数据库期末实验报告

17307130328 刘妍

Part I

1.实验简介

1.1 实验目的：

PostgreSQL 上的 Similarity Join 实现

1.2Postgresql 介绍

一款基于 POSTGRES 的关系对象数据库管理系统；支持 SQL 标准的大部分，并提供了许多现代化的特色，比如复杂查询、外键、触发器等；用户可以自主拓展功能，比如添加数据类型、函数、运算符等

1.3similarity join

即相似性查询。在通过诸如电话、地址等没有固定格式的属性，来连接两个表时， 绝对的相等" = "是不能满足连接需求的，就需要通过计算相似性来进行限制。计算相似性有以下两种方法：

- ☞ Levenshtein Distance：最小编辑距离；
- ☞ Jaccard Index：基于 bigram 为单元计算的 Jaccard 系数。

2.实验准备

configure 配置：设置 PostgreSQL 的安装目录为主目录下的 pgsql

```
./configure --enable-depend --enable-cassert --enable-debug CFLAGS="-O0"
--prefix=$HOME/pgsql
```

✧ postgresql 编译和安装

```
make&&make install
```

✧ 在磁盘上初始化一个数据库存储区域，即数据库集群

```
$HOME/pgsql/bin/initdb -D $HOME/pgsql/data --locale=C
```

✧ 启动数据库服务器

```
$HOME/pgsql/bin/pg_ctl -D $HOME/pgsql/data -l logfile start
```

✧ 创建 similarity 数据库并导入数据

```
$HOME/pgsql/bin/psql -p 5432 postgres -c 'CREATE DATABASE similarity;'
$HOME/pgsql/bin/psql -p 5432 -d similarity -f
/home/postgres/similarity_data.sql
```

✧ 进入 PostgreSQL

```
$HOME/pgsql/bin/psql similarity
```

✧ 开启计时

```
\timing
```

3.函数实现和说明

3.1 Levenstein_distance

3.1.1 实现原理：动态规划

可以证明，对于任意一种从 A 串经过删、改、添三种操作更改为 B 串的操作序列，它们之间的先后顺序没有影响。因此我们可以假设，从 A 串到 B 串的任意一种操作序列都是从左向右的。这样的话要想将 A[0-i]改变为 B[0-j]，最后一步有四种情况：

- 替换：A[0-i-1]已转换为 B[0-j-1]，但 A[i]!=B[j]，操作数+1
- 增加： A[0-i]已转换为 B[0-j-1]，需要增加 B[j] ，操作数+1
- 删除： A[0-i-1]已转换为 B[0-j]，需要删除 A[i] ，操作数+1
- 无操作：A[0-i-1]已转换为 B[0-j-1]，且 A[i]==B[j] ，操作数不变

最终操作数取上述最小值。时间复杂度 O（mn）。

3.1.2 具体步骤：

✧ 类型转换：

我在源码中找到了如下宏定义和函数定义：

```
#define TextDatumGetCString(d) text_to_cstring((text *) DatumGetPointer(d))
```

可以看到 TextDatumGetCString(d)能够将 text*类型转为 char*类型

✧ 定义一个二维数组 d，d[i][j]代表将 A[0-i]转变为 B[0-j]所需最少操作数

✧ 双层循环：

i=0: d[i][j]=j

j=0: d[i][j]=i

i≠0 且 j≠0：先判断当前索引指向字符是否相等，若相等则令临时变量 k=0 代表不需要操作，否则为 1 代表替换；取当前编辑距离 d[i][j]为 d[i - 1][j] + 1（删除），d[i][j - 1] + 1（增添）,d[i - 1][j - 1] + k（替换或无操作）中的最小值

