《数据库系统概论》实验报告

题目：实验九 查询优化	姓名	Vivian	日期	2006-1-1

实验内容及完成情况（写明你的优化方案）：

一．实验运行环境：

CPU：P3　800

内存：512MB

硬盘：30G

操作系统：WINDOWS 2000 ADVANCED SERVER

数据库系统：KingbaseES V4.1

二．设计数据库及其数据状况：

1．为本实验建立一个新的数据库，其中包括 Student、Course、SC 表和 STU、COU、S_C 表，它们的结构与《概论》书中的“学生课程数据库”类似。

2．本实验中，表 Student 共有 30 条记录，表 Course 共有 20 条记录，表 SC 共有 100 条记录；表 STU 共有 10000 条记录，表 COU 共有 100 条记录，表 S_C 共有 1000000 条记录。其中，Student、Course、SC 表已在自动建立的“学生课程数据库”中；STU、COU、S_C 表中的数据可以通过执行存储过程 INSERT_STU、INSERT_COU、INSERT_S_C，在建立的库中导入数据。

3．单表查询实验中，我们设计的数据情况如下：表 Student 中>20 岁的学生记录为 0 条，占总元组数的 0%；表 STU 中>20 岁的学生记录为 150 条,占总元组数的 1.5%。

（一） **单表查询**

【例 1】查询 Student 表中 20 岁以上学生的信息（表中元组数少，查询结果元组数所占比例小）

[例 1-1]直接查询

```
SELECT * FROM Student WHERE sage>20;
```

查看预查询计划：EXPLAIN SELECT * FROM STUDENT WHERE sage>20;

执行语句并查看查询计划：EXPLAIN ANALYZE SELECT * FROM STUDENT WHERE sage>20;

查询计划：

```
QUERY PLAN
------------------------------------------------------------------------------
Seq Scan on STUDENT    (cost=0.00..1.38 rows=1 width=51)   /*表示系统的预查询计划*/
        (actual time=0.000..0.000 rows=0 loops=1)          /*表示系统实际执行的查询计划*/
Filter: (SAGE > 20)                                        /*Student 表的过滤条件 sage>20*/
Total runtime: 0.000 ms                                    /*表示系统的实际执行时间*/
```

[例 1-2]建立索引后再查询

```
CREATE INDEX stuage ON Student(sage);
SELECT * FROM Student WHERE sage>20;
```

查询计划：

```
QUERY PLAN
```

Seq Scan on STUDENT (cost=0.00..1.38 rows=1 width=51) (actual time=0.000..0.000 rows=0 loops=1)

Filter: (SAGE > 20)

Total runtime: 0.000 ms

【例 2】查询 Student 表中 20 岁以下学生的信息（表元组数少，查询结果元组数所占比例大）

[例 2-1]直接查询

SELECT * FROM Student WHERE sage<20;

查询计划：

QUERY PLAN

Seq Scan on STUDENT (cost=0.00..1.38 rows=23 width=51) (actual time=0.000..0.000 rows=22 loops=1)

Filter: (SAGE < 20)

Total runtime: 0.000 ms

[例 2-2]建索引后再查询

CREATE INDEX stuage ON Student(sage);

SELECT * FROM Student WHERE sage<20;

查询计划：

QUERY PLAN

Seq Scan on STUDENT (cost=0.00..1.38 rows=23 width=51) (actual time=0.000..0.000 rows=22 loops=1)

Filter: (SAGE < 20)

Total runtime: 0.000 ms

【例 3】查询 STU 表中 20 岁以上学生的信息（表元组数多，查询结果元组数所占比例小）

[例 3-1]直接查询。

SELECT * FROM STU WHERE age>20;

查询计划：

QUERY PLAN

Seq Scan on STU (cost=0.00..356.16 rows=23 width=48) (actual time=20.000..30.000 rows=150 loops=1)

Filter: (AGE > 20)

Total runtime: 30.000 ms

[例 3-2]建立索引后再查询。

CREATE INDEX stuage ON STU(age);

SELECT * FROM STU WHERE age>20;

