PostgreSQL上的Similarity Join实现

实验报告

**16307130194 陈中钰**

**16307130215 刘晓黎**

16级 计算机科学技术学院

**Contents**

[1 分工 2](#_Toc518769357)

[2 基本状况 2](#_Toc518769358)

[3 算法原理及优化 3](#_Toc518769359)

[4 CREATE FUNCTION实现 5](#_Toc518769360)

[5 修改内核实现 11](#_Toc518769361)

[6 实验感想 17](#_Toc518769362)

# 分工

* CREATE FUNCTION实现：陈中钰；
* 修改内核实现：刘晓黎；
* 算法实现：一起讨论算法原理和优化，各自独立实现；
* 实验报告：一起写；

# 基本状况

* 1. PostgreSQL
* 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，而在这次实验中通过自主添加函数来实现Similarity Join。
  1. Similarity Join

也就是相似性连接。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，这时候需要允许实现连接的两个字符串存在一定的差异，那么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。
  1. 实现方式

一开始用的是CREATE FUNCTION的方式，虽然实现方式简单，但是这种方式的结果普遍较慢，后采用内核修改的方式实现。

* CREATE FUNCTION：遵循手册里的要求，书写C语言函数，并动态导入到数据库中；
* 内核修改：阅读源码，在源码上进行修改。
  1. 准备
* 安装：

./configure

make

（在w14\_postgresql-10.4文件夹的根目录处运行）

su

make install

adduser postgres

mkdir /usr/local/pgsql/data

chown postgres /usr/local/pgsql/data

su - postgres

/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data

/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &

* 重装：在实际操作当中，由于操作不当会使数据库产生异常，需要重装数据库。首先要清除make的结果，再删除data文件夹，再从make开始重复上述安装的步骤（不需要configure）。

make clean

rm -rf /usr/local/pgsql/data

（如果执行失败，可以运行sudo rm -rf /usr/local/pgsql/data）

make

……

* 导入数据：

/usr/local/pgsql/bin/psql –f

/home/zhongyuchen/Desktop/psql/w14\_pj2\_similarity\_data.sql

* 进入PostgreSQL：

/usr/local/pgsql/bin/psql

* 开启计时功能：

\timing

（在PostgreSQL中开启）

* 退出PostgreSQL：

Ctrl+z或者\q

# 算法原理及优化

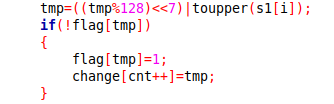
* 1. jaccard\_index

一开始，我们的想法很简单粗暴，为了去除字符串中重复的基本单元（由两个连续字符组成），只需要每次都将它与之前产生的所有字符串比较即可，比较两个字符串同理。这种实现方式的复杂度为O（n2+m2+m\*n），显然非常不合理，因此很快被我们弃掉。

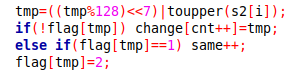
经过讨论，我们得到了一种O（m+n）的算法。以下为实现原理。

1. 将两个字符的ASCII码值映射到一个数组（设为flag），此映射为双射，（若映射到二维数组，则分别映射到其中一维，若映射到一维数组，则映射到两个ASCII码值拼接到一起的值）则该数组能够唯一每一个基本单元的出现情况。

2. 对A串处理，若已出现则跳过，反之令该位置的flag置1，总数加1，并记录该位置。



3. 对B串处理，若只在A串出现过（flag==1），则相同的数量加一；若未出现过，总数加1，则记录该位置。最后将该位置的flag置为2，保证当flag==1时，其只在A串出现过。（当flag==2时，说明已在B串出现过，不能重复统计）



以上就是该算法的实现原理，其中记录出现过的基本单元的目的是为了清0.

* 1. levenshtein\_distance

此函数的实现原理为动态规划，复杂度O（m\*n）。下面是具体原理。

1. 未优化版

可以证明，对于任意一种从A串经过三种操作更改为B串的操作序列（要求这一操作序列没有冗余，即每一步都是必须的，不可去掉），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[1--i]改变为B[0--j]，其最后一步只有三种情况：

* 替换

这说明A[1—i-1]已转换为B[1—j-1]，但A[i]!=B[j]，则操作数此基础上加1。

* 增加

这说明A[1—i]已转换为B[1—j-1]，需要增加B[j]，则操作数此基础上加1。

* 删除

这说明A[1—i-1]已转换为B[1—j]，需要删除A[i]，则操作数此基础上加1。

为了求最少的操作数，在三者中取最小值即可。（可以证明，如果A[i]==B[j]，则将A[1—i]转换为B[1—j]的最少操作数等于将A[1—i-1]转换为B[1—j-1]的操作数）

C:\Users\hs\AppData\Local\Temp\1530966049(1).png

2. 去掉相同前缀及后缀

这是一种比较简单的优化原理，实现方式也较为简单。

3. 利用上一次的编辑距离矩阵

这种动态规划的算法的瓶颈在于每次都需要计算一个m\*n的矩阵，有没有可能少计算一些东西呢？由于我们对两张表的连接需要做大量的编辑距离连接运算，也就是每个元祖对都要产生一个编辑矩阵，如果我们记住让一次的编辑矩阵，那么在计算下一个编辑矩阵时，我们就可以不用重复计算由相同前缀构成的编辑矩阵。这就是这一优化的产生思路。

具体的实现方式也极为简单，为了计算大矩阵，只需要更改边界条件，转为计算3个小矩阵即可。

实际上，如果说在做连接运算时，字符串是有序的，那么在进行连接时，相同的前缀长度将以递增方式出现，这将极大地减少该算法的运行时间。

4. 错误的优化

在这两种优化产生之前，我产生过一种奇怪的想法，那就是A和B串相同位置的字符如果相同，我就去掉它。这一想法的正确性从表面看起来没什么问题，（但也不一定对）因此我也试着尝试了一下，最后发现结果出现了误差。

经过思考，这种思路最终被我们证明为是错误的。原因是即使相同位置字符相同，也不一定将它们匹配，可能通过“错位”使得操作数更少。一个反例如下：cdaba->aba。很明显最少的操作数为2，但如果去掉相同位置的“a”，则操作数变为3.