3.1.3 代码

```
char* s1=TextDatumGetCString(str_01);
char* s2=TextDatumGetCString(txt_02);  //将Datum转为char*
int l1 = strlen(s1), l2 = strlen(s2);
int d[101][101];
for (int i = 0; i < l1+ 1; i++)
{
    for (int j = 0; j < l2 + 1; j++)
    {
        if (!i)
            d[i][j] = j;
        else if (!j)
            d[i][j] = i;
        else
        {
            //判断最后的字符是否相等
            int k = ((s1[i-1] == s2[j-1]) ? 0 : 1);
            d[i][j] = min(min(d[i - 1][j] + 1, d[i][j - 1] + 1), d[i - 1][j - 1] + k); //取最小值
        }
    }
}
result=d[l1][l2];
PG_RETURN_INT32(result);//将int转为Datum
```

3.1.4 时间上优化

在每次调用 levenshtein_distance 时都要开辟一个 d[101][2],于是在前面加上 static 的标识；这样在频繁调用该函数的时候时会保留这块空间，理论上能节省重复申请、释放空间的时间

3.1.5 空间上的优化

动态规划中每次需要用到的数据实际上只有前一列，所以可以将二维数组浓缩为一维数组 d，具体如下： 设置一个 pre 用来保存 d[i-1][j-1]；d[i-1][j-1]就是在上一次循环中 d[i]变化前的值；d[i-1][j]就是 d[i-1]；d[i][j-1]就是 d[i]。空间复杂度由 O(mn)->O(m)。代码如下：

```
for (int i = 0; i < l1+1; i++)
    d[i] = i;//预处理，与二维数组中预处理作用相同
for (int j = 1; j < l2+1; j++) {
    int pre = d[0];//保存d[i-1][j-1]
    d[0] = j; //预处理
    for (int i = 1; i < l1+1; i++) {
        int temp = d[i];//记住当前d[i]，因为这就是下一次循环中的d[i-1][j-1]
        if (s1[i - 1] == s2[j - 1])
            d[i] = pre;
        else d[i] = min(pre + 1, min(d[i] + 1, d[i - 1] + 1));
        pre = temp;
    }
}
```

3.1.6 三种实现时间比较：

```
similarity=# select count(*) from restaurantphone rp, addressphone ap where leve
nshtein_distance(rp.phone, ap.phone) < 4;
 count
-------
  3252
(1 row)

Time: 12026.754 ms
similarity=# select count(*) from restaurantphone rp, addressphone ap where leve
nshtein_distance2(rp.phone, ap.phone) < 4;
 count
-------
  3252
(1 row)

Time: 12339.121 ms
similarity=# select count(*) from restaurantphone rp, addressphone ap where leve
nshtein_distance3(rp.phone, ap.phone) < 4;
 count
-------
  3252
(1 row)

Time: 8293.420 ms
```

(图 1)

```
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where
 levenshtein_distance(ra.name, rp.name) < 3;
 count
-------
  2112
(1 row)

Time: 21816.618 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where
 levenshtein_distance2(ra.name, rp.name) < 3;
 count
-------
  2112
(1 row)

Time: 24010.295 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where
 levenshtein_distance3(ra.name, rp.name) < 3;
 count
-------
  2112
(1 row)

Time: 17563.077 ms
```

(图 2)

可以看到方案三相对于没有任何优化的方案一会快 4s，而方案二会比方案一慢。但理论上方案二是时间上的优化，方案三是在时间优化基础上对空间的优化。我的解释是：上述优化实际上对时间复杂度并没什么作用，时间上的差异属于正常随机误差。

3.2 jaccard_index

3.2.1 实现方法：

- 方案一：遍历两个字符串，分别进行分割和去重并保存在两个二维数组中，两层 for 循环获得相交集个数 inter，两个二维数组元素之和 - 相同元素得到并集个数 union，返回 inter/union，时间复杂复杂度为 O（n2+m2+m*n）。
- 方案二：利用字符串的 hash 函数：

字符串函数	冲突数
BKDRHash	0
SDBMHash	2
RSHash	2
APHash	0
ELFHash	1515
JSHash	779
DEKHash	863
FNVHash	2
DJBHash	832