查询计划：

QUERY PLAN

Index Scan using AGE on STU (cost=0.00..196.01 rows=57 width=48) (actual time=0.000..0.000 rows=150 loops=1) /*利用索引扫描表 STU*/

Index Cond: (AGE > 20) /*索引条件：AGE>20*/

Total runtime: 0.000 ms

[例 3-3]建立索引后增加限定条件再查询

```
SELECT * FROM STU WHERE age > 20 AND sex = 1;
```

查询计划：

```
QUERY PLAN
------------------------------------------------------------------------------
Index Scan using AGE on STU  (cost=0.00..196.15 rows=29 width=48) (actual time=0.000..0.000 rows=75 loops=1)
  Index Cond: (AGE > 20)
  Filter: (SEX = 1)
Total runtime: 0.000 ms
```

【例 4】查询 STU 表中 20 岁以下学生的信息（表元组数多，查询结果元组数所占比例大）

[例 4-1]直接查询。

```
SELECT * FROM STU WHERE age<20;
```

```
QUERY PLAN
------------------------------------------------------------------------------
Seq Scan on STU  (cost=0.00..356.16 rows=2949 width=48) (actual time=0.000..20.000 rows=7388 loops=1)
  Filter: (AGE < 20)
Total runtime: 60.000 ms
```

[例 4-2]建立索引后再查询。

```
CREATE INDEX stuage ON STU(age);
SELECT * FROM STU WHERE age<20;
```

查询计划：

```
QUERY PLAN
------------------------------------------------------------------------------
Seq Scan on STU  (cost=0.00..432.00 rows=7497 width=48) (actual time=10.000..21.000 rows=7388 loops=1)
  Filter: (AGE < 20)
Total runtime: 61.000 ms
```

注释：

1.查询计划中的第一个括号内表示的是系统的预查询计划。其中，cost 等号后的两个数字，分别代表“预计的启动时间”和“预计的总时间”（以磁盘页面存取为单位计量）；rows 代表“预计输出的行数”；width 代表“预计的行的平均宽度（以字节计算）”。

2.查询计划中的第二个括号内表示的是系统实际执行时的计划。其中，actual time 等号后的数字表示“在每个计划节点内部花掉的总时间”（以 ms 为单位）；rows 代表实际返回的结果行数；loops 代表进行查询循环的次数。（这是因为我们在显示查询计划是用的是 explain analyze，analyze 导致查询语句实际执行，而不仅仅是一个预期计划）

3.Total runtime 包括优化器启动和关闭的时间，以及花在处理结果行上的时间。对于 SELECT 查询，总运行时间通常只是比从顶层计划节点汇报出来的总时间略微大些。

4.在建立表之后，执行查询计划的步骤是：

- 执行命令：analyze <表名>。这实际上是系统对数据库的数据进行新的统计。这样做的原因是为了让查询优化器在优化查询的时候做出合理的判断，防止因为统计信息

的陈旧造成系统选择错误的查询计划。

- 选择查看预查询计划还是查看实际执行的查询计划。

  查看预查询计划的语句是： EXPLAIN <SQL 语句>

  查看实际查询计划的语句是：EXPLAIN ANALYZE <SQL 语句>

5.因为系统对每一个查询语句都会进行一次优化，每次运行时系统环境和数据也会变化，因此每次得到的查询计划数据也可能有所变化。

总结分析（有索引时不同条件下的查询执行计划）：

表的元组数量	元组数多	元组数少
查询结果的比例大	顺序扫描全表（有无索引的实际执行时间几乎相等）	顺序扫描全表（有无索引的实际执行时间几乎相等）
查询结果的比例小	利用索引扫描提取（使用索引比顺序扫描表的查询时间大大减少了）	顺序扫描全表（有无索引的实际执行时间几乎相等）

1.我们可以看到，以上四个查询中【例 1】【例 2】【例 4】有索引和没有索引的查询计划都是相同的，即都是顺序扫描全表；只有【例 3】在建立了索引后使用了索引进行扫描。可见，在建立索引后系统并不一定对所有的查询都使用索引。