* 1. 额外的函数levenshtein\_distance\_2

实际上，对于本次的查询语句，有一个重要信息，也就是编辑距离的限制（<d）没有被我们用到。通过查阅资料，我们最终找到了一个能够大幅度降低复杂度的算法（复杂度为O(min(n,m)\*d）。这一算法来自Ukkonen算法及其改进，它并不需要完整地算出levenshtein\_distance动态规划算法中的整个编辑矩阵。。其原理来自<http://www.berghel.net/publications/asm/asm.pdf>以及

<http://www.cs.helsinki.fi/u/ukkonen/InfCont85.pdf>两个网站。

由于该函数的实现并不满足这次pj的要求，加上时间限制，因此没能完整地看懂这一算法的实现原理，但从结果来看，它的正确性和高效性都得到了很好的保证。

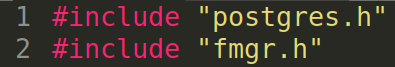
# CREATE FUNCTION实现

* 1. 实现流程
  2. 书写/修改代码

通过CREATE FUNCTION来实现函数的定义，只需要把函数写到一个文件中就可以了。而在这次实验中，把函数写到了similarity\_join.c文件中。书写代码的时候，要注意的是和C语言中不一样的部分，故只展示这些内容。

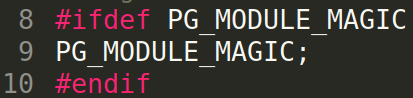
* 头文件

为了使用PostgreSQL的一些独特的内容，如psql的数据类型，则需要添加这两个头文件。所以这两个头文件是必须添加的。



* magic lock

为了保证动态加载的object file不会被加载到一个不相容的服务器中，PostgreSQL会检查magic lock的宏定义。如果有该宏定义存在，可以使服务器能够检查出明显的不相容性。



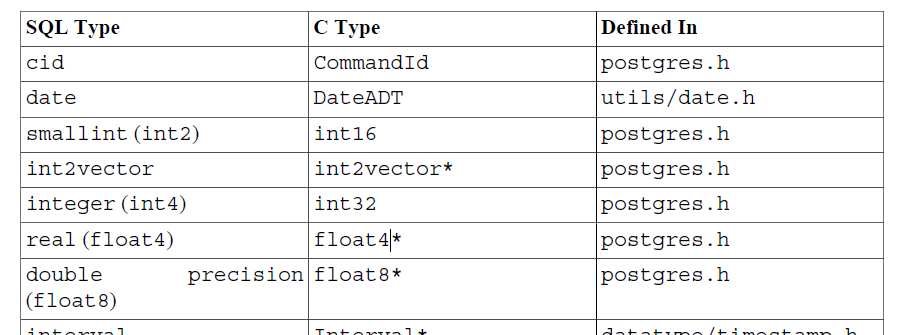
* 函数声明

psql有自己的函数声明方式，而且还必须要进行函数声明

C:\Users\ECHOES\AppData\Local\Temp\1530964070(1).png

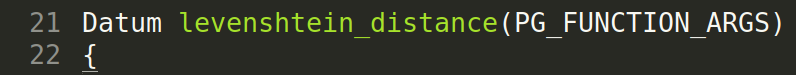
* 数据类型对应关系（部分）

在函数定义中，在引用函数参数、函数返回值时，不能直接使用C语言中的类型，而是要使用对应的SQL Type。对应查找手册中的表格即可。

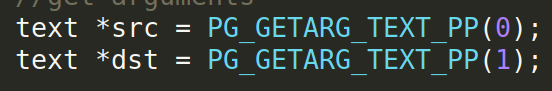


* 函数定义

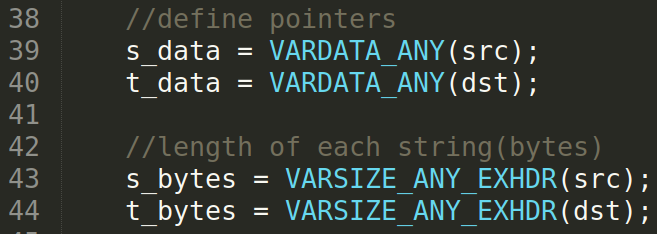
psql中的函数定义不需要写出函数返回值类型，只需统一用Datum。而函数参数用PG\_FUNCTINO\_ARGS表示，其中含有函数的全部参数。



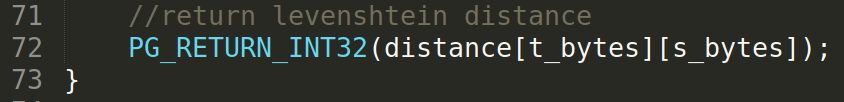
在取调用的参数时，要用PG\_GETARG来获取，还需要加上对应的数据类型和指示第几个参数的数字。如下图，TEXT指示了参数的类型。在这次实验中，函数调用的参数都是字符串，而根据手册中的对应关系，对应到psql中为text类型。而0表示这是第0个参数。



而对于text的数据类型，还可以通过函数获得text字符串的长度、指向text字符串的char \*指针。注意定义该指向数据库中的数据的指针时，必须要加上const的定义，来防止在操作过程当中不小心篡改了数据库中的数据。



