数据库引论 期末实验

实验报告

**Part I**

**Similarity Joins**

一、实验目的

①理解PostgreSQL源代码，理解PostgreSQL查询的执行过程。

②修改给定的PostgreSQL源代码，添加计算两个字符串的Levenshtein Distance和Jaccard Index的功能。

二、开发环境

操作系统：Ubuntu 20.04.4

数据库管理系统：PostgreSQL 9.1.3

三、查询执行流程

先从PostgreSQL的执行流程进行理解，以便我们后面理解如何添加自定义SQL查询函数。

1.服务器连接

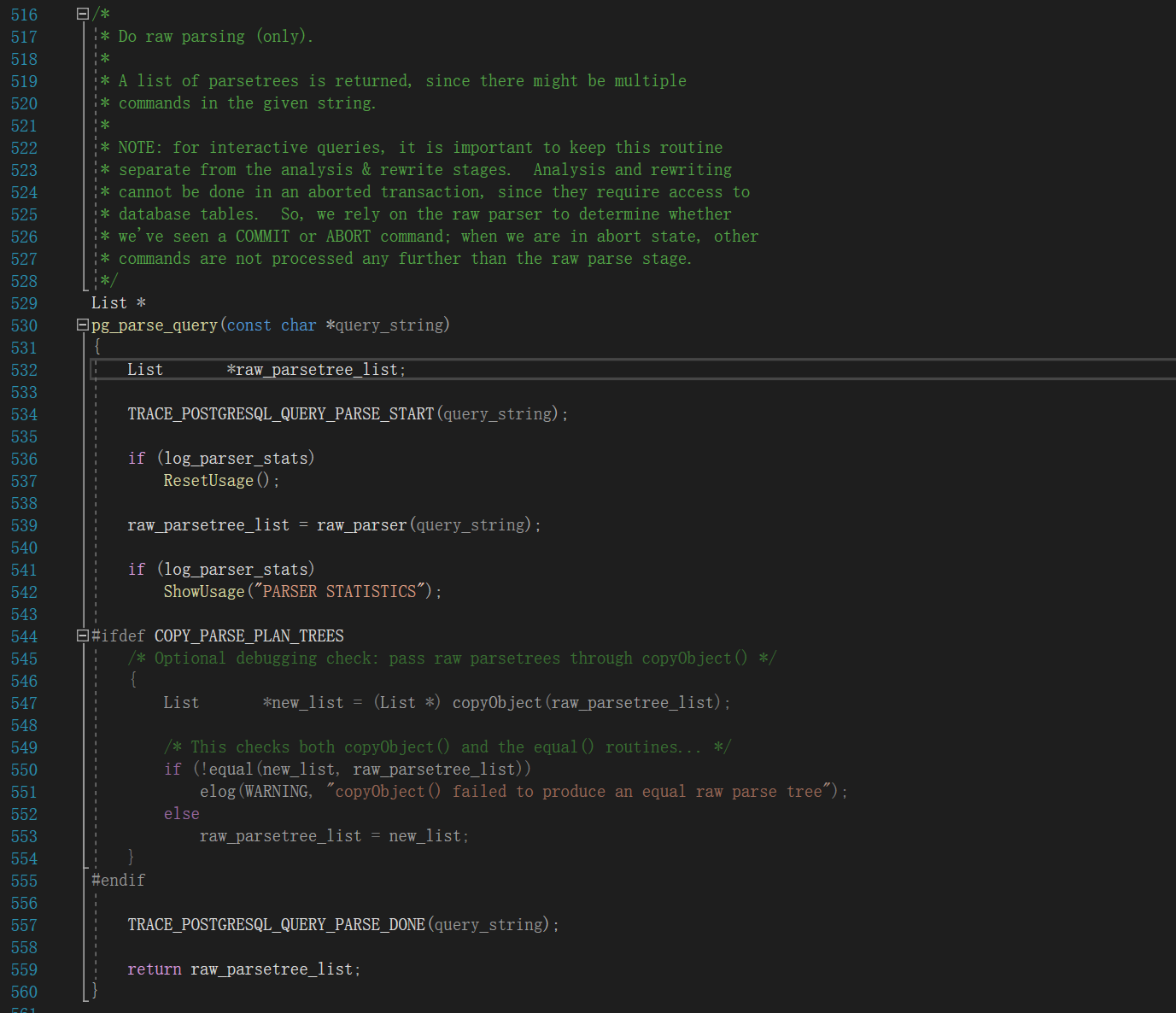
当符合PostgreSQL协议规范的客户端程序发出连接请求时，服务器进程被发出与客户端进行通信。建立了通信连接之后，客户端进程就可以向服务器进程发送查询了。

2.检查分析

查询文本发送到服务器进程后，服务器进程调用分析器进行分析，分为两个阶段：生分析(raw analysis)和语义分析(parse analysis)。

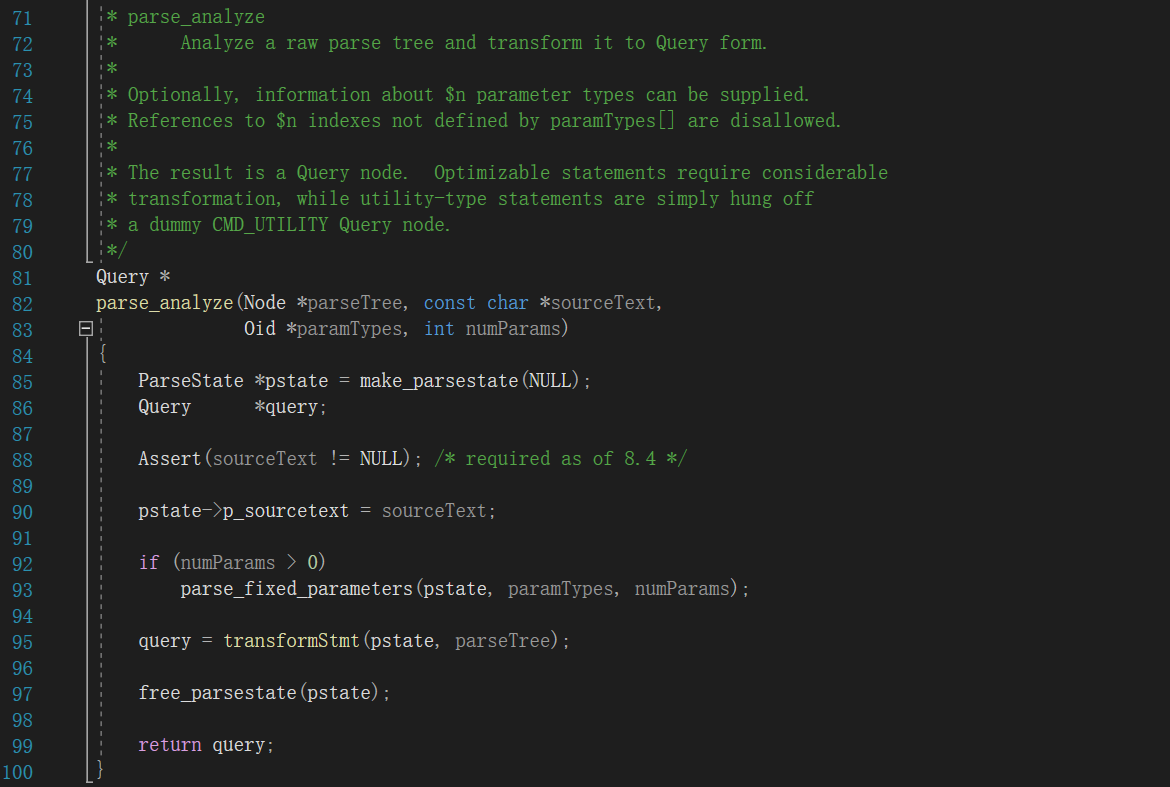
①raw analysis：string转换为生的语法分析树parseTree。这一步主要是判断语法的正确性，并对发现的关键字和标识符进行标记。

在src\backend\tcop\postgres.c中，找到函数pg\_parse\_query，该函数接受用户在终端输入的字符串，返回一个生的语法分析树列表。



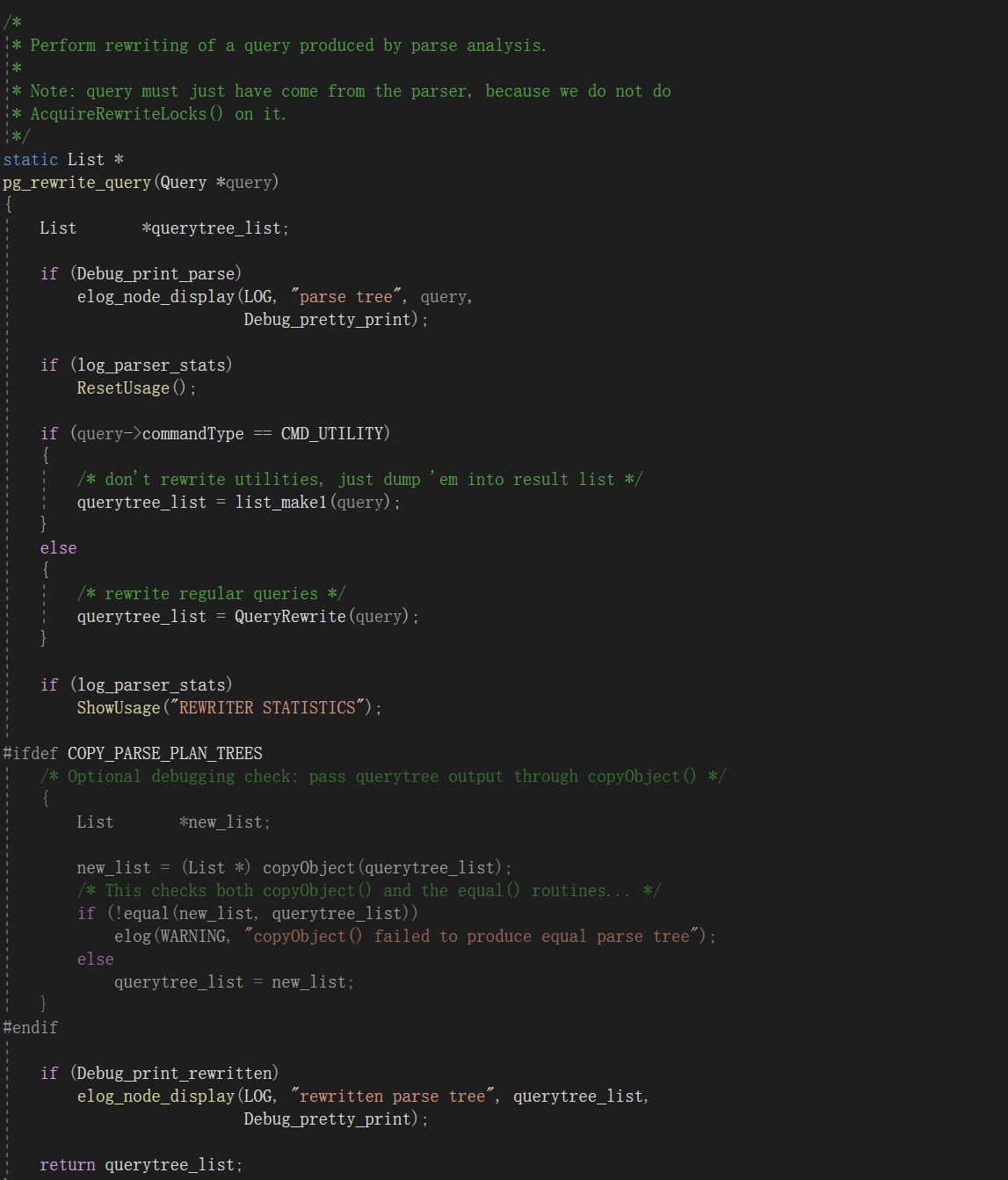
②parse analysis：将语法分析树列表重写为查询树列表

位于src\backend\parser\analyze.c中的函数pg\_analyze接受生的语法分析树结点和查询字符串，完成对查询的分析，返回一个查询类型的指针Query\*。这一步主要是分析查询中引用了哪些表、函数和操作符，并进行相应的转换。



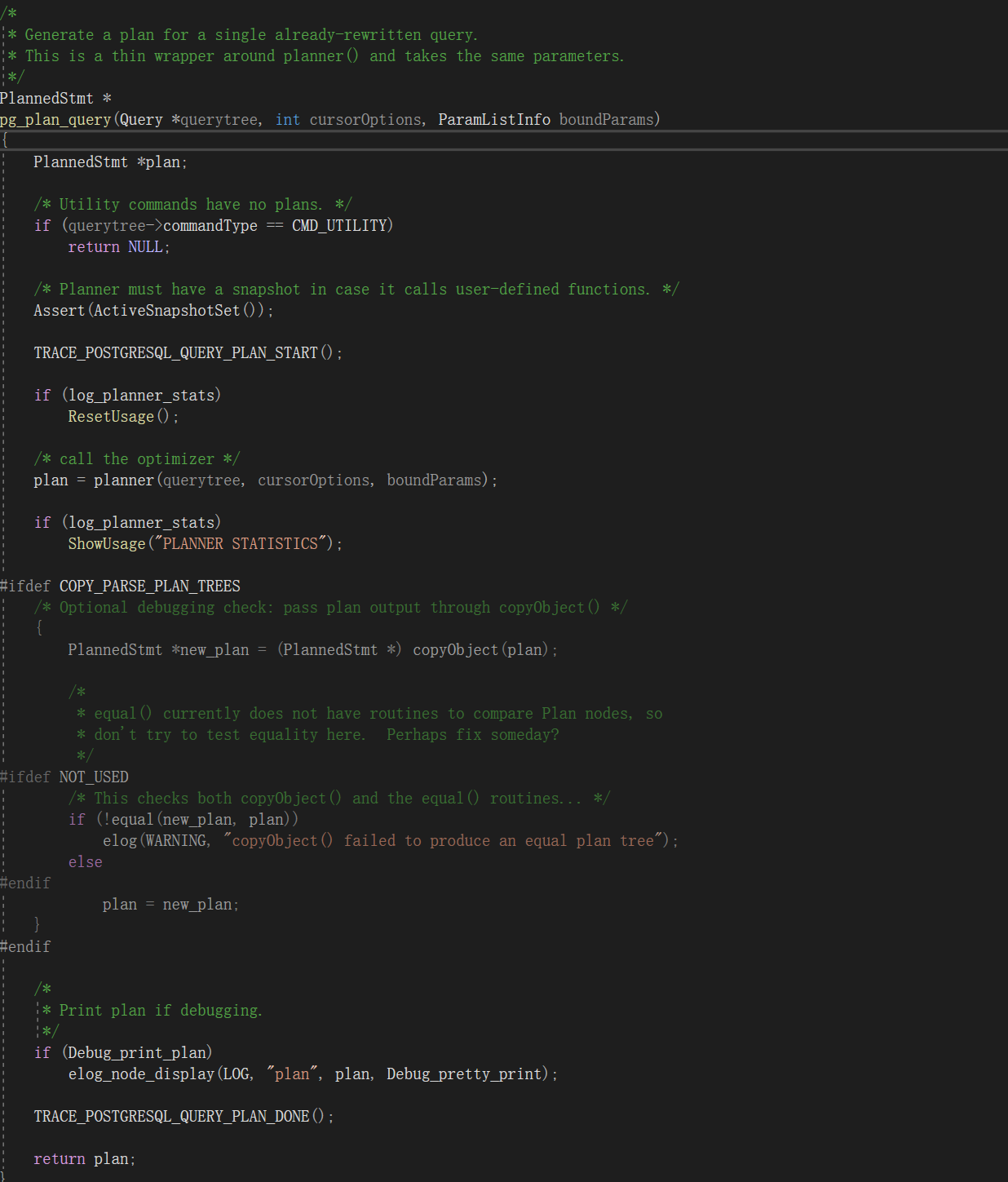
3.重写

将上一阶段生成的Query\*进行重写优化，生成一个查询树列表。这一步由函数pg\_rewrite\_query完成。



4.生成查询执行计划

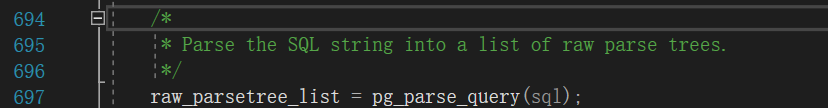
优化器/规划器接收查询树列表，生成相应的查询执行计划。这一步由函数pg\_plan\_queries完成，其中pg\_plan\_queries就是函数pg\_plan\_query对整个list的调用。