参考网络上对各种字符串哈希函数的冲突率测评（如下图）：
我采用了冲突数最少的 BKDRHash 函数（实际上在性能测评中也是最好的）作为哈希函数，将分割后的两个字符的 ASCII 码值映射到一个数组上。首先创建一个一维标记数组 flag，对 s1 每两个字符获取哈希值 temp，并判断 flag[temp]，若是 0，则置 1 并 union++，若是 1 则跳过；接着对 s2 进行类似处理，但是当 flag[temp]=1 时，inter++。时间复杂度为 O（m+n），但相对方案一因为 flag 的存在空间复杂度也增大。

具体代码：
改造后的 BKDRHash 函数：

```
unsigned int _BKDRHash(char *str)
{
    unsigned int seed = 131;
    //因为每个元素只有两个字符，所以舍弃了BKDRHash原本的循环，直接得出hash值
    unsigned int hash = str[0] * seed + str[1];
    return hash;
}
```

处理 s1:

```
int temp = _BKDRHash(s);
if (flag[temp] == '\0')//未出现过
{
    _union++;//并集+1
    flag[temp] = 'y';//标记为出现过
}
```

处理 s2:

```
if (flag[temp] == '\0')
{
    _union++;
    flag[temp] = 'n';
}
else if (flag[temp] == 'y')//遇到被标记为s1的元素
{
    inter++;
    flag[temp] = 'n';//防止因为s2中重复的元素使得inter偏大
}
```

- 方案三：保证不会发生冲突的 hash 函数

  将两个字符的 ASCII 码值进行移位拼接，能够保证该映射为双射：

```
int temp = (toupper(s[0])<<7)|toupper(s[1])
```

  其余部分同方案二

  三种方案时间比较：

```
similarity=# select count(*) from restaurantphone rp, addressphone ap where jacc
ard_index(rp.phone, ap.phone) > .6;
 count
-------
  1653
(1 row)

Time: 18908.505 ms
similarity=# select count(*) from restaurantphone rp, addressphone ap where jacc
ard_index2(rp.phone, ap.phone) > .6;
 count
-------
  1653
(1 row)

Time: 8685.060 ms
similarity=# select count(*) from restaurantphone rp, addressphone ap where jacc
ard_index3(rp.phone, ap.phone) > .6;
 count
-------
  1653
(1 row)

Time: 8428.487 ms
```

(图 1)

```
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where
 jaccard_index(ra.name, rp.name) > .65;
 count
-------
  2398
(1 row)

Time: 40373.915 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where
 jaccard_index2(ra.name, rp.name) > .65;
 count
-------
  2398
(1 row)

Time: 10876.058 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where
 jaccard_index3(ra.name, rp.name) > .65;
 count
-------
  2398
(1 row)

Time: 10569.984 ms
```

(图 2)

可以看到方案二、三相对没有任何优化的方案一时间会大幅缩短，且方案一时间越长，优化效果越明显。

4、PostgreSQL 内部机制理解

4.1 大体步骤：

- ✧ 接收 SQL 语句：后端的 main 函数是 PostgresMain (tcop/postgres.c)，接收前端发送过来的查询。
- ✧ 构文解析：接收的 SQL 查询是单纯的文字，要转换成比较容易处理的内部形式构文树 parser tree,这个处理称为构文解析，发生在'parser'模块。这个阶段只能够使用文字字面上得来的信息，所以只要没语法错误之类的错误，即使是 select 不存在的表也不会报错。构文处理的入口在 raw_parser (parser/parser.c)。
- ✧ 分析处理：构文树解析完后，会转换为查询树(Query tree)。这个时候会访问数据库，检查表是否存在，如果存在则把表名转换为 OID。进行分析处理的模块是'analyzer'。分析处理的模块的入口在 parse_analyze (parser/analyze.c)
- ✧ 重写：PostgreSQL 会通过查询语句的重写实现视图(view)和规则(rule),这个处理的模块是'rewrite'，重写的入口在 QueryRewrite (rewrite/rewriteHandler.c)。
- ✧ 查询优化：通过解析查询树，可以实际生成计划树。该处理称为'执行计划处理'，关键是要生成估计能在最短的时间内完成的计划树(plan tree)。这个步骤称为'查询优化', 完成这个处理的模块称为'optimizer'。执行计划处理的入口在 standard_planner (optimizer/plan/planner.c)。
- ✧ 执行：运行执行计划树里面的步骤。完成这个处理的模块称为 'Executor', 执行器的入口为 ExecutorRun (executor/execMain.c)。下图为调用 executor 的伪码：

```
处理查询得到 Query Plan 和初始化必要的数据结构
调用 Executor 的 ExecStart 初始化
调用 Executor 的 ExecRun 执行 Query Plan
调用 Executor 的 ExecEnd 清理
```

