_name[1]:line1}#簇集内存中一次只能放下两张表中各一个元组，key为表名，value为对应元组
result=pd.DataFrame(columns=[selects['attr'][i] for i in range(0,len(select_name))])#结果存到result中
cnt=0
test=0
while l0:
    line0=line0.strip().split("\t")
    line0[0]=int(line0[0])
    dict_line[table_name[0]]=line0
    while l1:
        flag=0
        line1=line1.strip().split("\t")
        line1[0]=int(line1[0])
        dict_line[table_name[1]]=line1
        for i in range(0,len(terms)):#判断两个元组是否满足所有条件
            t_line=dict_line[terms['table1'][i]]
            if terms['char'][i]==0:
                if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] != terms['second_term'][i]:
                    flag+=1
            elif terms['char'][i]==1:
                if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] > terms['second_term'][i]:
                    flag+=1
            elif terms['char'][i]==2:
                if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] < terms['second_term'][i]:
                    flag+=1
            elif terms['char'][i]==3:
                if (terms['table1'][i]==terms['table2'][i]):
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] == terms['second_term'][i]:
                        flag+=1
                else:#处理等值连接属性
                    t_line2=dict_line[terms['table2'][i]]
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] == t_line2[attr_to_int[terms['table2'][i]][terms['second_term'][i]]]:
                        flag+=1
            elif terms['char'][i]==4:
                if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] >= terms['second_term'][i]:
                    flag+=1
            elif terms['char'][i]==5:
                if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] <= terms['second_term'][i]:
                    flag+=1
        if flag==len(terms):#如果所有条件都满足，就将两张表中select中对应的属性的值加入result
            t_arow=[]
            for j in range(0,len(select_name)):
                t_arow.append(dict_line[selects['table'][j]][attr_to_int[selects['table'][j]][selects['attr'][j]]])
            result.loc[cnt]=[t_arow[k] for k in range(0,len(t_arow))]
            cnt+=1
        line1=file1.readline()
        l1=line1
    file1.close()
    file1=open(table_name[1],encoding='UTF-8-sig')
    line1=file1.readline()
    l1=line1
    line0=file0.readline()
    l0=line0
result.to_csv("result_loop_join.csv",mode='a',encoding='UTF-8')

```

图 2.2.8

(3) 运行结果

【说明】由于这种方法查询效率很低，因此测试时强制让其只查询 2 个结果，来看看查询效过。

测试用例

- 测试语句

```

select journals.id, journal, title
from papers,journals

```

```
where journals.name = papers.journal and papers.id<500 and class = 'A';
```

- 运行结果

结果显示查询满足条件的两个元组都耗时 24.16s，效率是很低

```
请输入查询语句:  
select journals.id, journal, title  
from papers, journals  
where journals.name = papers.journal and papers.id<500 and class = 'A';  
查询的表名为: ['journals.txt', 'papers.txt']  
查询的属性为: ['journals.id', 'journal', 'title'] False  
查询的条件为:  
first_term char second_term      table1      table2  
0       name    3     journal  journals.txt  papers.txt  
1       id      2        500   papers.txt  papers.txt  
2       class   3          A  journals.txt  journals.txt  
区分表名后查询的属性为:  
attr      table  
0      id  journals.txt  
1  journal  papers.txt  
2  title   papers.txt  
  
正在执行查询, 请稍后.....  
  
查询已成功执行, 结果请看result_loop_join.csv文件  
查询用时 24.16 s, 查询结果返回 2 行
```

图 2.2.9

- 结果查看

	A	B	C	D	E
1		id	journal	title	
2	0	55	文艺研究	y6gc9snsdayjfxg	
3	1	64	中国农史	28bbzhraqor0hu0	
A					

图 2.2.10

b. 归并连接

代码详情请见附件中 merge_join.py

(1) 程序执行流程

(2) 核心代码说明

思路：首先先对两个表按照连接属性进行升序排列。然后循环遍历两个表，若外表的连接属性小于内表，则外表指针加一；若内表的连接属性小于外表，则内表指针加一。若连接属性值相等，则进行下一步判断，看是否满足所有的条件。将满足所有条件的元组对应 select 要求的元组加入 result 中。

```

#先对两个表按照连接属性排序,默认升序排列
table1=table1.sort_values(by='name')
table2=table2.sort_values(by=name2)
#注意DataFrame的行索引在排序后也改变了,因此要重新将行索引变为1-cnt1
table1.index=[k1 for k1 in range(1,cnt1)]
table2.index=[k2 for k2 in range(1,cnt2)]
i_1=1
i_2=1
cnt=1
while i_1 < cnt1:#结束条件:读到文件末尾
    if i_2 == cnt2:#如果第二个表已经读完也结束
        break
    while i_2 < cnt2:
        if table1['name'][i_1] < table2[name2][i_2]:#若左边表的连接值比右边表小,则跳过左边的表
            i_1+=1
        elif table1['name'][i_1] > table2[name2][i_2]:#若右边表的连接值比左边小,则跳过右边的表
            i_2+=1
        else:#若左边的表的连接值与右边的相等,则比较其他条件
            line0=[]
            line1=[]
            for ii in range(0,len(dict_data[table_name[0]])):
                line0.append(table1[dict_data[table_name[0]][ii]][i_1])
            for jj in range(0,len(dict_data[table_name[1]])):
                line1.append(table2[dict_data[table_name[1]][jj]][i_2])
            dict_line={table_name[0]:line0,table_name[1]:line1}#读入连接值相同的两个表的元组
            flag=0
            for i in range(0,len(terms)):
                t_line=dict_line[terms['table1'][i]]
                if terms['char'][i]==0:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] != terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==1:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] > terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==2:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] < terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==3:
                    if (terms['table1'][i]==terms['table2'][i]):
                        if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] == terms['second_term'][i]:
                            flag+=1
                    else:
                        t_line2=dict_line[terms['table2'][i]]
                        if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] == t_line2[attr_to_int[terms['table2'][i]][terms['second_term'][i]]]:
                            flag+=1
                elif terms['char'][i]==4:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] >= terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==5:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] <= terms['second_term'][i]:
                        flag+=1
            if flag==len(terms):#如果所有条件都满足,就将两张表中select中对应的属性的值加入result
                t_arow=[]
                for j_t in range(0,len(select_name)):
                    t_arow.append(dict_line[selects['table'][j_t]][attr_to_int[selects['table'][j_t]][selects['attr'][j_t]]])
                result.loc[cnt]=[t_arow[k] for k in range(0,len(t_arow))]
                cnt+=1
            i_2+=1
result.to_csv("result_merge_join.csv",mode='a',encoding='UTF-8')

```

图 2.2.11

(3) 运行结果

测试用例

- 测试语句

```

select journals.id, journal, title
from papers,journals
where  journals.name = papers.journal  and papers.id<500 and class = 'A';

```

- 运行结果

结果显示查询时间为 7.16s (包括排序时间), 结果返回 183 行。

```

请输入查询语句:
select journals.id, journal, title
from papers, journals
where journals.name = papers.journal and papers.id < 500 and class = 'A';
查询的表名为: ['journals.txt', 'papers.txt']
查询的属性为: ['journals.id', 'journal', 'title'] False
查询的条件为:
    first_term char second_term      table1      table2
0      name   3      journal  journals.txt  papers.txt
1      id     2      500    papers.txt  papers.txt
2      class   3          A  journals.txt  journals.txt
区分表名后查询的属性为:
    attr      table
0      id  journals.txt
1  journal  papers.txt
2  title   papers.txt
读取数据中, 请稍后.....
读取结束! 两个表的大小分别是: 4742 行、 1001 行

正在执行查询, 请稍后.....
查询已成功执行, 结果请看result_merge_join.csv文件
查询用时 7.16 s, 查询结果返回 183 行

```

图 2.2.12

- 结果查看

【说明】由于对 journals 表和 papers 表的 name 和 journal 属性进行排序，因此排序后 id 为乱序，索引连接后的结果 id 也为乱序，但可以看到 journal 属性为有序的。

	A	B	C	D	E
1		id	journal	title	
2	1	188	东北史地	v56fdmw9aozdn1m	
3	2	3779	东北财经大	so25h7kvt7z8bmg	
4	3	1659	中俄关系的	ciwf5xesv4rlp2m	
5	4	514	中华儿女	310nlchz1vowsto	
6	5	1389	中国专利与	xwnktdd1knc4gec	
7	6	3876	中国人造板	16am7gla064q7zm	
8	7	4510	中国价格监	xnnwa22iqk4an3i	
9	8	4267	中国会计评	f1wm1nvecsosoff	
10	9	64	中国农史	28bbzhraqor0hu0	
11	10	4139	中国农垦	y40flrukpm11gbu	
12	11	4064	中国农村金	24nnah3cb6w864z	
13	12	3975	中国制造业	o13787t208dnsn0	
14	13	3540	中国卫生人	sq0wcgsuye3trxm	
15	14	1139	中国司法	ts6zek3mbk5wap0	
16	15	3911	中国城市经	opxg4a5aa1lh9mu	
17	16	4028	中国安防	y39h5h01hdzg2h0	
18	17	4498	中国审计信	jzbj39mldesuzh4	
19	18	353	中国文化遗	st4qxt95nfre4ra	
20	19	4345	中国服饰	yjpvkf5u01ritme	
21	20	4055	中国标准化	gdemsugh8z402nh	
22	21	2628	中国校外教	j8egjqi6o57vxtw	
23	22	1222	中国民政	y189ysegmbb73kp	
24	23	1770	中国民族	18cswiy8izm9qgo	
25	24	3806	中国现代中	jx54zcl8g6abrrtr	
26	25	2210	中国现代教	w0k3ceelwamy3if	
27	26	3008	中国电子教	gr90u7sdyawocmw	
28	27	4086	中国纤检	hgvytq412cwc0qf	
29	28	4597	中国经济信	7jchfs038185vic	
30	29	4514	中国经济学	nw8p2p6juocvqec	

图 2.2.12

c. 散列连接

代码详情请见附件中 hash_join.py

(1) 程序执行流程

(2) 核心代码说明

- 哈希函数

将字符串哈希到大小为 numHash 的桶里。

```
def HashFunction(name, numHash):#哈希函数，传入name表示连接属性的值，numHash表示哈希桶的个数
    key=hash(name)#利用python自带的hash函数，将字符串哈希成大整数（32位整数，有负数）
    #在网上找的Redis常用的哈希函数，处理32位整数的哈希情况，冲突率比较低
    key+=~(key<<15)
    key=key^(key>>10)
    key+=(key<<3)
    key^=(key>>6)
    key+=~(key<<11)
    key^=(key>>16)
    return key%numHash#最后取模，将值放在哈希桶里
```

图 2.2.13

- 进行哈希

将哈希表设为字典，key 为哈希桶的编号，value 为哈希到该桶里的元组的编号（即行索引）。然后对两个表进行扫描，将 name 和 journal/author 属性列上的值进行哈希。