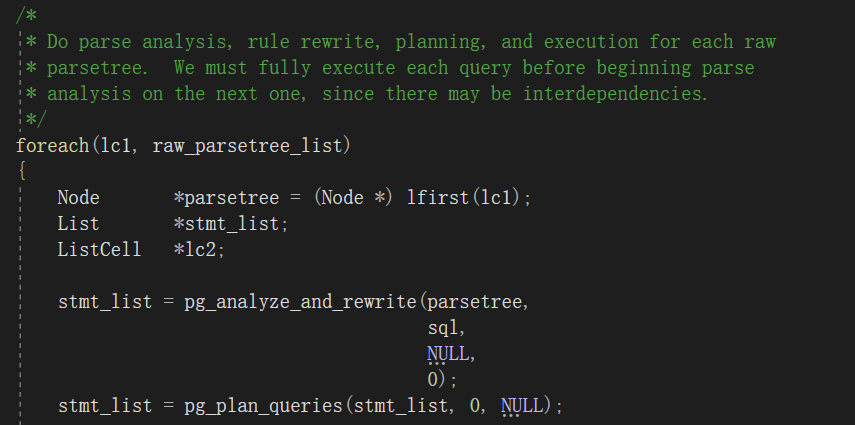
5.执行查询计划

执行器递归访问生成的查询计划。这部分内容见src\backend\executor\execMain.c，主要经过ExecutorStart()、ExecutorRun()、ExecutorFinish()、ExecutorEnd()这4个函数。

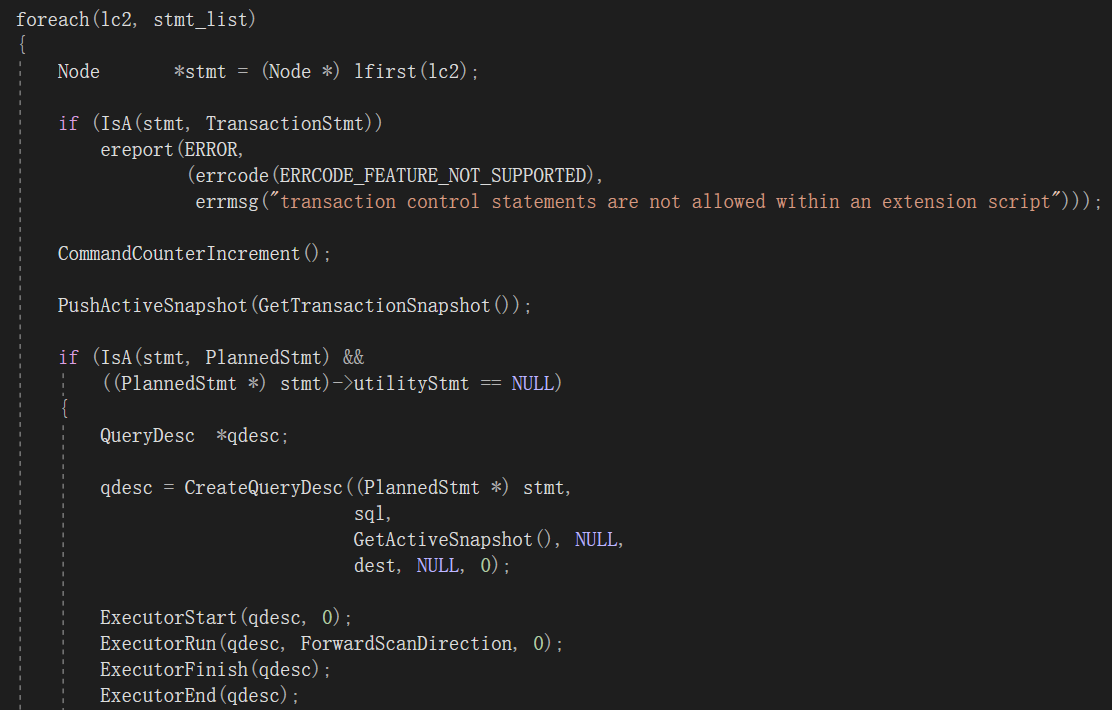
在src\backend\commands\extension.c中能找到执行单次查询的函数exec\_sql\_string（实际PostgreSQL执行的是src\backend\tcop\postgres.c的exec\_simple\_query函数，更加复杂，但流程大体一致），可以验证：



首先用pg\_parse\_query，将sql语句转为生的语法分析树列表。



对每一生的语法分析树调用pg\_analyze\_and\_rewrite转换为查询树列表（这里pg\_analyze\_and\_rewrite也在src\backend\tcop\postgres.c中，实际就是先后调用pg\_analyze和pg\_rewrite这两个函数），再通过pg\_plan\_queries生成查询执行计划列表。



对每个查询执行计划，先通过CreateQueryDesc获取查询的元组，再经过执行的四个阶段。至此，一次查询就完成了。

四、系统源码理解

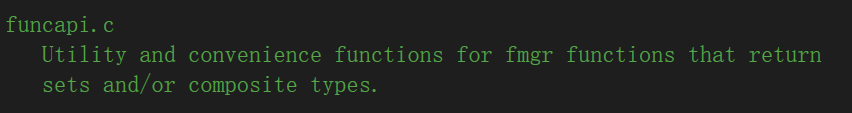
我们的目的是实现Levenshtein Distance和Jaccard Index这两个自定义函数功能，现在我们已经对PostgreSQL的查询执行流程有了大致的了解，现在需要关心的问题在于：如何添加并实现自定义函数？

基于以上问题，先对源码进行结构和功能上的定性分析，再讨论如何添加我们需要的这两个自定义函数。

（一）相关源代码功能简要分析

主要相关源代码只有两个文件：funcapi.c和pg\_proc.h。

①src\backend\utils\fmgr\funcapi.c



存放返回集合或复合类型的函数。我们需要在这里添加两个函数的定义。

②src\include\catalog\pg\_proc.h



系统关系的定义及初始内容，存放了数据库中的内部函数信息。我们需要在这里添加两个函数的声明。

事实上，要添加自定义函数，我们只需要关心funcapi.c和pg\_proc.h，分别添加函数定义和声明即可。因为之前提到过，使用的SQL语句会被pg\_parse\_query函数分析生成语法分析树。在pg\_parse\_query函数中调用了生分析函数raw\_parser（位于src\backend\parser\parser.c），该函数对SQL语句进行关键词的识别，对函数会生成相应的FuncCall变量。之后按照流程生成分析树。在分析树生成查询树的过程中，根据生成的FuncCall变量，会对系统表pg\_proc.h进行查找、判定需要的最优函数，并添加到执行计划中。用于查找的函数为ParseFuncOrColumn，可以在src\backend\parser\parse\_func.c中找到。（关于这部分内容，具体可参考<https://cloud.tencent.com/developer/article/1157311>）

（二）添加自定义函数

①在funcapi.c中添加函数定义

添加的函数形式为：

这里涉及到两个PostgreSQL自定义类型：Datum和PG\_FUNCTION\_ARGS。

在src\include\postgres.h中，可以找到Datum类型定义为：



点进去查看unintptr\_t的定义为：



所以postgres中的Datum类型实际就是unsigned int类型。

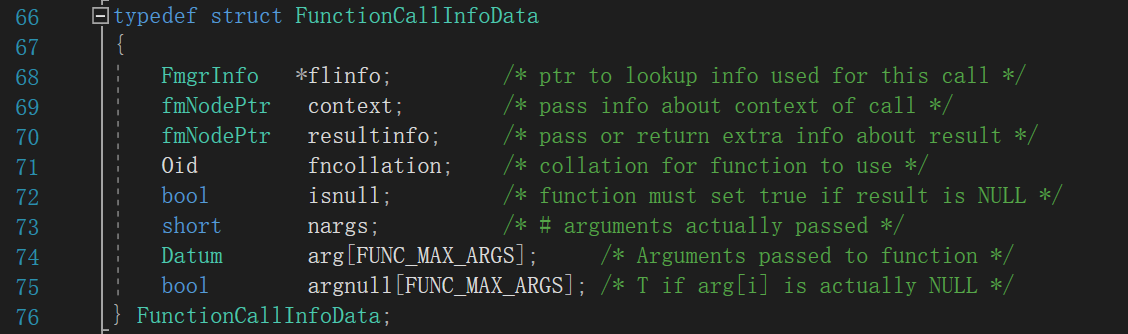
对于PG\_FUNCTION\_ARGS，在src\include\fmgr.h中找到其定义为：



接着追查FunctionCallInfo的定义：



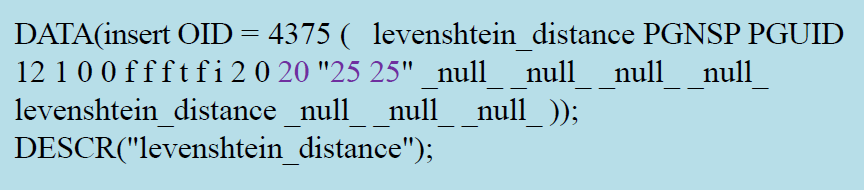
可以看到FunctionCallInfo是FunctionCallInfoData的指针类型。查看FunctionCallInfoData类型，其定义为如图的结构体。



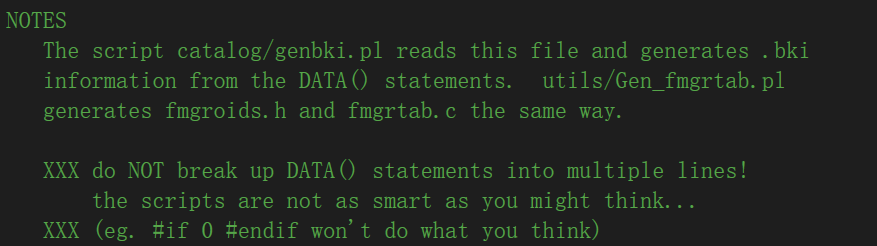
内部细节我们不需要过深掌握，大致了解即可。

②在pg\_proc.h中进行函数声明

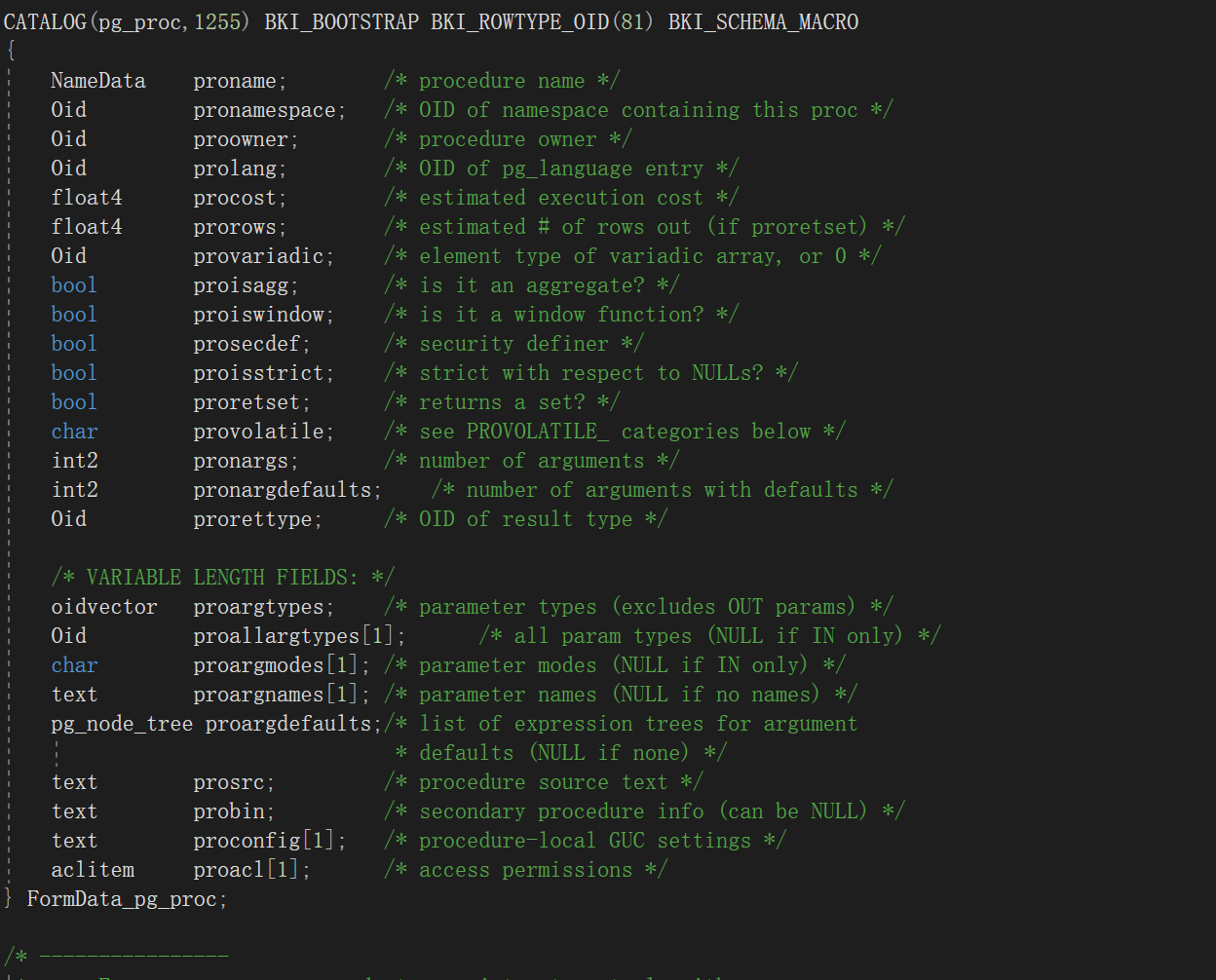
声明类型为：



查看pg\_proc.h给出的说明：

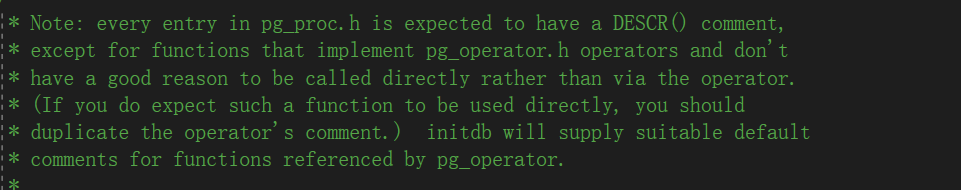


给出的DATA()声明会被读取并转换。（注意不要将DATA()声明分成多行）其中，DATA()中insert后的声明要按照CATALOG的形式给出，如下图：



可以将其与PPT所给出的DATA()声明一一对应，这里不深入讨论了。

此外，根据要求，除了不被显示调用的操作符函数外，每一个声明都应有一个DESCR()批注。



对于Levenshtein Distance，这句批注是”levenshtein\_distance”。

五、实验原理

（一）Levenshtein Distance

①定义

Levenshtein Distance用于衡量两个字符串的相似程度，其定义为：

其中，一次编辑定义为：

①添加一个字符；

②删去一个字符；

③替换一个字符。

可以从定义看出，Levenshtein Distance越小，两个字符串越相似。当Levenshtein Distance=0时，两字符串完全相同。

②算法实现

设有一个大小足够的二维数组d，其中d[m][n]保存了s的前m个字符组成的字符串与t的前n个字符组成的字符串的Levenshtein Distance。

首先有d[i][0]=d[0][i]=i，删除所有字符是变成空字符串的最快方法。

基于动态规划的思想，之后开始递推。记si表示s前i个字符组成的字符串。考虑以下2种情况：

a.s的第i个字符等于t的第j个字符

这说明从si变到tj所需要的编辑次数和si-1到tj-1的编辑次数是一致的，因为最后一位不需要更改，从而d[i][j]=d[i-1][j-1]，完成了递推。

b.s的第i个字符不等于t的第j个字符

若已知d[i-1][j]，则从si变到tj可以先删去s的第i个字符，再从si-1变到tj；

若已知d[i][j-1]，则从si变到tj可以先从si变到tj-1，再在tj-1的末尾插入一个字符变为tj。

若已知d[i-1][j-1]，则从si变到tj可以先删去s的第i字符，从si-1变到tj-1，再在tj-1的末尾插入一个字符变成tj。（先删去再插入即替换）

由Levenshtein Distance的定义知编辑次数应最小，所以最终d[i][j]应取三者最小值再加1。

从而相应的状态转移方程为：

我们所需要的结果就是d[m][n]。

该算法的时间复杂度为。

（二）Jaccard Index

①定义

Jaccard Index同样用于衡量两个字符串的相似程度，记两个字符串分别为和，长为和，则令集合

A和B集合中的元素为连续两个字符构成的有序二元组，且对于开头字符和结尾字符补上’$’符号，比如，对于字符串”apple”：

A={$a,ap,pp,pl,le,e$}

则Jaccard Index定义为：

可以看出，两个字符串越相似，则Jaccard Index越大。当Jaccard Index=1时，说明两个字符串完全相同；Jaccard Index=0时，说明两个字符串没有任何相同之处(|A∩B|=0)。