- ✧ 执行结果返回给前端。返回到步骤一重复执行。

4.2 具体函数调用

- ✧ PostgreMain 接收到查询语句，调用 exec_simple_query 函数；
- ✧ exec_simple_query ->pg_parse_query ->pg_parse_query ->raw_parser 函数进行词法和语法分析，产生构文树（parser tree）
- ✧ exec_simple_query->pg_analyze_and_rewrire->parse_analyze 进行语义分析生成 Query 结构体（query tree），再将该结构体传递给 pg_rewrite_query 进行查询重写；

- ✧ 查询重写完成以后，进入 pg_plan_queries 函数进行查询规划。pg_plan_queries->pg_plan_query->planner-> standard_planner ->subquery_planner 和 set_plan_references 完成计划树的生成、优化；
- ✧ 在计划树生成之后，调用 ExecStart 函数初始化查询计划树-> ExecRun ->standard_ExecRun 函执行

5.实验反思和感想

1、一开始修改了源码之后，我并未重新安装 PostgreSQL，是因为我在按照手册做的时候没有理解这些步骤的意义。配置(configure)、编译(make)、安装(make install)是源码的安装一般步骤。

2、在添加了优化后的几个版本函数之后，忘记在 “src/include/catalog/pg_proc.h” 中添加 “DATA(insert OID = 4375……”语句，导致调用出错；而添加了相应语句后却因为没有正确修改（修改了前一部分的函数名却未修改后一部分）而导致不同的函数名调用的都是同一个函数，根本原因是因为没有理解插入这一语句的含义。在拜读别人的实验报告后，得知这是注册函数，括号内的数字包括参数个数，返回参数的类型以及两个参数的类型。25 表示字符串、700 表示单精度浮点数、23 表示 32 位整数、16 表示布尔类型。

3、在重新安装 PostgreSQL 时，在创建 similarity 数据库时出现如下错误

```
zcr@zcr-XPS-13-9360:~/postgresql$ $HOME/pgsql/bin/psql -p 5432 postgres -c 'CREA
TE DATABASE similarity'
psql: FATAL:  could not open relation mapping file "global/pg_filenode.map": No
such file or directory
```

在网络上搜索得到如下答案：

> Stop the database process, remove all remaining files. If it is a logocal backup (`pg_dump`), run `initdb` to create a new PostgreSQL cluster and restore. If it is a physical backup, restore it and recover to a point before the problem happened. – Laurenz Albe Nov 9 '17 at 8:35

大致是我没有正确关闭上一个进程，于是我查看当前含 postgres 的进程：

```
zcr@zcr-XPS-13-9360:~/postgresql$ ps aux | grep postgres
zcr       16888  0.0  0.0  21536  1148 pts/0    S+   11:23   0:00 grep --color=au
to postgres
zcr       20467  0.0  0.1  70256 11572 pts/0    S    10:26   0:00 /home/zcr/pgsql
/bin/postgres -D /home/zcr/pgsql/data
```