2.【例 3】在建立索引后使用索引的原因分析：在[例 3-2]中，我们把 WHERE 条件变得足够有选择性，即表中总的元组数非常大而结果数所占比例非常小（5%-10%）。系统优化器经过代价估算后得到利用索引将比全表顺序扫描快的结果。因为有索引，这个计划将只需要访问 150 条记录，所以选择使用索引。

3. [例 3-2]和[例 3-3]的开销比较分析：在[例 3-3]中，我们增加了查询的限定条件，可以看出，增加的条件减少了预计的结果行数（从 57 减少为 29），但是却没有减少开销（预计总时间分别为 196.01ms 和 196.15ms），因为查询时仍然需要访问相同的行。性别上不同值的个数只有男、女两个，不合适建立索引。sex = 1 只能做为对索引中检索出的元组的过滤器，实际上开销还略微增加了一些以反映这个检查。

（二）多表查询

【例 5】嵌套查询（表元组数少）

[例 5-1]查询选修了 2 号课程的学生姓名。

```
SELECT Sname
FROM Student
WHERE Sno IN
      (SELECT Sno
        FROM SC
        WHERE Cno= '2');
```

查询计划：

```
QUERY PLAN
------------------------------------------------------------------------------
Hash IN Join  (cost=22.51..23.97 rows=1 width=21) (actual time=0.000..0.000 rows=16 loops=1)
/*对 Student 表和 Hash 表进行连接*/
  Hash Cond: ("outer".SNO = "inner".SNO)
  →Seq Scan on STUDENT  (cost=0.00..1.30 rows=30 width=32) (actual time=0.000..0.000 rows=30
  loops=1)
```

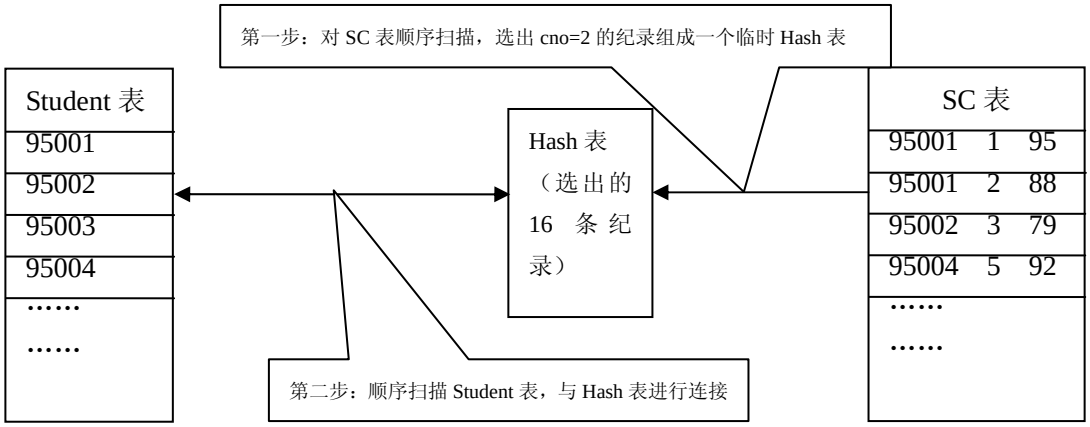
→Hash (cost=22.50..22.50 rows=5 width=18) (actual time=0.000..0.000 rows=0 loops=1)
/*对 SC 表的扫描结果进行哈希*/

→Seq Scan on SC (cost=0.00..22.50 rows=5 width=18) (actual time=0.000..0.000 rows=16 loops=1)

Filter: (CNO = '2'::BPCHAR)

Total runtime: 0.000 ms

分析：这是一个不相关子查询，内层查询的查询条件不依赖于外表的值，系统经过优化分析后会产生与连接查询相同的处理顺序。（如下图）



首先对内层表 SC 根据查询条件进行顺序扫描，得到 16 条记录，把这些记录放在一个在内存的散列表里，然后对表 Student 做一次顺序扫描，对每一条 Student 记录检测上面的散列表，即 Hash IN Join，寻找可能匹配的行。