②算法实现

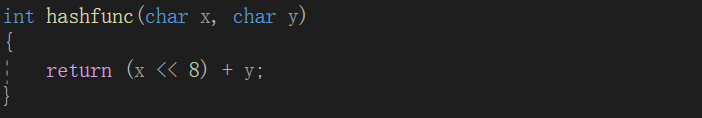
最简单最直接的想法就是模拟这个过程，即先用字符串s和t产生对应的二元组集合A和B，再分别去计算A∩B和A∪B的大小，最后作商。算法的重点在于：

a.如何去除集合中的重复？

b.如何高效计算交集和并集的大小？

首先是二元组集合的产生。实际上，我们并不需要实际产生这样的集合，而是考虑构建一个以s的二元组为key的Hash表。一方面起到了去重的作用，另一方面，在检查t的每个二元组时就能在时间内知道正在检查的二元组是否已经在s中出现过，而不用再扫描一遍s。

考虑对于二元组xy（代指两个字符，不特指x和y），我们希望让一个xy对应到唯一的一个Hash值，这样当对s生成Hash表后，遍历t时只需要查看对应二元组的Hash值的内容就能判断是否有重复了，而避免再对s进行一次扫描。一个可行的hash函数是，将两个字符的二进制表示连在一起共16位并表示为一个整数，hash值的范围为0到65535(即0xFFFF)，且确保了hash(x,y)=hash(x’,y’)当且仅当x=x’且y=y’。具体的hash函数可以写成这样：



考虑开辟大小为65536的数组h作为hash表，对字符串s进行扫描，并把发现的二元组对应hash值位置的值置1（无论是否已经为1都置1，这一步包含了对A集合的去重）。另外此时要开始计算并集的大小，如果检查到hash值为0，说明这个二元组还没有被放入A∪B，从而令并集大小+1，否则并集大小不变。

接下来对t扫描，检索对应的hash位置的值：为1时，说明该二元组在s中也出现过，令交集大小+1，并把该hash值改为2（这一步是对A∩B去重，保证交集中每个元素只出现一次）；为0时，说明该二元组在s中没有出现过，令并集大小+1，并把该hash值改为3（同理对A∪B去重）。

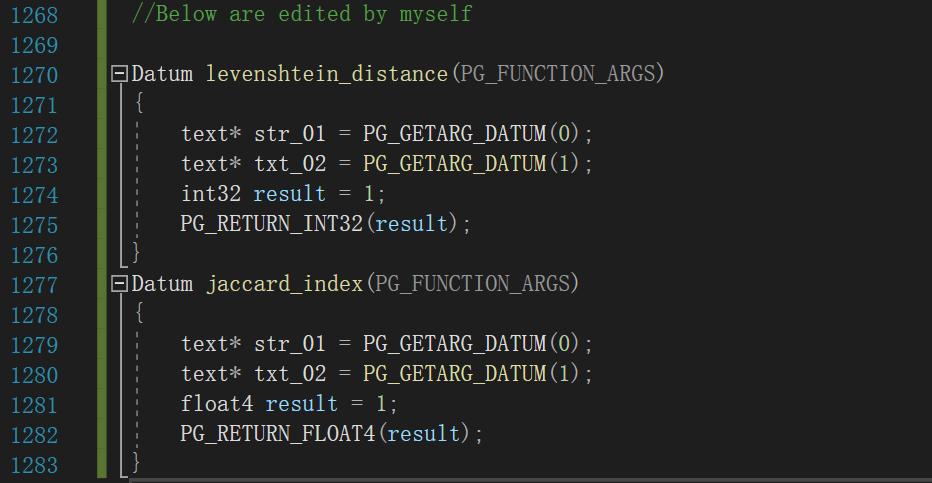
这个思路虽然做到了hash值唯一，但由于开辟数组较大，如果每次都要初始化这么多元素则耗费的时间很可能已经超过了计算交集大小的时间，且也造成了很大的空间浪费，从而这样的算法显得不够实用。

我们可以考虑用一个clean数组标记h数组被修改的下标，因为s和t的长度不超过100，从而被修改的下标不超过202个，从而clean数组大小只需要不小于202即可。在第二次调用Jaccard index时，就可以根据clean数组来将h数组恢复到初始状态了，从而极大地降低了每次初始化的时间。

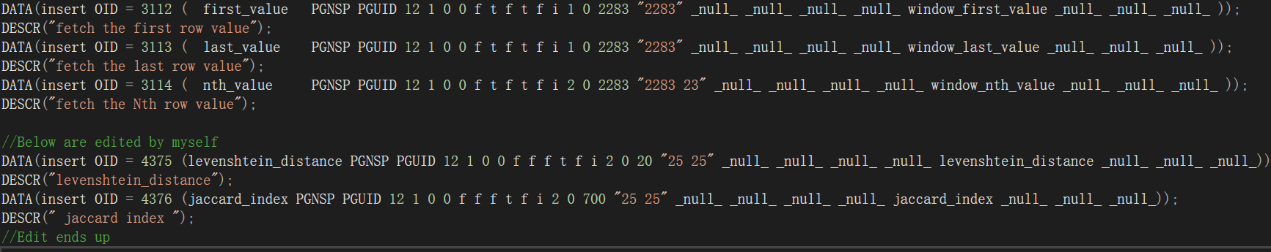
该算法的时间复杂度为。

六、具体实现

先找到funcapi.c（位于src\backend\utils\fmgr），添加Levenshtein Distance和Jaccard Index两个函数



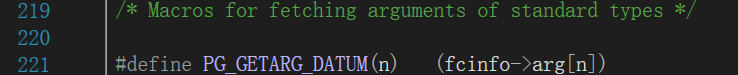
并在pg\_proc.h（位于src\include\catalog）中添加相应的声明，如图。



现在对两个函数进行具体实现：

①Levenshtein Distance

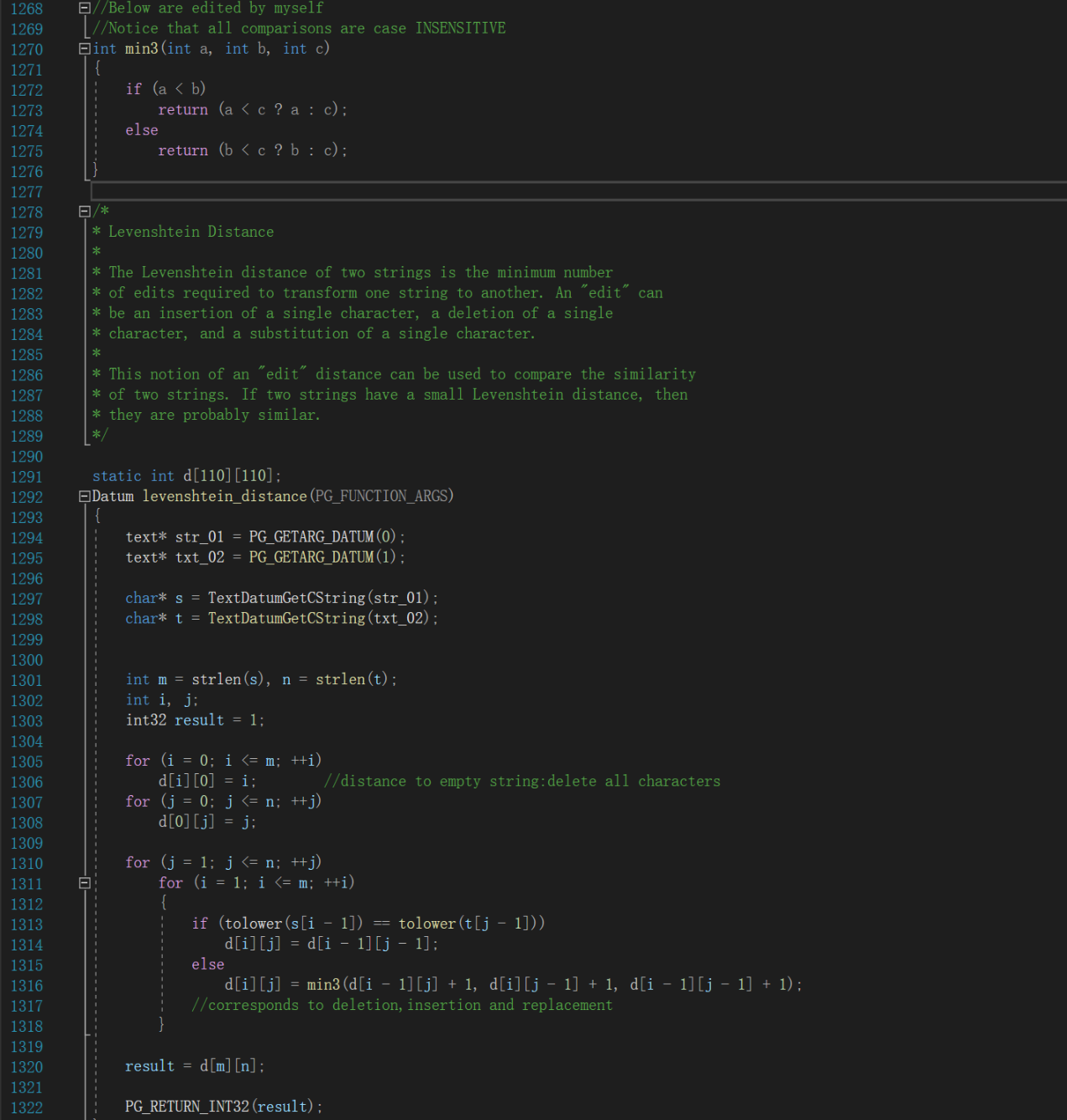
在进行算法的编写之前，首先必须理解PostgreSQL源码中相应的宏定义和自定义类型。

PG\_GETARG\_DATUM是一个宏定义，在src\include\fmgr.h中能找到其定义：

前面已经指出，fcinfo是FunctionCallInfoData\*类型变量，其指向的arg成员为传入函数的参数，所以PG\_GETARG\_DATUM(n)的作用为取得传入函数的第n个参数。在这里，PG\_GETARG\_DATUM(0)和PG\_GETARG\_DATUM(1)取得了传入函数的两个字符串，它们都是text\*类型的，这是PostgreSQL的自定义数据类型。我们需要用相应函数先把text\*类型转换成C语言的char\*类型，再执行算法。这里可以用TextDatumGetCString来转换。

PG\_RETURN\_INT32传入INT32（即C语言的int）类型，将结果返回Datum(unsigned int)类型。

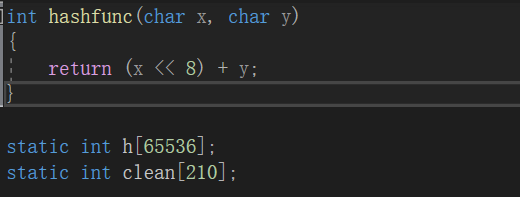
具体算法按照上述思路不难写出，相应注释已给出。



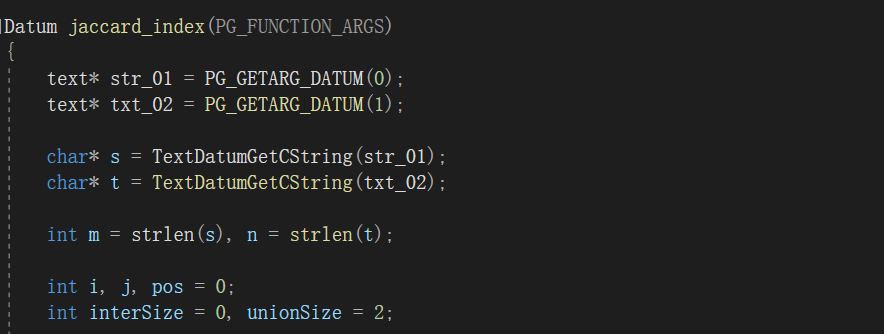
需要注意对里面的字符调用tolower函数，统一转换成小写形式比较，从而达到大小写不敏感的要求。另外注意开辟的数组d是static的，可以进行多次使用，减少了每次初始化的时间。

②Jaccard Index

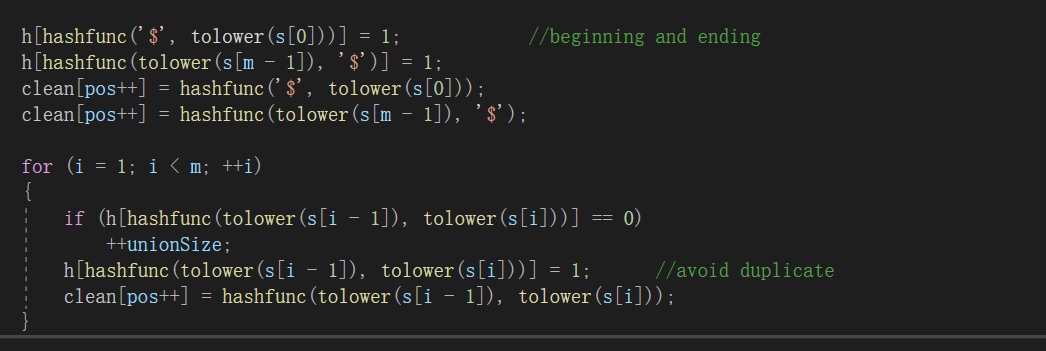
具体算法如图：



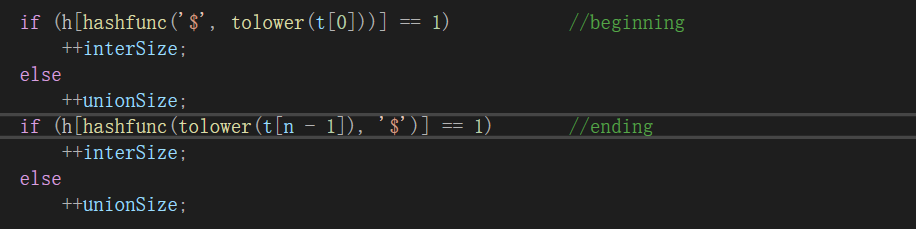
使用的hash函数，以及开辟的数组h和clean。这两个数组都是静态数组，同样是从减少初始化的角度考虑。



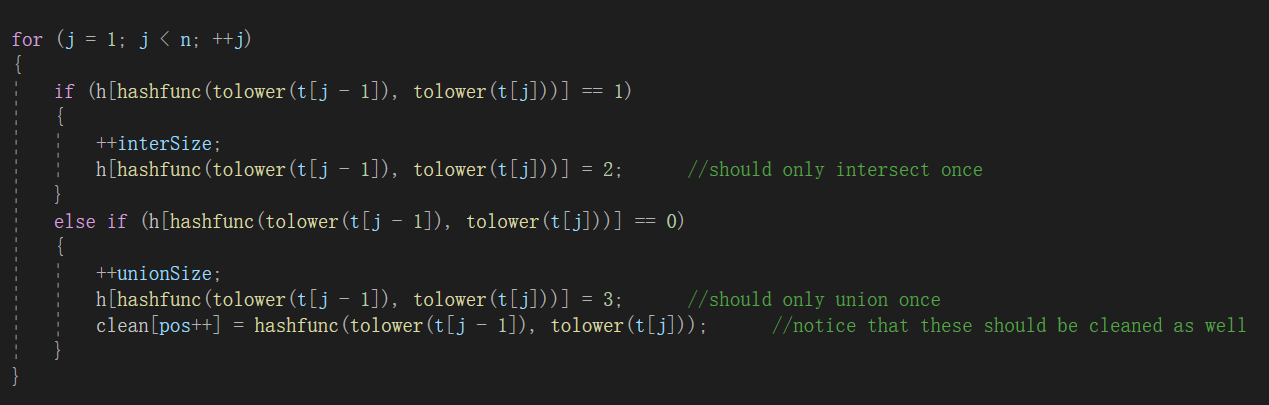
初始设定，注意unionSize为2，表示s字符串的开头和结尾已经被计入。



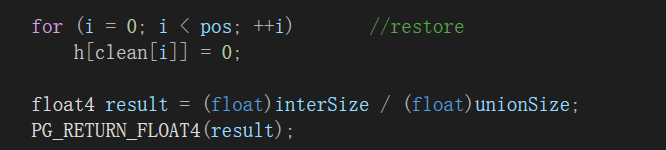
遍历s字符串，设置h数组和clean数组的值。因为字符串开头和结尾的二元组含有’$’，这在字符串其他位置一定不出现，所以不考虑这两个位置的重复情况。



检查t字符串的开头和结尾。如果该二元组在s中出现过则交集大小+1，否则并集大小+1。这里不用对A∩B和A∪B去重，理由同上。



对t字符串内部的二元组检查。这里在检测到h数组对应值为1或0时就要对h数组对应值修改，保证t中相同的二元组在交集、并集中只出现一次。另外，检测到h数组对应值为0时还需要把这个hash值记录到clean数组中，因为对h数组的值做了修改，之后需要恢复回来。h数组对应值为1时虽然也修改了h数组的值，但由于值为1表示这个二元组在s中出现过，说明该hash值肯定已经被记录过了，所以不需要再记录一次。



利用clean数组记录的hash值恢复h数组到全为0的初始情况，返回计算结果。

七、实验测试

（一）环境配置

在终端中进行环境配置：

①设置configure

cd postgresql-9.1.3

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

（可能需要先安装gcc,make,libreadline等）

②编译安装

make

make install

③初始化数据库

（需要先退出postgresql-9.1.3目录)