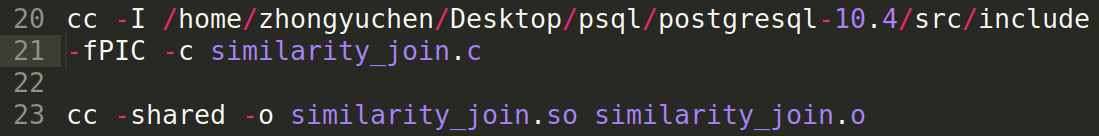
在返回时，也要根据返回值的具体类型来返回。在返回的是普通的int类型，再通过查手册，发现对应的类型是INT32，再使用INT32类型来返回。



* 动态空间申请与释放

在psql中申请同台空间需要用palloc函数，而释放空间则需要用pfree。它们的用法和C语言中对应的malloc和free的用法是一样的。

* 对应的算法已经在上文进行了叙述，再这里就不再重复了。
  1. 编译



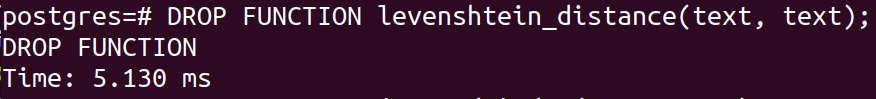
* 编译过程：书写的程序是在.c文件中，那么首先要由.c文件生成对应的共享库.so文件，再.so文件由生成对应的.o文件；
* 编译需要在.c文件所在的文件夹进行；
* 查找手册，发现在Linux上的编译采用的是cc；
* .c文件 -> .so文件

查找手册可得生成PIC的compiler flag是 –fPIC。但如果按照手册中的格式直接运行，会报错，因为编译器并不知道include的postgres库的对应文件在哪里。那么就要加入-I的标志，并在后面加入psql的include文件夹的路径，然后就能生成对应的.so文件了。

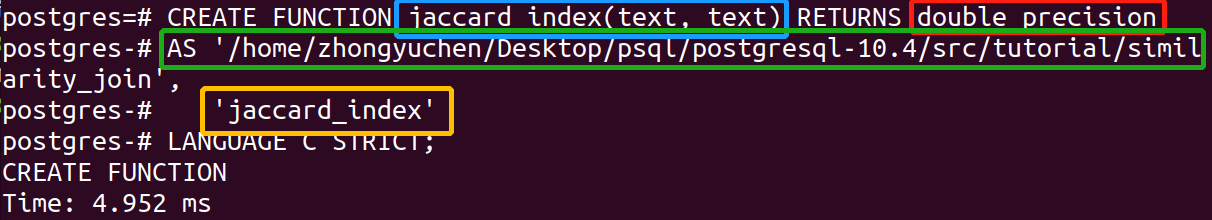
* .so文件 -> .o文件

生成共享库的compiler flag是shared，直接运行就能生成对应的.o文件。

* 1. DROP FUNCTION
* 如果是修改过了代码，那么要重新导入函数。而在重新导入函数前，需要把原来导入数据库的函数卸掉，则需要用DROP FUNCTION指令（在psql中运行）。

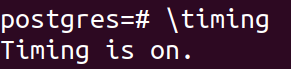


* 经过测试，在重新编译过.c文件后、在CREATE FUNCTION之前，为了保证导入的是新的函数，不仅要DROP FUNCTION，还必须要退出psql，再重新进入psql。否则导入的函数还依然会是原先的函数。
  1. CREATE FUNCTION

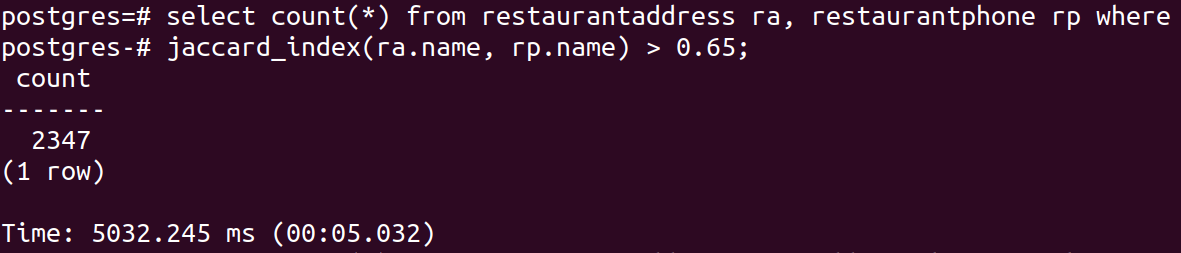


以下是CREATE FUNCTION的格式：

* CREATE FUNCTION篮框中的是在psql中使用该函数时要输入的名字；
* RETURN红框中的是返回值的SQL Type，可以通过查表获得；
* AS绿框中的是含有函数的.c文件的路径，而黄框中的是.c文件中函数的名字。
* LANGUAGE C STRICT指定了函数是strict的，也就是系统会自动判断，在调用参数为null时，返回null值，而不需要在代码中显式地进行判断。否则要用PG\_ARGISNULL()来判断调用参数是否为null，并对应返回null值。
  1. 测试
* 在正式测试之前，需要打开计时功能，之后每执行一条一条sql指令都会输出对应的运行时间；