其中，读取 SC 和建立散列表是散列连接的全部启动开销，因为我们在开始读取 Student 之前不可能获得任何输出行。这个连接的总的预计时间同样还包括检测散列表 30 次的 CPU 时间（Student 共 30 条记录）。但是检测 30 次不代表着需要对 22.50 乘 30，散列表在这里只需要设置一次。

另外，需要注意的是，Hash 中实际执行的 rows 值为 0，这是因为将记录存放在散列表的过程只是一个存储再分布的过程，而没有对元组进行选择、投影等操作，因此系统认为得出来的元组数为 0。而实际上，并不是扫描出来的元组数消失了，元组还是没有变化的。

问题：在 Student 表和 SC 表中，都根据主码建立了主码索引，为什么在扫描时不使用索引进行扫描？

这一结果产生的原因可能与前面单表查询中结果分析一样，是因为 Student 表和 SC 表的元组数都非常小，使用索引进行查询比全表扫描的代价更大，所以使用的是顺序扫描。这一猜想我们在接下来的部分，对大数据量的表进行相同的试验后再进行验证和分析。

[例 5-2]查询没有选修 1 号课程的学生姓名。

```
SELECT Sname
FROM Student
WHERE NOT EXISTS
        (SELECT *
         FROM SC
         WHERE Sno = Student.Sno AND Cno = '1');
```

查询计划：

```
QUERY PLAN
------------------------------------------------------------------------------
```

Seq Scan on STUDENT (cost=0.00..751.30 rows=15 width=21) (actual time=0.000..0.000 rows=19 loops=1)

Filter: (NOT (subplan))

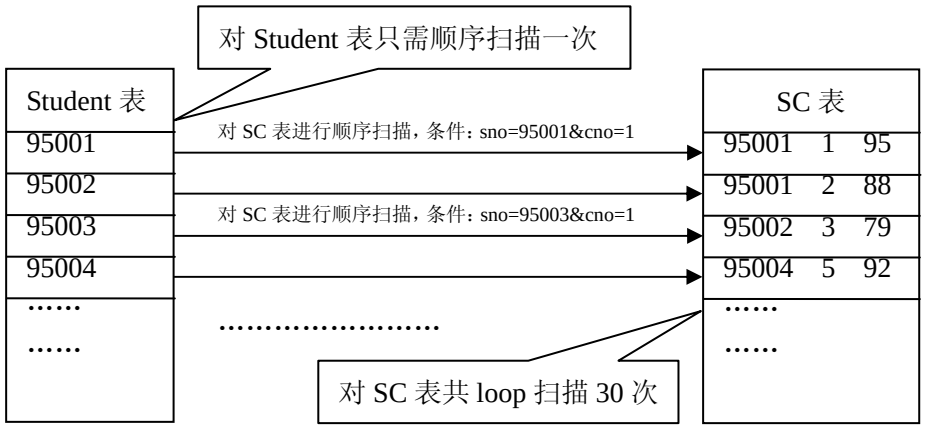
SubPlan /*子查询计划*/

→Seq Scan on SC (cost=0.00..25.00 rows=1 width=34) (actual time=0.000..0.000 rows=0 loops=30)

Filter: ((SNO = $0) AND (CNO = '1'::BPCHAR))

Total runtime: 0.000 ms

分析：这个查询是一个相关子查询，内层查询的查询条件依赖于外表的值（Student.Sno），因此系统处理这一查询的方式与不相关子查询就有所不同。执行顺序见下图：



首先是对外表 Student 顺序扫描，每次取一个记录，根据这个记录的 Sno 执行一次内层查询，如果 WHERE 子句返回值 false，则输出这个 Sno 对应的 Sname 值．如此反复，直到处理完 Student 表中所有的记录。因此可以看到，Seq Scan on SC 的实际执行 loops 数为 30，即为 Student 表的元组数；而 Seq Scan on STUDENT 的 cost 的值也大略等于 Seq Scan on SC 的 cost 乘 30。

【例 6】嵌套查询（表元组数多）

[例 6-1]查询选修了 2 号课程的学生姓名。

```
SELECT name
FROM STU
WHERE num IN
      (SELECT s_num
        FROM S_C
        WHERE c_num= ‘2’);
```

