相似度查询实验报告

罗翔 17307130191

解润芃 17307130196

1 系统与源码理解

1.1 funcapi.c文件

在该文件中需要实现Levenshtein Distance和Jaccard Index两种相似度度量的函数，下面以Levenshtein Distance函数讲解对源码的理解。

```
Datum levenshtein_distance(PG_FUNCTION_ARGS)
{
    text *str_01 = PG_GETARG_DATUM(0);
    text *txt_02 = PG_GETARG_DATUM(1);

    char *str1 = TextDatumGetCString(str_01);
    char *str2 = TextDatumGetCString(txt_02);

    Just for implementation

    PG_RETURN_INT32(result);
}
```

函数开头的PG_GETARG_DATUM(n)是获得函数的第n个参数，TextDatumGetCString则将获得的参数从自定义的形式转换成C语言中的字符串便于后续操作，然后实现相似度匹配的算法来度量两个字符串之间的相似度。PG_RETURN_INT32(result)则是将result作为函数调用结果返回。

1.2 pg_proc.h文件

在该文件中，我们需要声明在funcapi.c函数中写的两个函数，仍以Levenshtein Distance的函数声明为例进行说明。

```
DATA(insert OID = 4375 ( levenshtein_distance PGNSP PGUID 12 1 0 0 0 f f f f t f i u 2 0 20 "25 25" _null_ _null_ _null_ _null_ _null_ levenshtein_distance _null_ _null_ _null_ ));
DESCR("levenshtein_distance");

```

刚开始我们只是参照普通班同学的声明方式，发现在编译过程中会报错：函数参数个数不匹配。在张作柏同学的帮助下，发现因为普通班同学实验的postgressql版本为9.1.3，而我们的版本则为10.4，现版本的函数声明需要29个参数而9.1.3只需要25个参数。而在该文件的头部有对函数的定义与参数的详细说明即CATALOG(pg_proc,1255) BKI_BOOTSTRAP BKI_ROWTYPE_OID(81) BKI_SCHEMA_MACRO，再将两个版本的函数定义对比之后，发现现版本较以往版本多了4个参数分别为：protransform，proleakproof，proparallel，protrftypes。在根据现有版本的其他函数调用完成参数补充后成功编译。

值得指出的是，我们在后续优化时将Levenshtein Distance给定阈值作为第三个参数传入函数作为剪枝的参照，这里同样需要重新声明一个函数，并且更改参数个数与参数类型pronargs，proargtypes。

2 基本设计思路及其实现方法

2.1 Levenshtein_distance

下面以两字符串$s_1 s_2 \dots s_n$和$t_1 t_2 \dots t_m$为例说明Levenshtein_distance度量方式，当考察$s_1 s_2 \dots s_i$到$t_1 t_2 \dots t_j$的最快变换方式时，令s指标游动，t指标一定，如果s_i与t_j相同，则显然与$s_1 s_2 \dots s_{i-1}$到$t_1 t_2 \dots t_{j-1}$的最快变化等价；若不相同，则有以下三种方式得到变换：

- $s_1 s_2 \dots s_i$删除s_i变换为$s_1 s_2 \dots s_{i-1}$，再转变为$t_1 t_2 \dots t_j$
- $s_1 s_2 \dots s_i$转换为$t_1 t_2 \dots t_{j-1}$再添加t_j
- $s_1 s_2 \dots s_i$将s_i转换为t_j再进一步转换为$t_1 t_2 \dots t_j$

这个过程使用直接枚举的方法复杂度是不可承受的，但是其可以通过维护一个矩阵进行动态规划来得到结果。

2.2 Jaccard_index

通过比较两字符串bigram集合的重合情况来比较两字符串的近似度，选择bigram的原因是为了一定程度上保留词语的语义（上下文顺序）。可以通过哈希来快速匹配，对第一个字符串遍历时即建立一个哈希表，遍历第二个字符串时用同样的哈希函数映射。显然为了消除冲突，我们需要每个bigram的哈希值尽量不同，因为ASCII码共128个元素，所以可以考虑维护一个n的哈希列表并使用$h = (129 * a + b)\%n$函数得到一张哈希表，n的选择可以用来控制哈希表的疏密程度。当然n可以等于128*128，这样则不存在哈希的情况。哈希函数可以退化为一个标准的二维表，将第一个字符作为横坐标，第二个字符作为纵坐标访问。但是当使用的字符个数越来越多时，此方法不具有拓展性，因为当字符数逐渐增多时，数据将逐渐稀疏，哈希的思想会非常重要。

