

# PJ2 PostgreSQL上的Similarity Join实现

宁晨然 17307130178 田睿 17300750084

## 一、系统与源码理解

首先需要对postgresql进行系统上的理解，不需要很深入的理解完整的架构，而是通过查询理解postgresql的执行过程。postgresql是一种对象关系数据库服务器(数据库管理系统)，在数据库交互命令行直接输入sql语句，可以实现查询处理。

### (一)、系统理解

1. 服务器连接：需要建立应用程序到postgresql服务器的连接，进入交互界面。
2. 分析器阶段：根据应用程序发送来的查询，核对语法并且创建一个查询树。
3. 重写系统：接受由分析阶段创建的查询树并且搜索可以应用到该查询树上的存储在系统表里的规则，进行规则转换。
4. 优化：优化器接收查询树后创建查询计划，将查询计划输入执行器。
5. 执行：执行器递归地访问规划树，按照规划制定的方式检索数据行，最后返回生成的数据行。

我们更加关心的是：如何处理一个单次查询？如何评价一次查询？

对于postgresql的一次查询流程

```
execute_sql_string(src/backend/commands/extension.c)
```

分为以下几个方面：

①PLSQL语句string生成语法树parseTree。用户在应用程序（linux命令行）输入的string的SQL语句转换为原始语法树（parseTree），利用函数pg\_parse\_query()完成sql语句到语法树的转换，返回到一个语法树list。

```
/*
 * Do parse analysis, rule rewrite, planning, and execution for each raw
 * parsetree. We must fully execute each query before beginning parse
 * analysis on the next one, since there may be interdependencies.
 */
foreach(lc1, raw_parsetree_list)
{
    Node      *parsetree = (Node *) lfirst(lc1);
    List      *stmt_list;
    ListCell   *lc2;

    stmt_list = pg_analyze_and_rewrite(parsetree,
                                      sql,
                                      NULL,
                                      0);
    stmt_list = pg_plan_queries(stmt_list, 0, NULL);
}
```

②生成查询树queryTree。语法树转换成查询树时，根据深度优先遍历所有语法树节点，创建一个Query对象。由于可能生成多种查询树，所以会生成查询树列表queryTreeList，pg\_analyze\_and\_rewrite函数(src/backend/tcop/postgresql.c)负责把生的语法树分析后，重写出查询树列表。

③查询树转计划语句树planStmtTree。循环查询树List，调用pg\_plan\_queries把查询树转换成PlannedStmt，为重写的查询列表生成计划。

④计划语句树转换为计划执行状态树planStateTree，并且转换后的结果放到queryDesc->estate->es\_subplanstates中。在执行器执行过程中，根据es\_subplanstates链表和树形结构进行执行。

```
foreach(lc2, stmt_list)
{
    Node      *stmt = (Node *) lfirst(lc2);

    if (IsA(stmt, TransactionStmt))
        ereport(ERROR,
            (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
              errmsg("transaction control statements are not allowed within an extension script")));

    CommandCounterIncrement();


    PushActiveSnapshot(GetTransactionSnapshot());

    if (IsA(stmt, PlannedStmt) &&
        ((PlannedStmt *) stmt)->utilitystmt == NULL)
    {
        QueryDesc *qdesc;

        qdesc = CreateQueryDesc((PlannedStmt *) stmt,
                                sql,
                                GetActiveSnapshot(), NULL,
                                dest, NULL, 0);

        ExecutorStart(qdesc, 0);
        ExecutorRun(qdesc, ForwardScanDirection, 0);
        ExecutorFinish(qdesc);
        ExecutorEnd(qdesc);

        FreeQueryDesc(qdesc);
    }
}
```



其中执行的四个函数：ExecutorStart/ExecutorRun/ExecutorFinish/ExecutorEnd。依次执行，将真正执行计划执行的查询。对于查询的评估，PostgreSQL为每个收到的查询设计一个查询规划。选择正确的查询结构和数据属性的规划对执行效率至关重要，可以使用EXPLAIN命令查看查询规划器创建的任何查询。

## (二)、源码理解

postgresql框架下代码很多，无需关心查询过程处理细节，需要关注的是：

1. 如何添加外部函数levenshtein\_distance/jaccard\_index?
2. 何处调用函数?
3. 如何评估函数开销?

### A. 部分源码分析

#### ① -src/backend/utils/fmgr/funcapi.c

用于返回集合或复合类型的函数。需要添加的levenshtein\_distance和jaccard\_index函数均在此处。

#### ② -src/backend/executor/execMain.c

真正执行qdesc中的query操作时，需要经过execMain中的四个函数。

#### ③ -src/backend/executor/execProcnode.c

包括为给定节点类型调用适当的初始化、获取元组、清理的调度函数。

#### ④ -src/backend/executor/execScan.c

用于通用的关系扫描。传递一个节点和一个指向函数的指针，从关系中返回一个元组，然后检查条件是否合适，进行投影元组。

⑤ `-src/backend/executor/execTuples.c`

处理可更新插槽的例程。这些用于与元组相关联的资源管理（例如，释放磁盘缓冲区中元组的缓冲区管脚，或释放临时元组占用的内存）。

⑥ `-src/include/catalog/pg_proc.h`

系统“程序”关系（pg\_proc）的定义以及关系的初始内容。可以理解为名字空间。

⑦ `-src/include/utils/builtins.h`

对内置类型的操作声明。外部函数的声明在此处。

其中插入函数最重要的是 `funcapi.c`，`builtins.h`，`pg_proc.h`。

由此可以发现，可以通过如下步骤插入两个算法函数：

① **funcapi.c**里面添加函数主体。

```
Datum levenshtein_distance(PG_FUNCTION_ARGS)
```

（实际上，不在funcapi.c中插入自定义的函数也是可以的，但我们如果需要加入自己新写的.c文件，则需要修改相应的makefile文件，添加编译命令，并确保.c中引用的库包括：`postgres.h`，`utils/builtins.h`；）

参考官方文档：<https://www.postgresql.org/docs/9.3/xfunc-c.html>

我们可以使用macros 来将复杂的参数定义以及返回值类型进行简化，这属于V1定义方式。“PG\_FUNCTION\_INFO\_V1(funcname)”而在函数体中我们也要将 sql 类型与 c 类型定义进行转换，这也在后面的代码实现中有所体现。

② **builtin.h**中声明函数。

```
extern Datum levenshtein_distance(PG_FUNCTION_ARGS);
extern Datum jaccard_index(PG_FUNCTION_ARGS);
```