查询计划：

QUERY PLAN

Nested Loop (cost=22.51..24.54 rows=1 width=24) (actual time=2323.000..3064.000 rows=10000 loops=1)

→HashAggregate (cost=22.51..22.51 rows=1 width=4) (actual time=2323.000..2374.000 rows=10000 loops=1) /*对 S_C 表选择出来的结果做聚集*/

→Seq Scan on S_C (cost=0.00..22.50 rows=5 width=4) (actual time=0.000..2253.000 rows=10000 loops=1)

Filter: (C_NUM = 2)

→Index Scan using STU_NUM_50304_KEY on STU (cost=0.00..2.01 rows=1 width=28) (actual time=0.045..0.048 rows=1 loops=10000) /*利用主码索引对 STU 表进行扫描*/

Index Cond: (STU.NUM = "outer".S_NUM)

Total runtime: 3074.000 ms

分析：在 STU 表上使用了索引，因为 STU 表的元组数足够大，系统在分析后选择了代价较小的索引查找方法。

这个例子产生的查询计划是一个嵌套循环，并且是将选择完后的 S_C 表作为外表对 STU 表进行循环扫描。这样做的原因是对 S_C 进行过滤之后，产生的中间结果元组数不比 STU 表的元组数多很多，而 STU 表上有选择率较高的索引 STU_NUM_50304_KEY，在查询时可以利用索引 STU_NUM_50304_KEY 对元组快速定位抓取。对 S_C 表进行选择之后共产生 10000 条记录，因此需要对 STU 表进行 10000 次的循环索引查找。

[例 5-1]和[例 6-1] 是 2 个相同的查询语句，系统实际执行的查询计划是不同的。原因是，[例 5-1]中 Student 表和 SC 表都很小而[例 6-1]中操作的表都很大。说明 KingbaseES 的查询优化器是依据统计信息进行代价估算来选择高效的操作算法或存取路径的，从而产生了不同的查询计划。

[例 6-2-1]查询没有选修 1 号课程的学生姓名。

```
SELECT name
FROM STU
WHERE NOT EXISTS
        (SELECT *
         FROM S_C
         WHERE s_num = STU.num AND c_num = 1);
```

查询计划：

```
QUERY PLAN
------------------------------------------------------------------------------
Seq Scan on STU (cost=0.00..250407.00 rows=5000 width=24)(actual time=9695461.000..9695461.000 rows=0 loops=1)
  Filter: (NOT (subplan))
  SubPlan
    →Seq Scan on S_C  (cost=0.00..25.00 rows=1 width=12) (actual time=969.430..969.430 rows=1 loops=10000)
          Filter: ((S_NUM = $0) AND (C_NUM = 1))
Total runtime: 9695461.000 ms
```

分析：从这个实例可以看到，在没有任何索引，数据量非常大的表上进行相关子查询所耗费的时间非常长（约为 2.7 小时），代价将是非常大的。因为查询的过程是从外表取记录，对内表检测是否符合 EXIST 条件的重复过程，需要对内表进行多次循环扫描。

我们可以考虑通过建立索引、改写查询语句等方法对该查询进行一些优化。以下分别尝试了四种优化方式，可以和[例 6-2-1]进行查询计划和查询代价比较。

[例6-2-2]在S_C表的属性s_num上建立索引snumindex，然后进行查询。

```
CREATE INDEX snumindex ON S_C (s_num) ;
SELECT name
FROM STU
```

```
    WHERE NOT EXISTS
            (SELECT *
             FROM S_C
             WHERE s_num = STU.num AND c_num = 1);
```

查询计划：

```
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Seq Scan on STU  (cost=0.00..46598.39 rows=5044 width=24) (actual time=1823.000..1823.000 rows=0 loops=1)
  Filter: (NOT (subplan))
  SubPlan
    ->  Index Scan using SNUMINDEX on S_C  (cost=0.00..4.60 rows=1 width=12) (actual time=0.177..0.177 rows=1 loops=10000)
          Index Cond: (S_NUM = $0)
          Filter: (C_NUM = 1)
Total runtime: 1823.000 ms
```