3 关键代码说明

3.1 Levenshtein_distance

```
for(int i = 1; i <= len1; i++)
    for(int j = 1; j <= len2; j++)
        if(s[i] == t[j])                                    //如果匹配上则看左上角的点
            distance[i][j] = distance[i - 1][j - 1];
        else                                                //否则找最快变化
            distance[i][j] = mymin(distance[i - 1][j] + 1, //insert
                                   distance[i][j - 1] + 1, //delete
                                   distance[i - 1][j - 1] + 1); //substitute
```

大致的思想已经通过上面代码的注释中体现了出来。这里需要强调的一点是这个表格并不是单调的，其会因为左上角的值而发生非单调情况。同是由于基本的动态规划要求，嵌套循环是不可避免的。

3.3 Jaccard_index

```
for(p = newstr1; *(p + 1)!= '\0'; p++)
{
    int a = *p;
    int b = *(p + 1);
    int h = 103 * a + b;//因为不区分大小写所以基数减少26
    if(visit[h][0] == 0)//bigram在第一个字符串中第一次出现
        ++sum, visit[h][0] = 1;//对第一个字符串遍历将bigram总数+1，并标记相应的bigram
}
for(p = newstr2; *(p + 1)!= '\0'; p++)
{
    int a = *p;
    int b = *(p + 1);
    int h = 103 * a + b;
    if(visit[h][1] == 0)//bigram在第二个字符串中第一次出现
    {
        if(visit[h][0])//如果第一个字符串中同样出现，则匹配成功
            join++;
        else//否则bigram总数+1
            ++sum;
        visit[h][1] = 1;//标记bigram
    }
}
float4 result = ((float)join) / sum;
```

Jaccard distance相较于Levenshtein distance要容易理解的多。上面的代码先进行一次对第一个字符串的哈希，然后在对第二个字符串哈希并且判断是否同样在第一个字符串中出现。如是可以同时构建交集与并集。当然也可以通过容斥原理获得其中的一个值然后求解第二个值，但是本质而言，这两种方法并没有减少任何复杂度，原因是由于一个字符串中的bigram可能重复，所以导致计算集合大小的操作仍然需要记录一个哈希表。

4 实验与结果

4.1 Levenshtein distance

Levenshtein采用的查询为以下三条：

```
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;
```

结果分别为：3252， 2130， 2592.用时为9172ms, 17696ms, 35306ms.可以看到这个查询是相当慢的，其很大程度是因为存在的嵌套for语句。

4.2 Jaccard distance

Jaccard distance使用以下三条：

```
select count (*)  from restaurantphone rp, addressphone ap where
jaccard_index(rp.phone, ap.phone) > .6;
select count (*)  from restaurantaddress ra, restaurantphone rp where
jaccard_index(ra.name, rp.name) > .65;
select count (*)  from restaurantaddress ra, addressphone ap where
jaccard_index(ra.address, ap.address) > .8;
```

结果分别为：1653， 2398， 2186. 而检索所用的时间是4521ms, 5083ms, 6555ms。由于没有特殊的初始化，所以不会有Levenshtein这么高的初始化代价。检索使用时间相应小一点。

5 性能优化

5.1 带有剪枝的动态规划

显然我们可以发现在上述动态规划的过程中，两字符串可能出现$s_1 s_2 \dots s_i$到$t_1 t_2 \dots t_j$最快变化所需次数已经出给定的阈值的情况，对于这些点我们可以选择不访问以减小复杂度，也就是剪枝操作。

具体实现类似广度搜索，先维护一个队列，每次操作从队列中读出一个参照点对它周围的三个点（右，下，右下）进行更新，一个点最多会被周围三个点（左，上，左上）更新，显然更新矩阵的过程与基础方法相反。

因为我们要得到的是最快路径，所以对矩阵中每个参照点的数值更新都应是单调下降（以得到最小值）的操作，因此在参照点对周围三个点完成更新后，如果周围三个点中某一点的数值小于阈值其最终结果一定小于阈值，即为符合条件的点，选择压入队列。