```

#进行哈希
#确定哈希桶个数
hashNum1=hashNum[table_name[0]]
hashNum2=hashNum[table_name[1]]
hashValue1={}#哈希桶，每个桶里存下对应元组的行序号
hashValue2={}#key为哈希桶的编号，value为哈希到该桶里的元组的编号（即行索引）
#初始化哈希表，每一个value初始化为空列表
for i in range(0,hashNum1):
    t=[]
    hashValue1[i]=t
for j in range(0,hashNum2):
    t=[]
    hashValue2[j]=t
#对第一个表的name进行哈希
for i in range(1,cnt1):
    value=HashFunction(table1['name'][i],hashNum1)
    hashValue1[value].append(i)
#确定第二表的属性列名（由于前面已经将表名排序，因此第二个表一定是papers表）
if table_name[0]=='journals.txt':
    name2='journal'
elif table_name[0]=='authors.txt':
    name2='author'
#对第二个表(papers)的author或journal进行hash
for i in range(1,cnt2):
    value=HashFunction(table2[name2][i],hashNum2)
    hashValue2[value].append(i)

```

图 2.2.14

- 连接表

遍历两个哈希桶，若两个哈希表对应的同一个编号的哈希桶中有一个为空，则跳过，因为不存在这样的元组。若都存在，但连接条件不等，则跳过。注意由于哈希桶中可能有多个（一般不超过 3 个）元组编号，因此要遍历每一个哈希桶中的元素去读取，将元组编号存入 dict_line 中表名对应的列表中。最后对于每一组连接条件相等的元组，判断其是否满足所有的条件，将满足条件的元组对应的 select 中的属性列的值加入 result 中。

```

//对两个哈希表进行扫描
cnt=0
for i_h in range(0,hashNum['papers.txt']):
    if hashValue1[i_h] is None:#若两个哈希表对应的同一个编号的哈希桶中有一个为空，则跳过，因为不存在这样的元组
        pass
    if hashValue2[i_h] is None:
        pass
    for j in range(0,len(hashValue1[i_h])):#遍历第一个哈希表
        for k in range(0,len(hashValue2[i_h])):#遍历第二个哈希表
            index1=hashValue1[i_h][j]
            index2=hashValue2[i_h][k]
            if table1['name'][index1]!=table2['name2'][index2]:#连接条件不相等，pass
                pass
            line0=[]
            line1=[]
            for ii in range(0,len(dict_data[table_name[0]])):#由于哈希桶中可能有多个（一般不超过3个）元组编号，因此要遍历去读取
                line0.append(table1[dict_data[table_name[0]][ii]][index1])
            for jj in range(0,len(dict_data[table_name[1]])):
                line1.append(table2[dict_data[table_name[1]][jj]][index2])
            dict_line=[table_name[0]:line0,table_name[1]:line1]#连接条件相等时，读出两个元组存入dict_line中
            flag=0
            for i in range(0,len(terms)):#循环判断元组是否满足所有条件
                t_line=dict_line[terms['table1'][i]]
                if terms['char'][i]==0:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] != terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==1:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] > terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==2:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] < terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==3:
                    if (terms['table1'][i]==terms['table2'][i]):
                        if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] == terms['second_term'][i]:
                            flag+=1
                    else:
                        t_line2=dict_line[terms['table2'][i]]
                        if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] == t_line2[attr_to_int[terms['table2'][i]][terms['second_term'][i]]]:
                            flag+=1
                elif terms['char'][i]==4:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] >= terms['second_term'][i]:
                        flag+=1
                elif terms['char'][i]==5:
                    if t_line[attr_to_int[terms['table1'][i]][terms['first_term'][i]]] <= terms['second_term'][i]:
                        flag+=1
            if flag==len(terms):#如果所有条件都满足，就将两张表中select中对应的属性的值加入result
                t_arrow=[]
                for j_t in range(0,len(select_name)):
                    t_arrow.append(dict_line[selects['table'][j_t]][attr_to_int[selects['table'][j_t]][selects['attr'][j_t]]]])
                result.loc[cnt]=[t_arrow[k] for k in range(0,len(t_arrow))]
                cnt+=1
if distinct==True:
    result=result.drop_duplicates()#进行distinct操作
    result.index=[k for k in range(0,len(result))]#去重后重新给行索引编号
result.to_csv("result_hash_join.csv",mode='a',encoding='UTF-8')
timeEnd=time.time()#结束计时
runTime=float('%.2f' % (timeEnd-timeStart))
print("查询已成功执行，结果请看result_hash_join.csv文件")

```

图 2.2.15

(3) 运行结果

测试用例

- 测试语句

```

select journals.id, journal, title
from papers,journals
where  journals.name = papers.journal  and papers.id<500 and class = 'A';

```

- 运行结果

结果显示查询时间为 2.38s（包括哈希时间），结果返回 183 行。

```

请输入查询语句:
select journals.id, journal, title
from papers, journals
where journals.name = papers.journal and papers.id<500 and class = 'A';
查询的表名为: ['journals.txt', 'papers.txt']
查询的属性为: ['journals.id', 'journal', 'title'] False
查询的条件为:
    first_term char second_term      table1      table2
0      name    3      journal  journals.txt  papers.txt
1      id     2      500    papers.txt  papers.txt
2      class   3      A    journals.txt  journals.txt
区分表名后查询的属性为:
    attr      table
0  id  journals.txt
1  journal  papers.txt
2  title  papers.txt
读取数据中, 请稍后.....
读取结束! 两个表的大小分别是: 4742 行、 1001 行
正在执行查询, 请稍后.....
查询已成功执行, 结果请看result_hash_join.csv文件
查询用时 2.38 s, 查询结果返回 183 行

```

图 2.2.16

- 结果查看

【说明】由于是对 journals 表和 papers 表的 name 和 journal 属性进行哈希的，哈希后顺序会被破坏，因此连接后的结果也是无序的。

	A	B	C	D	E
1		id	journal	title	
2	0	826	孔学研究	cxblf1milereyoyj	
3	1	3013	黑龙江教育	rhlkqe8o0wjiyorg	
4	2	418	杂文月刊	i8pqwchzj12bb102	
5	3	2428	百色学院学	gpt6ccpdu66xs6m	
6	4	2021	山东农业大	oopheyww7t4r9mf	
7	5	4055	中国标准化	gdemsugh8z402nh	
8	6	4707	投资与合作	ieymwwxge85a7kp	
9	7	188	东北史地	v56fdmw9aozdn1m	
10	8	129	汉语学报	217grbk016442e0	
11	9	1840	西安电子科	7hv4map11j1shde	
12	10	2141	深圳信息职	8xbhn18picjc71n	
13	11	536	今日民航	nnchgcxfa7itq25	
14	12	2900	湖南教育	g609scwuksa4xvr	
15	13	4105	发展	n4hzn2q5isugaxd	
16	14	4105	发展	k70b9tryo9exhz6	
17	15	1364	学校党建与	a142fm56g4w6gd8	
18	16	1996	江西师范大	ixryfxqstccgpon	
19	17	388	中国西部	ggow59mrr58sifd	
20	18	2861	体育博览	jex6mh0b5nh5eio	
21	19	2367	武汉工程职	edycaxii802sbt0	
22	20	3877	湖南商学院	bsc4v1zdco44vkj	
23	21	2678	数学学习与	kiicc0nezl1ziff4	
24	22	2455	安徽职业技	p2q18bu8kzqupj2	
25	23	1222	中国民政	y189ysegmhb73kp	
26	24	4580	特区展望	o5rd033k8up5ylx	
27	25	3637	当代财经	ufeofjdzwnte011	
28	26	4132	农业发展与	b9z7h6bm33xb30f	
29	27	67	自然辩证法	ac4wzi77elraqs5	
30	28	739	江河文学	8hlfwydh1338ri9	

图 2.2.17

综合对比

表 2.2 三种连接对比

连接类型	嵌套循环连接	归并连接	哈希连接
运行时间	>>24s	7.16s	2.38s
返回行数	2	183	183

结果分析：从上述连接结果可以看出，对于同一个查询语句，哈希连接效率最高，归并次之，嵌套循环连接排在最后。

(三) 查询执行计划

1、选择运算

给定 SQL 语句如下：