看到进程 20467 我想起手册上说要关闭 PostgreSQL 服务器，而我只是 ctrl+D 退出了 psql，于是把 pid 为 20467 的进程杀死，重新开启服务器。在这之后我重新安装之前都会输入“$HOME/pgsql/bin/pg_ctl -D $HOME/pgsql/data stop”退出服务器，就不会出现类似错误，证明确实是由此引起。

4、最初 jaccard 的方案二、三虽然时间快很多，但答案也大很多，发现算法存在漏洞：在对第二个字符串进行处理时，没有考虑到第二个字符串中重复出现的也会被计入相同数目中，于是将对 s2 的处理修改为只有当遇到被标记为 s1 的元素才令交集数++，并且要让当前元素哈

希值对应的 flag 值标记为 s2。

```
else if (flag[temp] == 'y')//遇到被标记为s1的元素
{
    inter++;
    flag[temp] = 'n';//防止因为s2中重复的元素使得inter偏大
}
```

5、在 make 时出现很多与我修改代码无关的 warning，为此困扰很久，实际上并不影响后续 make install

6、对于源码中后端部分，虽然其中很多细节我并不清楚，但是掌握了整体流程，理解了 PostgreSQL 操作数据库的内部机制，有助于数据库的理论学习，比如我现在知道学习的执行计划选择在实际操作中属于哪一个模块，又是如何和其他模块合作的。

7、 学习到有关的 Similarity Join 的各种实现方式；深入了解到各种 Levenshtein Distance、Jaccard Index 的计算算法，并学会对算法进行分析和优化。

参考：

- 《各种 Hash 函数冲突率分析》https://blog.csdn.net/qq7366020/article/details/8730425
- 《PostgreSQL 执行引擎简介》http://www.postgres.cn/news/viewone/1/203
- 《动态规划二维降一维》https://blog.csdn.net/gcola007/article/details/77413553
- 《could not open relation mapping file “global/pg_filenode.map”》https://stackoverflow.com/questions/47196126/could-not-open-relation-mapping-file-global-pg-filenode-map

Part II

1.实验简介

1.1 实验目的：

PostgreSQL 上的 Block Nested Loop Join 实现

1.2Block Nested Loop Join 原理

PostgreSQL 中的嵌套循环（Nested Loop Join）伪代码如下：

```
for (each tuple i in outer relation) {
  for (each tuple j in inner relation) {
    if (join_condition(tuple i, tuple j) is true)
      emit (tuple i, tuple j)
    else
      continue
  }
}
```

缓存块嵌套循环（Block Nested Loop Join）通过一次性缓存多条外表数据到 Join Buffer 里，然后拿 join buffer 里的数据批量与内层表的数据进行匹配，伪代码如下：

```
for (each block of tuples B in outer relation) {
  for (each tuple j in inner relation) {
    for (each tuple i in block B) {
      if (join_condition(tuple i, tuple j) is true)
        emit (tuple i, tuple j)
      else
        continue
    }
  }
}
```

较 Nested Loop Join 的改进就在于可以减少内表的扫描次数。

2.实验准备

✧ 在开启数据库服务器前修改 pgsql/data/postgresql.conf，使 LOG 输出到 pgsql/data/pg_log/中

```
vim $HOME/pgsql/data/postgresql.conf;
set logging_collector = on;set log_disconnections = on
```

✧ 在输入查询语句前关闭另外两种连接方式：Hash Join 和 Merge Join

```
set enable_hashjoin to off;set enable_mergejoin=off;
```

3.具体实现

3.1 思路：记外部关系为 R，内部关系为 S

- 开辟一个缓存块存储 R 的元祖 `static TupleTableSlot *block[size];`
- 一个最外层 for 循环 L1 控制 R 元祖读取；在 L1 内判断 full 即 block 是否已满：若未满则读取 R 的下一条存到 block 中，使 block 索引 pos++，再判断此时 block 是否已满，若是则令 full=1,pos=0；若 block 已满则进入第二层循环 L2