③ **pg\_proc.h**中注册到名字空间。

```
DATA(insert OID = 4000 ( levenshtein_distance PGNSP PGUID 12 1 0 f f f t f i 2 0 20 "25 25"
_null _null _null _null levenshtein_distance _null _null _null ));
DESCR("levenshtein_distance");
DATA(insert OID = 4001 ( jaccard_index PGNSP PGUID 12 1 0 f f f t f i 2 0 700 "25 25" _null
_null _null _null jaccard_index _null _null _null ));
DESCR("jaccard_index");
```

仔细对应pg\_proc各属性的性质并观察其他函数的插入规则。可以发现：

OID 为每个函数独有的标识，只需要保证插入的函数的OID与先前已定义的函数都不同即可；

```
NameData proname;          /* procedure name */
Oid       pronamespace;    /* OID of namespace containing this proc */
Oid       proowner;        /* procedure owner */
Oid       prolang;         /* OID of pg_language entry */
float4    procost;         /* estimated execution cost */
float4    prorows;         /* estimated # of rows out (if prorowset) */
Oid       provariadic;     /* element type of variadic array, or 0 */
```

```

bool    proisagg;           /* is it an aggregate? */
bool    proiswindow; /* is it a window function? */
bool    proisstrict; /* security definer */
bool    proisstrict; /* strict with respect to NULLs? */
bool    proretset;         /* returns a set? */
char    provolatile; /* see PROVOLATILE_ categories below */

```

proname 属性要与定义的函数名严格一一对应，若有细微差别函数无法连接；  
所有函数的 pronamespace 的 OID 都为 PGNSP，proowner 的 OID 都为 PGUID，  
prolang、procost、prorows、provariadic 分别为 12、1、0、0；

```

/*
 * If function is not marked "proisstrict" in pg_proc, it must check for
 * null arguments using this macro. Do not try to GETARG a null argument!
 */

```

考虑以上属性：对于函数 levenshtein\_distance 以及 jaccard\_index，它们不为聚集函数，proisagg 为 f；不是 window function，proiswindow 为 f，不涉及 security definer，proisstrict 为 f；

```

/*
 * Symbolic values for provolatile column: these indicate whether the result
 * of a function is dependent *only* on the values of its explicit arguments,
 * or can change due to outside factors (such as parameter variables or
 * table contents). NOTE: functions having side-effects, such as setval(),
 * must be labeled volatile to ensure they will not get optimized away,
 * even if the actual return value is not changeable.
 */
#define PROVOLATILE_IMMUTABLE    'i' /* never changes for given input */
#define PROVOLATILE_STABLE      's' /* does not change within a scan */
#define PROVOLATILE_VOLATILE    'v' /* can change even within a scan */

```

由以上注释可知，proisstrict 控制函数参数是否严格不为空，由于我们希望我们的函

```

int2    pronargs;           /* number of arguments */
int2    pronargdefaults; /* number of arguments with defaults */
Oid      prorettype;         /* OID of result type */

```

数不会接收到NULL的参数，因此设置 proisstrict 为 t；由于函数返回 int 或 float，与 set 无关，因此 prorettest 为 f；

```

DATA(insert OID = 20 (      int8      PGNSP PGUID      8 FLOAT8PASSBYVAL b N f t \054
0  0 1016 int8in int8out int8recv int8send - - - d p f 0 -1 0 0 _null_ _null_ ));
DESCR("~18 digit integer, 8-byte storage");

```

参考 postgresql 对于 provolatile 的定义，我们希望我们的 return 值只受到输入的影

响，不会随意改变，因此将 provolatile 置为 'i'；

由于我们需要传入两个字符串作为参数，所以 pronargs 为 2，而这两个函数没有默认的参数，pronargdefaults 为 0，

第一个函数 levenshtein\_distance 返回值为 int32，在 pc\_type.h 中我们可以找到对该类型的定义，因此返回值 oid 为 20：

```
DATA(insert OID = 700 ( float4 PGNSP PGUID 4 FLOAT4PASSBYVAL b N f t \054 0
0 1021 float4in float4out float4recv float4send - - i p f 0 -1 0 0 _null_ _null_
));
DESCR("single-precision floating point number, 4-byte storage");
```

第二个函数 jaccard\_index 返回值为 float4，在 pc\_type.h 中我们可以找到对该类型的定义，因此返回值 oid 为 700：

```
oidvector proargtypes; /* parameter types (excludes OUT params) */
Oid proallargtypes[1]; /* all param types (NULL if IN only) */
```

参考postgresql对于provolatile的定义，我们希望我们的return值只收到输入的影响，不会随意改变，因此将provolatile置为 'i'

接下来需要考虑输入参数的类型，继续在 pc\_type.c 中查找对于 text\* 类型的定义：可以发现其对应的 oid 为 25；则 proargtypes 为 "25 25"

```
DATA(insert OID = 25 ( text PGNSP PGUID -1 f b S t t \054 0 0 1009
textin textout textrecv textsend - - i x f 0 -1 0 100 _null_ _null_ ));
DESCR("variable-length string, no limit specified");
```

## B. 如何调用该函数

在select语句中直接使用函数名称即可，语法树会根据pg\_proc.h中的名字空间解析函数名，然后定位该函数的内容，执行函数。所以可以写代码忽略调用过程，只关心算法实现。

## C. 如何评估函数开销

postgresql对每个QueryDesc提供了Instrmentation\*成员totaltime，其中保存了一次查询任务（忽略生成语法树等操作的执行过程）开始时间、结束时间、执行元组个数等信息，可以通过\timing调用出时间输出函数，每次执行sql后自动打印执行时间。

当然还有单独的explain体系，可以在sql语句加上EXPLAIN命令，或者EXPLAIN ANALYZE，查看查询规划器创建的查询。

# 二、设计思路与实现方案

## （一）、操作步骤

### 1. 环境配置

Linux 18.04.2 + PostgreSQL 9.1.3：下载源码解压后，先使用代码设置configure内容：

(在这一步指令中我们将赋予 \$HOME/pgsql 作为相对路径以便于我们导航到postgresql)

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

然后使用以下指令将数据库初始化:

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

之后可以创建数据库database, 使用make && make install将源码中全部makefile编译, 后使用以下指令开启postgresql服务器:

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

下一步使用psql命令连接上服务器, 为了测试数据, 可以建立相似度的数据库。

```
>$ $HOME/pgsql/bin/psql postgres
>$ create database similarity
```

此处导入数据similarity.sql

```
>$ $HOME/pgsql/bin/psql -d similarity -f /home/chtty627/similarity_data.sql
```