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

④启动服务器

$HOME/pgsql/bin/pg\_ctl -D $HOME/pgsql/data -l logfile start

⑤连接服务器，创建similarity数据库

$HOME/pgsql/bin/psql postgres

create database similarity;

⑥退出数据库，导入数据到similarity数据库

$HOME/pgsql/bin/psql -d similarity -f /home/postgre/similarity\_data.sql

⑦进入similarity数据库

$HOME/pgsql/bin/psql similarity

至此就可以开始进行查询测试了。

如果修改了代码，需要重新编译安装，则需要先执行：

a.停止数据库：

$HOME/pgsql/bin/pg\_ctl -D $HOME/pgsql/data -l logfile stop

b.删除数据库：

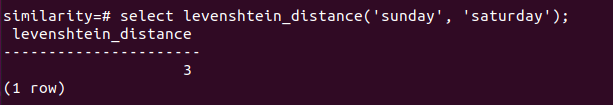
rm -r $HOME/pgsql

执行完这两步后再从上述的第①步开始执行即可。（注意执行到初始化数据库时，需要删除—locale=C文件夹）

（二）查询测试

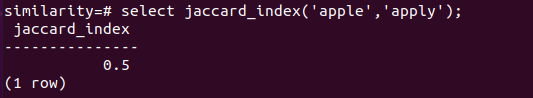
先对给出的单次查询

select levenshtein\_distance('sunday', 'saturday');

进行测试：

结果正确无误。再测试Jaccard Index:

select jaccard\_index(‘apple’,’apply’);

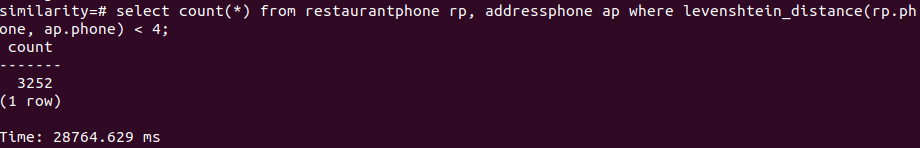


同样得到了正确的结果。

接下来，对给定的大数据集进行测试。在测试之前，先用\timing打开计时来衡量程序的执行性能。



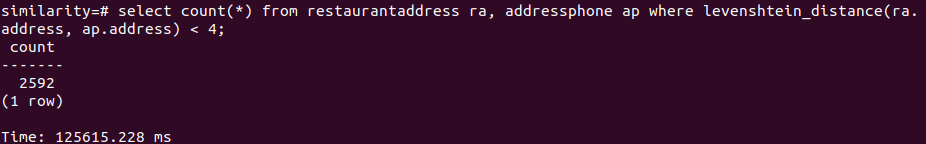
①select count(\*) from restaurantphone rp, addressphone ap where levenshtein\_distance(rp.phone, ap.phone) < 4;



②select count(\*) from restaurantaddress ra, restaurantphone rp where levenshtein\_distance(ra.name, rp.name) < 3;

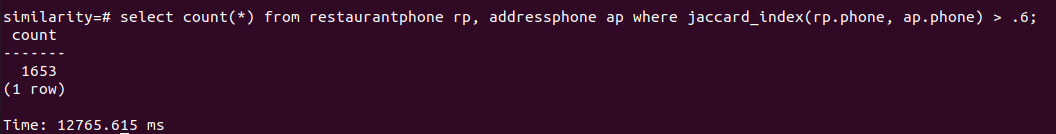


③select count(\*) from restaurantaddress ra, addressphone ap where levenshtein\_distance(ra.address, ap.address) < 4;

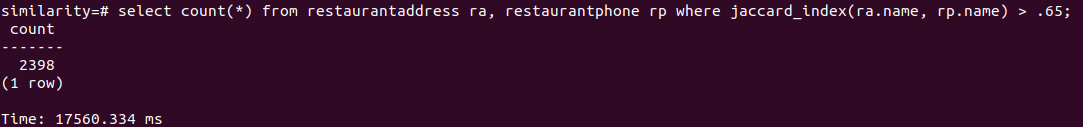


|  |  |  |  |
| --- | --- | --- | --- |
| Levenshtein Distance | 查询① | 查询② | 查询③ |
| 查询结果 | 3252 | 2130 | 2592 |
| 查询时间 | 28765ms | 61908ms | 125615ms |

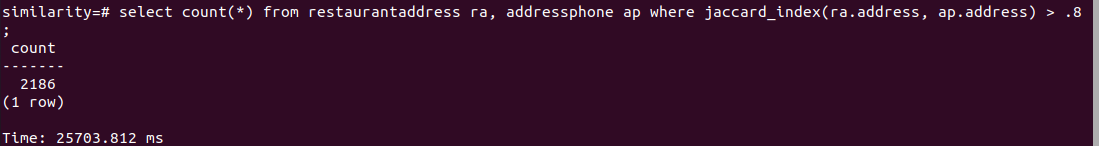
④select count(\*) from restaurantphone rp, addressphone ap where jaccard\_index(rp.phone, ap.phone) > .6;



⑤select count(\*) from restaurantaddress ra, restaurantphone rp where jaccard\_index(ra.name, rp.name) > .65;



⑥select count(\*) from restaurantaddress ra, addressphone ap where jaccard\_index(ra.address, ap.address) > .8;



|  |  |  |  |
| --- | --- | --- | --- |
| Jaccard Index | 查询④ | 查询⑤ | 查询⑥ |
| 查询结果 | 1653 | 2398 | 2186 |
| 查询时间 | 12766ms | 17560ms | 25704ms |

接下来，先退出数据库，执行以下3条查询，将查询结果写到文件：

①$HOME/pgsql/bin/psql similarity -c "SELECT ra.address,

ap.address, ra.name, ap.phone FROM restaurantaddress ra, addressphone ap

WHERE levenshtein\_distance(ra.address, ap.address) < 4 AND (ap.address

LIKE '%Berkeley%' OR ap.address LIKE '%Oakland%') ORDER BY 1, 2, 3,

4;" > levenshtein.txt

②$HOME/pgsql/bin/psql similarity -c "SELECT rp.phone, ap.phone, rp.name, ap.address

FROM restaurantphone rp, addressphone ap

WHERE jaccard\_index(rp.phone, ap.phone) > .6 AND

(ap.address LIKE '%Berkeley%' OR ap.address LIKE '%Oakland %')

ORDER BY 1, 2, 3, 4;" > jaccard.txt

③$HOME/pgsql/bin/psql similarity -c "SELECT ra.name, rp.name, ra.address, ap.address, rp.phone, ap.phone

FROM restaurantphone rp, restaurantaddress ra, addressphone ap

WHERE jaccard\_index(rp.phone, ap.phone) >= .55 AND

levenshtein\_distance(rp.name, ra.name) <= 5 AND

jaccard\_index(ra.address, ap.address) >= .6 AND

(ap.address LIKE '%Berkeley%' OR ap.address LIKE

'%Oakland%')ORDER BY 1, 2, 3, 4, 5, 6;" > combined.txt

结果见附件的levenshtein.txt、jaccard.txt和combined.txt。

八、算法优化

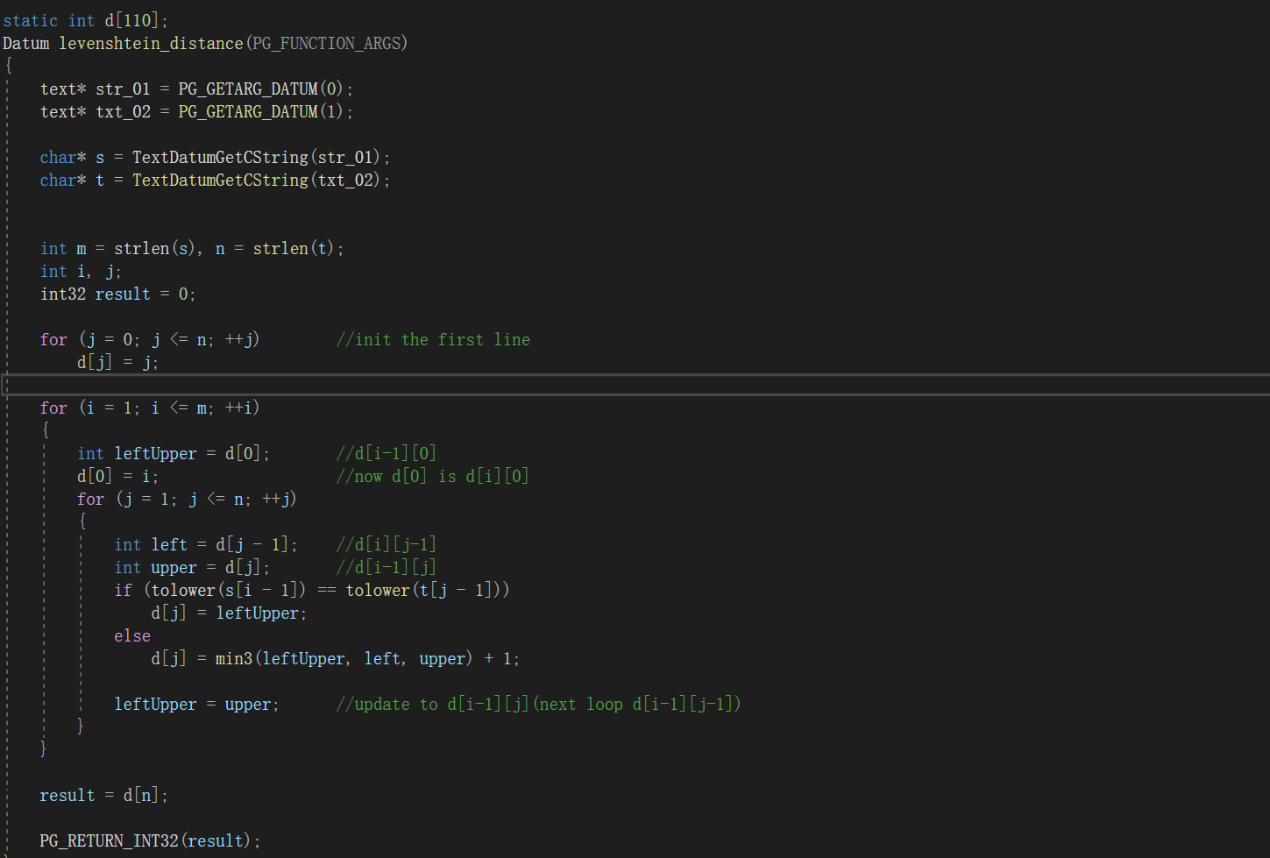
（一）Levenshtein Distance

观察原始算法可知，实际上d[i][j]的获取只与d[i-1][j-1]、d[i-1][j]和d[i][j-1]这三个变量有关，从而原始的d数组可以缩小，减小空间开销和初始化的时间开销。另外也注意到，PPT给出的原始算法外层循环为j，内层为i，这样的程序的空间局部性较差，比如d[i][j]和d[i+1][j]有可能在不同的块中，使得Cache命中率降低而浪费了时间。

具体地，原本的d[110][110]数组可以缩减为一维数组d[110]，数组每次保存的是原本一行的值，每次都是用上一行的数组来计算出下一行的数组。这样，假如进入第i次循环，当前的d数组保存的是原本第i-1行的值，那么计算d[i][j]时，已有的d数组可分为左右两部分：

左边红色括号扩起的部分是已经被更新过的部分，它们代表了原本的d[i][0],d[i][1],…,d[i][j-1]；右边蓝色括号扩起的部分还没有被更新，它们代表了原本的d[i-1][j],d[i-1][j+1],…。我们不难看出，更新后d[j]就对应了d[i][j]，它需要的三个值：①原本的d[i-1][j]，对应了还未被更新的d[j]；②原本的d[i][j-1]，对应了已经更新过的d[j-1]；③原本的d[i-1][j-1]，这个值并不保存在d数组中，所以需要用一个额外的临时变量leftUpper来保存，刚进入这行的循环(j=0)时，leftUpper=d[0]（对应d[i-1][0]），每次更新j后，就令leftUpper=原本的d[j]（对应d[i-1][j]，在下一次内循环中对应d[i-1][j-1]），这样完成了左上角元素d[i-1][j-1]的保存。

对应的实际空间复杂度降低至。代码如下所示：

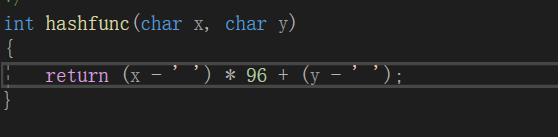


另外，对于时间复杂度的优化，我查阅了一些资料，发现是存在最终可优化至的算法的，其中d是编辑距离。不过算法原理和实现都比较复杂，就不做深入研究了。参考<https://stackoverflow.com/questions/4057513/levenshtein-distance-algorithm-better-than-onm>

（二）Jaccard Index

算法的时间复杂度为，基本没有什么可优化空间了。主要问题在于开辟的hash数组太大，造成初始化时间浪费和空间浪费太大。

考虑到使用的ASCII字符仅128个，去除掉不能显示的特殊字符后就更少了。查找ASCII表可知，从32开始的空格字符是第一个可见字符，前面的不可见字符是不会出现在字符串中的，所以实际需要的hash表只要有96×96=9216的大小即可。相应地，hash函数就可以修改为：



将h数组由65536的大小改为9216的大小。除此之外对主体部分不做更改。

(三)优化测试

优化测试的结果如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| Levenshtein Distance | 查询① | 查询② | 查询③ |
| 查询结果 | 3252 | 2130 | 2592 |
| 原用时 | 28765ms | 61908ms | 125615ms |
| 优化后用时 | 21474ms | 49178ms | 103319ms |
| Jaccard Index | 查询④ | 查询⑤ | 查询⑥ |
| 查询结果 | 1653 | 2398 | 2186 |
| 查询时间 | 12766ms | 17560ms | 25704ms |
| 优化后用时 | 11852ms | 17103ms | 22363ms |

对Levenshtein Distance，用时有一定的减少，可能主要是由于改变了内外层的循环顺序，同时也减少了每次初始化的时间。

对Jaccard Index，用时均减少但幅度较小，因为本身的优化是针对空间的，所以用时减少不多是意料之中的事。

九、实验感想与体会

本次实验的难点主要在如何理解PostgreSQL执行单次查询的过程上，我查阅了相当多资料，也学到了很多PostgreSQL内部实现的相关内容。

对于Levenshtein Distance和Jaccard Index的算法实现，这部分由于可参考的资料比较多，所以实现起来相对来说难度不是很大。当然算法可优化的点还比较多，但考虑到时间因素以及本次实验的主要目的就到此为止了。

对于Linux系统下的环境配置，一开始这也是让我比较感到迷惑的一个点，可能是因为我自己在虚拟机上安装过PostgreSQL，所以make、make install后总是无法连接到服务器。于是我只能将虚拟机彻底删掉重新安装了一遍。另外，由于程序语言是C语言，而我一开始带着C++的习惯来写（比如在for循环里定义临时变量）导致不能顺利运行，也花了我很多时间。在实验过程中也遇到过很多完全搞不清楚的错误，比如提示找不到数据库服务器等，一般只有采取重启或删掉源代码文件重新编译安装的方式解决。

总的来说，本次实验从实际需求出发，从源代码角度添加自定义PostgreSQL函数，不仅让我更深入地了解了PostgreSQL函数的调用和执行过程，在实现函数的过程中也综合运用了数据结构、计算机组成与体系结构等课程的知识，加深了本学期所学的如查询处理等章节的理解。

十、参考资料

1.<https://cloud.tencent.com/developer/article/1157311>

2. <https://blog.csdn.net/driftingman/article/details/122759503>

3.<https://blog.csdn.net/chunchengli0734/article/details/101024624>

4.<https://www.zzzyk.com/show/b83de30b0de60c27.htm>

5.<https://www.cnblogs.com/vagabond/p/9546083.html>

6.<https://leetcode.cn/problems/edit-distance/>

7.<https://stackoverflow.com/questions/4057513/levenshtein-distance-algorithm-better-than-onm>