- L2 控制 S 元祖读取，先判断 pos 是否为 0，若是则代表一个缓存块已经遍历完，需要读取 S 的下一条元祖；进入第三层循环 L3（继续）遍历缓存块与当前 S 元祖进行匹配。
- 在 L3 内，若匹配成功，则返回匹配的元祖；否则继续匹配。当全部匹配失败时，令 pos=0 代表一个缓存块已经遍历完，并退出 L3
- 对于最后可能剩下的不足 blockSize 数目的 S 元祖，匹配方法类似，但当 S 遍历完了之后不需要再读 R 更新 block 而是返回 NULL 表示不再所有匹配结束不再调用该函数

3.2 细节

- 在每次一个缓存块满了之后通过 ExecReScan(innerPlan);控制 S 从头开始读取
- 因为是每匹配到一条就会返回，也就是在一次函数调用中不会得到所有的匹配元祖，那么在下一次进入该函数时应当保存上一次调用结束后的状态，本次实验中通过静态变量实现状态保存，如下图：

```
static int ini=0;
static int full=0;
static int pos=0;
static int over=0;
static TupleTableSlot *block[size];
```

ini 标记 block 是否已经初始化

full 标记 block 是否已满

pos 标记 block 中匹配到了哪一条

over 标记大部分匹配完成，只剩下不足 blockSize 数目的匹配

- block[i]的初始化是通过已有的 MakeTupleTableSlot()函数创建一个空的 TupleTableSlot*；通过已有的 ExecCopySlot(block[pos], outerTupleSlot)将 R 的元祖指针存到缓存块中，免去不必要的复制；在取出 block[i]赋给 outerTupleSlot 时可直接赋值，不必再调用上述 ExecCopySlot（）函数；在最后一次函数调用退出前通过 ExecDropSingleTupleTableSlot（）释放空间

4.实验结果：

blockSize=1

```
 count
-------
   451
(1 row)

Time: 24179.073 ms
```

blockSize=2

```
 count
-------
   451
(1 row)

Time: 21812.091 ms
```

blockSize=8

```
 count
-------
   451
(1 row)

Time: 16887.346 ms
```

blockSize=64

```
 count
-------
   451
(1 row)

Time: 16806.133 ms
```

blockSize=128

```
 count
-------
   451
(1 row)

Time: 17430.851 ms
```

blockSize=1024

```
 count
-------
   451
(1 row)

Time: 18209.118 ms
```

现象解释：

Block=1 即 nested loop，内层关系扫描时间成本为 O(R*S)。而 block nested loop 的循环 L3 循环次数即 block 被扫描次数为：⌈(S * C)/blockSize ⌉。次数随着 blockSize 的增大而减少, 直到一个 block 能够容纳所有的 R 元祖，再增大 blockSize，query 的速度就不会再变快了。

5.实验反思和感想

1．一开始修改了 blockSize，时间却没什么变化。询问助教，在连接语句前加上 explain analyze 发现 join 操作采取的是 hash join，于是关闭 hash join 和 merge join。关于 explain 的原理：查询计划的结构是一个计划结点的树。explain 给计划树中每个结点都输出一行，显示基本的结点类型和计划器为该计划结点的执行所做的开销估计。而通过使用 analyze 选项，显示真实的返回记录数和运行每个规划节点的时间。

2、一开始不知道日志输出到何处，搜索后得知在 pgsql/data/pg_log 目录下，但是并没有找到这个目录，只有 pg_xlog 和 pg_clog 目录，原来是因为 postgresql.conf 文件中并未将 logging_collector 打开。当我尝试不直接通过修改 postgresql.conf 而是在进入 similarity 数据库后通过 set 命令修改时提示：

```
similarity=# set logging_collector = on;set log_disconnections = on;
ERROR:  parameter "logging_collector" cannot be changed without restarting the s
erver
ERROR:  parameter "log_disconnections" cannot be set after connection start
```

在修改了.conf 也就是配置文件后，应当重启服务器。

3、在实验时出现如下错误

```
STATEMENT:  select *from student,grade where student.id=grade.id;
TRAP: FailedAssertion("!(slot->tts_tupleDescriptor != ((void *)0))", File:
"execTuples.c", Line: 338)
```