## 2. 插入外部函数levenshtein\_distance和jaccard\_index

操作方式同“源码理解”中的插入函数, 此处注意注册到命名空间中的OID不能重复, 选择了4000和4001, 测试无重复。

## 3. 重启测试

测试内容同PPT上内容相同, 先测试比较小的数据。

select jaccard\_index('sunday','saturday'); select levenshtein\_distance('sunday','sunday');测试无误后, 再测试后面的大数据集。

```
select count(*) from restaurantphone rp, addressphone ap where levenshtein_distance(rp.phone, ap.phone) < 4;
select count(*) from restaurantaddress ra, restaurantphone rp where levenshtein_distance(ra.name, rp.name) < 3;
select count(*) from restaurantaddress ra, addressphone ap where levenshtein_distance(ra.address, ap.address) < 4;
select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
```

## (二)、设计思路

按照以上内容测试无误后, 则可以开始进行funcapi.c中的主函数编辑。设计思路则为:

- ① 根据PG\_GETARG\_DATUM(0/1)获取函数中的两个参数, text\_to\_cstring()转换为字符串
- ② 对两个字符串进行运算。
- ③ 测试时间并优化

### 三、关键代码说明

#### 1. levenshtein\_distance

##### 【算法原理】

从一个字符串转变为另一个字符串是很自然的顺序过程，联想到归纳的方法。

以字符串的长度为归纳对象：

考虑长度为  $l_1$  的字符串  $s_1$  变到长度为  $l_2$  的字符串  $s_2$  的过程，基于每次允许的编辑操作：

（假设  $s_{1\ l_1-1}$  为  $s_1$  字符串去掉尾字符、 $s_{2\ l_2-1}$  为  $s_2$  字符串去掉尾字符）

- 从空字符到非空字符

在过程中，若碰到空字符串到非空字符串的转换，假设非空字符串长度为  $l$ ，只需要  $l$  步不断插入字符即可。

- 将一个字符替换成另一个字符

假设从  $s_{1\ l_1-1}$  变到  $s_{2\ l_2-1}$  步骤已知：

若  $s_1$ 、 $s_2$  最后一个字符相同，无需替换，则  $s_1$  变到  $s_2$  等价于从  $s_{1\ l_1-1}$  变到  $s_{2\ l_2-1}$  的过程；

若不同，则等价于从  $s_{1\ l_1-1}$  变到  $s_{2\ l_2-1}$  后替换最后一个字符的过程

- 插入一个字符

假设从  $s_1$  变到  $s_{2\ l_2-1}$  步骤已知

则只需要在  $s_1$  末尾插入  $s_2$  末尾字符，即可完成  $s_1$  转变到  $s_2$ ；

- 删除一个字符

假设从  $s_{1\ l_1-1}$  变到  $s_2$  步骤已知

则只需要在  $s_1$  末尾删除一个字符，再由  $s_{1\ l_1-1}$  变到  $s_2$ ，即可完成  $s_1$  转变到  $s_2$ ；

记  $s_{1[0:i]}$  以及  $s_{2[0:j]}$  之间的Levenshtein距离为  $LevenD^{s_1, s_2}_{(i, j)}$

经过总结，我们可以归纳的到：

$$LevenD^{s_1, s_2}_{(i, j)} = \begin{cases} i + j & i = 0 \text{ or } j = 0 \\ \text{Min} \begin{cases} LevenD^{s_1, s_2}_{(i-1, j)} + 1 \\ LevenD^{s_1, s_2}_{(i, j-1)} + 1 \\ LevenD^{s_1, s_2}_{(i-1, j-1)} + 1 \end{cases} & \text{Otherwise} \end{cases} \quad (s_1 \neq s_2)$$

##### 【算法实现】

```
#define MIN(a,b) ((a)<(b))?(a):(b)
/* TODO: levenshtein_distance & jaccard_index */

static int d[100][100];
static bool init_d = true;

void init_distance()
{
    for (int i = 0; i <= 100; i++)
    {
        d[i][0] = i;
        d[0][i] = i;
    }
    init_d = false;
}
```



【1】 在代码中，我们用二维int类型矩阵d来记录逐步求得的 Levenshtein distance；在测试中，我们假设取得的字符串长度  $\leq 100$ （这个假设在后续也将成立）；由于 C90 中数组的长度不可由变量指定，我们将d设计为静态全局变量。由于在【原理】部分提到的由空字符串到非空字符串转变的过程。数组d的第一列（0），第一行（0）的结果对于所有字符串是统一的。我们使用 init\_distance 函数对其进行初始化。由于每一次计算不同字符串 s1 以及 s2，都不会修改这些数据。初始化仅需要一次，由全局静态变量 init\_d 来控制。

```
Datum levenshtein_distance(PG_FUNCTION_ARGS)
{
    text * str_01 = PG_GETARG_DATUM(0);
    text * txt_02 = PG_GETARG_DATUM(1);
    int32 result = 1;
    char * s = text_to_cstring(str_01);
    char * t = text_to_cstring(txt_02);
    int m = strlen(s), n = strlen(t);

    if(init_d) init_distance(); //initialize the whole distance

    for (int j = 1; j <= n; j++)
        for (int i = 1; i <= m; i++)
            if (s[i] == t[j])
                d[i][j] = MIN(d[i - 1][j] + 1, MIN(d[i][j - 1] + 1, d[i - 1][j - 1]));
            else
                d[i][j] = MIN(d[i - 1][j] + 1, MIN(d[i][j - 1] + 1, d[i - 1][j - 1] + 1));

    result = d[m][n];
    PG_RETURN_INT32(result);
}
```

【2】 进入levenshtein\_distance函数：

- A. 首先，接收参数：参考官方文档：<https://www.postgresql.org/docs/9.3/xfunc-c.html>；我们在用 C 内核以及V1方式定义函数后，函数体内的第一步就是接收参数，在这里，我们可以指明参数的类型（实际上已经隐式地定义过了）：  
PG\_GETARG\_DATUM(0) / PG\_GETARG\_TEXT\_P(0) 都可以用来接收第一个参数；
- B. 我们需要将接收到的text类型参数转化为char \* 类型（C语言是没有针对 text\* 类型的函数的，如果直接处理将会出现问题），我们可以直接调用 postgresql 中已经定义的函数 text\_to\_cstring（在 verlena.c 中定义，utils\builtins.h 中外部声明了该函数）  
（A、B 操作在 jaccard\_index中也会重复，不再加以解释）
- C. 如果要对 d 数组进行初始化，则需要先初始化 init\_distance
- D. 随后使用两层循环，不断重复如【原理】部分的算法。知道最后生成一个  $(m + 1) * (n + 1)$  的数组。我们所要求的距离就是 d[m][n] 的值