```
USE DBCOURSE
SELECT TITLE FROM PAPERS;
```

(1) 通过下述两种操作，使得系统选择不同的选择运算操作（即表扫描和索引扫描）。

在原始语句的情况下进行表扫描，执行时间为 1min12s.

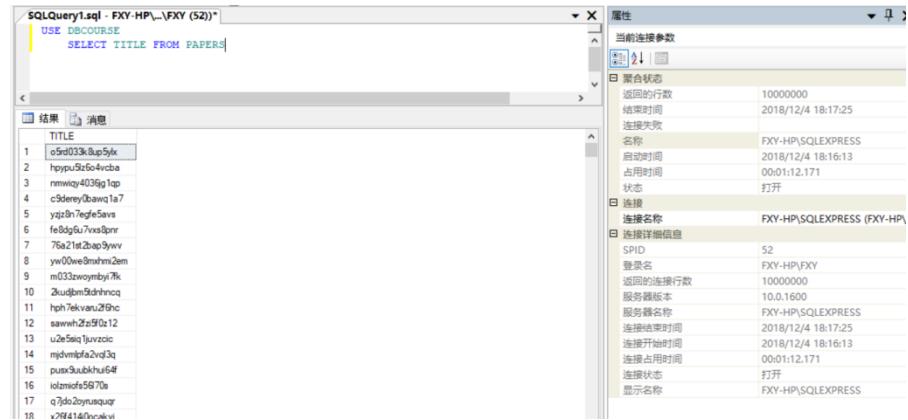


图 3.1.1

- a. 对相应数据集进行构建索引等操作；
- 首先对查询的属性列 title 建立索引

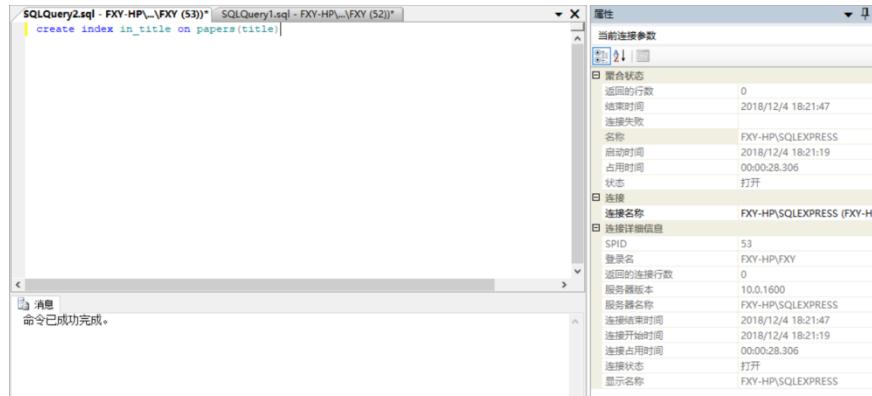


图 3.1.2

- 显示执行计划，如图 3.1.3 所示，该查询将会使用索引扫描

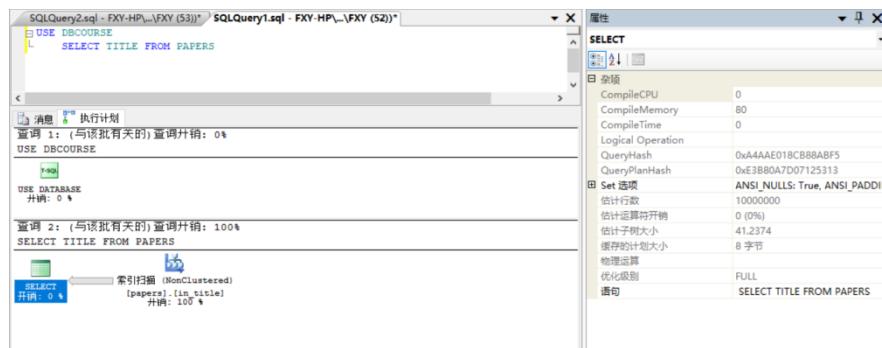


图 3.1.3

- 图 3.1.4 显示在建立索引后，查询运行时间为 1min9s，比表扫描稍微快一些。

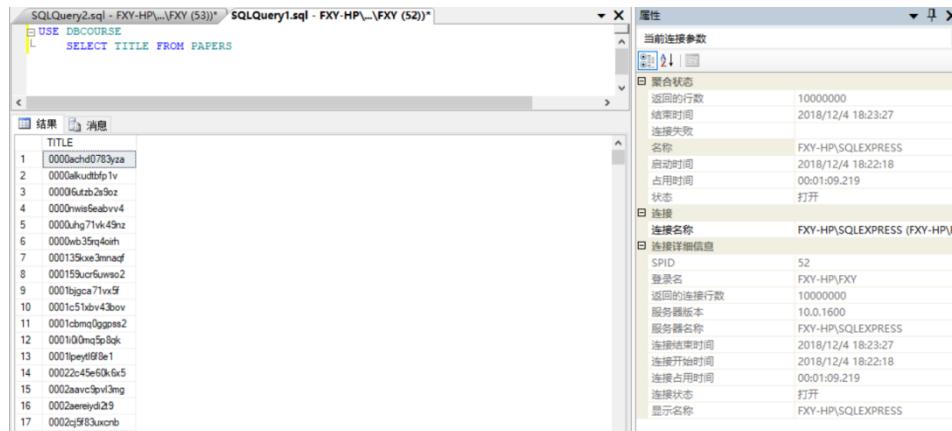


图 3.1.4

表 3.1.1 表扫描与索引扫描对比

扫描方式	表扫描	索引扫描
耗时	1min12s	1min9s

结果分析：对于查询所有行时，在查询的属性列上加上非聚集索引可以增加查询速度。

b. 通过对相应查询的选择条件等进行调整来实现数据集大小的控制。

根据选择率可以控制执行计划为表扫描还是索引扫描。设选择条件分别为 $id \leq 100$ （选择率=0.001%）和 $id \leq 5000000$ （选择率=50%）

- 首先在 id 上建立索引。

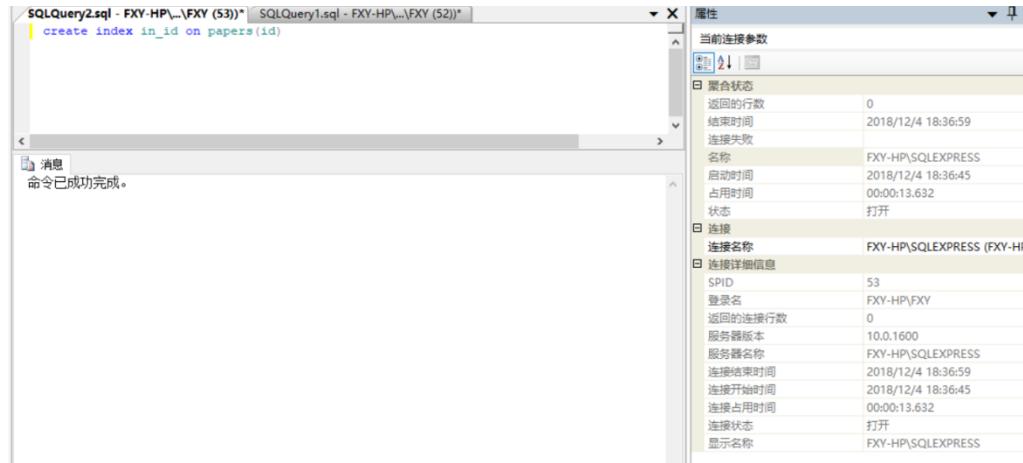


图 3.1.5

情形 1: $id \leq 100$

- 查看执行计划，如图 3.1.6 示，该查询运行了索引扫描。

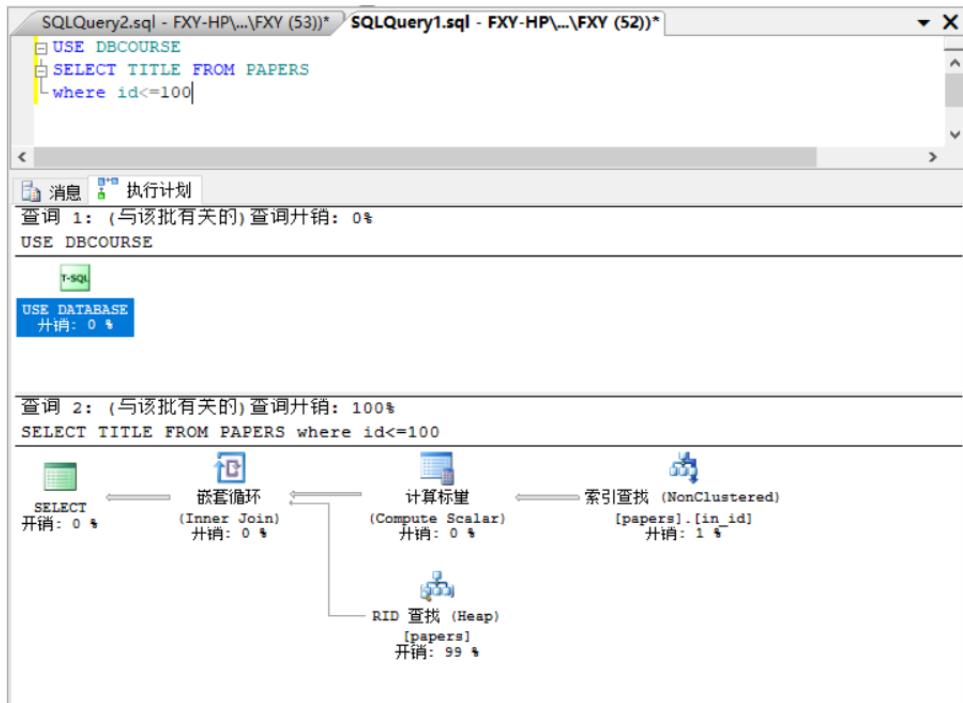


图 3.1.6

- 图 3.1.7 显示运行结果，查询用时 137ms。

The screenshot shows the SQL Server Management Studio interface. In the top window, a query is run:

```
USE DBCOURSE
SELECT TITLE FROM PAPERS
where id<100
```

The results pane displays 20 rows of data from the PAPERS table, where id < 100. The properties pane on the right shows connection details for the current session.

图 3.1.7

情形 2: id<=5000000

- 执行计划, 图 3.1.8 显示运行表扫描。

The screenshot shows the execution plan for the query:

```
USE DBCOURSE
SELECT TITLE FROM PAPERS
where id<=5000000
```

The execution plan details two steps: a '缺少索引' (Missing Index) step and a '表扫描' (Table Scan) step. The '表扫描' step is highlighted with a yellow background.

图 3.1.8

- 图 3.1.9 为运行结果, 查询时间为 40.458s

The screenshot shows the results of the query and its properties. The results pane shows the same 20 rows of data as in Figure 3.1.7. The properties pane on the right shows connection details for the current session, with the '连接' (Connection) section highlighted.

图 3.1.9

表 3.1.2 不同选择条件下执行不同执行计划的情况

情形	$\text{id} \leq 100$	$\text{id} \leq 5000000$
选择率	0.001%	50%
执行时间	137ms	40.458s

结果分析：

- 当查询语句中选择率很低时（一般低于 10%），用索引扫描会更快。
- 当选择率比较高时，即使在属性列上建有索引，DBMS 会选择表扫描进行查询；这是因为频繁进行索引查询的代价和要高于只扫描表一次。

(2) 通过修改默认执行计划比较不同扫描

a. 表扫描

修改 SQL Server 默认执行计划，使原来使用表扫描的选择运算采用索引扫描。并对比上述两种选择运算在该查询中的查询执行计划、查询执行结果和查询执行具体时间，以估算的形式给出特定语句的执行代价，说明 SQL Server 默认选择表扫描的原因。

- 将表扫描强制转为索引扫描的方式：在表名后面加上语句 **with (index(index_name))**
- 执行的 SQL 语句如下：