分析：在属性 s_num 上建立索引之后，系统在对 S_C 表进行扫描时就不再是顺序扫描了，而是对每一条外表 STU 记录的 num 值利用索引 snumindex 定位，然后再根据过滤条件 c_num=1 查找是否符合 EXIST 条件。可以看到，在[例 6-2-1]中，每对 S_C 表扫描一次所花费的时间为 969.430ms，而使用了索引之后，定位扫描的时间仅为 0.177ms，大大减少了查询时间。

[例 6-2-3]如果我们改变表 S_C 的定义，在创建时将 s_num 和 c_num 定义为主码。

```
    CREATE TABLE S_C(
    s_num    INT,
    c_num    INT,
    grade    INT,
    PRIMARY KEY (s_num,c_num));
    SELECT name
    FROM STU
    WHERE NOT EXISTS
            (SELECT *
             FROM S_C
             WHERE s_num = STU.num AND c_num = 1);
```

查询计划为：

```
QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on STU  (cost=0.00..53325.91 rows=5044 width=24) (actual time=300.000..300.000 rows=0 loops=1)
  Filter: (NOT (subplan))
  SubPlan
    ->  Index Scan using S_C_1041627_PKEY on S_C  (cost=0.00..5.27 rows=1 width=12) (actual time=0.016..0.016 rows=1 loops=10000)
          Index Cond: ((S_NUM = $0) AND (C_NUM = 1))
Total runtime: 300.000 ms
```

分析：由于在定义表时将 s_num 和 c_num 定义为了主码，因此系统将在 s_num 和 c_num 上建立一个主码索引 S_C_1041627_PKEY。和[例 6-2-3]相比，在查询时，在对外表记录的 num 值利用索引定位之后，c_num＝1 不是当作一个过滤条件对所有与外表 num 值相等的记录进行过滤，而能够直接通过 c_num 上的索引快速定位，判断是否存在着 c_num＝1 的记录。将对 S_C 表每次扫描的时间降低到 0.016ms，查询时间进一步减少，得到更优的查询计划。

由此，我们可以看出，正确合适的使用索引是提高查询效率的一个有效手段，能够使查询时间得到数量级上的改进。

[例 6-2-4]利用临时表将嵌套查询改写成连接查询。

```
SELECT name
FROM (
    SELECT DISTINCT s_num
    FROM S_C
    WHERE c_num = 1)
        AS TEMP RIGHT JOIN STU ON TEMP.s_num = STU.num
WHERE TEMP.s_num IS NULL;
```

查询计划：

```
QUERY PLAN
---------------
Merge Left Join  (cost=22.60..254.72 rows=10088 width=24) (actual time=2353.000..2353.000 rows=0 loops=1)                    /*左连接，连接条件为STU.num = S_C.s_num*/
  Merge Cond: ("outer".NUM = "inner".S_NUM)
  Filter: ("inner".S_NUM IS NULL)
  ->  Index Scan using STU_NUM_17388_KEY on STU  (cost=0.00..206.88 rows=10088 width=28) (actual time=0.000..80.000 rows=10000 loops=1)
  ->  Sort  (cost=22.60..22.61 rows=1 width=4) (actual time=2193.000..2203.000 rows=10000 loops=1)
        Sort Key: "TEMP".S_NUM
        ->  Subquery Scan "TEMP"  (cost=22.56..22.59 rows=1 width=4) (actual time=2033.000..2143.000 rows=10000 loops=1)
              ->  Unique  (cost=22.56..22.58 rows=1 width=4) (actual time=2033.000..2093.000 rows=10000 loops=1)          /*因为s_num要求唯一，因此需要进行唯一性检查*/
                    ->  Sort  (cost=22.56..22.57 rows=5 width=4) (actual time=2033.000..2063.000 rows=10000 loops=1)      /*对s_num进行排序以便下一步排除重复值*/
                          Sort Key: S_NUM
                          ->  Seq Scan on S_C  (cost=0.00..22.50 rows=5 width=4) (actual time=0.000..1983.000 rows=10000 loops=1)
                                Filter: (C_NUM = 1)
Total runtime: 2363.000 ms
```