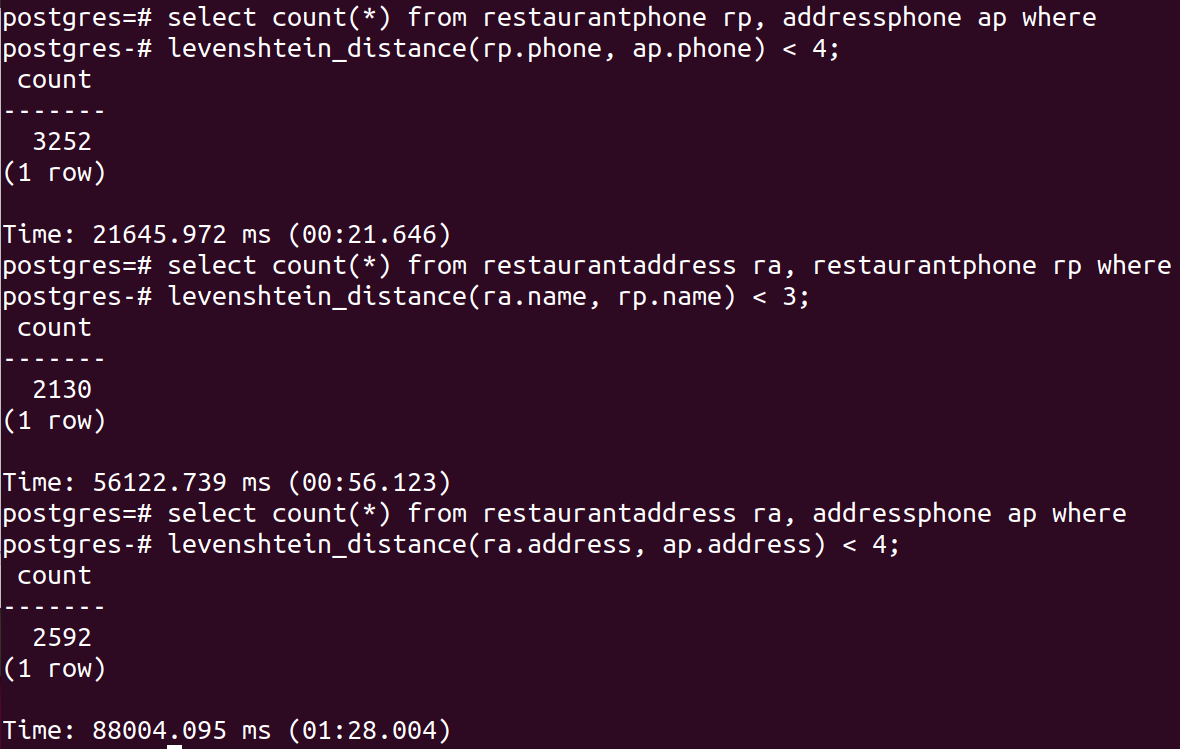
* 正式测试：直接输入对应的sql语句运行，会输出语句的运行结果。

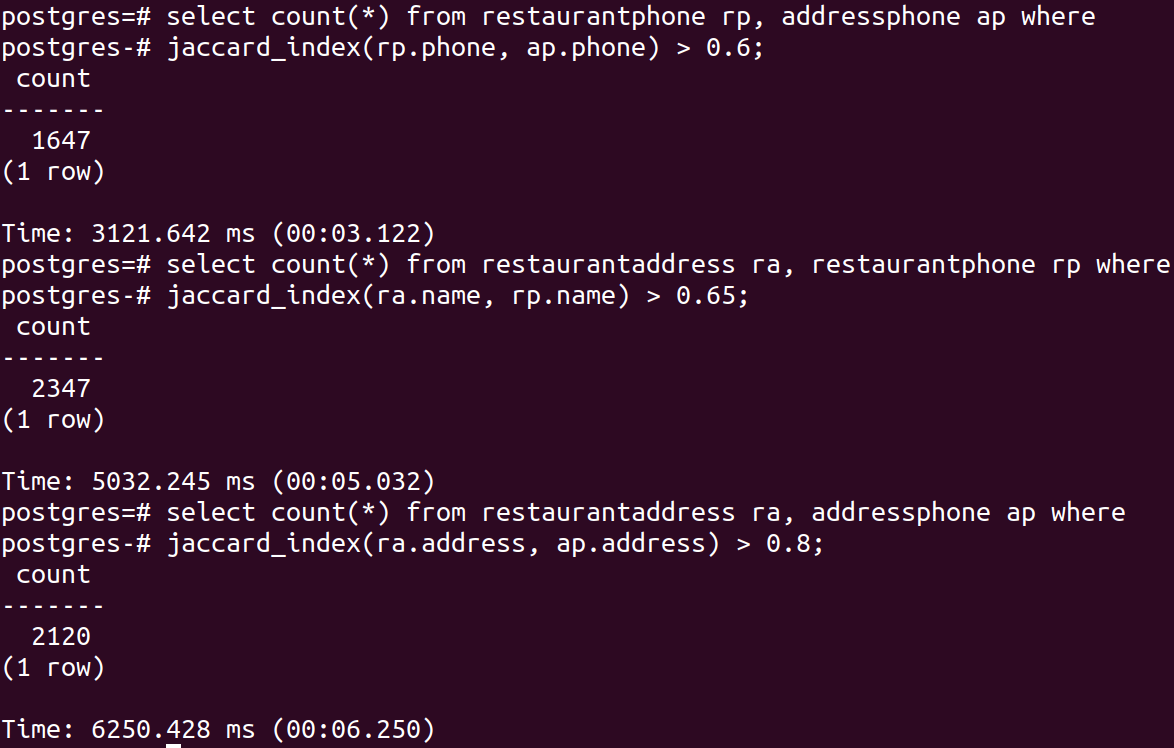


* 1. 调试
* 调试主要通过肉眼调试。通过观察代码、观察输出实例，来找到错误。比如可以输出where levenshtein\_distance(ra.name, rp.name) = 1的例子的对应名字的数据，通过分析异常结果来找到代码的错误；
* 在已有正确正确程序的情况下，还可以通过输出where levenshtein\_distance\_true(ra.name, rp.name) = 1 &

levenshtein\_distance\_false(ra.name, rp.name) != 1的结果，来快速找到levenshtein\_distance\_false函数的错误例子，并对其进行分析，进而找到代码的错误。