#### 【算法性能】

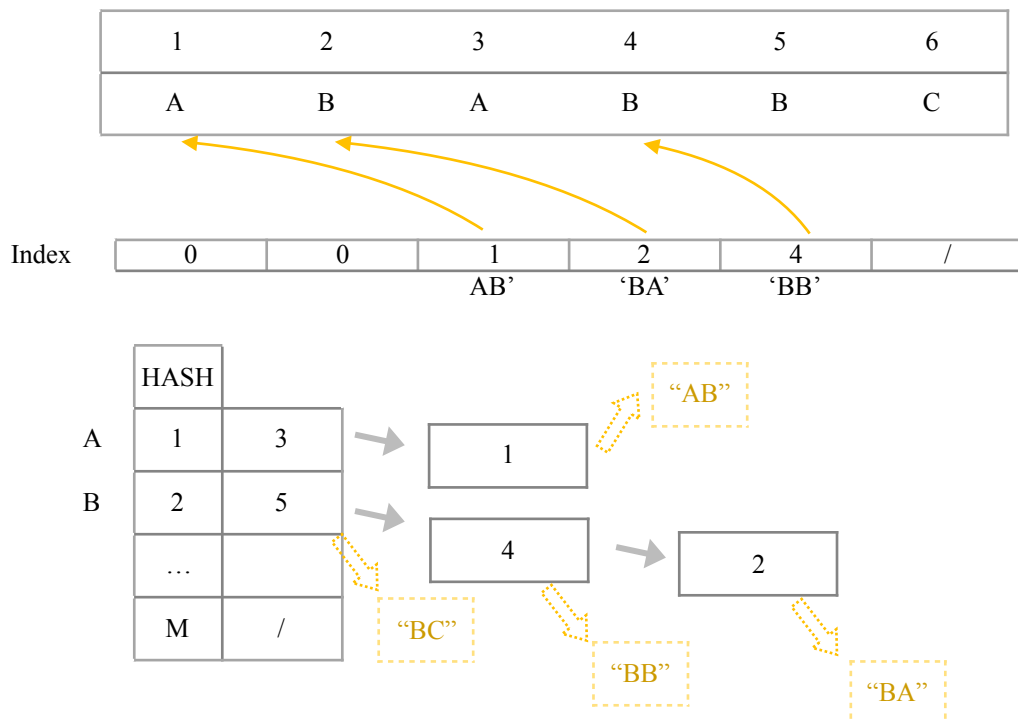
初始化数组有恒定的 200 的开销；对于每一次计算，复杂度都为  $O(m * n)$ ；在这里使用静态数组可以免去每次查询时再创建数组的开销，由于每一次d数组都会被完全重新写入，并不会读取上一次写入的数据，因此可省去memset的步骤（这一开销减少较为有限，后文的测试中将有所体现）。

## 2. jaccard\_index

#### 【算法原理】



1. 首先，结合 jaccard index 的计算方法，我们需要计算的部分就是 1) 字符串 s1 以及 s2 中相交两位字符的个数 2) 字符串 s1 以及 s2 中两位字符的并集
2. 考虑到相似性，我们需要支持大小写字母模糊匹配；考虑到集合的性质，我们需要考虑“去重”的问题，对同一字符串中出现多次的二元字符串只记录一次；
3. 由于需要判断两元字符串是否在之前已经出现，并且判断 s2 中两元字符串是否在 s1 中出现，如果暴力遍历比较显然不是一个好方法（这在后文优化的部分将有体现）；为了便于找到更加相似的二元字符串进行比较，以减少比较次数，在这一部分，我们使用 Hash 的方式，现将每个二元字符串的首字母映射到一个 Hash 桶中，直接将具有相同 Hash 值的二元字符串进行比较。而由于 C 并没有动态的存储容器，使用 malloc 动态分配空间的开销也很大。Hash 桶中的溢出链的维护方式采用静态链表。只需要在对应字符串的区间内维护即可：
4. 以字符串 ABABBC 为例，它投射到 HASH 中的情况如图所示。对应字符起始位置的静态链表也见下图：



#### 【算法实现】

```

#define min(x,y,z) ((x < y) ? ((z < x)? z : x) : ((z < y)? z : y))
#define check_char(x,y) (((x == y) || ((x == y + 32) && (x <= 'z' && x >= 'a' && y >= 'A' && y <= 'Z'))) || ((x == y - 32) && (x <= 'Z' && x >= 'A' && y >= 'a' && y <= 'z')) ? 1 : 0)
#define M 53
/* by value */

int hash(char c){
    return (((c > 'a' && c <= 'z') ? c - 'a' : c - 'A') + 'a') % M;
}

```

【1】 判断两字符是否相等的函数，为了更好地提升 similarity index 的性能，在算法中我

们支持大小写模糊匹配，所以判断两个字符是否相等有以下两种成立情况：

- 1) 两字符ASCII一致，即字符完全一样
- 2) 两字符一个为小写字母另一为大写字母，他们都转换为小写字母后为同一字符

使用 `check_char` 宏定义来计算这一布尔值，可以减少函数的调用开销

hash函数，我对两元字符串的首字符采用hash函数，hash所得到的桶中存储相应的字符组起始位置。值得注意的是，需要将同一字母（不区分大小写）投射到同一值；

```
static int ref1[M];
static int ref2[M];
static int index1[100];
static int index2[100];
```

【2】定义静态数组 `ref1`、`ref2` 代表分别对字符串 `s2` 以及 `s1` 哈希后桶中储存的字符索引。由于 C 中对于动态空间的管理代价较高，不考虑采用动态数组的方式添加溢出链。因此需要建立静态索引 `index1`、`index2` 来静态索引hash桶中溢出的二元字符串的起始位置。

```
Datum jaccard_index(PG_FUNCTION_ARGS)
{
    text * t1 = PG_GETARG_DATUM(0);
    text * t2 = PG_GETARG_DATUM(1);
    char * s1 = text_to_cstring(t1);
    char * s2 = text_to_cstring(t2);
    int l1 = strlen(s1);
    int l2 = strlen(s2);

    memset(ref1, 0, sizeof(ref1));
    memset(ref2, 0, sizeof(ref2));
    memset(index1, 0, sizeof(index1));
    memset(index2, 0, sizeof(index2));

    int start1, start2, i, j, k;
    int countu = 0;
    int countn = 0;
```