```
USE DBCOURSE
SELECT TITLE
FROM PAPERS with(index (in_id))
where id<=5000000
```

- 查询执行计划，如图 3.1.10 所示，可以看出此时该查询将进行索引查找：

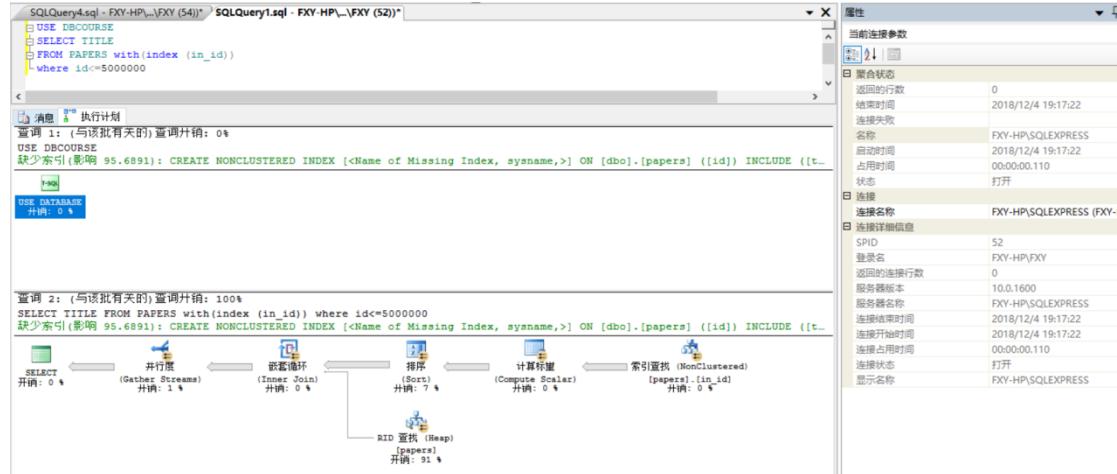


图 3.1.10

- 图 3.1.11 为运行结果，查询时间为 44s

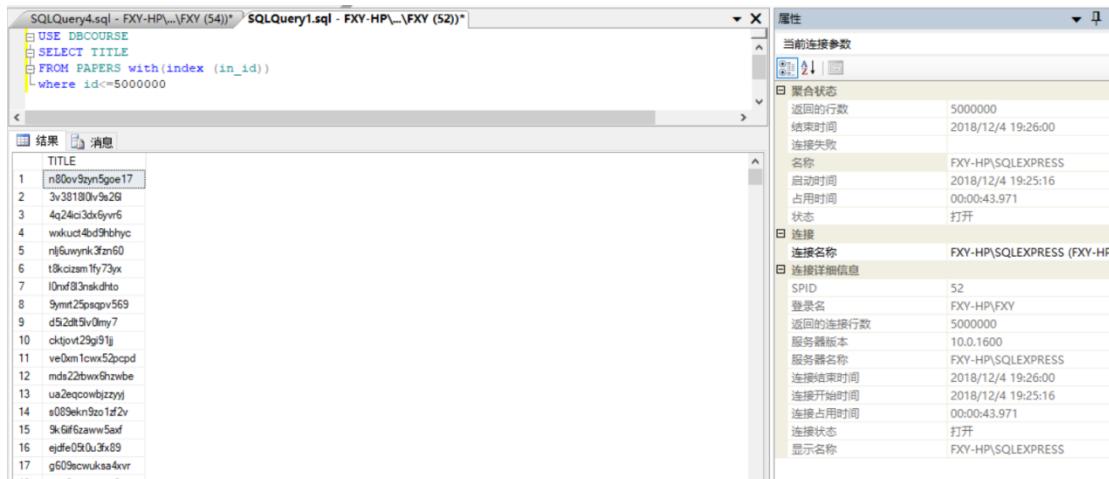


图 3.1.11

表 3.1.3 默认为表扫描与强制索引扫描的对比

扫描方式	表扫描 (DBMS 默认)	索引扫描
耗时	40.458s	43.971s

- 估算执行代价:

表扫描	$E_{table_scan} = b_r = 5000000$
索引扫描 (辅助索引, 非等值比较)	$E_{index_scan} = HT_i + LB_i * SF + n_r * SF = 7500006$

索引扫描分析: id 有 10000000 个不同的值, 假设每个结点放 20 个指针对, 每个结点至少半满, 故叶节点有 1000000~5000000, LB_i=5000000, 且树高为 HT_i=log(10000000,20)=6, SF=50%, 因此 E=6+5000000*50%+10000000*50%=7500006>5000000

- 为何默认为表扫描:

当选择率比较高时, 即使在属性列上建有索引, DBMS 会选择表扫描进行查询; 这是因为频繁进行索引查询的代价和要高于只扫描表一次。

b. 索引扫描

修改 SQL Server 默认执行计划, 使原来使用索引扫描的选择运算采用表扫描。并对比上述两种选择运算在该查询中的查询执行计划、查询执行结果和查询执行具体时间, 以估算的形式给出特定语句的执行代价, 说明 SQL Server 默认选择索引扫描的原因。

- 将索引扫描强制转为表扫描的方式: 在表名后面加上 **with (index (0))**
- 执行的 SQL 语句:

```
USE DBCOURSE
SELECT TITLE
FROM PAPERS with(index (0))
where id<=100
```

- 查询执行计划, 图 3.1.12 显示将执行表扫描:

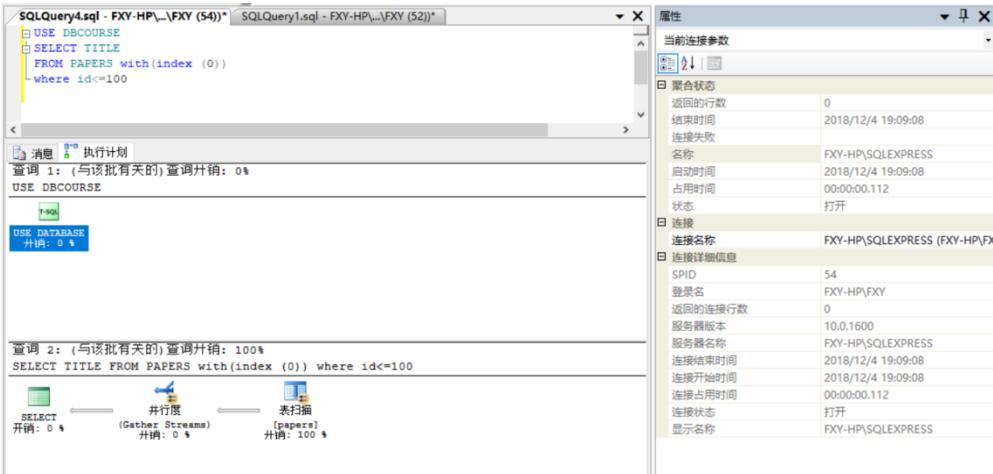


图 3.1.12

- 图 3.1.13 显示运行结果，运行时间 11s:

SQLQuery1.sql - FXY-HP\...\FXY (54)"		SQLQuery1.sql - FXY-HP\...\FXY (52)"																											
USE DBCOURSE	SELECT TITLE	FROM PAPERS with(index (0))	where id<=100																										
结果																													
<table border="1"> <thead> <tr> <th>TITLE</th> </tr> </thead> <tbody> <tr><td>1 c9d033k8up5yk</td></tr> <tr><td>2 hpypp9z6o4vba</td></tr> <tr><td>3 nmwqy-036g1ap</td></tr> <tr><td>4 c3dreyjbaunqta7</td></tr> <tr><td>5 ygd8t7egfesava</td></tr> <tr><td>6 fe8dgfuvxx8prn</td></tr> <tr><td>7 78a21at2bg9yvv</td></tr> <tr><td>8 yw00we9mhmz2em</td></tr> <tr><td>9 m033zwonyby7k</td></tr> <tr><td>10 2kuobm9dhhncq</td></tr> <tr><td>11 hgh7ekwanz9hc</td></tr> <tr><td>12 sawwh2z9f0x12</td></tr> <tr><td>13 u2e5eq1juvcoc</td></tr> <tr><td>14 njdvmlph2wv3q3</td></tr> <tr><td>15 pux3uakbhjw4f</td></tr> <tr><td>16 idzmoif5970s</td></tr> <tr><td>17 q7ds20ynusquq</td></tr> <tr><td>18 x2f4140ccavkj</td></tr> <tr><td>19 kubihqb8244r</td></tr> <tr><td>20 5n5pbh2rn3vgb</td></tr> <tr><td>21 fwott2z9659re</td></tr> <tr><td>22 d19e5ahg5nts9</td></tr> <tr><td>23 powoxth49ha</td></tr> <tr><td>24 t54d3mr8031nt</td></tr> <tr><td>25 1vox4nijsqnl8q</td></tr> </tbody> </table>				TITLE	1 c9d033k8up5yk	2 hpypp9z6o4vba	3 nmwqy-036g1ap	4 c3dreyjbaunqta7	5 ygd8t7egfesava	6 fe8dgfuvxx8prn	7 78a21at2bg9yvv	8 yw00we9mhmz2em	9 m033zwonyby7k	10 2kuobm9dhhncq	11 hgh7ekwanz9hc	12 sawwh2z9f0x12	13 u2e5eq1juvcoc	14 njdvmlph2wv3q3	15 pux3uakbhjw4f	16 idzmoif5970s	17 q7ds20ynusquq	18 x2f4140ccavkj	19 kubihqb8244r	20 5n5pbh2rn3vgb	21 fwott2z9659re	22 d19e5ahg5nts9	23 powoxth49ha	24 t54d3mr8031nt	25 1vox4nijsqnl8q
TITLE																													
1 c9d033k8up5yk																													
2 hpypp9z6o4vba																													
3 nmwqy-036g1ap																													
4 c3dreyjbaunqta7																													
5 ygd8t7egfesava																													
6 fe8dgfuvxx8prn																													
7 78a21at2bg9yvv																													
8 yw00we9mhmz2em																													
9 m033zwonyby7k																													
10 2kuobm9dhhncq																													
11 hgh7ekwanz9hc																													
12 sawwh2z9f0x12																													
13 u2e5eq1juvcoc																													
14 njdvmlph2wv3q3																													
15 pux3uakbhjw4f																													
16 idzmoif5970s																													
17 q7ds20ynusquq																													
18 x2f4140ccavkj																													
19 kubihqb8244r																													
20 5n5pbh2rn3vgb																													
21 fwott2z9659re																													
22 d19e5ahg5nts9																													
23 powoxth49ha																													
24 t54d3mr8031nt																													
25 1vox4nijsqnl8q																													
消息																													
返回的行数 100 结束时间 2018/12/4 19:15:21 连接失败 名称 FXY-HP\SQLEXPRESS 启动时间 2018/12/4 19:09:08 占用时间 00:00:11.954 状态 打开																													
连接																													
连接名称 FXY-HP\SQLEXPRESS (FXY-HP\FX SPID 54 登录名 FXY-HP\FXY 返回的连接行数 0 服务器版本 10.0.1600 服务器名称 FXY-HP\SQLEXPRESS 连接结束时间 2018/12/4 19:09:08 连接开始时间 2018/12/4 19:09:08 连接占用时间 00:00:0.112 连接状态 打开 显示名称 FXY-HP\SQLEXPRESS																													
属性																													
当前连接参数																													

图 3.1.11

表 3.1.4 默认为索引扫描和强制表扫描的对比

扫描方式	索引扫描 (DBMS 默认)	表扫描
耗时	0.137s	11s

- 估算执行代价:

表扫描	$E_{table_scan} = b_r = 5000000$
索引扫描 (辅助索引, 非等值比较)	$E_{index_scan} = HT_i + LB_i * SF + n_r * SF = 156$

索引扫描分析: id 有 10000000 个不同的值, 假设每个结点放 20 个指针对, 每个结点至少半满, 故叶节点有 1000000~5000000, LB_i=5000000, 且树高为 HT_i=log(10000000,20)=6, SF=50%, 因此 E=6+5000000*0.001%+10000000*0.001%=156<<5000000

- 为何默认为索引扫描:

当查询率较低时, 进行索引扫描代价更低, 原因在于只需进行少数几次 B+树的查找就

可以找到全部满足的元组，且因为 B+树一般比较低，一次树的搜索代价很小，因此索引扫描的总代价就很低；而表扫描需要将整个表扫描一遍才能找到需要的结果，代价相对较高。

2、连接运算

给定 SQL 语句如下：

```
USE DBCOURSE  
SELECT DISTINCT PAPERS.ORG  
FROM PAPERS JOIN AUTHORS ON PAPERS.AUTHOR=AUTHORS.NAME
```

(1) 通过下述两种操作，使得系统选择不同的连接运算操作（即嵌套循环连接、归并连接和散列连接）。

- 对相应数据集进行构建索引等操作；
- 通过对相应查询的选择条件等进行调整来实现数据集大小的控制。

首先，在连接的属性列上建立索引