Part II

Block Nested Loop Joins

一、实验目的

①修改PostgreSQL源代码，将嵌套循环连接优化为块嵌套循环连接。

②理解块嵌套循环连接的实现过程，对实验结果的性能进行比较、评判。

二、实验环境

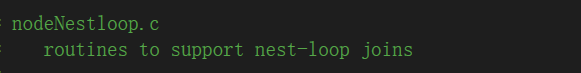
操作系统：Ubuntu 20.04.4

数据库管理系统：PostgreSQL 9.1.3

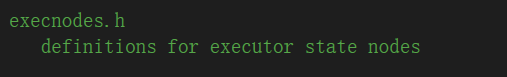
三、相关源代码理解

主要涉及的代码文件为：

①src\backend\executor\nodeNestedloop.c

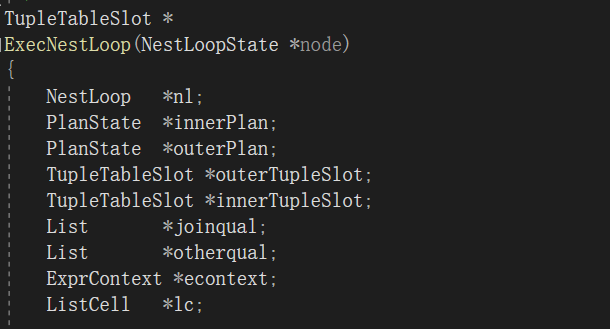


主要包含了对嵌套循环连接的相应函数支持。

②src\include\nodes\execnodes.h

对执行状态结点的定义。

我们首先需要对PostgreSQL的循环嵌套连接过程进行理解，这个过程主要由nodeNestedloop.c的ExecNestedLoop完成。除此之外还有一些的如初始化函数、结束清理内存函数等辅助函数。

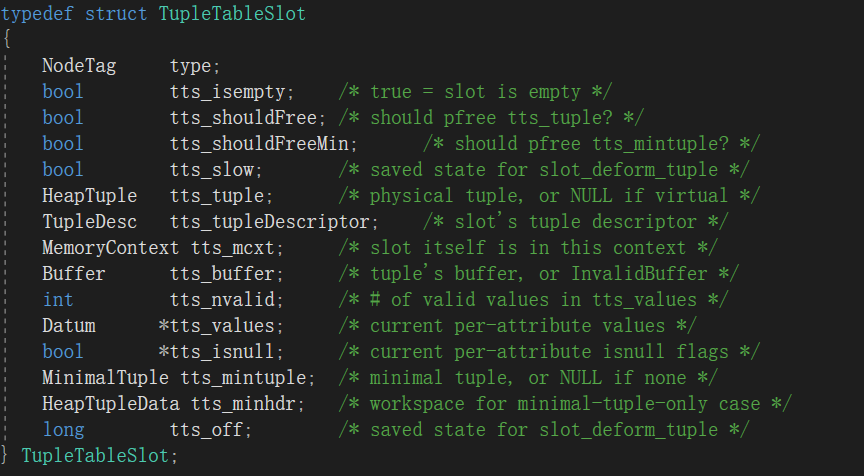


我们首先必须要对使用的数据类型有一个基本的了解。

①TupleTableSlot

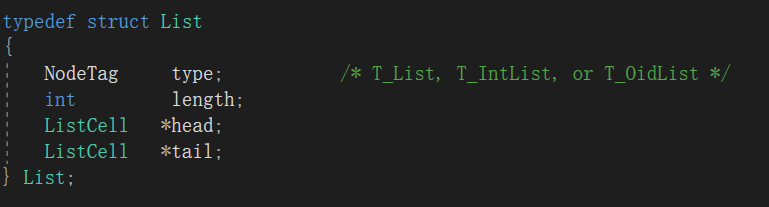
TupleTableSlot用于存储元组相关信息。

在src\include\executor\tuptable.h中可以找到相关定义。



②List

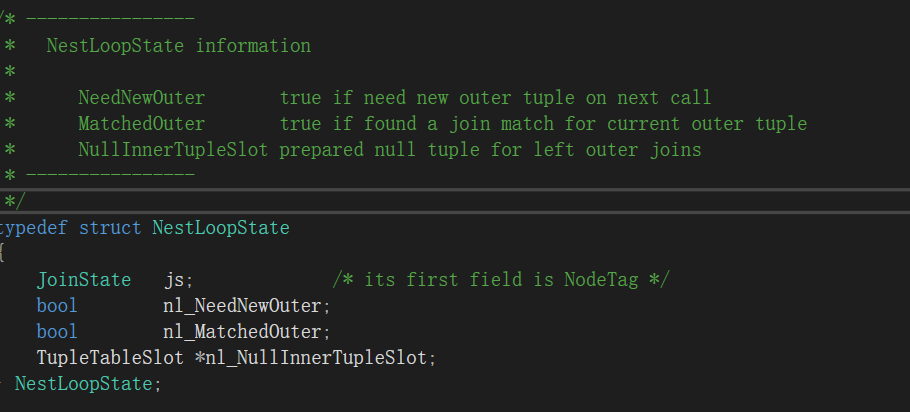
ListCell形成的链表，定义于src\include\nodes\pg\_list.h。



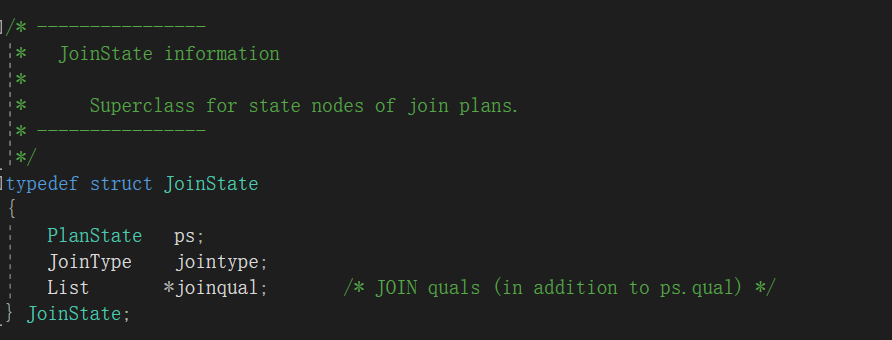
其中ListCell可以存储指针值，或int值，或Oid值。

③NestloopState

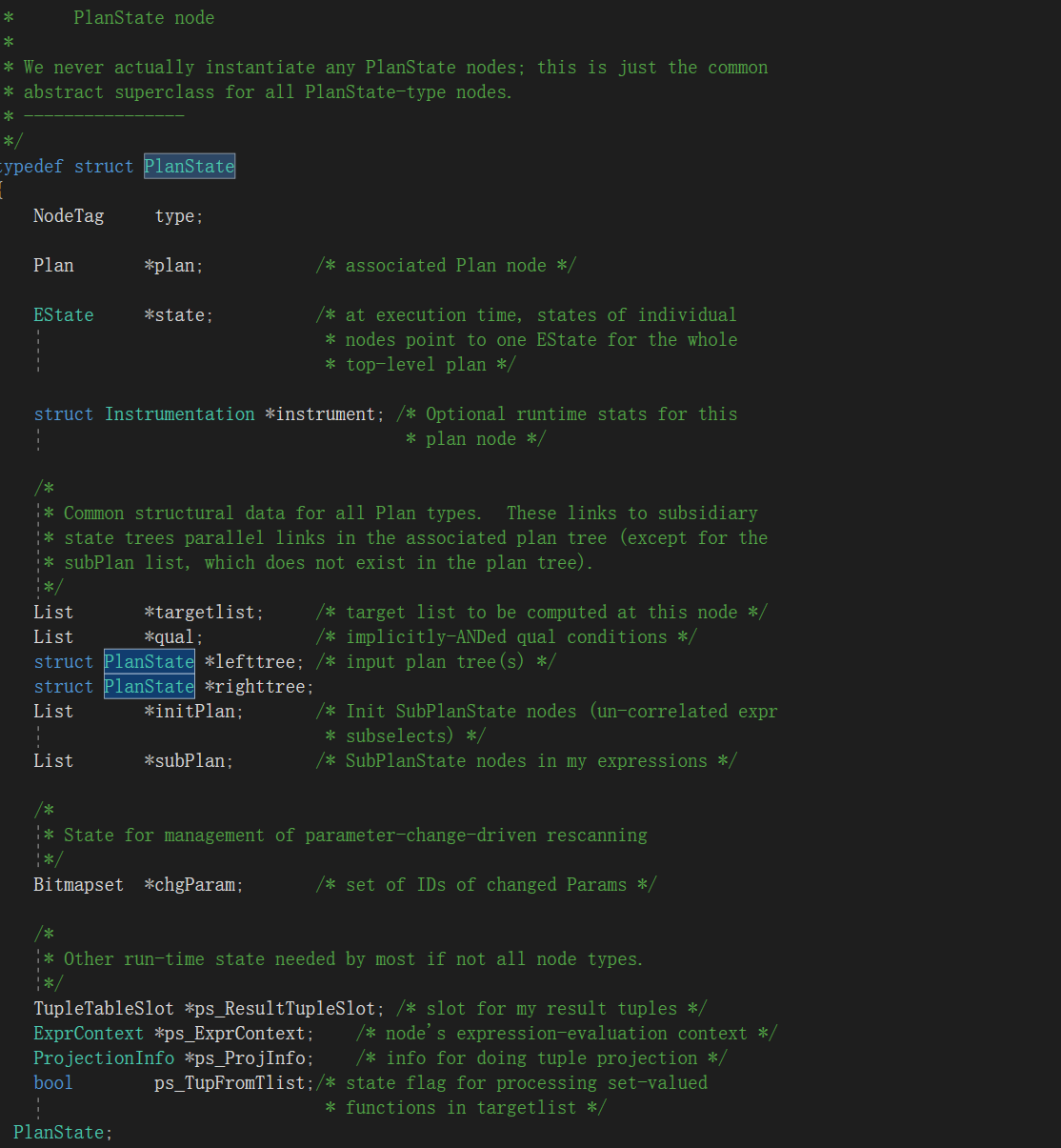
定义于src\include\nodes\execnodes.h。



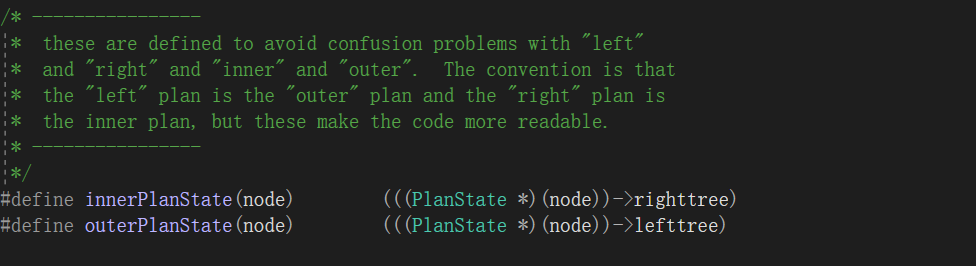
其中，JoinState 是各种连接计划状态结点的父类。



其中PlanState是所有状态结点的抽象父类。



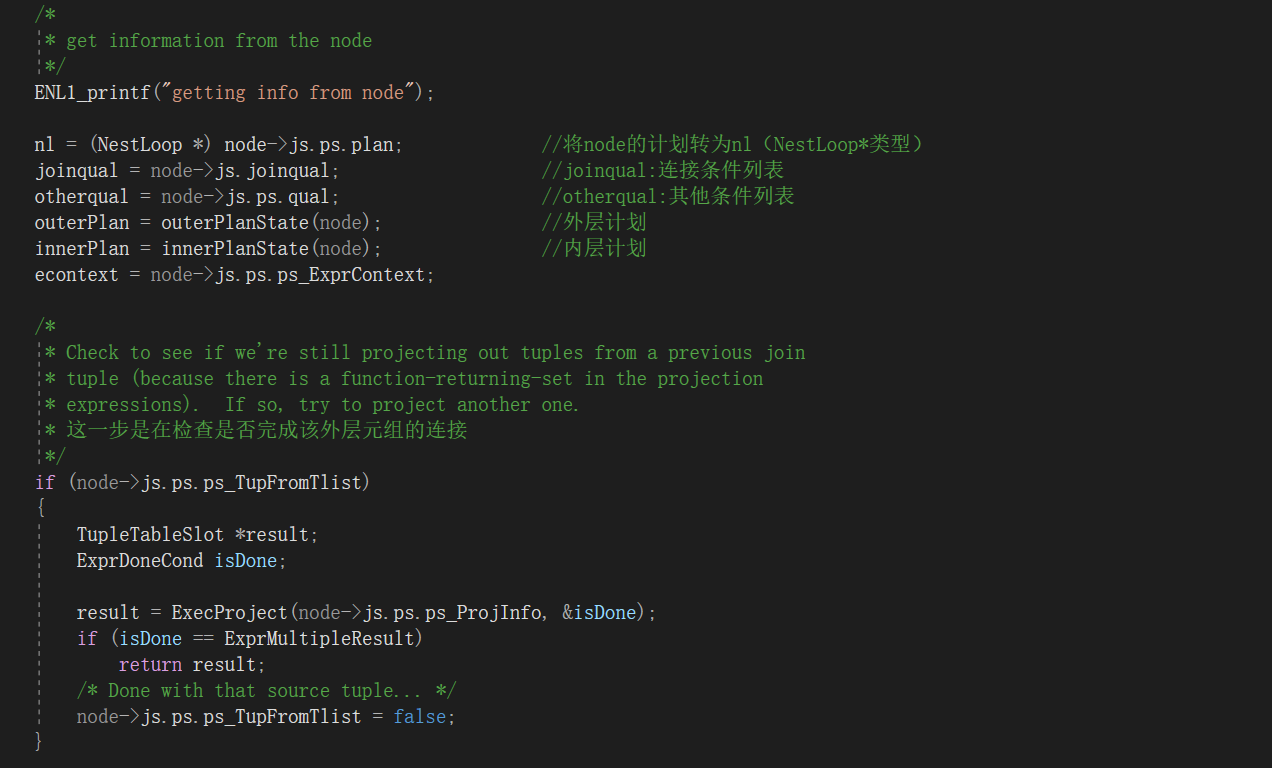
从下面的宏定义，可以看出PostgreSQL对内层和外层的处理方式：将内层计划树作为计划右子树，外层计划树作为计划左子树。