* 1. 运行结果





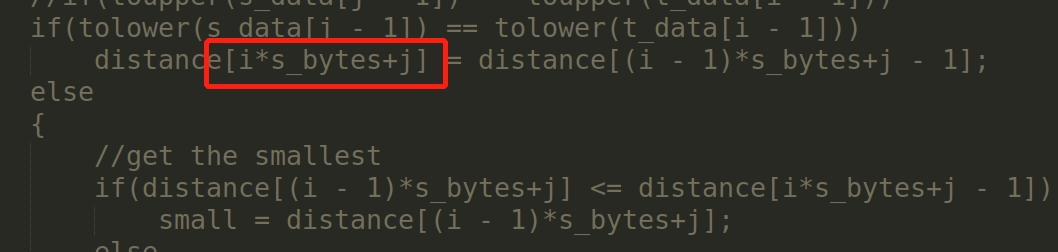
* 1. Levenshtein Distance优化

以下统一用levenshtein\_distance(ra.address, ap.address) < 4的例子进行计时。

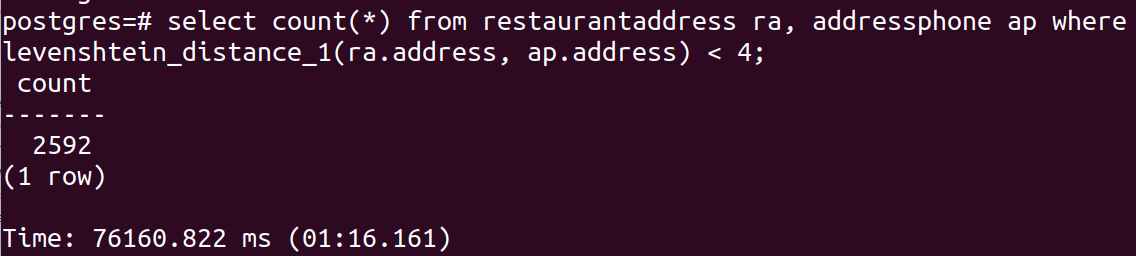
* + 1. 空间生成
* 最开始的时候，数组都是用palloc来动态生成的，但是和malloc类似，动态生成空间耗时很大；
* 于是改成用C语言定义数组的方式来定义普通数组，并进而在前面加上static的标识；
* 那么，在每次调用函数时都会保留这同一块空间，能节省掉重复申请空间和释放空间的时间；
* 只是针对select count(\*) from restaurantaddress ra, addressphone ap where

levenshtein\_distance(ra.address, ap.address) < 4;运行的耗时，就能从平均约2:20缩减到平均约1:30，减少了近1分钟，优化的效果很好；

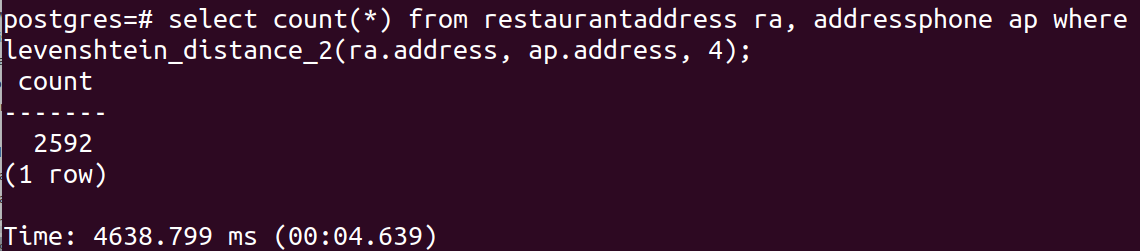
* + 1. 二维数组改为一维数组



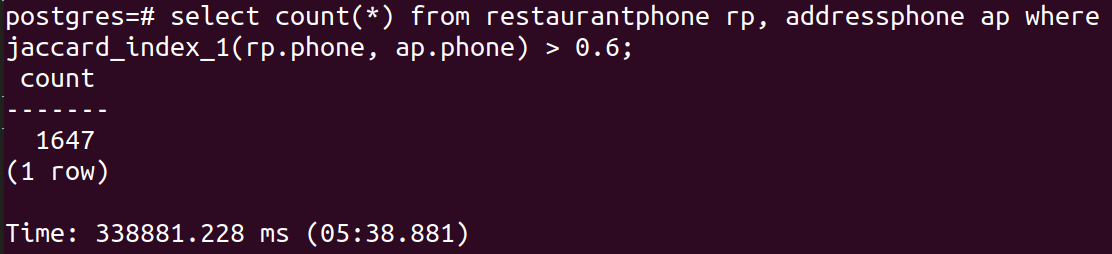
* 函数中使用的数组是二维的，尝试把数组改成相同大小的一维数组，并在使用时访问一维数组对应的位置，查看是否有时间上的优化；
* 结果并没有很大的差异（不再截图了）。
  + 1. 消除两个字符串在两端相同的部分
* 两个字符串把左端两端相同的部分都去掉，可以大大减少递推矩阵递推的大小；
* 但是结果仍然没有很大的差异（不再截图了）。
  + 1. 利用上一次的编辑距离矩阵
* 由于编辑距离矩阵是static的，里面的数据可以保留到下一次函数调用中，那么可以利用上一次的数据，来减少重复的运算；
* 结果能在平均1:30的基础上缩减到平均1:15，效果还是很好的。