```
create index in_name on authors(name)  
create index in_author on papers(author)
```

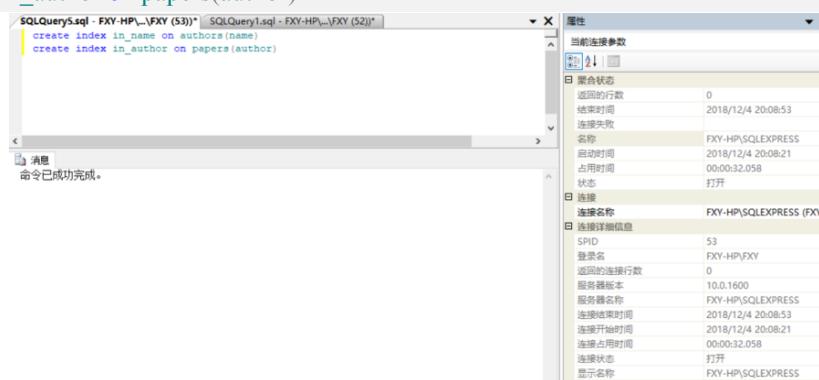


图 3.2.1

这部分内容详情可见下面(2)中 a,b,c 三部分中每部分的默认执行部分。

- 当在原语句上加上 where 子句“WHERE SEX='女' AND PAPERS.ID<=500 AND AGE BETWEEN 20 AND 30”后，默认执行计划改为嵌套循环连接；
- 当在原语句连接属性 papers.author 和 authors.name 上建立聚集索引，且 where 子句加上“SEX='女'”，最后加上 order by authors.name，这样后默认执行计划改为归并连接；
- 直接使用原语句，默认执行计划为散列连接。

(2) 通过修改默认执行计划比较不同连接方式

根据 Microsoft 官方手册，修改查询的连接方式可以通过 **OPTION** 子句实现，其参数有三个 **LOOP JOIN, HASH JOIN, MERGE JOIN**，分别表示嵌套循环连接、哈希连接、归并连接。

a. 嵌套循环连接

修改 SQL Server 默认执行计划，使原来使用嵌套循环连接的连接运算采用归并连接和散列连接。并对比上述三种连接运算在该查询中的查询执行计划、查询执行结果和查询执行具体时间，以估算的形式给出特定语句的执行代价，说明 SQL Server 默认选择嵌套循环连接的原因。

①默认执行计划为嵌套循环连接

- SQL 语句:

```
USE DBCOURSE
```

```
SELECT DISTINCT PAPERS.ORG,AUTHORS.NAME
FROM PAPERS JOIN AUTHORS ON AUTHOR=NAME
WHERE SEX='女' AND PAPERS.ID<=500 AND AGE BETWEEN 20 AND 30
```

- 查询执行计划, 图 3.2.2 显示该查询的默认执行计划为嵌套循环:

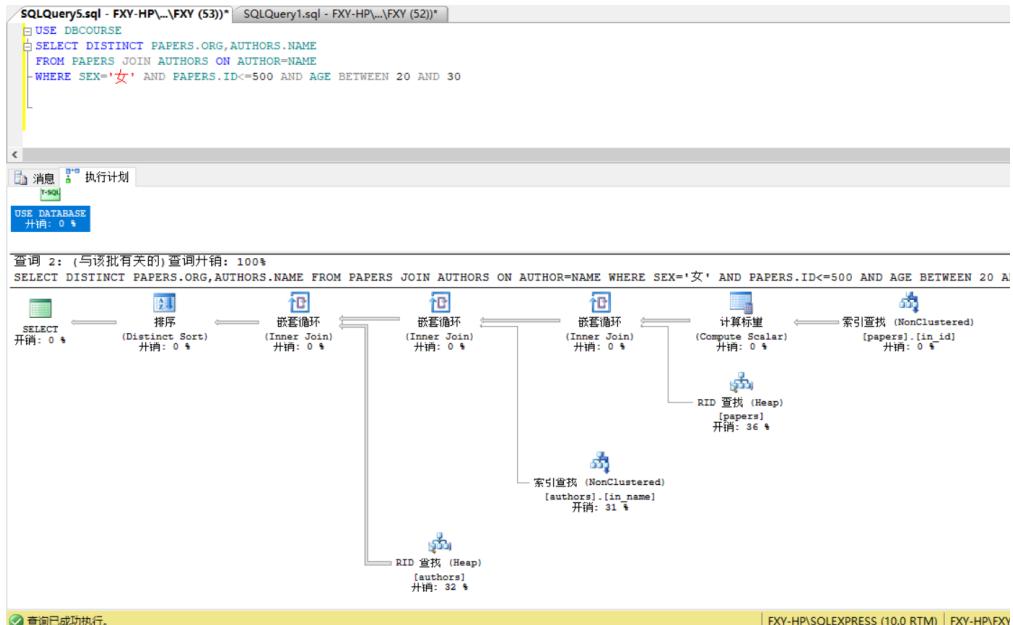


图 3.2.2

- 运行结果, 图 3.2.3 显示耗时 215ms, 返回 78 行

ORG	NAME
1 00009pgjb3qg4hg4drg49h2kf4f6gw8bj537ad4atrpum75...	刘炳欣
2 0xhg4g4q7pe3762u407wm70w9a63jherh63or5pyv21u...	钟九洲
3 0zantjt14ac1599k5v3vbfh038n1q1bnwva75j5ev47oba5n1...	卢方明
4 1a7lywpm5g11gwovyf61gg2ahv105lraey7vafalun0emer...	吕耀华
5 20kcm3wk17y0weeb1qsttc24mzx12x4a6918aqkhs92...	张承铭
6 3073iuults2v7nnq5nk7twdc2ghp4m8ybqjtb6av8vfg...	李中祥
7 3394428dhwk8keeiypagne5yuhfph5d4dm0h43pg9fbzq...	萧国亮
8 5m66zenq0kmvb0m6a6x77qxezydrns5dhbn2eacsaye5...	谢佑平
9 5yk60ccuz25u19b6rcpf4q1qlrwwuavxae3c69lyz0qfbouc...	李明国
10 607qdrcwiwl5d3ecwpd1uicwu8kwogwh1o4gco1vqsvoy9zu...	赵鹰
11 682wn10mve5odidfaur4jona2ci948e4d1chwx0tphh9yadg...	陈效林
12 6n3tz7d284yf111374uv2nqdpcciliv1ycpcfey5a0v0jean...	孟令超
13 6p9v9nrcexdsbw39ctvhedv2zq1ymcfk6msgfmf6cf64...	刘细军
14 6w7c3av1x10qy7dfy2mlcx2udjdtg1p05jgnr2edumar0c32g...	苏德强
15 6walsc42v09b525fc423v95bek3ndavik45mx3uktwq2...	付麟

图 3.2.3

②归并连接

- 在查询语句最后加上 **OPTION(MERGE JOIN)**, 可强制将执行计划变为归并连接。SQL 语句如下:

```
USE DBCOURSE
```

```
SELECT DISTINCT PAPERS.ORG,AUTHORS.NAME
FROM PAPERS JOIN AUTHORS ON AUTHOR=NAME
WHERE SEX='女' AND PAPERS.ID<=500 AND AGE BETWEEN 20 AND 30
```

OPTION(MERGE JOIN)

- 查询执行计划，从图 3.2.4 可以看到此时的查询计划为归并连接，而非默认的嵌套循环连接。

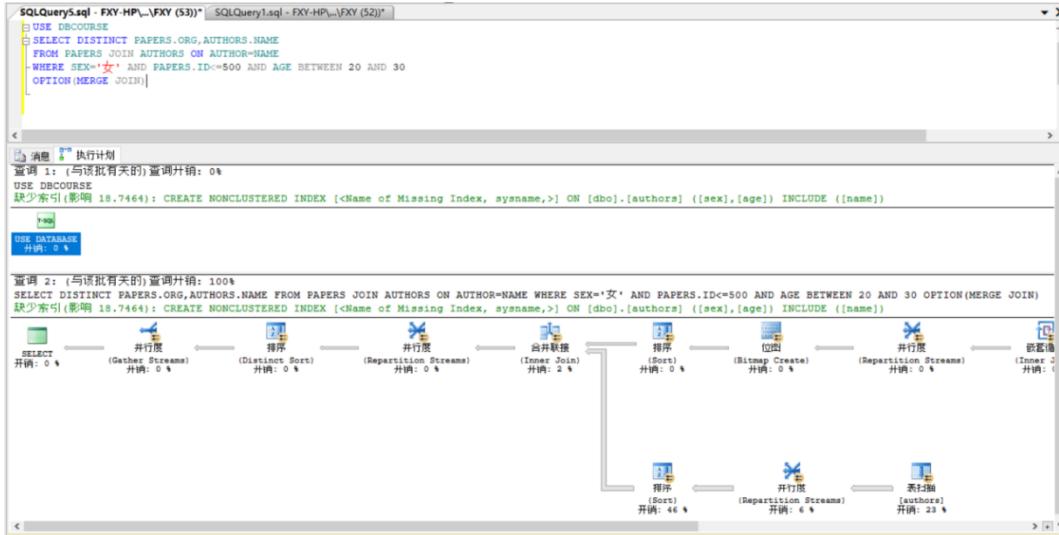


图 3.2.4

- 运行结果，图 3.2.5 显示耗时 243ms，返回 78 行

ORG	NAME
1	刘鹤欣
2	孟令超
3	刘继军
4	付娟
5	郭起华
6	王礼杰
7	姚雪丹
8	葛小杰
9	陆理群
10	常梦云
11	步丑男
12	杨宗义
13	周安昌
14	张孟才
15	孙长群
16	吕晨
17	和万祥
18	李振杰
19	耿宝军

图 3.2.5

③ 散列连接

- 同样，在查询语句最后加上 **OPTION(HASH JOIN)**，强制进行哈希连接