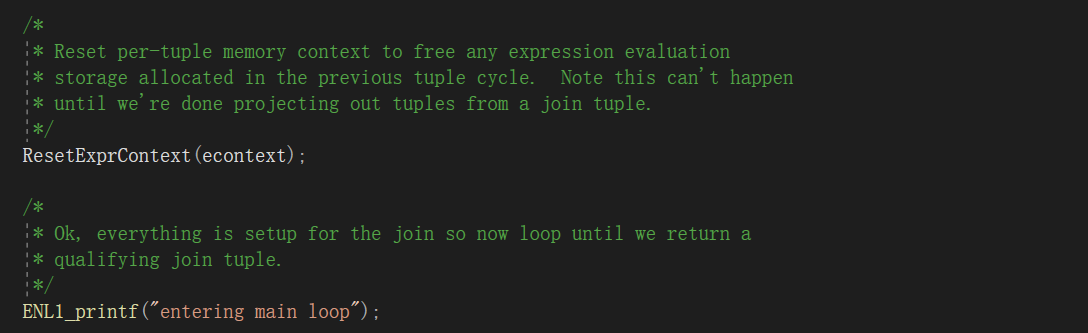


④ExprContext

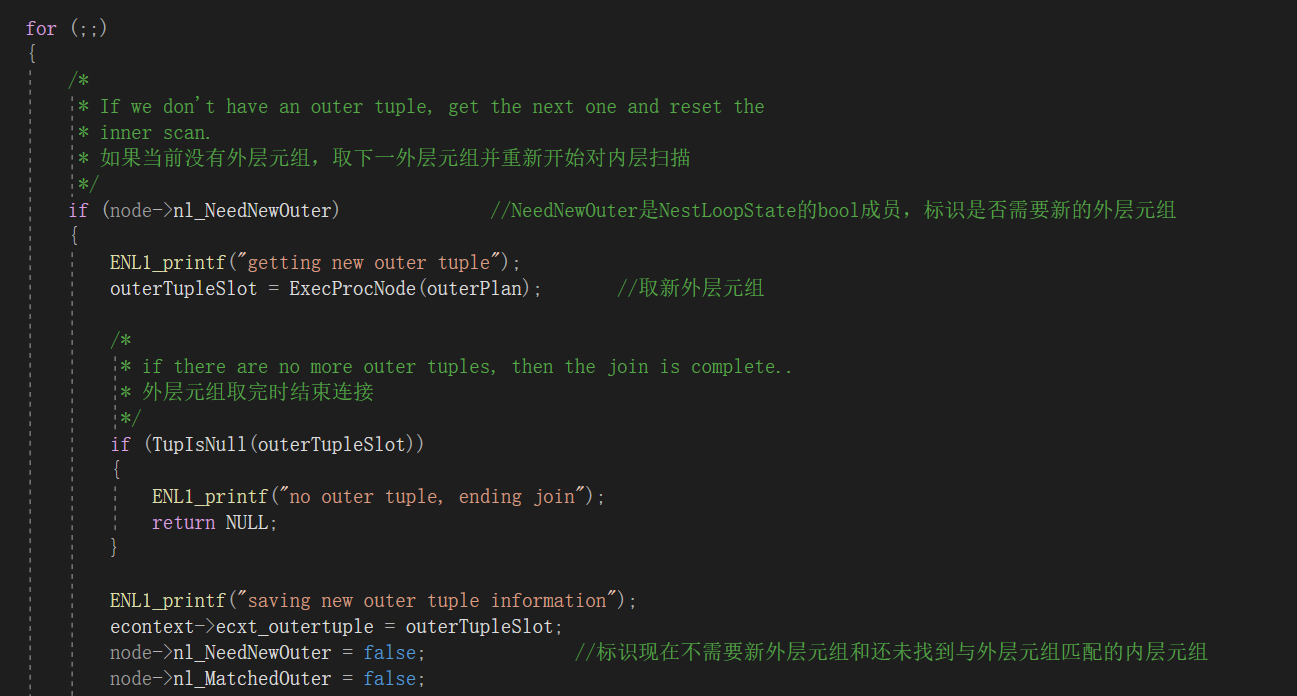
用于保存表达式，这里用来保存了一个外层元组和一个内层元组。

以下为完整的ExecNestedloop函数：

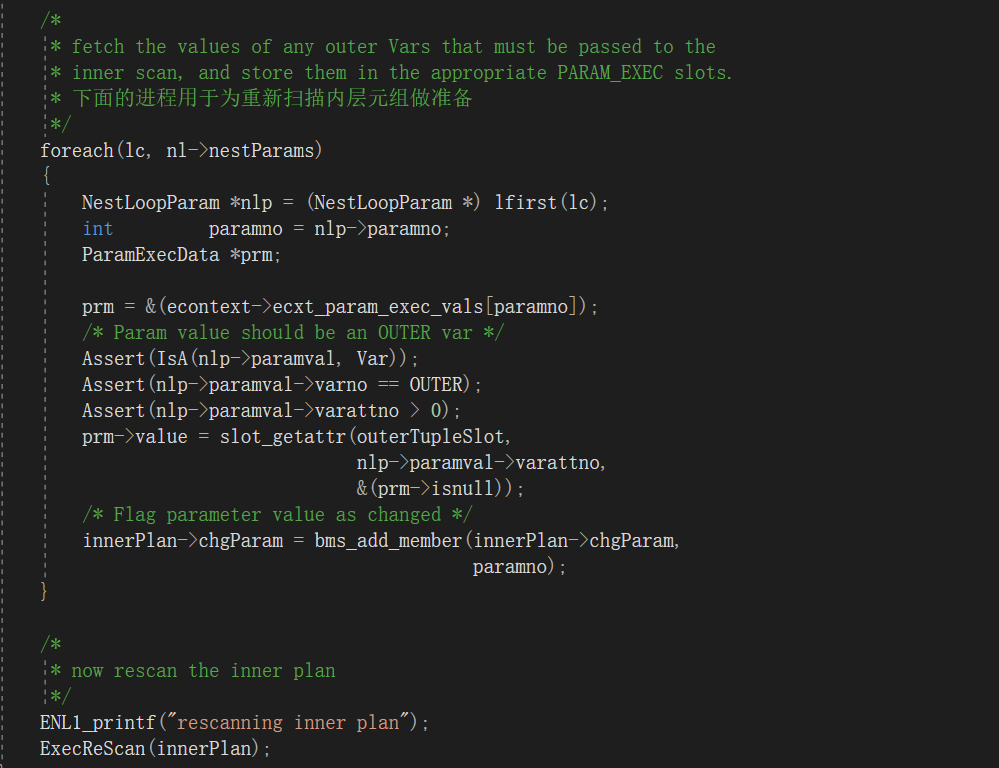




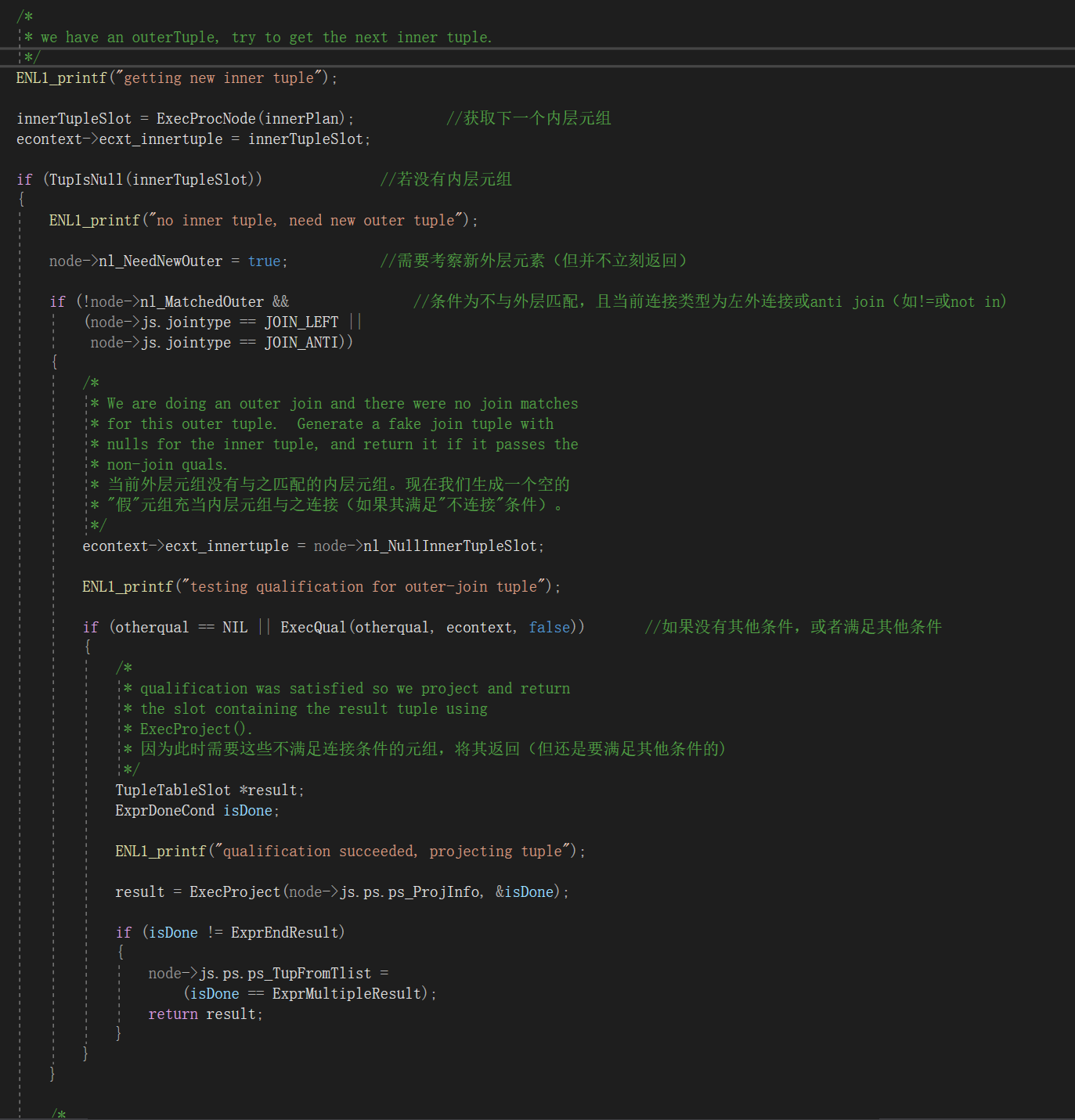
前面都是进行准备工作。



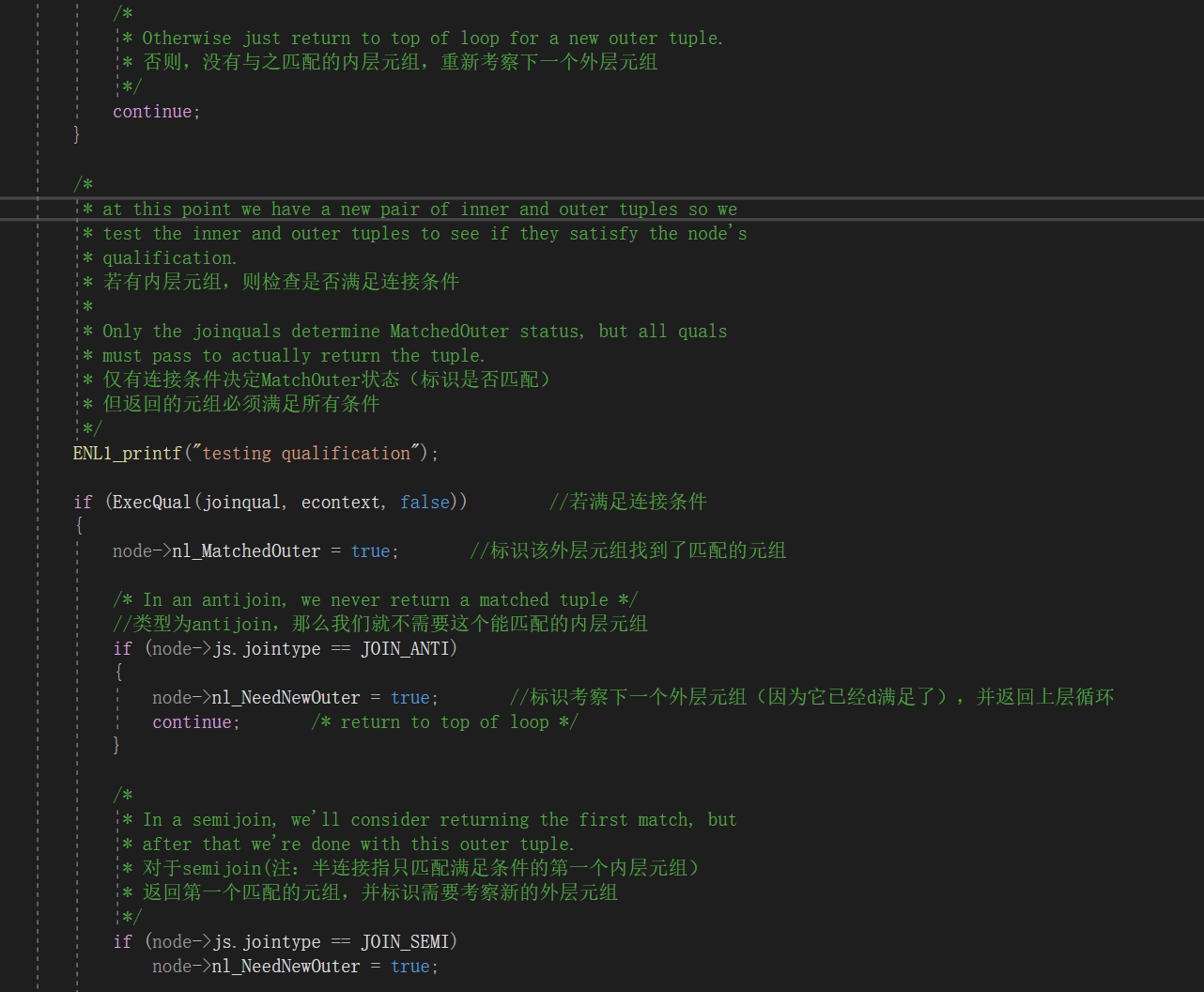
进入主循环。检查是否需要外层元组，如果是就取新的外层元组（这里用了ExecProNode函数，从执行计划得到相应的元组）。若发现取到的外层元组为空（说明取完了），则整个连接已经结束，结束循环；否则，将取到的外层元组配置到context的outertuple，并设置相关bool变量。



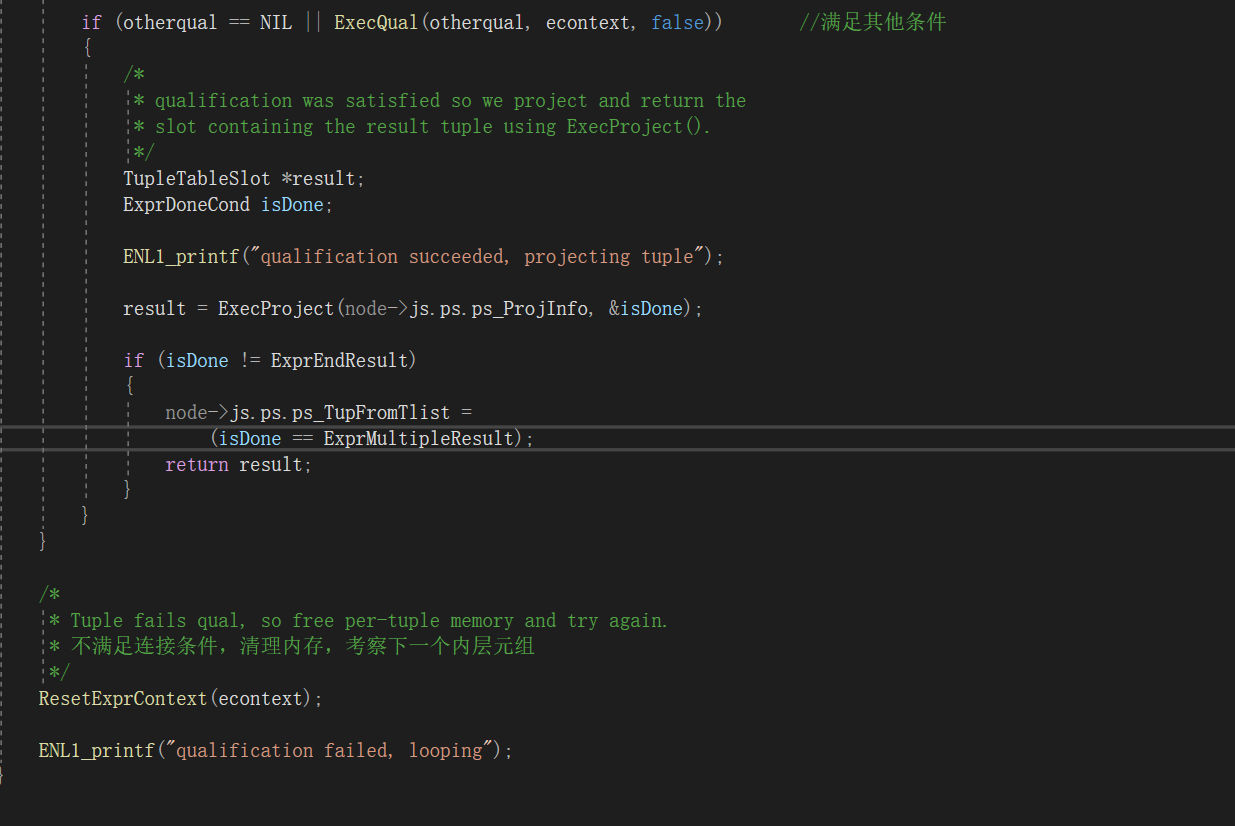
拿了新外层元组时需要重新扫描内层元组，所以这里先做准备工作，再调用ExecReScan=函数重新扫描内层元组。



尝试取下一个内层元组，同样是通过ExecProNode函数获取，并配置context的innertuple。若没有，则说明已考察完所有的内层元组，需要重新取外层元组。但注意不要立刻回到上层循环，在标记需要新外层元组后，检查该外层元组是否已经找到匹配（考察了所有内层元组）及连接类型是否为左外连接或Anti Join（类似not in或不等于一类），如果是则说明所有内层元组都不与这个外层元组匹配。此时就算不匹配也应保留相应的外层元组信息，相应地，内层元组就作为一个空元组与该外层元组连接，再考察其他条件，如果满足就进行连接。这里连接使用了ExecProject函数，执行指定的连接操作。



取到外层元组和内层元组后，调用ExecQual函数(src\backend\executor\execQual.c)检查该context的资格。如果满足就标记当前外层元组匹配，并对anti join和semi join特殊处理：如果是anti join，则不需要这个匹配的内层元组，应该立即去考察下一个外层元组（因为该外层元组已经满足了一个连接，所以不应出现在结果）；如果是semi join，则对每个外层元组，返回第一个与之匹配的内层元组匹配，从而标记需要新外层元组，在完成本次循环后就立刻取新外层元组重新考察。



如果检查也满足其他条件，则进行连接，否则清空context，重新循环。

四、块嵌套循环算法的具体实现

（一）相关定义修改与补充

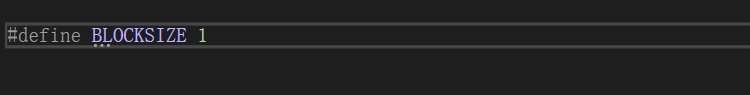
ExecNestedloop所实现的循环嵌套算法的核心伪代码如下，可以看到就是一个简单的二重循环。

现在考虑块循环嵌套算法的核心伪代码：

以上内容可以参考教材12.5.1和12.5.2。块循环嵌套的算法与教材上的有些不一样，只对外层进行了块循环。可以发现，两个算法的一个重要区别在于，算法1是先对i考察再对每个j考察，而算法2正好相反，这点在后面块循环嵌套算法的实现十分重要。

现在我们考虑：如何把ExecNestedloop函数改为执行块循环嵌套？

首先当然需要设定块大小，这由我们自己设定，可以用#define写在.c文件的开头。比如，设块大小为1。



在原本的循环嵌套中，主要使用NestLoopState变量的相关成员来提供需要的信息，如bool变量nl\_NeedNewOuter标识是否需要新外层元组，nl\_MatchedOuter标识当前考察的内层元组是否与外层元组匹配。现在，我们需要修改NestLoopState类型的定义以提供我们所需要的支持：