【3】开始定义jaccard 函数：由于每一次传入的参数不同，字符串长度不一，内容也不一致。所以每次都需要通过memset一系列静态数组以确保需要用到的索引位已经清空置0。初始化 `countu`（交集的元数）、`countn`（并集的元数）；

```
for(i = 0; i <= l1; i++){
    if(i == 0 || i == l1) countn++;
    else{
        start1 = ref1[hash(s1[i - 1])];
        if(start1 == 0) {ref1[hash(s1[i - 1])] = i; countn++;}
        else{
            for(j = start1; j = index1[j - 1]){
                if(j == 0 || (check_char(s1[j - 1], s1[i - 1]) &&
                    check_char(s1[j], s1[i])))
                    break;
            }
            if(j == 0){
                index1[i - 1] = start1;
                ref1[hash(s1[i - 1])] = i;
                countn++;
            }
        }
    }
}
```

【4】对第一个字符串 `s1` 处理：

1. 对于开头以及结尾单独字符组成的二元字符，由于其特殊性，不需要和其他字符进行比较；并且，直接加入并集，`countn++`；

2. 对于由两个字符组成的二元字符串，需要考虑是否重复，先计算它的hash值，查看 hash 值对应的桶：

- i. 若桶中索引值为 0，代表这是第一个出现的具有该 hash 值的二元字符串，直接将其索引编号加入桶中，countn++
- ii. 若桶中存在不为 0 的索引值，则代表之前出现过相似的二元字符串；则需要结合 Hash 桶中存储的 ref 值以及字符串相对应的静态索引 index 来遍历桶中所有元素，由于得到的是“起始位置”，所以要将该位置转化为对应的字符，两两比较，若一致，则代表已经出现，停止检索；若检索到最后得到索引值为0，且没有找到完全一致的两元字符串，则代表现在的字符串是唯一的，countn ++，并将桶中的 ref 更新为当前字符串的其实地址，并把当前字符对应的静态索引存储桶中原本的索引，以确保更新桶中的静态索引链；

```
for(k = 0; k <= l2; k++){
    if(k == 0){
        if(check_char(s2[0], s1[0])) countu++;
        else countn++;
    }
    else if(k == l2){
        if(check_char(s1[l1 - 1], s2[l2 - 1])) countu++;
        else countn++;
    }
    else{
        start1 = ref1[hash(s2[k - 1])];
        start2 = ref2[hash(s2[k - 1])];
        if(start2 == 0) {ref2[hash(s2[k - 1])] = k;}
        else{
            for(j = start2; j = index2[j - 1]){
                if(j == 0 || (check_char(s2[j - 1], s2[k - 1]) &&
                    check_char(s2[j], s2[k]))) break;
            }
            if(j == 0){
                index2[k - 1] = start2;
                ref2[hash(s2[k - 1])] = k;
            }
        }
        if(ref2[hash(s2[k - 1])] == k){
            if(start1 != 0){
                for(j = start1; j = index1[j - 1]){
                    if(j == 0) {
                        countn++;
                        break;
                    }
                    else if (check_char(s1[j - 1], s2[k - 1]) && check_char
                        (s1[j], s2[k])){
                        countu++;
                        break;
                    }
                }
            }
        }
    }
}
```

#### 【4】对第二个字符串 s2 处理：

1. 对于开头以及结尾单独字符组成的二元字符串，考虑其特殊性，只需要将开头字符与 s1 的开头字符比较，结尾字符与 s1 的结尾字符比较，如果相同，则将二元组加入交集，countu++；若不同，则加入并集，countn++；
2. 对于其他二元字符串，与字符串 s1 的处理相似的是，首先需要判断当前字符串是否重复：
  - i. 如果重复，则继续考虑下一字符串（是否重复的算法与s1是一致的，只是采用 s2 对应的 hash桶 ref2 以及静态索引 index2）；
  - ii. 如果不重复，则需要考虑 s1 中是否存在相同的两元字符串，将其得到的 hash 值映射到 s1 的 hash桶 rf1 中，根据桶中的静态索引一一比较 s1 中具有相同

hash 值的字符串，如果找到相同的，加入交集，countu++; 否则，加入并集，countn++;

```
PG_RETURN_FLOAT4((float)countu / countn);  
}
```

【5】最后的结果即为  $\text{countu} / \text{countn}$ ，注意要将类型强制转换为 float;

### 【算法性能】

使用hash桶 + 静态索引的算法开销与字符串中字符的分布有关，假设每个桶中平均有p个索引链，m 为 s1 的长度，n 为 s2 的长度，则可估计的复杂度为： $O(2 * p * m + p * n)$ ;

考虑其他算法（见下文）的复杂度：

Brute Force:  $O((n^2 - n) / 2 + (m^2 - m) / 2 + m * n)$

Quick Sort + Merge:  $O(n \log n + m \log m + m + n)$

最后得到的结果是Hash + Static Link表现最优，因为在实际的应用中，p值非常小，并且我们考虑了对两元字符串两个字符进行 Hash 以及 首字符 Hash两种不同的方式，最后是首字符Hash的方法具有略微的优势（见下文）。

## 四、实验与结果

### 1. levenshtein\_distance

① static静态数组 + memset

```
similarity=# \timing  
Timing is on.  
similarity=# select count(*) from restaurantphone rp, addressphone ap where levenshtein_distance(rp.phone, ap.phone) < 4;  
count  
-----  
3252  
(1 row)  
  
Time: 12344.738 ms  
similarity=#  
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where levenshtein_distance(ra.name, rp.name) < 3;  
count  
-----  
2313  
(1 row)  
  
Time: 20156.379 ms  
similarity=# select count(*) from restaurantaddress ra, addressphone ap where levenshtein_distance(ra.address, ap.address) < 4;  
count  
-----  
2874  
(1 row)  
  
Time: 38353.983 ms
```

	查询1	查询2	查询3
结果	3252	2313	2874
时间(ms)	12344	20156	38353

② static静态数组 + 初始化

```

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where levenshtein_distance(rp.phone, ap.phone) < 4;
count
-----
3252
(1 row)

Time: 11711.933 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where levenshtein_distance(ra.name, rp.name) < 3;
count
-----
2313
(1 row)

Time: 20642.471 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where levenshtein_distance(ra.address, ap.address) < 4;
count
-----
2874
(1 row)

Time: 40932.444 ms

```

	查询1	查询2	查询3
结果	3252	2313	2874
时间(ms)	11711	20642	40932

【结论】memeset对速度的影响是极小的，是否采用init\_distance的方式对整体性能影响较小

## 2. jaccard\_index

### ① Brute Force

```

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
count
-----
1653
(1 row)

Time: 12200.501 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
count
-----
2398
(1 row)

Time: 24274.595 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
count
-----
2186
(1 row)

Time: 44585.124 ms