分析：在本实例中，我们将原有的嵌套查询改写成为一个连接查询，将 S_C 表中所有选修了 1 号课程的学生学号形成一个临时 TEMP，与 STU 表进行外连接，如果存在着无法满足连接条件的元组，即连接后 TEMP.s_num 为 NULL 值的元组，则是没有选修 1 号课程的学生记录。虽然改写后的查询计划层次更多了，但查询时间大大减少，达到了优化的目的。

另外，需要注意的一点是，我们在些查询语句时，是将 TEMP 表与 STU 表进行右外连接，

而系统在经过代价分析之后，选择将 STU 表作为外表，TEMP 表作为内表，因此将语句中的右外连接改写成为实际执行的左外连接。

[例 6-2-5]利用集合查询和连接查询的方式改写原有查询语句。

```
SELECT name
FROM STU
EXCEPT
SELECT DISTINCT name
FROM STU LEFT JOIN S_C ON (STU.num = S_C.s_num)
WHERE c_num = 1;
```

查询计划：

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
SetOp Except  (cost=22368.58..22466.97 rows=1968 width=24) (actual time=3405.000..3405.000 rows=0 loops=1)                    /*集合差操作，由两个结果集合相减得到最后结果*/
  ->  Sort  (cost=22368.58..22417.78 rows=19678 width=24) (actual time=3295.000..3345.000 rows=20000 loops=1)
        Sort Key: NAME
        ->  Append  (cost=0.00..20965.12 rows=19678 width=24) (actual time=0.000..2815.000 rows=20000 loops=1)
              ->  Subquery Scan "*SELECT* 1"  (cost=0.00..305.76 rows=10088 width=24) (actual time=0.000..91.000 rows=10000 loops=1)
                    ->  Seq Scan on STU  (cost=0.00..204.88 rows=10088 width=24) (actual time=0.000..40.000 rows=10000 loops=1)
              ->  Subquery Scan "*SELECT* 2"  (cost=20515.51..20659.36 rows=9590 width=24) (actual time=2513.000..2684.000 rows=10000 loops=1)
                    ->  Unique  (cost=20515.51..20563.46 rows=9590 width=24) (actual time=2513.000..2623.000 rows=10000 loops=1)
                          ->  Sort  (cost=20515.51..20539.48 rows=9590 width=24) (actual time=2513.000..2563.000 rows=10000 loops=1)
                                Sort Key: STU.NAME
                                ->  Merge Join  (cost=19505.37..19881.26 rows=9590 width=24) (actual time=2053.000..2263.000 rows=10000 loops=1)
                                      Merge Cond: ("outer".NUM = "inner".S_NUM)
                                      ->  Index Scan using STU_NUM_17388_KEY on STU  (cost=0.00..206.88 rows=10088 width=28) (actual time=0.000..120.000 rows=10000 loops=1)
                                      ->  Sort  (cost=19505.37..19529.35 rows=9590 width=4) (actual time=2053.000..2053.000 rows=10000 loops=1)
                                            Sort Key: S_C.S_NUM
                                            ->  Seq Scan on S_C  (cost=0.00..18871.13 rows=9590 width=4) (actual time=0.000..1993.000 rows=10000 loops=1)
                                                  Filter: (C_NUM = 1)
Total runtime: 3435.000 ms
```

分析：在本实例中，我们利用了集合查询中的EXCEPT差操作，用全部学生记录减去选

修了1号课程的学生记录即得到我们要查询的结果，没有选修1号课程的学生记录。因此，整个查询计划包含两个子查询计划，第一个子查询计划是顺序扫描STU表选择属性列name；第二个子查询计划是一个左外连接，查询选修了1号课程的name唯一值，查询过程与[例6-2-4]相似。最后根据两个子查询的结果，进行集合操作，得到最终查询结果。可以看到，查询效率也得到了很大的提高。

从[例6-2-2]至[例6-2-5]可以看出，一个效率低下的查询计划，可以通过建立索引、改写语句等多种方式对它进行优化，优化后的查询结果将得到很大的改善（从2.7小时降低到数秒甚至300ms）。因此，如何选择最优的查询计划要求我们进行更多的分析和实践才能有所体会。