①nl\_NeedNewOuter是用于表示是否需要新外层元组的，而现在由于循环层次的改变，现在取inner元组在相对于取outer元组的外层，我们需要bool变量来表示是否需要新内层元组，即把nl\_NeedNewOuter改为nl\_NeedNewInner。

②同样，考虑循环次序，取outer块在相对于取inner元组的外层，所以需要bool变量表示是否需要新块，即nl\_NeedNewBlock。

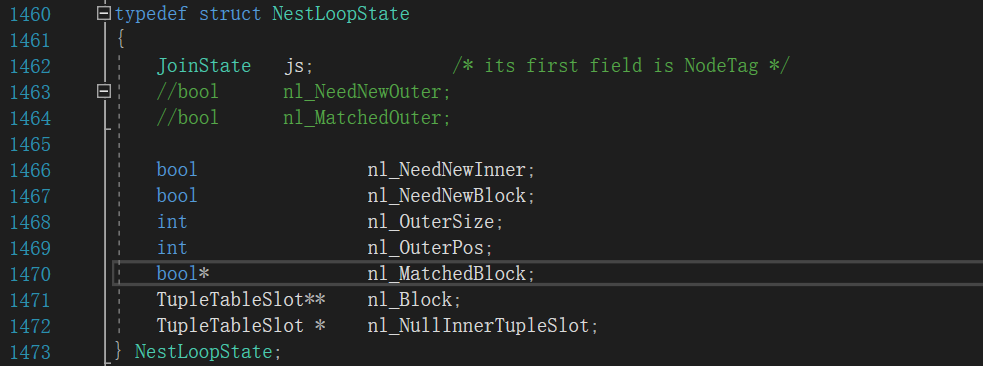
③我们需要一个变量来保存实际取的外层块。因为块是元组的集合，我们用元组的数组表示，而单个元组在PostgreSQL中的表示是TupleTableSlot\*类型，所以nl\_Block的类型应为TupleTableSlot\*\*。

④需要变量记录nl\_Block的元素个数，即该块的外层元组个数，记为int类型变量nl\_OuterSize。

⑤需要变量记录当前处理到哪个外层元组，记为int类型变量nl\_OuterPos。

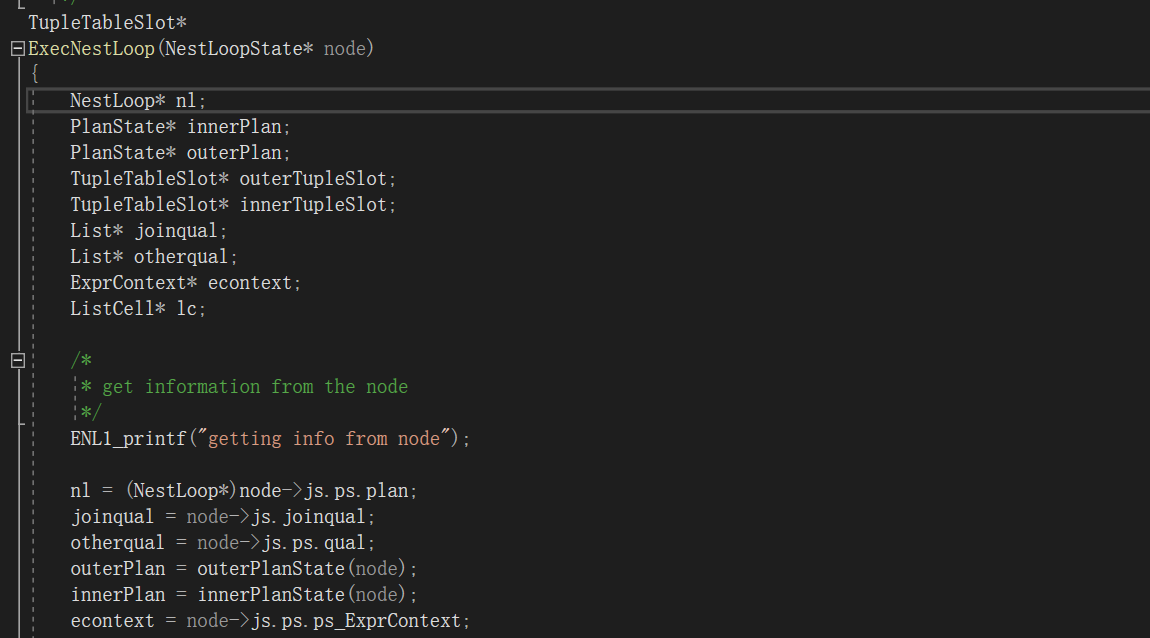
⑥原本的MatchedOuter是标记是否存在一个内层元组与该外层元组匹配的，现在我们将这个变量扩展为一个数组，用来标记一个块的外层元组的匹配情况。它应是bool类型的数组，即bool\*类型，记为nl\_MatchedBlock。

以下是对execnodes.h所做的修改：

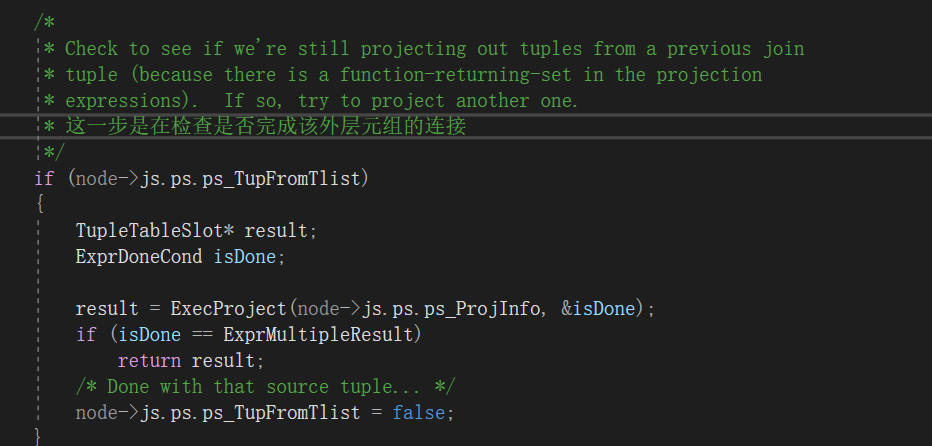


（二）对ExecNestedloop函数的具体修改

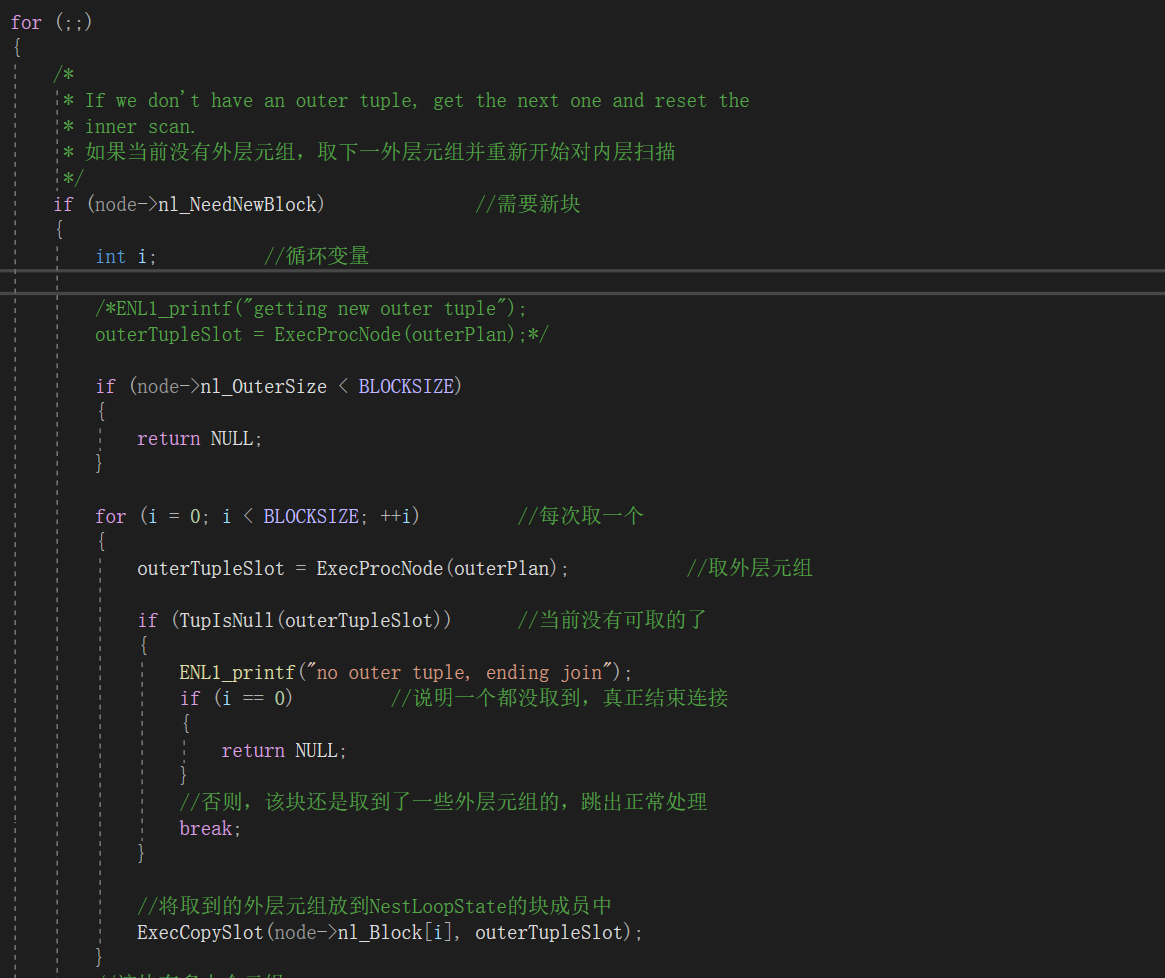
现在我们尝试对ExecNestedloop函数进行更改。



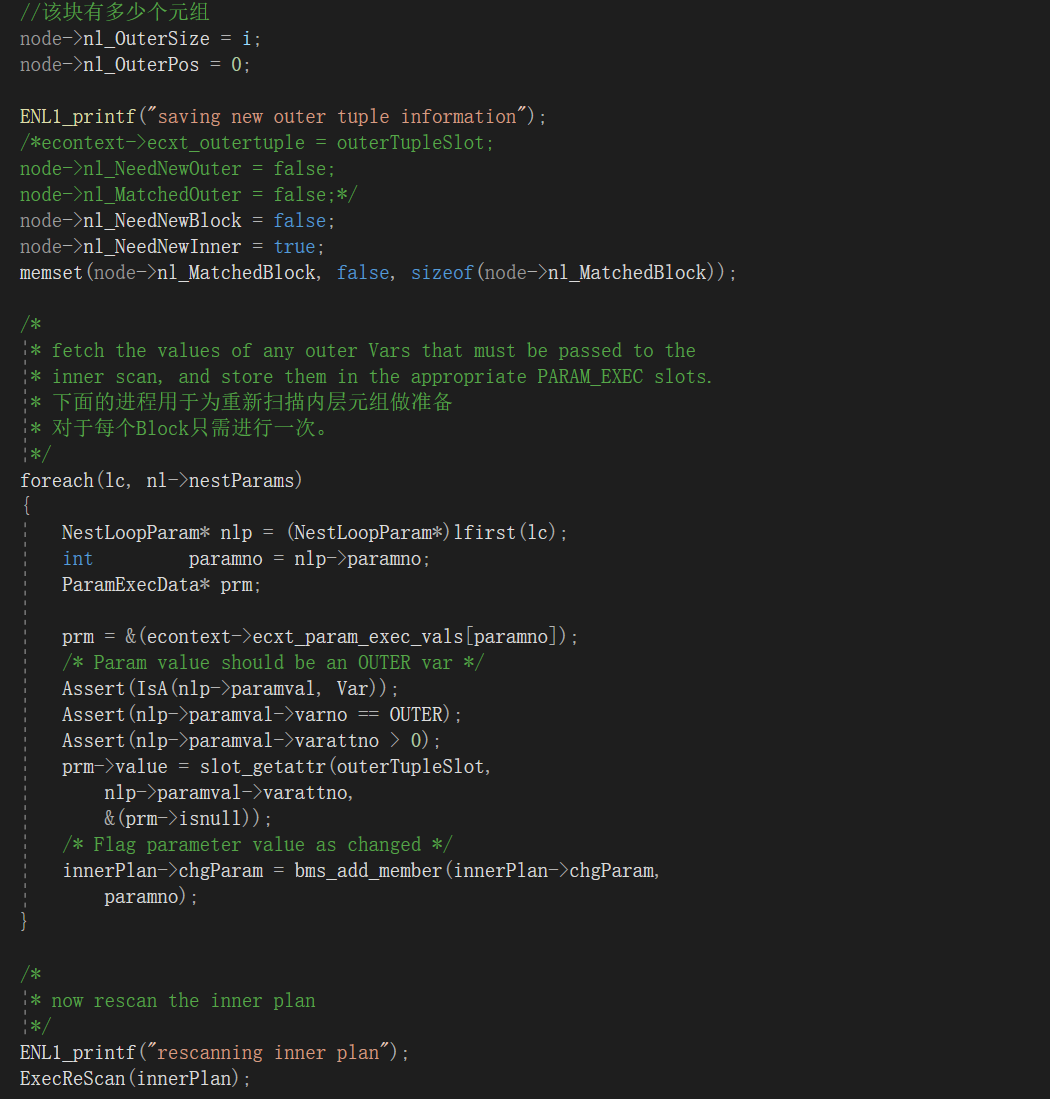
前面这些获取信息的部分没有需要更改的地方，保持原样。



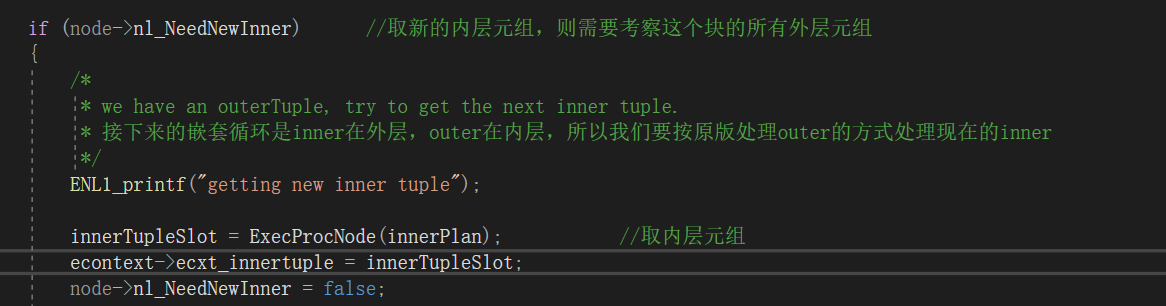
检测部分，不做更改。



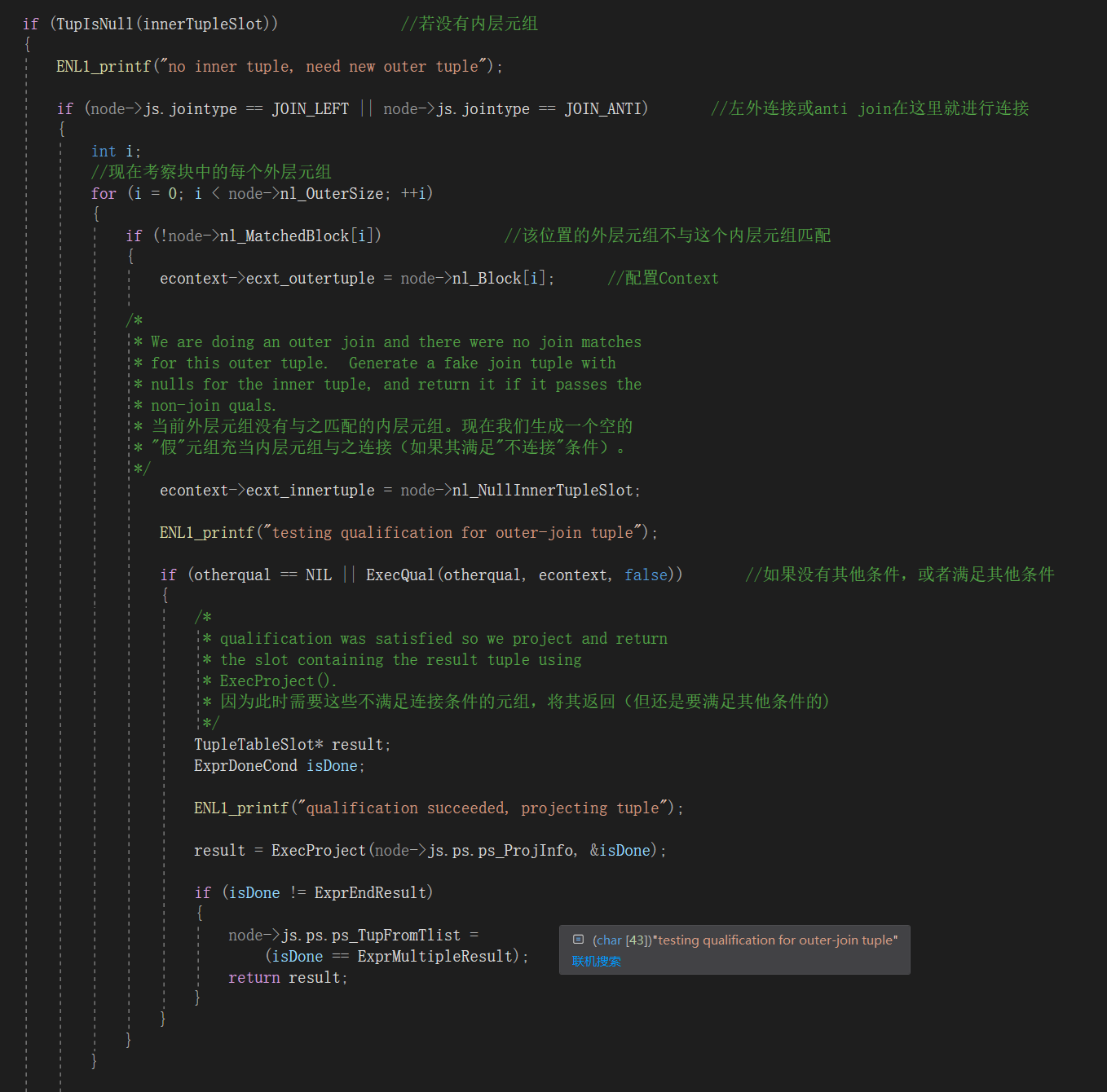
接下来进入主循环(即for(;;))。node->nl\_NeedNewBlock标识了是否需要新块，如果为true，则按BLOCKSIZE取外层元组到块中。这里如果中间检测到外层元组为空，说明已经取完了所有外层元组，如果i为0，说明该块一个外层元组都没取到，就直接结束，否则说明这个块取到了少于BLOCKSIZE的一些元组，按照正常流程继续执行。最后将取到的元组放入node的块node->nl\_Block中。这里还有一个检查是否OuterSize<BLOCKSIZE的，如果成立说明上次取的元组数都不够一个块，那么这次肯定已经取完了所有外层元组，结束循环。