```

### ② Hash + M = 53

```

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
count
-----
1653
(1 row)

Time: 6860.428 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
count
-----
2391
(1 row)

Time: 10863.338 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
count
-----
2196
(1 row)

Time: 14833.912 ms

```

### ③ Hash + M = 103

```

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
count
-----
1653
(1 row)

Time: 6779.537 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
count
-----
2391
(1 row)

Time: 10415.210 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
count
-----
2196
(1 row)

Time: 13853.398 ms

```

#### ④ Hash + M = 131

```

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
count
-----
1653
(1 row)

Time: 7015.794 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
count
-----
2391
(1 row)

Time: 10530.191 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
count
-----
2196
(1 row)

Time: 14077.952 ms

```

#### ⑤ Hash2c + M = 53

```

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
count
-----
1653
(1 row)

Time: 6964.955 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
count
-----
2398
(1 row)

Time: 9947.087 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
count
-----
2186
(1 row)

Time: 14070.510 ms

```

#### ⑥ Quicksort + Merge

```

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
count
-----
1653
(1 row)

Time: 15493.826 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
count
-----
2397
(1 row)

Time: 30419.955 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
count
-----
2186
(1 row)

Time: 46200.839 ms

```

```

similarity=# EXPLAIN ANALYZE select count(*) from restaurantphone rp, addressph
one ap where levenshtein_distance(rp.phone, ap.phone) < 4;
QUERY PLAN

-----
Aggregate  (cost=13647.43..13647.44 rows=1 width=0) (actual time=11703.012..117
03.012 rows=1 loops=1)
-> Nested Loop  (cost=0.00..13030.03 rows=246960 width=0) (actual time=0.057
..11701.213 rows=3252 loops=1)
    Join Filter: (levenshtein_distance((rp.phone)::text, (ap.phone)::text)
    < 4)
-> Seq Scan on addressphone ap  (cost=0.00..35.76 rows=1176 width=68)
(actual time=0.015..3.754 rows=2429 loops=1)
-> Materialize  (cost=0.00..30.45 rows=630 width=118) (actual time=0.0
00..0.176 rows=2463 loops=2429)
    -> Seq Scan on restaurantphone rp  (cost=0.00..27.30 rows=630 wi
dth=118) (actual time=0.010..0.791 rows=2463 loops=1)
    Total runtime: 11703.204 ms
(7 rows)

```

## ⑦ Static + Quicksort + Merge

```

similarity=# timing
Timing is on.
similarity=# select count(*) from restaurantphone rp, addressphone ap where jacc
ard_index(rp.phone, ap.phone) > 0.6;
count
-----
1653
(1 row)

Time: 14392.393 ms
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where
jaccard_index(ra.name, rp.name) > 0.65;
count
-----
2397
(1 row)

Time: 28698.061 ms
similarity=# select count(*) from restaurantaddress ra, addressphone ap where ja
ccard_index(ra.address, ap.address) > 0.8;
count
-----
2186
(1 row)

Time: 42585.701 ms

```

	查询1(ms)	查询2(ms)	查询3(ms)
1	12200	24274	44585
2	6860	10863	14833
3	6779	10415	13853
4	7015	10530	14077
5	6964	9947	14070
6	15493	30419	46200
7	14392	28698	42585

【结论】算法3表现出最好的性能，所耗时间约为brute force的一半；值得思考的是，quicksort + merge 的方法并不能比暴力算法更优的表现，即便是采用静态数组对字符串索引排序，消耗的时间依旧很长，这样的原因可能是因为在测试中，数据数据字符串长度较短，暴力循环和递归排序所进行的比较次数是相近的。

## 五、性能优化

### • levenshtein\_distance

静态与非静态数组

#### 【优点】

使用静态数组，而不是直接在函数中定义临时变量，可以减少调用函数中分配内存的时间；并且在levenshtein中不需要通过每一次memset来维护数组（尽管从实验结果中可以看到memset由于其优越性对于整体时间的影响十分小）

#### 【Query Plan】

见下图：



```

similarity=# EXPLAIN ANALYZE select count(*) from restaurantaddress ra, address
phone ap where levenshtein_distance(ra.address, ap.address) < 4;
QUERY PLAN

-----
Aggregate  (cost=26486.95..26486.96 rows=1 width=0) (actual time=40764.658..407
64.658 rows=1 loops=1)
-> Nested Loop  (cost=0.00..25286.45 rows=480200 width=0) (actual time=0.086
..40762.257 rows=2874 loops=1)
    Join Filter: (levenshtein_distance((ra.address)::text, (ap.address)::te
xt) < 4)
-> Seq Scan on restaurantaddress ra  (cost=0.00..37.25 rows=1225 width
=68) (actual time=0.009..5.488 rows=2439 loops=1)
-> Materialize  (cost=0.00..41.64 rows=1176 width=68) (actual time=0.0
00..0.176 rows=2429 loops=2439)
    -> Seq Scan on addressphone ap  (cost=0.00..35.76 rows=1176 widt
h=68) (actual time=0.008..0.534 rows=2429 loops=1)
    Total runtime: 40764.816 ms
(7 rows)

```

```

similarity=# EXPLAIN ANALYZE select count(*) from restaurantaddress ra, restaur
antphone rp where levenshtein_distance(ra.name, rp.name) < 3;
QUERY PLAN

-----
Aggregate  (cost=14214.88..14214.89 rows=1 width=0) (actual time=21384.912..213
84.912 rows=1 loops=1)
-> Nested Loop  (cost=0.00..13571.75 rows=257250 width=0) (actual time=43.12
0..21382.907 rows=2313 loops=1)
    Join Filter: (levenshtein_distance((ra.name)::text, (rp.name)::text) <
3)
-> Seq Scan on restaurantaddress ra  (cost=0.00..37.25 rows=1225 width
=68) (actual time=0.040..5.249 rows=2439 loops=1)
-> Materialize  (cost=0.00..30.45 rows=630 width=118) (actual time=0.0
00..0.190 rows=2463 loops=2439)
    -> Seq Scan on restaurantphone rp  (cost=0.00..27.30 rows=630 wi
dth=118) (actual time=0.011..0.560 rows=2463 loops=1)
    Total runtime: 21385.101 ms
(7 rows)