```
USE DBCOURSE
```

```
SELECT DISTINCT PAPERS.ORG,AUTHORS.NAME
FROM PAPERS JOIN AUTHORS ON AUTHOR=NAME
WHERE SEX='女' AND PAPERS.ID<=500 AND AGE BETWEEN 20 AND 30
OPTION(HASH JOIN)
```

- 查询执行计划，从图 3.2.6 可以看到此时将执行哈希连接而非嵌套循环连接

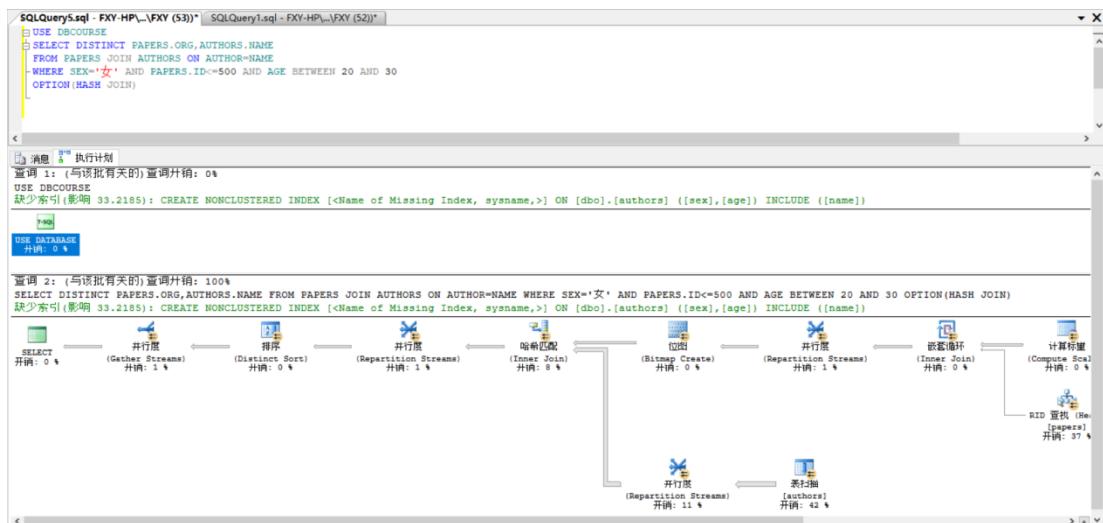


图 3.2.6

- 运行结果，耗时 245ms，返回 78 行。

ORG	NAME
1	30731uota2v7mnq5mk7tjwcd2ghpf4m8ygbxgtbr6av3pvfq...
2	33944z28dnw8keeyepgns5yhrhp5dk44dm043pg9nfzq...
3	5n66zenq0kmnb03m6x7q7nve2ydm5sdhbn2e2csayje5...
4	6w7c9avx10y7dfy2mlcx2udjct1p05ygnr2edumar0c32...
5	8log0cy4o77s0x7axn7i9wgeg56975mgfhltoymzldhnn3...
6	9cmdzvny6q1cqsc4770y1cb8pnpd9kabm30nbne8o8f95...
7	c0eed0fhxwic8cqjuyykuq6a6v906qasqtb339h6714n09...
8	cmzeb53knh5uggyfrvibv132y0244yqdphy1nx7ad05739...
9	ga6d3k3alawm0091156952g2nv4npaprsq131an7aya7555...
10	g45eq2dogt64n1tquwf9vu3a42pk1uh3aqy7oywaf1ko9...
11	ij5wnvml8ad2x2d9tp2738q157y585g1045da6edg29qt...
12	keyfyhyd505s97uor9eznru1532h5vqpf7dyxfv3uqer7f...
13	kld9ccp436mykn8577x9t8myskgokuec37h3p2wbovpb...
14	nwlarpfr4rqnmvcogpq5hvhra5u4dwvk2cwcvwet46wwas...

图 3.2.7

表 3.2.1 默认为嵌套循环连接

连接方式	嵌套循环连接(默认)	归并连接	散列连接
查询时间	215ms	243ms	245ms
查询结果	78 行	78 行	78 行

- 估算执行代价：

嵌套循环连接 (默认)	$E_{loop_join} = b_a + n_a * c + E_{index_scan} = 3399$
归并连接	$E_{merge_join} = b_p + b_a = 5524$
散列连接	$E_{hash_join} = b_p + 3 * b_a = 6572$

【说明】

嵌套循环连接: authors 表中 name 约有 524000 个不同的值，每个节点放放 20 个指针对，每个结点至少半满，故叶节点有 26200~52400，LBi=52400，而 HTi= $\log(524000, 20)=5$ ，因此 $c=5+1=6$ 。np=10000000,na=524000,fp=2000,fa=1000 则 bp=5000,ba=524. 设满足 where 条件

的元组数为 N, N<=500,且从中筛选出性别为女且年龄在 20-30 之间的元组, 因此假 N=200, 即通过选择后 authors 表的 na 变为 200, 则 ba=3.故总代价为 $6*200+3=1203$ 。再加上选择的代价, 利用索引扫描的公式计算如下:

$$E_{index_scan} = HT_i + LB_i * SF + n_r * SF = 5 + 52400 * 0.00038 + 5240000 * 0.00038 = 2196$$

因此总代价为 $2196+1203=3399$

归并连接: 连接代价为 $bp+ba=5524$

散列连接: 按照公式 $bp+3*ba$ 代入计算可得 6572。

- 默认选择嵌套循环连接的原因:

连接方式中内部循环的表有序(也就是有索引), 并且外部循环表的行数要小于内部循环的行数, 否则查询分析器就更倾向于散列连接。而且在实验中也发现当 where 子句中条件很多时, 其他条件相同的情况下, DBMS 更倾向于嵌套循环连接而非散列(可对比 c. 中默认情况为散列连接的 SQL 语句, 比本次查询语句少了 where 子句), 原因可能是散列连接更适合数据量大且无索引的情况, 而当查询条件比较多且建有索引, 查询分析器可能认为最终查询的结果数据量很小, 因此更倾向于选择嵌套循环连接。

b. 归并连接

修改 SQL Server 默认执行计划, 使原来使用归并连接的连接运算采用嵌套循环连接和散列连接。并对比上述三种连接运算在该查询中的查询执行计划、查询执行结果和查询执行具体时间, 以估算的形式给出特定语句的执行代价, 说明 SQL Server 默认选择归并连接的原因。

①默认为归并连接

- 归并连接需要满足一些条件

i. 连接的属性列已经有序, 可由聚集索引和非聚集索引来保证归并连接的两端有序。因此先在属性列上建立聚集索引:

```
~vsD368.sql - FXY-HP\...|FXY (51)* ~vs851F.sql - FXY-HP\...|FXY (52)*
create clustered index clu_p_author on papers(author)
create clustered index clu_a_name on authors(name)

消息
命令已成功完成。
```

图 3.2.8

但执行计划却选择了哈希连接, **原因可能**是即使建立索引且保证有序, 但是由于结果表太大哈希效率依然比归并连接效率高。因此进行第 ii 种方案。

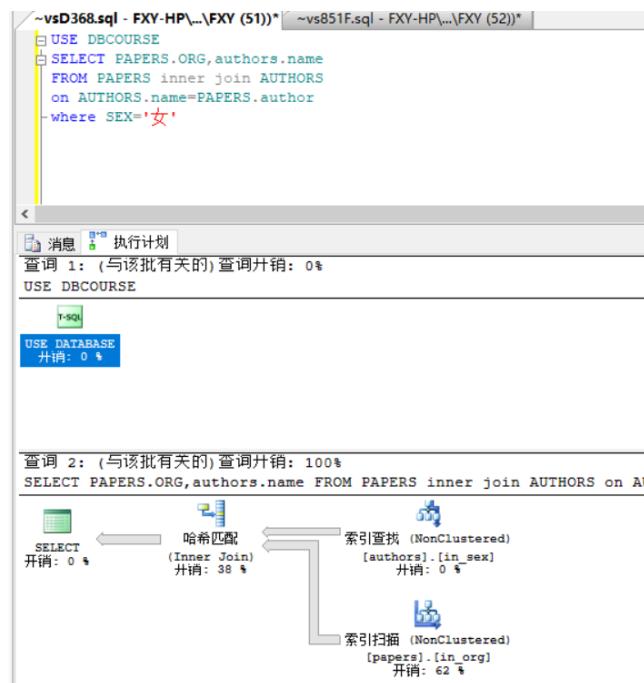


图 3.2.9

ii. 查询中存在 order by,group by,distinct 等可能导致查询分析器不得不进行显式排序，那么对于查询分析器来说，反正都已经进行显式排序了，那么直接利用排序后的结果进行成本更小的 MERGE JOIN 会是更好的选择。

- 在查询中加入 order by 子句，SQL 语句如下：

```

USE DBCOURSE
SELECT PAPERS.ORG,authors.name
FROM PAPERS inner join AUTHORS
on AUTHORS.name=PAPERS.author
where SEX='女'
order by authors.name

```

- 查询执行计划，如图 3.2.10 所示此时执行计划成功变为归并连接。

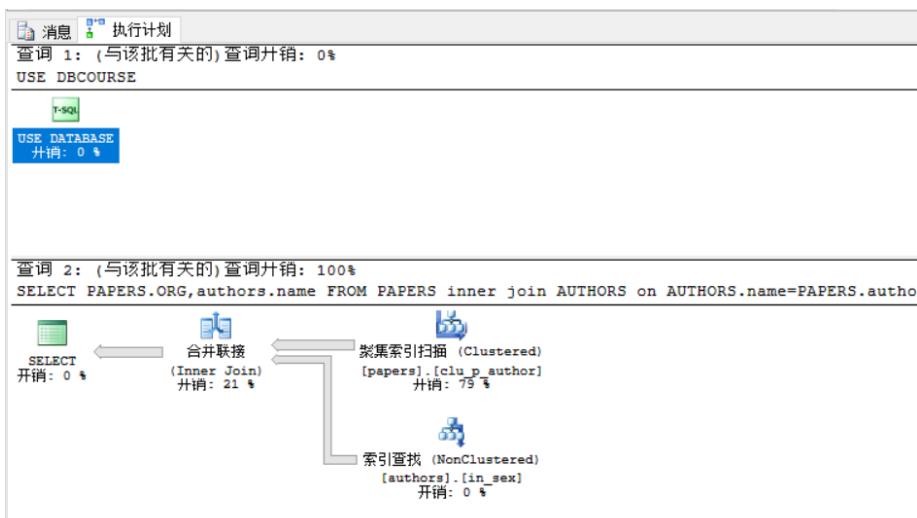


图 3.2.10

- 运行结果，1min3s，返回 4994258 行

图 3.2.11

②散列连接

- 在查询语句中加上 **option(hash join)**, 会使 DBMS 强制执行散列连接

USE DBCOURSE

```
SELECT PAPERS.ORG,authors.name  
FROM PAPERS inner join AUTHORS  
on AUTHORS.name=PAPERS.author  
where SEX='女'  
order by authors.name  
option(hash join)
```

- 查询执行计划，从图 3.2.12 可以看到此时执行散列连接

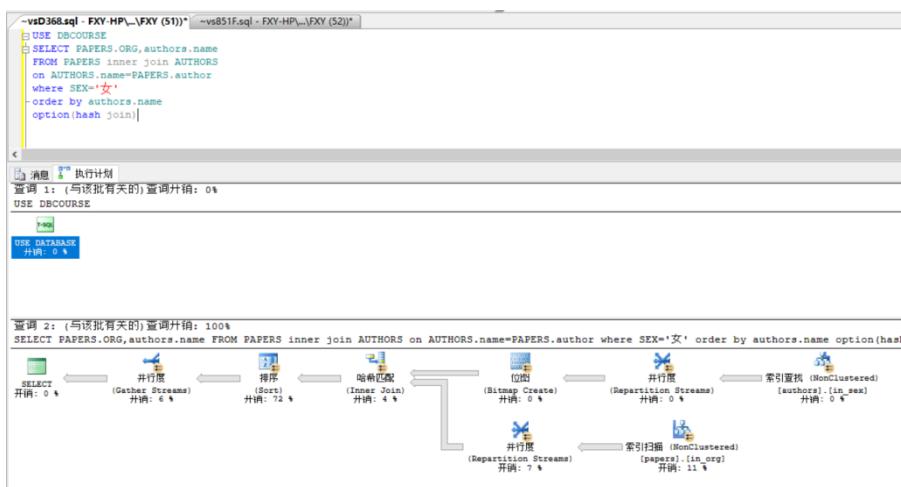


图 3.2.12

- 查看运行结果，结果显示耗时 1min24s，返回 4994258 行：