错误发生于 ExecStoreTuple()函数中，代码如下：

```
Assert(slot->tts_tupleDescriptor != NULL);
```

而只有 ExecCopySlot()会调用 ExecStoreTuple()，且由日志输出可知是

```
ExecCopySlot(block[i%size], outerTupleSlot);
```

此处的 block[i%size]的属性 tts_tupleDescriptor 为 NULL，而 block 中元素的初始化是通过 MakeTupleTableSlot()实现的，于是查看 MakeTupleTableSlot()发现，在初始化一个空的 TupleTableSlot 时会把 tts_tupleDescriptor 属性置为 null。于是尝试通过 ExecSetSlotDescriptor()函数把 block[i]的 tts_tupleDescriptor 属性置成和 outerTupleSlot 一样，可行。

4、最初自然地以为在一次函数调用中就可以返回所有的匹配成功的元祖，但在看原始代码时发现似乎只要匹配到一条就会 return，但并未深究，仍然按照一次调用的想法修改，结果在查询时久久没有结果。查看日志后，发现在不断地进入->匹配成功->退出->又进入的循环往复中，于是我意识到可能是多次调用该函数。这个想法是解释得通的：传入的参数 node*有一个属性 nl_NeedNewOuter，标识是否可以取外部关系 R 的下一个元祖，相当于记录了上次退出时的外部关系状态；而且读取元祖是直接通过函数 ExecProcNode（），并没有一个在函数内部定义的记录元祖读取位置的变量，猜测是通过 node 的某一个属性记录。事实上，查看 ExecNestLoop()函数的作用说明"outerTuple contains current tuple from outer relation and the right son(inner relation) maintains "cursor" at the tuple returned

previously. This is achieved by maintaining a scan position on the outer."就可以明白确实是多次调用。于是修改代码，引入静态变量保存上次退出时的状态。

5、实验时出现了如下错误：

```
STATEMENT:  select *from student,grade where student.id=grade.id;
LOG:  index:1

STATEMENT:  select *from student,grade where student.id=grade.id;
LOG:  server process (PID 15284) was terminated by signal 11: Segmentation fault
```

搜索得知 Segmentation fault 是访问了不存在的内存，由日志记录可推出发生于如下语句：

```
ExecCopySlot(outerTupleSlot,block[p]);
```

是因为 outerTupleSlot 没有分配空间，而 ExecCopySlot 的作用不是得到一个引用而是产生一个副本，所以要求事先开辟空间。考虑到这个并不需要有副本的功能，仅仅是一个别名，不需要在 block 销毁之后还存在，于是把 copy 改成直接赋值；但对于另一处 copy(如下)则不可改成赋值：

```
ExecCopySlot(block[i%size], outerTupleSlot);
```

因为匹配成功 return 之后 outerTupleSlot 会损毁，但 block 是静态要保存，所以不能是引用而应该有自己的空间分配产生一个副本。

6、一开始没有释放 block 开辟的空间，在每次查询后都会 warning：

```
WARNING:  TupleDesc reference leak: TupleDesc 0x7fe562671558 (16401,-1) still referenced
WARNING:  TupleDesc reference leak: TupleDesc 0x7fe562671558 (16401,-1) still referenced
```

但我没有引起注意，直到最后磁盘空间不足（no space left on device）导致编译失败。 输入命令 sudo apt autoremove --purge snapd 清理磁盘，并在代码中最后一次调用退出前加上释放空间的函数 ExecDropSingleTupleTableSlot（） 。

参考

- PostgreSql 的 Explain 命令详解 https://toplchx.iteye.com/blog/2091860
- ubuntu 下如何查看 postgresql 的运行日志 https://blog.csdn.net/hesongGG/article/details/52913894
- /dev/loop0-/dev/loop29 占用 100%解决办法 https://blog.csdn.net/Allyli0022/article/details/89882157