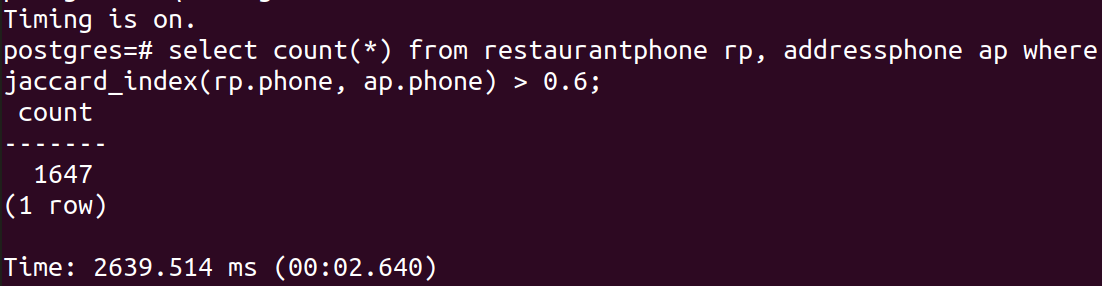
* + 1. 把编辑距离限制作为参数加入到函数中
* 当把编辑距离限制作为参数加入到函数中后，尽管不符合项目的要求，但是的确能有很好的结果；
* 结果能从1:30的基础上缩减到平均0:04，效果很好。
* 注意，在CREATE FUNCTION时，调用参数4的SQL Type是integer，而返回值SQL Type是boolean。



* 1. Jaccard Index优化
     1. 空间优化
* 由于在测试levenshtein distance时，不用palloc来生成空间，而是用static int来生成数组，会有很好的结果，因此在写jaccard index函数时，就直接用了static int的数组定义方式；
* 但是有一个缺点是，由于每次用数组之前到需要使数组为0，那么，在使用该数组前，需要对static数组清空，否则该空间会保留着之前的数据（求levenshtein distance的矩阵是直接覆盖的，因此不需要对矩阵进行清0）；
* 通过遍历整个二维数组的方式来进行清0；
* 结果约为5:40左右。



* + 1. 优化的清0方式
* 如果对数组进行遍历清0，那么相当于复杂度从O(m + n)，上升为O(n \* n)级别，使得耗时很长；
* 而且实际上，数组需要清0的地方并不多；
* 于是用一个数组来记录非0的点，并在计算出jaccard index之后，只对记录数组中记录的点进行清0，使得复杂度回到O(m + n)；
* 使得时间从平均5:40下降到0:02，效果十分的好。



* 1. CREATION FUNCTION的优劣
* 优点：在有C语言的基础下，简单容易操作，修改过函数之后重新导入函数快；
* 缺点：整体耗时长，而且对于一些优化并没有明显的作用。

为了寻求更好、更快的运行结果，于是接下来，尝试通过修改内核来实现Similarity Join。

# 修改内核实现

* 1. 系统与源码理解

实际上，想要完成本次pj，并不需要对源码有深刻的理解。但在使用gdb调试程序的过程中，为了稍微理解每个函数在做什么，我还是查阅了相关资料，知道了一些关于查询语句在源码中的执行方式。以下结合资料谈谈我的查询语句运行机制的理解：

1. 在postgresql的后台服务进程postgres接受到查询语句后，进入PostgresMain函数中的exec\_simple\_query函数；

2. exec\_simple\_query调用pg\_parse\_query函数，pg\_parse\_query再调用raw\_parser函数进行词法和语法分析，产生分析树；

3. exec\_simple\_query调用pg\_analyze\_and\_rewrire函数。pg\_analyze\_and\_rewrire函数函数调用parse\_analyze进行语义分析生成Query结构体（查询树），再将该结构体传递给pg\_rewrite\_query进行查询重写（根据转换规则将原始的查询转换为新的查询树）；

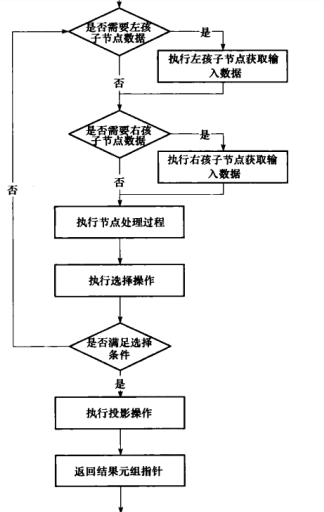
4. 查询重写完成以后，进入pg\_plan\_queries函数进行查询规划。pg\_plan\_queries进入pg\_plan\_query函数，pg\_plan\_query函数进入planner函数，它负责查询计划的生成；

5. planner函数调用standard\_planner函数，standard\_planner函数通过调用subquery\_planner和set\_plan\_references分别完成计划树的生成、优化、清理；

6. 在查询语句转变为执行计划之后，exec\_simple\_query函数调用PortalRun函数进行计划执行；

7. PortalRun函数根据select查询类型进入PortalRunSelect函数;

8. 在初始化查询计划树（调用ExectorStart函数后），PortalRunSelect函数调用ExecutorRun函数，ExecutorRun函数调用standard\_ExecutorRun函数，standard\_ExecutorRun又调用ExecutePlan进行计划执行；