```

- jaccard\_index

- i. Brute Force

```

for(i = 0; i <= l1; i++){
    if(i == 0 || i == l1) countn ++;
    else{
        for(j = 1; j < i; j++){
            if(check_char(s1[j - 1], s1[i - 1]) && check_char(s1[j], s1[i])) break;
        }
        if(j == i) countn ++;
    }
}

for(i = 0; i <= l2; i++){
    if(i == 0){
        if(check_char(s1[0], s2[0])) countu ++;
        else countn ++;
    }
    else if(i == l2){
        if(check_char(s1[l1 - 1], s2[l2 - 1])) countu ++;
        else countn ++;
    }
    else{
        for(j = 1; j < i; j++){
            if(check_char(s2[j - 1], s2[i - 1]) && check_char(s2[j], s2[i]))
                break;
        }
        if(j == i){
            for(k = 0; k < l1 - 1; k++){
                if(check_char(s1[k], s2[i - 1]) && check_char(s1[k + 1], s2
[i])){
                    countu ++;
                    break;
                }
            }
            if(k == l1 - 1) countn ++;
        }
    }
}
PG_RETURN_FLOAT4((float)countu / countn);

```

【算法解释】

- i. 对于 s1，对于每一个二元字符串，遍历查找其之前是否有与其重复的字符串。
- ii. 对于 s2，对于每一个二元字符串，遍历查找其之前是否有与其重复的字符串，若无，则遍历 s1，查找s1中是否有相同的二元字符串；

【缺点】查找了大量冗余的字符串；

【复杂度】 $O((n^2 - n) / 2 + (m^2 - m) / 2 + m * n)$

## ii. Hash and Static List

### i. Bucket 的大小

Hash 桶的数目与数据分布的散度有关；在测试中我们得到的最优的桶的大小为 103；

### ii. 非静态数组与静态数组

将静态索引以及桶数组设置为静态数组可以有效地减少每次调用函数时开辟临时变量的开销（鉴于memset函数性能较优）

### iii.Hash or Hash2c

```
int hash2c(char a, char b){
    return (((a >= 'A' && a <= 'Z') ? a - 'A' : a - 'a') * 3 + ((b >= 'A' && b <= 'Z') ? b - 'A' : b - 'a') + 500) % M;
}
```

在实验中，我们还考虑了将二元字符串一齐映射Hash值的方法，但这一方法与只考虑一个字符的方法相比，没有太大的性能差异，因此，我们最后选择只对首字母 hash 的方法；

【优点】可以有效减少查找次数，只查找具有相同hash值的二元字符组；

【复杂度】 $O(2 * p * m + p * n)$

## iv. Sort and Merge

### i. 非静态数组与静态数组

由于quicksort使用递归的方法，因此，如果每一次递归都要传入整个数组，开销将很大，鉴于C中没有传引用等方法，最好的选择就是将排序的索引数组设置为静态全局变量。这样就可以直接调用，不会产生传递数组时压栈的开销；

## 【算法解释】

```
bool less(char x, char y){
    if(x <= 'Z' && x >= 'A' && y >= 'a' && y <= 'z')
        return x < y - 'a' + 'A';
    else if(x <= 'z' && x >= 'a' && y >= 'A' && y <= 'Z')
        return x < y - 'A' + 'a';
    else
        return x < y;
}

bool greater(char x, char y){
    if(x <= 'Z' && x >= 'A' && y >= 'a' && y <= 'z')
        return x > y - 'a' + 'A';
    else if(x <= 'z' && x >= 'a' && y >= 'A' && y <= 'Z')
        return x > y - 'A' + 'a';
    else
        return x > y;
}
```

【1】less 和 greater 函数帮助quick sort排序，由于大小写模糊，less 和 greater需要特别设计，以确保大小写字母的大小关系是正确的。即将所有大写字母都转化为小写，再进行比较；

```

void swap_one(int a,int b)
{
    int temp;
    temp = sort1[a];
    sort1[a] = sort1[b];
    sort1[b] = temp;
}

void swap_two(int a,int b)
{
    int temp;
    temp = sort2[a];
    sort2[a] = sort2[b];
    sort2[b] = temp;
}

```

## 【2】swap函数帮助快排实现交换

```

void quickSort_one(const char* s, int start, int end){
    int Base, arrMiddle;
    int tempStart = start, tempEnd = end;
    if(tempStart >= tempEnd) return;
    Base = sort1[start];
    while(start < end){
        while(start < end && (greater(s[sort1[end]], s[Base]) || (check_char(s[sort1[end]], s[Base]) && greater(s[sort1[end] + 1], s[Base + 1]))))
            end--;
        if(start < end){
            swap_one(start, end);
            start++;
        }
        while(start < end && (less(s[sort1[start]], s[Base]) || (check_char(s[sort1[start]], s[Base]) && less(s[sort1[start] + 1], s[Base + 1]))))
            start++;
        if(start < end){
            swap_one(start, end);
            end--;
        }
    }
    sort1[start] = Base;
    arrMiddle = start;
    quickSort_one(s, tempStart, arrMiddle - 1);
    quickSort_one(s, arrMiddle + 1, tempEnd);
}

```

【3】快速排序采用典型的算法，但是排序的数组中存储的并不是字符串，而是记录字符串起时位置的索引，因此，比较大小时，需要将该数组中存放的值再次作为字符串索引的值；

```

quickSort_two(s2, 0, l2 - 2);

for(i = 0, j = 0; ){
    while(i < l1 - 2 && check_char(s1[sort1[i]], s1[sort1[i + 1]]) &&
        check_char(s1[sort1[i] + 1], s1[sort1[i + 1] + 1])) i++;
    while(j < l2 - 2 && check_char(s2[sort2[j]], s2[sort2[j + 1]]) &&
        check_char(s2[sort2[j] + 1], s2[sort2[j + 1] + 1])) j++;
    if(i != l1 - 1 && j != l2 - 1 && check_char(s1[sort1[i]], s2[sort2[j]])
        && check_char(s1[sort1[i] + 1], s2[sort2[j] + 1])){
        countu++;
        countn++;
        i++;
        j++;
    }
    else if(i != l1 - 1 && j != l2 - 1 && (less(s1[sort1[i]], s2[sort2[j]])
        || (check_char(s1[sort1[i]], s2[sort2[j]]) && less(s1[sort1[i] + 1],
        s2[sort2[j] + 1])))){
        countn++;
        i++;
    }
    else if(i != l1 - 1 && j != l2 - 1){
        countn++;
        j++;
    }
    else if(i == l1 - 1 && j != l2 - 1){
        countn++;
        j++;
    }
    else if(i != l1 - 1 && j == l2 - 1){
        countn++;
        i++;
    }
    else break;
}

if(check_char(s1[0], s2[0])){
    countu++;
    countn++;
}
else countn += 2;

if(check_char(s1[l1 - 1], s2[l2 - 1])){
    countu++;
    countn++;
}
else countn += 2;