```

USE DBCOURSE
SELECT PAPERS.ORG,authors.name
FROM PAPERS inner join AUTHORS
on AUTHORS.name=PAPERS.author
where SEX='女'
order by authors.name
option(loop join)

```

图 3.2.13

③嵌套循环连接

- 强制执行嵌套循环连接：在查询语句后加上 **option(loop join)**

```
USE DBCOURSE
```

```

SELECT PAPERS.ORG,authors.name
FROM PAPERS inner join AUTHORS
on AUTHORS.name=PAPERS.author
where SEX='女'
order by authors.name
option(loop join)

```

- 查看执行计划，从图 3.2.14 可以看到，此时将执行嵌套循环连接。

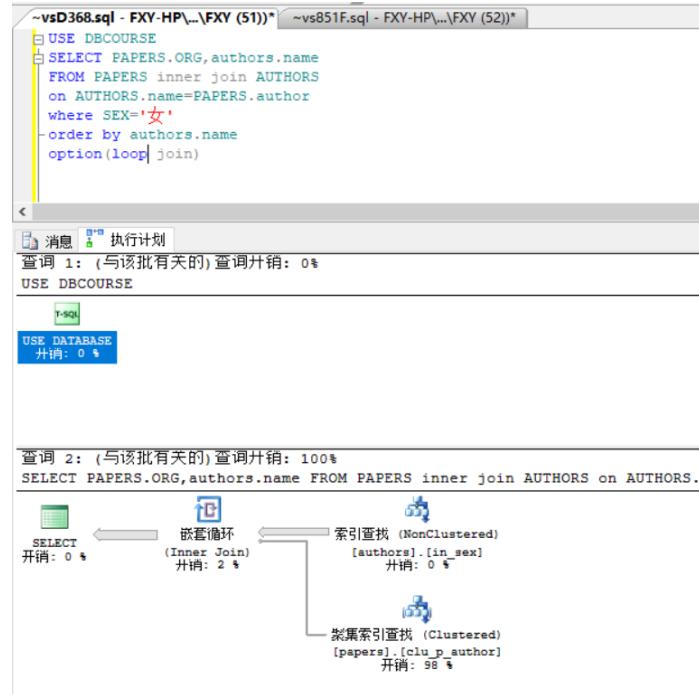


图 3.2.14

- 查看运行结果，耗时 3min28s，返回 4994258 行：

```

USE DBCOURSE
SELECT PAPERS.ORG,authors.name
FROM PAPERS inner join AUTHORS
on AUTHORS.name=PAPERS.author
where SEX='女'
order by authors.name
option (loop join)

```

ORG	name
1	阿巴耶夫
2	阿巴耶夫
3	阿巴耶夫
4	阿巴耶夫
5	阿巴耶夫
6	阿巴耶夫
7	阿巴耶夫
8	阿巴耶夫
9	阿巴耶夫
10	阿巴耶夫
11	阿巴耶夫
12	阿巴耶夫
13	阿巴耶夫
14	阿巴耶夫
15	阿巴耶夫
16	阿碧

图 3.2.15

表 3.2.2 默认为归并连接

连接方式	归并连接（默认）	散列连接	嵌套循环连接
查询时间	1min3s	1min24s	3min28s
查询结果	4994258 行	4994258 行	4994258 行

- 估算执行代价：

归并连接（默认）	$E_{merge_join} = b_p + b_a = 5524$
散列连接	$E_{hash_join} = b_p + 3 * b_a + 2 * b_p * \log_2(b_p) = 27800$
嵌套循环连接	$E_{loop_join} = b_a + n_a * c = 1572262$

【说明】

归并连接：连接代价为 $b_p + b_a = 5524$ 。

散列连接：代价分为两部分。一部分是连接得代价，用公式 $b_p + 3 * b_a$ 可以算得为 6572。另一部分是排序得代价，根据排序代价的公式算得代价为 21800，两者加起来为总代价 27800。

嵌套循环连接：authors 表中 name 约有 524000 个不同的值，每个节点放放 20 个指针对，每个结点至少半满，故叶节点有 26200~52400，LBi=52400，而 HTi= $\log(524000, 20)=5$ ，因此 $c=5+1=6$ 。 $np=10000000, na=524000, fp=2000, fa=1000$ 则 $bp=5000, ba=524$. 设满足 where 条件的元组数为 na ,且从中筛选出性别为女的元组，因此假设选择一半，即通过选择后 authors 表的 na 变为 $524000 * 0.5 = 262000$ ，则 $ba=262$. 故总代价为 $6 * 262000 + 262 = 1572262$ 。由于 authors.name 已经建立好索引，因此读取的时候已经有序，且嵌套循环连接不会改变顺序，故不再需要额外排序。

- 默认选择归并连接的原因：

DBMS 执行归并连接计划有两种情况，一种是连接属性上已经排好序（即建立好索引），另一种是有 order by 子句等。这里的查询语句中包含 order by，导致查询分析器不得不进行显式排序，那么对于查询分析器来说，反正都已经进行显式排序了，那么直接利用排序后的结果进行成本更小的归并连接会是更好的选择。

c. 散列连接

修改 SQL Server 默认执行计划，使原来使用散列连接的连接运算采用嵌套循环连接和归并连接。并对比上述三种连接运算在该查询中的查询执行计划、查询执行结果和查询执行具体时间，以估算的形式给出特定语句的执行代价，说明 SQL Server 默认选择散列连接的原因。

①默认为散列连接

- SQL 语句：

```
USE DBCOURSE
SELECT DISTINCT PAPERS.ORG
FROM PAPERS JOIN AUTHORS ON PAPERS.AUTHOR=AUTHORS.NAME
```

- 查看执行计划，显示运行散列连接。

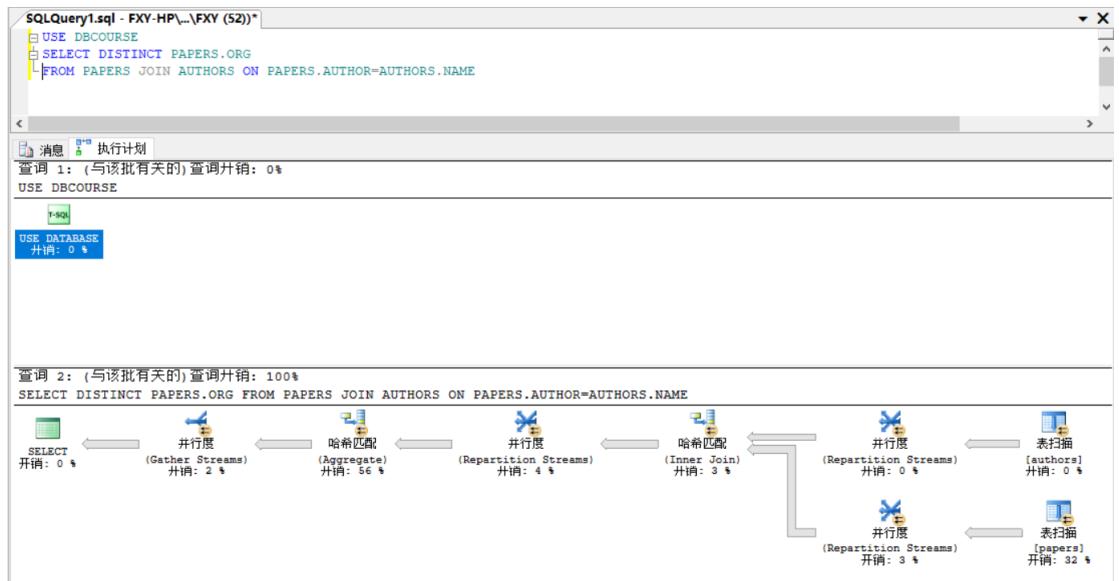


图 3.2.16

- 运行结果显示，执行时间 1min57s，返回 10000000 行

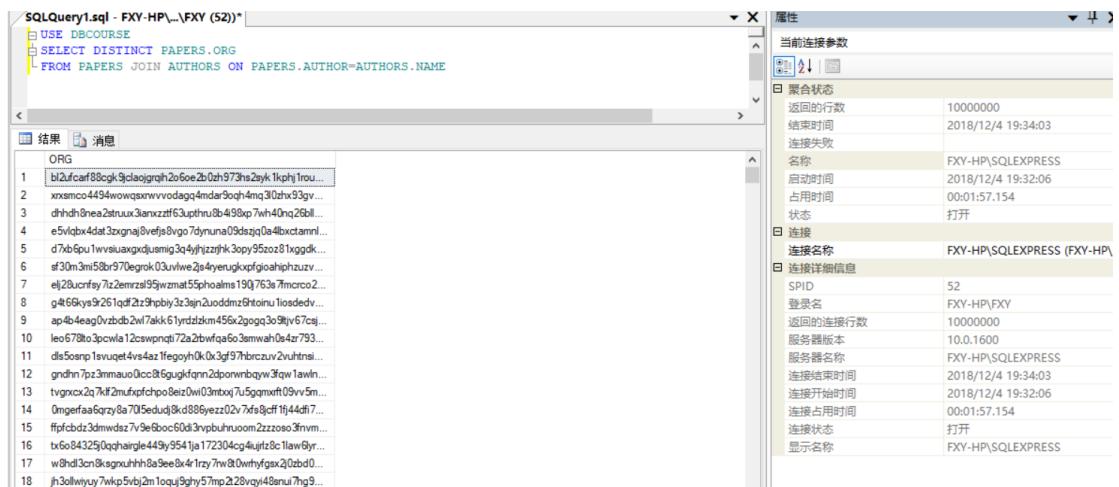


图 3.2.17

②归并连接

- 强制执行归并连接：在查询语句后加上 **OPTION(MERGE JOIN)**