 9. ExecutePlan函数是一个循环，它通过ExecProcNode函数从计划节点中获取一个元组，然后对该元组进行相应的处理。ExecProcNode的执行过程如下图所示：

10. 当所有元组都被获得以后，进行一些清理工作，整个查询语句就结束了。

* 1. 设计思路与实现方案（改源码方式）

在未修改任何源码之前，我尝试运行了这6条查询语句，发现postgresql所报错误为未找到该函数，也就是说，我的主要任务就是将这两个函数添加到源码中，并让postgresql能够识别它即可。接下来的思路分为以下阶段：

（一）、gdb调试阶段

一开始我的想法是找到postgresql第一次调用该函数的位置，通过它间接知道应该在哪个源文件中添加函数。主要的过程如下：

1.在make时添加—enable-debug参数

2.删除O2优化

这一点至关重要，否则几乎无法调试。由于没有找到合适的命令行参数，我直接将makefile.golbal文件中的CFLAGS变量中的-O2改为了-O0。

3.设置断点调试

通过不断的设置断点，我最终找到了postgresql报错的源头。这是因为它在调用func\_get\_detail（）函数时返回了FUNCDETAIL\_NOTFOUND。到这一步后我的思路就进行不下去了，仔细想想，好像无论我怎么深入，它报错的源头应该都只是在源代码中没有出现相应的参数（或许是函数的id），我不可能通过这一点就找到在哪添加函数。

虽然这一阶段以失败告终了，但我因此熟悉了大量gdb调试的方法，也对后面的debug有一定的帮助。

注意：在将O2优化去掉（改为O0）后，如果只是重新执行make是无效的，必须首先执行一次make clean操作，再执行make，否则不会有任何变化。

（二）、奇思妙想阶段

在阅读postgresql的文档时，我发现它在第9章描述了许多内置函数，包括数学函数、字符串函数等等。我突然想到，只要我模仿其中一个函数的源码实现，不就能够实现我想要的函数吗？这一阶段的主要过程如下：

1. 测试内置函数是否能够用于查询语句

我用其中的几个函数尝试运行了几条查询语句，发现都能成功，这让我更加坚定了自己的想法。

2. 选定参照函数

由于我们需要实现的是对字符串处理的函数，因此我选定了btrim函数作为参照对象。其功能为从字符串的开头和结尾删除指定字符。

3. 运用sublime寻找该函数出现的位置

通过查询功能，我知道btrim出现在以下源文件中：

* postgresql-10.4\src\backend\catalog\postgres.bki
* postgresql-10.4\src\backend\parser\gram.c
* postgresql-10.4\src\backend\parser\gram.y
* postgresql-10.4\src\backend\utils\adt\oracle\_compat.c
* postgresql-10.4\src\backend\utils\fmgroids.h
* postgresql-10.4\src\backend\utils\fmgrprotos.h
* postgresql-10.4\src\backend\utils\fmgrtab.c
* postgresql-10.4\src\bin\psql\describe.c
* postgresql-10.4\src\include\catalog\pg\_proc.h

其中gram.c、gram.y、describe.c文件很明显跟函数的实现无关，因此我只需对剩下6个源文件更改即可。（实际上，通过后面的工作，我知道其实我并不需要更改postgres.bki，它是在编译的过程中自己创建的）

4. 修改细节（以jaccard\_index为例）

* fmgrprotos.h

此源文件只是对函数的全局函数外部使用的声明，添加语句（extern Datum jaccard\_index(PG\_FUNCTION\_ARGS);）即可。

* fmgroids.h

此源文件是对内置函数设置oid，通过模仿，添加语句（#define F\_JACCARD\_INDEX 9997）即可.

* fmgrtab.c

此源文件是内置函数的管理表，通过模仿，添加语句（{ 9997, "jaccard\_index", 2, true, false, jaccard\_index },）即可。（其中的数字2的含义是我通过比较其它内置函数知道它表示该函数参数个数的）

* pg\_proc.h

此源文件供catalog/genbki.pl读取，并被转换为postgres.bki，是注册函数的核心所在。通过分析btrim的记录（DATA(insert OID = 884 ( btrim PGNSP PGUID 12 1 0 0 0 f f f f t f i s 2 0 25 "25 25" \_null\_ \_null\_ \_null\_ \_null\_ \_null\_ btrim \_null\_ \_null\_ \_null\_ ));），并将它与其它记录对比，可以知道其中的2表示参数个数，第一个25表示返回参数的类型，而冒号中的两个数字则表示两个参数的类型。同时可以得知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。

综上所述，我在此插入了记录（DATA(insert OID = 9997 ( jaccard\_index PGNSP PGUID 12 1 0 0 0 f f f f t f i s 2 0 700 "25 25" \_null\_ \_null\_ \_null\_ \_null\_ \_null\_ jaccard\_index \_null\_ \_null\_ \_null\_));

DESCR("jaccard\_index");）

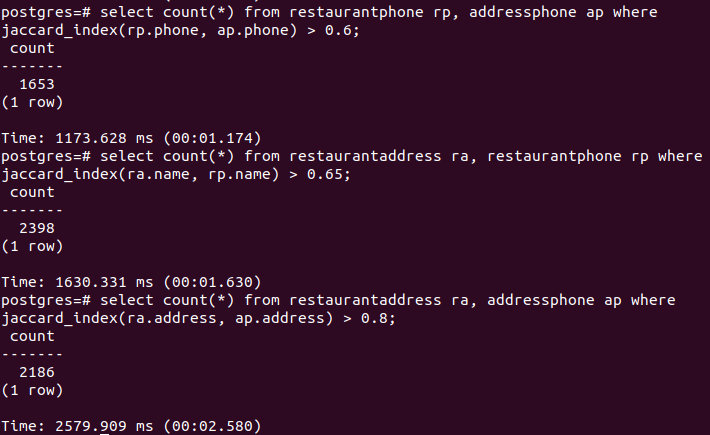
* oracle\_compat.c

这是btrim函数真正定义的地方。为了实现相关功能需要添加一些头文件（upper函数需要<ctype.h>，abs函数需要<math.h>），具体函数的实现这里就不再展开。