```

【4】在merge的过程中，操作主要如下：

- i. 如果s1、s2指针同时指向结尾，结束；

- ii. 如果s1、s2指针没有指向结尾，对于s1、s2排好序的索引，先往后查找看是否有重复的二元字符串，直至下一个字符串与当前不同；（如果重复，则出栈）
- iii. 如果s1、s2指针没有指向结尾，比较s1、s2的当前字符串：
  - i. 若s2的更小，s2的指针后移（s2出栈）；countn++
  - ii. 若s1更小，s1的指针后移（s1出栈）；countn++
  - iii. 若s1、s2相同，同时后移（同时出栈）；countu++，countn++
  - iv. 跳转到（ii）
- iv. 如果有一指针指向结尾，则只后移另一字符串排序数列的指针；

【优点】避免重复查找以及比较，通过排序一次性解决问题

【复杂度】  $O(n \log n + m \log m + m + n)$

【Query Plan】

```
similarity=# EXPLAIN ANALYZE select count(*) from restaurantphone rp, addressph
one ap where jaccard_index(rp.phone, ap.phone) > 0.6;
QUERY PLAN
-----
Aggregate (cost=13647.43..13647.44 rows=1 width=0) (actual time=7037.450..7037
.450 rows=1 loops=1)
-> Nested Loop (cost=0.00..13030.03 rows=246960 width=0) (actual time=0.016
..7036.649 rows=1653 loops=1)
    Join Filter: (jaccard_index((rp.phone)::text, (ap.phone)::text) > 0.6::
double precision)
    -> Seq Scan on addressphone ap (cost=0.00..35.76 rows=1176 width=68)
(actual time=0.005..1.733 rows=2429 loops=1)
        -> Materialize (cost=0.00..30.45 rows=630 width=118) (actual time=0.0
00..0.158 rows=2463 loops=2429)
            -> Seq Scan on restaurantphone rp (cost=0.00..27.30 rows=630 wi
dth=118) (actual time=0.004..0.514 rows=2463 loops=1)
    Total runtime: 7037.584 ms
(7 rows)
```

```
similarity=# EXPLAIN ANALYZE select count(*) from restaurantaddress ra, address
phone ap where jaccard_index(ra.address, ap.address) > 0.8;
QUERY PLAN
-----
Aggregate (cost=26486.95..26486.96 rows=1 width=0) (actual time=14007.411..140
07.411 rows=1 loops=1)
-> Nested Loop (cost=0.00..25286.45 rows=480200 width=0) (actual time=0.033
..14006.082 rows=2196 loops=1)
    Join Filter: (jaccard_index((ra.address)::text, (ap.address)::text) > 0
.8::double precision)
    -> Seq Scan on restaurantaddress ra (cost=0.00..37.25 rows=1225 width
=68) (actual time=0.010..2.746 rows=2439 loops=1)
        -> Materialize (cost=0.00..41.64 rows=1176 width=68) (actual time=0.0
00..0.165 rows=2429 loops=2439)
            -> Seq Scan on addressphone ap (cost=0.00..35.76 rows=1176 widt
h=68) (actual time=0.007..0.552 rows=2429 loops=1)
    Total runtime: 14007.681 ms
(7 rows)
```

```
similarity=# EXPLAIN ANALYZE select count(*) from restaurantaddress ra, restaur
antphone rp where jaccard_index(ra.name, rp.name) > 0.65;
QUERY PLAN
-----
Aggregate (cost=14214.88..14214.89 rows=1 width=0) (actual time=10562.367..105
62.367 rows=1 loops=1)
-> Nested Loop (cost=0.00..13571.75 rows=257250 width=0) (actual time=11.34
0..10561.223 rows=2391 loops=1)
    Join Filter: (jaccard_index((ra.name)::text, (rp.name)::text) > 0.65::d
ouble precision)
    -> Seq Scan on restaurantaddress ra (cost=0.00..37.25 rows=1225 width
=68) (actual time=0.010..2.265 rows=2439 loops=1)
        -> Materialize (cost=0.00..30.45 rows=630 width=118) (actual time=0.0
00..0.166 rows=2463 loops=2439)
            -> Seq Scan on restaurantphone rp (cost=0.00..27.30 rows=630 wi
dth=118) (actual time=0.006..0.552 rows=2463 loops=1)
    Total runtime: 10562.521 ms
(7 rows)
```

## 六、其他

### (一)、分工

本次PJ2由两人合作完成:

1、宁晨然 17307130178:

1) 环境配置及试运行 2) 框架代码阅读和整体分析 3) levenshtein\_distance/jaccard\_index算法和优化思路 4) 结果测试与分析

2、田睿 17300750084

1) 环境搭建和算法剖析 2) 插入外部函数解析 3) levenshtein\_distance/jaccard\_index算法并多种优化方式

### (二)、实验难点

本次实验在源码阅读、环境配置和算法优化上耗费很多时间。

源码阅读上, postgresql的框架很大, 刚上手很难找到我们需要的函数、需要修改的内容。在阅读了网络上一些框架解读后, 发现只需要从sql的执行过程入手即可, 也无需关心算法语法树的优化过程, 而外部函数的插入有固定的模式。

环境配置, 一开始装入数据库后, 并且插入函数后, 发现无法找到对应OID的函数, 也遇到了函数执行时无法连接到数据库的情况, 还有数据库无法连接(开服务器问题)。解决方法是删除整个数据库, 从configure重新开始、重启电脑等, 对于函数执行过程中无法连接数据库, 是因为函数体中有不规范的C语言, 例如数组长度为变量、int声明在for循环初始化中等, 有许多C语言规范的问题导致了连接中断。

算法优化, 对于第一个算法很简单, 但是第二个由于牵涉到集合操作, 而C语言特别不擅长处理动态数组的问题, 也没有可行的容器进行集合删去重复、并交操作, 所以采取了多种算法思路, 中间花费了很多时间。

### (三)、实验感想

本次实验总体而言很顺利, 两人合作分工进行代码剖析、讨论, 有时一人没有看懂的框架另一人提纲挈领的点拨能加快阅读代码的速度, 并且讨论算法时两人会有更多不同的思路, 经过讨论再到实现就有了不同的方案的解读。

算法实现方面使用了很多数据结构和数据库的综合内容, 是对学过的知识的活学活用, 加深了对于数据库本学期内容的理解。

### (四)、实验环境

Gcc version 7.4.0

系统: Ubuntu18.04.2

内存: 2G

参考资料

[1]: <https://www.cnblogs.com/flying-tiger/p/8039055.html>

[2]: <https://blog.csdn.net/tencupofkaiwater/article/details/80967948>

[3]: <https://www.postgresql.org/docs/9.3/xfunc-c.html>