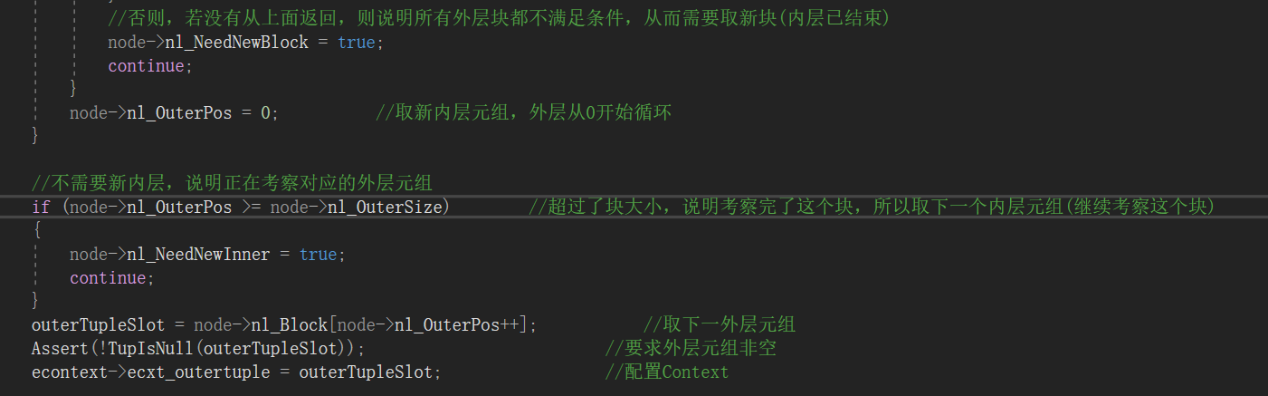
现在设置相应的变量。对于原来的ReScan进程，现在我们只需要对每个Block执行一次。



现在到了取内层元组的阶段。注意一点在于：块循环嵌套中，实际是inner在外，而outer在内，相当于调换了二者原来的次序。



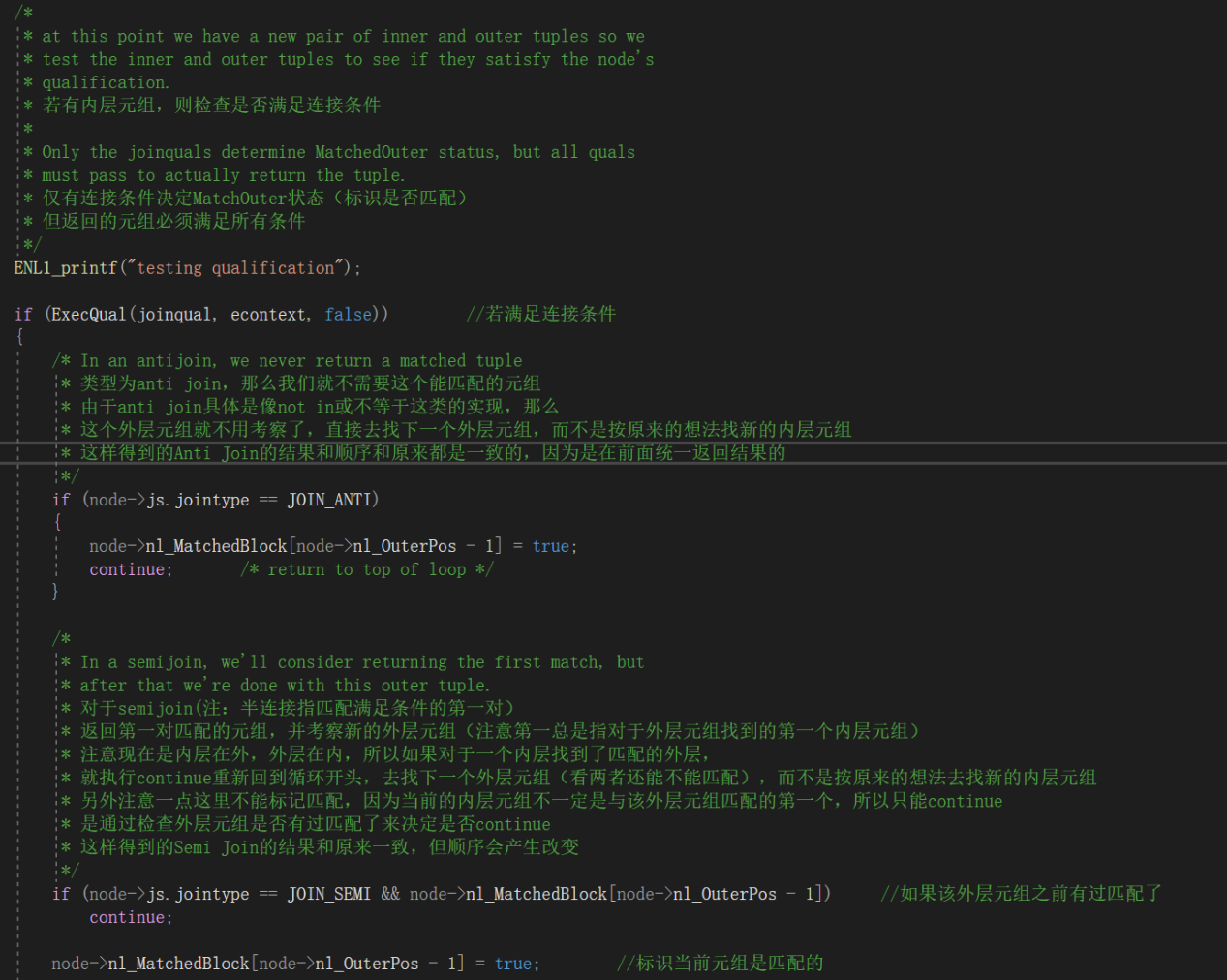
若检测到内层元组为空，则说明对于这个块，我们已经检查完了所有的内层元组。换言之，对于所有的内层元组，都已经考察过了该块的所有外层元组。同样是考虑Left Join和Anti Join这两类特殊情况。现在我们考察该块的每个外层元组的nl\_MatchedBlock，如果为false说明每个内层元组都不与之匹配，从而满足这两类特殊连接的条件。之后也是一样检查特殊条件。



如果没有从上面返回，说明该块的所有外层元组都不能满足Left Join或Anti Join的条件，则需要一个新块（因为此时内层已经考察完为空元组了）。

如果没有触发内层为空，则正常取到了一个内层元组，所以外层重新开始循环。

如果没有进入上面的if，即不需要新内层元组，则正常考察下一个外层元组，如果考察完了(Pos>=Size)就标记需要新内层元组，继续考察该块的外层元组（和块循环算法保持一致），否则就正常取下一外层元组。



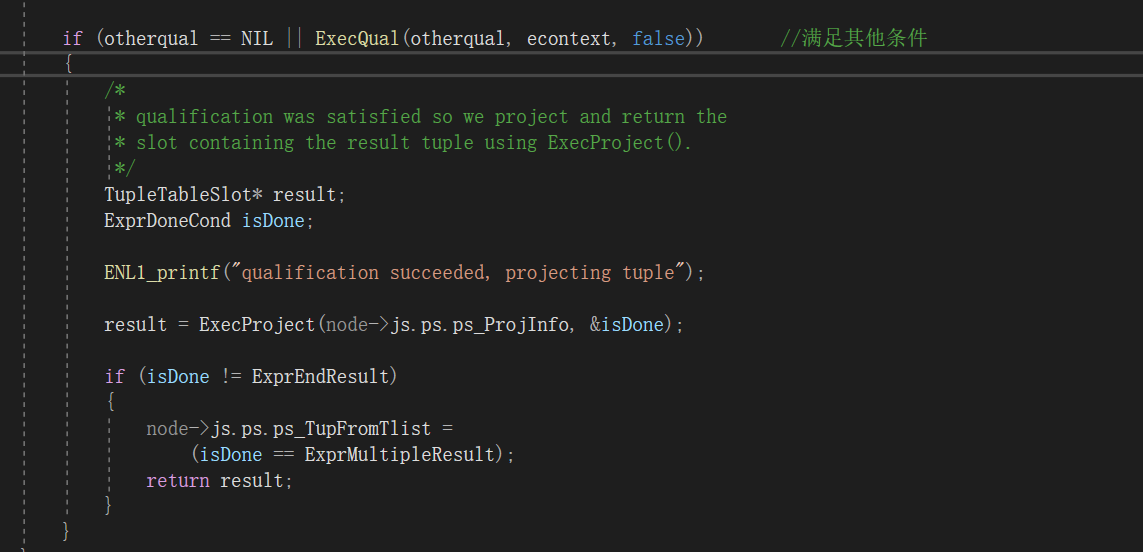
现在已经取到了一对外层和内层的元组，需要检测它们是否匹配。若满足连接条件，分为以下三类情况：

①Anti Join：则这对元组不应加入到结果，直接返回，但需要标记当前元组是存在匹配的。

②Semi Join：检查是否有过匹配，执行continue，继续执行循环。（因为已经有过匹配了，所以不用再考虑该外层元组了）

③其他情况：标记该外层元组存在匹配。

需要注意这里数组的下标是Pos-1，因为之前取外层元组时Pos已经+1了，所以实际上我们考察的那个外层元组的位置是现在的Pos再减1。



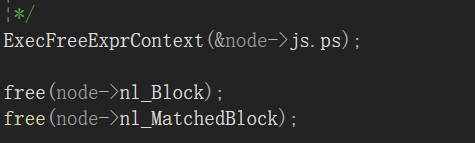
如果也满足其他条件，就照常返回。

对初始化函数ExecInitNestLoop的改动：



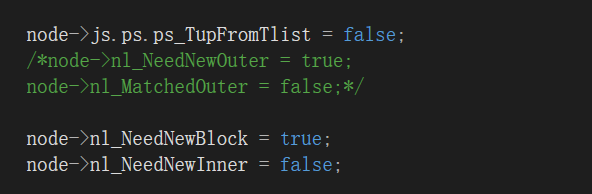
需要的初始变量的设定。

对ExecEndNestLoop的改动：



添加对node的这两个数组的释放。

对ExecReScanNestLoop的改动：



对于每个块只应扫描一次，所以NeedNewBlock为true，NeedNewInner=为false。

五、实验测试

将BLOCKSIZE分别设为1，2，8，64，128，1024，测试执行指定连接查询所用时间和结果，并与原版的普通嵌套循环查询的执行结果和性能进行对比。

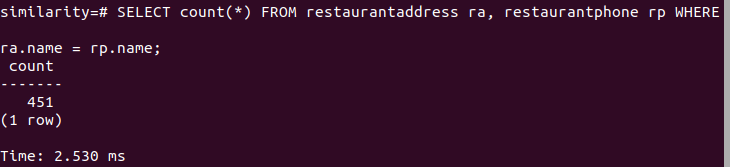
（一）原版查询

打开\timing后，执行指定查询

SELECT count(\*) FROM restaurantaddress ra, restaurantphone rp WHERE

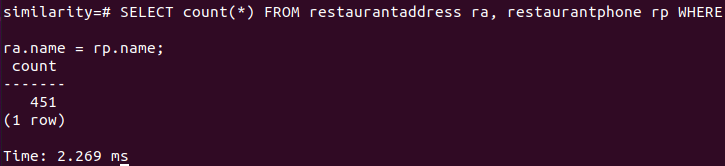
ra.name = rp.name;

两次，并记录第2次的用时：

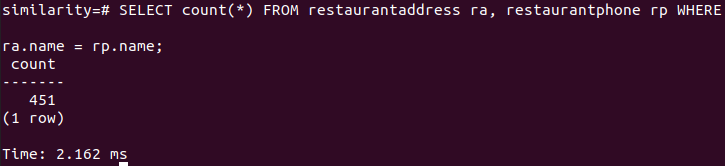


（二）使用块嵌套查询

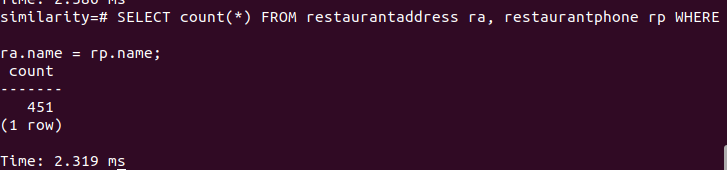
①BLOCKSIZE=1



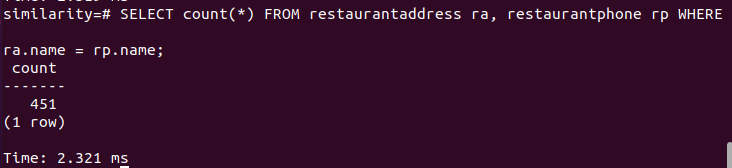
②BLOCKSIZE=2



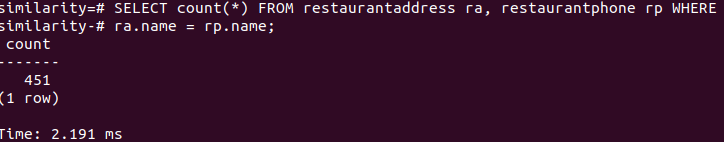
③BLOCKSIZE=8



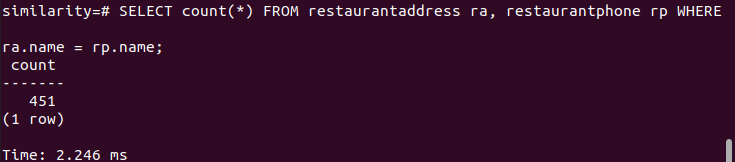
④BLOCKSIZE=64



⑤BLOCKSIZE=128



⑥BLOCKSIZE=1024



（三）分析

经查询，两个关系的元组个数分别为2439和2463。

可以看到，无论是使用普通的循环嵌套连接，还是块循环嵌套连接，得到的结果都一致，且用时都非常短，非常相近，基本看不出什么差别。可能是由于本身连接的元组数太少（只有几百个），所以体现不出差别。

五、实验感想与体会

因为时间比较仓促，实验的Part2进行的并不够深入，仅是主要参考GitHub上代码进行复现和理解。实验得到的结果也显得不太令人满意，我甚至不太清楚改动是否生效了（但编译时确实有相应报错并进行过修改后再成功编译、安装、执行了）。只能说是进行一个简单的尝试吧。

六、参考资料

1.<https://zhuanlan.zhihu.com/p/456245221>

2.<https://github.com/cosmic-jc/pg_block_join/tree/master/src>