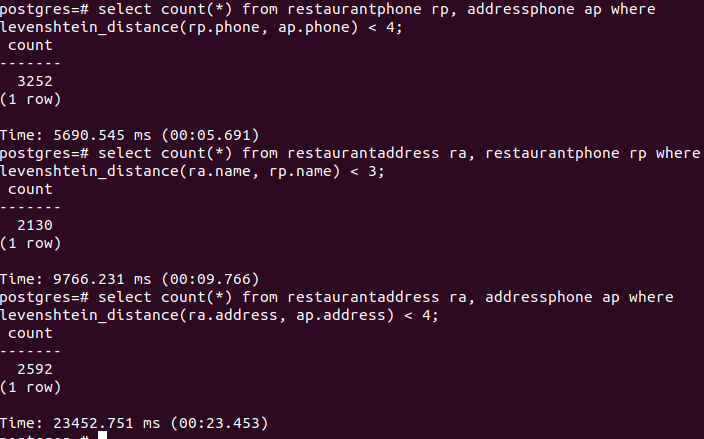
（三）.实验结果展示

相比于create\_function的方式，此方法极大的提高了运行速度。具体结果如下：

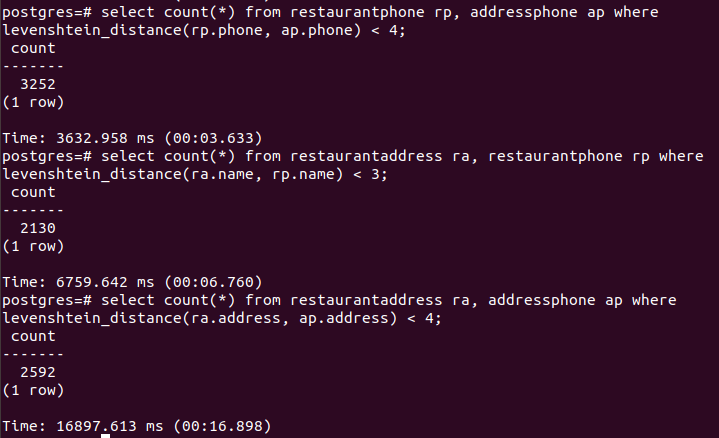
1. jaccad\_index



2. levenshtein\_distance

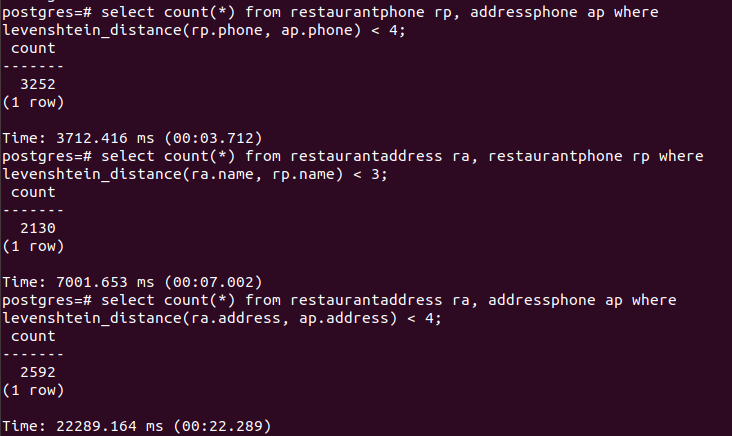


（未优化）



（去掉相同前缀及后缀）

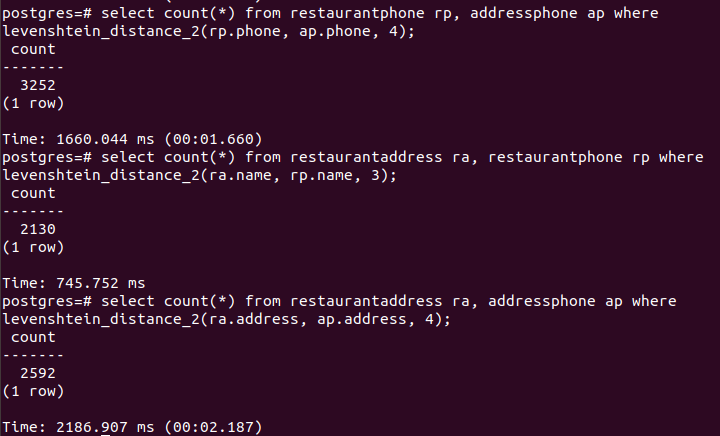
* 这种优化效果在此实现方式下效果较好，提升了30%-40%的速度。源码中保留此版本。



（利用上一次的编辑距离矩阵）

* 这种优化方式在此实现方式下效果也还可以，但稍稍逊色于上一种（即去掉相同的前后缀）的方法。如果字符串的排列是有序的话，相信它表现得应该更好，因为这样的话几乎每一次都能充分的利用上一次的编辑距离矩阵（由于相同的前缀长度递增）。

3. levenshtein\_distance\_2



* 这种利用给出的编辑距离限制的方法表现卓越。

# 实验感想

* 接触到数据库的实际实现，并学会安装、使用数据库，把实际所学应用到实际当中；
* 学会实现额外的数据库功能；
* 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；
* 在日后可以尝试通过修改内核，来实现下半学期重点讲的各类查询优化，不仅能进一步深入学习查询优化各类算法，并且能够应用到实际当中去。