```
USE DBCOURSE
```

```

SELECT DISTINCT PAPERS.ORG
FROM PAPERS JOIN AUTHORS ON PAPERS.AUTHOR=AUTHORS.NAME
OPTION(MERGE JOIN)

```

- 查看执行计划，图 3.2.18 显示将执行归并连接

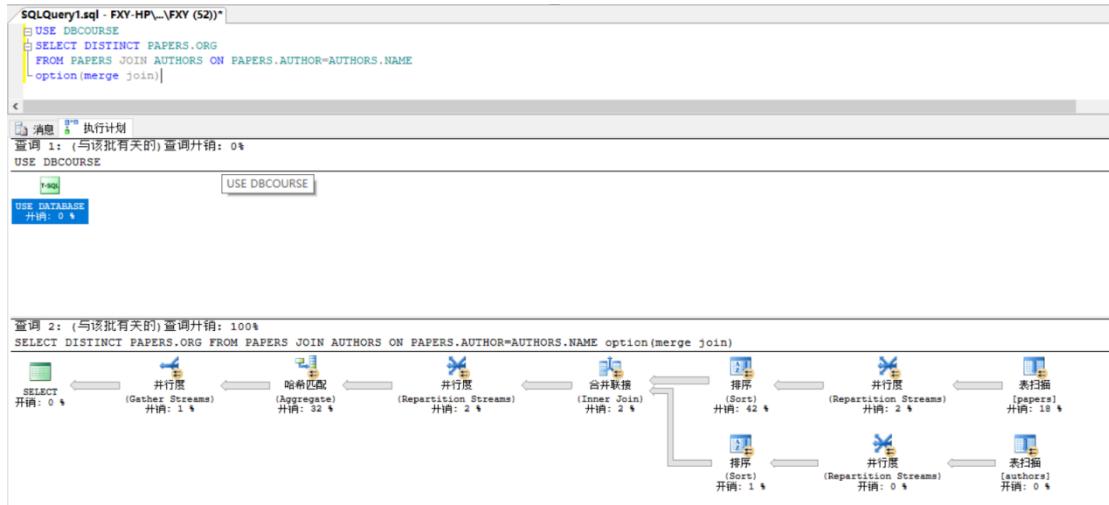


图 3.2.18

- 运行结果显示耗时 2mins26s，返回 10000000 行

属性	值
当前连接参数	
聚合状态	返回的行数: 10000000 结束时间: 2018/12/4 19:45:17 连接失败: 0 名称: FXY-HP\SQLEXPRESS 启动时间: 2018/12/4 19:42:51 占用时间: 00:02:26.307 状态: 打开
连接	连接名称: FXY-HP\SQLEXPRESS (FXY-HP\FXY) SPID: 52 登录名: FXY-HP\FXY 返回的连接数: 10000000 服务器版本: 10.0.1600 服务器名称: FXY-HP\SQLEXPRESS 连接结束时间: 2018/12/4 19:45:17 连接开始时间: 2018/12/4 19:42:51 连接占用时间: 00:02:26.307 连接状态: 打开 显示名称: FXY-HP\SQLEXPRESS

图 3.2.19

③嵌套循环连接

- 强制进行嵌套循环连接：在语句后加上 **OPTION(LOOP JOIN)**

```

USE DBCOURSE
SELECT DISTINCT PAPERS.ORG
FROM PAPERS JOIN AUTHORS ON PAPERS.AUTHOR=AUTHORS.NAME
OPTION(LOOP JOIN)

```

- 查看执行计划，图 3.2.20 显示将执行嵌套循环连接

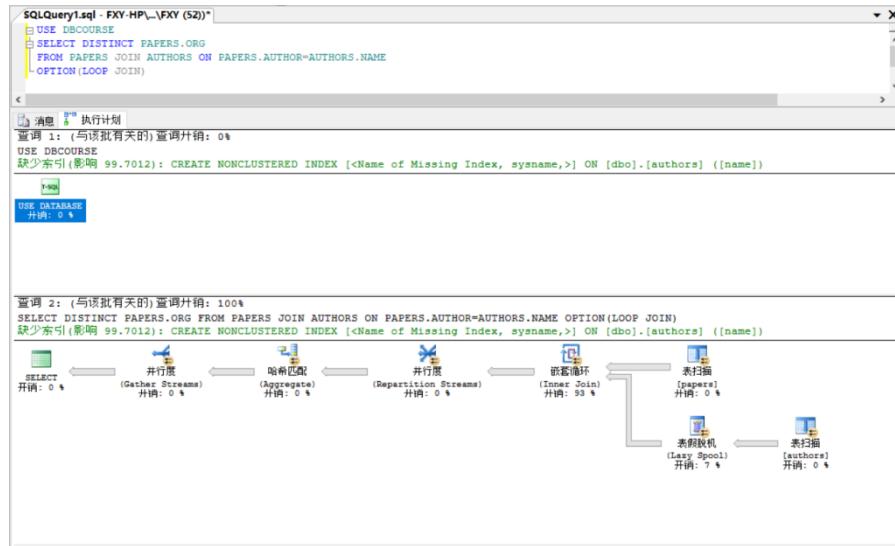


图 3.2.20

- 执行结果，用时已经超过 4mins，用时均比哈希连接和归并连接大，因此取消了查询。



图 3.2.21

表 3.2.3 默认为散列连接

连接方式	散列连接（默认）	归并连接	嵌套循环连接
查询时间	1min57s	2mins26s	>4mins
查询结果	10000000 行	10000000 行	10000000 行

- 估算执行代价：

散列连接（默认）	$E_{hash_join} = b_p + 3 * b_a = 6572$
归并连接	$E_{merge_join} = b_p + b_a + 2 * (b_p * \log_2 b_p + b_a * \log_2 b_a) = 143391$
嵌套循环连接	$E_{loop_join} = b_a + n_a * c = 3344524$

【说明】

散列连接（默认）：按照公式 $b_p + 3 * b_a$ 计算得到为 6572。

归并连接：连接代价为 $b_p + b_a = 5524$ ，排序代价根据估算公式得为 137867，总代价为 143391。

嵌套循环连接：authors 表中 name 约有 524000 个不同的值，每个节点放 20 个指针，

每个结点至少半满，故叶节点有 $26200 \sim 52400$, $LBi=52400$, 而 $HTi=\log(524000,20)=5$, 因此 $c=5+1=6$ 。 $np=10000000, na=524000, fp=2000, fa=1000$ 则 $bp=5000, ba=524$. 设满足 where 条件的元组数为 $na=524000$, 则 $ba=524$. 故总代价为 $6*524000+524=3144524$ 。

- 默认选择散列连接的原因：

当查询涉及到大量数据时，且数据一般无序时，散列连接的效果要好于归并连接和嵌套循环连接。这个查询涉及到大量的数据，且没有 where 子句条件进行再删选（即不会用到 where 子句涉及的属性列上的索引）就不会倾向选择嵌套循环连接；且连接属性是无序的，因此也不会选择归并连接。

(四) SQL 调优

1、原始语句的执行

- SQL 语句：

```
select n_name, sum(l_extendedprice*(1-l_discount)) as revenue
from customer,orders,lineitem,supplier,nation,region
where c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= '1994-01-01'
    and o_orderdate < '1995-01-01'
group by n_name
order by revenue desc;
```

- 运行结果，耗时 1920ms

N_NAME	revenue
INDONESIA	58355388
VIETNAM	58272781
CHINA	56572386
INDIA	54835573
JAPAN	47801690

图 4.1

2、优化处理

上次建立索引只在主键上建立了索引，且查询在临时表上查询。这次在所有涉及到的属性列上建立索引，采用了改变表连接顺序的优化方法（从小表到大表）且连接条件也按照表的顺序排列。

```
create index clu_c_custkey on customer(c_custkey)
create index clu_o_orderkey on orders(o_orderkey)
create index clu_s_suppkey on supplier(s_suppkey)
create index clu_n_nationkey on nation(n_nationkey)
create index clu_r_regionkey on region(r_regionkey)
create index non_c_nationkey on customer(c_nationkey)
create index non_s_nationkey on supplier(s_nationkey)
create index non_o_custkey on orders(o_custkey)
create index non_o_orderdate on orders(o_orderdate)
create index non_n_regionkey on nation(n_regionkey)
create index non_n_name on nation(n_name)
create index non_l_orderkey on lineitem(l_orderkey)
create index non_l_suppkey on lineitem(l_suppkey)
create index non_l_extendedprice on lineitem(l_extendedprice)
create index non_l_discount on lineitem(l_discount)
```

- 执行结果，显示耗时 420ms

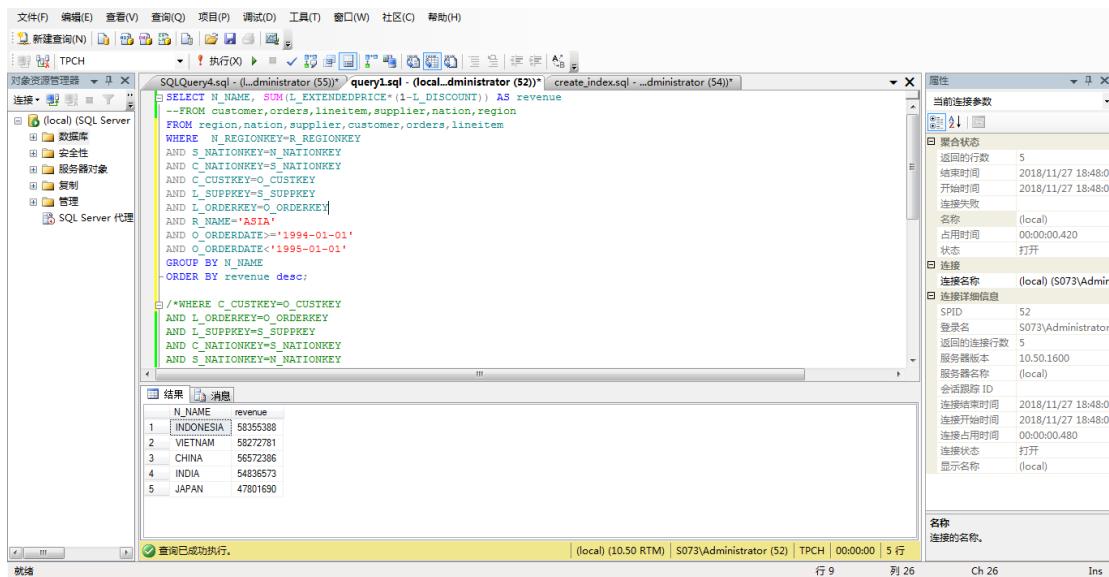


图 4.2

五、实验总结

(一) 查询优化基本思路

- 1.当查询语句较为复杂时，画出语法树分析语法。基于语法树优化后写出 SQL 语句，可以提高查询效率；
- 2.注意表的连接顺序，连接顺序不同对查询结果影响很大；
- 3.在合适的属性列上建立不同类型的索引，可以加快查询效率；
- 4.必要的时候可以使用临时表，如对一个表或几个表需要反复查询，但是查询只进行一次的时候不建议如此，因为建立临时表的代价也很大；

(二) 不足与展望

由于时间有限，在算法设计部分目前只对 authors 表上的 id 和 age 属性（因为为 int 值）建立了 B+ 树索引。程序一进去就开始自动建立索引。这是 B+ 树索引的程序不足的地方。未来需要不断完善，能对所有类型的属性建立索引。