为了保证当从队列中读出一个参照点时，已经完成了对它的更新，则压入队列时应保证右下的参照点是最后压入队列的，这样也可以保证矩阵的右下角是最后一个更新的点也即是结果。如果最终结果大于阈值，则在过程中的某一个点时队列将清空，因此我们只需要判断最后更新的一个点是否为右下角的点即可。

复杂度的优化上，主要是通过引入阈值来减少了参照点的个数来实现的。在这个优化下，三种查询的时间分别为：7621ms, 8707ms, 15985ms。而检索的结果和没有优化过的算法是相同的，但是使用的时间大幅度减少。在第三个查询中可以看见，使用的时间直接减小到了原来的一半。

相关代码

```
//注明：下面的更新操作是指将更新值与被更新点的当前值相比较取较小的值
int front = 0, end = 0;
que[end][0] = 0;
```

```
que[end++][1] = 0;                                  //压入队列
visit[0][0] = 1;
distance[0][0] = 0;
while(front < end)
{
    i = que[front][0];
    j = que[front++][1];                            //从队列中读出一个参照点
    if(j + 1 < len2)                                //判断有无越过边界
    {
        distance[i][j + 1] = min(distance[i][j + 1], distance[i][j] + 1);//参照点对其右侧和下方的点的更新都是+1操作
        if(distance[i][j + 1] < tex && visit[i][j + 1] == 0)//如果当前值小于阈值则压入队列，并标记
        {
            que[end][0] = i;
            que[end++][1] = j + 1;
            visit[i][j + 1] = 1;
        }
    }

    if(i + 1 < len1 && j + 1 < len2)
    {
        if(s[i + 1] == t[j + 1])//参照点对右下方的点的更新需判断是否匹配，匹配则更新为参照点的值，否则为参照点的值+1
            distance[i + 1][j + 1] = min(distance[i][j], distance[i + 1][j + 1]);
        else
            distance[i + 1][j + 1] = min(distance[i][j] + 1, distance[i + 1][j + 1]);
        if(distance[i + 1][j + 1] < tex && visit[i + 1][j + 1] == 0)
        {
            que[end][0] = i + 1;
            que[end++][1] = j + 1;
            visit[i + 1][j + 1] = 1;
        }
    }
}
bool result;
result = i == len1 - 1 && j == len2 - 1;//最终判断是否到右下角
```

5.2 使用更小的哈希

Jaccard的空间复杂度为O(m+n+128*128)。由于m，n都小于100，所以其空间复杂度主要由哈希表决定。在我们的观察中，对于一个正常的序列而言，其应当出现的字符包括数字、大小写字母、括号等标准符号。而在ASCII码中，存在大量的与相似度无关的字符，如ASCII码中前32号不可见字符。这一部分字符对于哈希是没有帮助的，所以可以自然地从哈希表中删去。如此一来，删掉相关字符之后的哈希表仅为60*61即可，比原有的128*129空间缩小了3/4。

在这个情况下，三种查询的时间变为3291ms,4027ms, 4635ms。而正常的速度是4521ms, 5083ms, 6555ms。而在这个程度上进一步缩小哈希，降低准确度以获得更小的空间，例如采用1601的空间，则准确度变为100%, 99.7%, 99.1%，但是事实上速度反而下降，分别为3600ms, 3987ms, 4645ms。我们认为由于缩小空间而出现额外的一步整除操作带来了较高的损耗。但是空间缩小一倍，准确率相当高，并且速度没有大幅度下降时，所以当空间复杂度十分重要时，这样的策略是值得考虑的。

除此之外，可以将原来的两个哈希表合为一个，仅需要获取一次地址，0表示都未访问过，1表示其中第一个访问过，2表示第二个访问过。这样可以正常地获得所有的结果。自然速度也较原来的略快。三次检索分别为：3088ms, 3682ms, 4349ms。

5.3 一些其他优化

这一部分优化非常琐碎，比如避免频繁取地址、减少不必要的判断、有效初始化等等，有一定效果，但是效果不如前两种改善明显。

5.4 时间对照

Levenshtein distance

时间（ms）	第一样例	第二样例	第三样例
基本动态规划	9172	17696	35306
带有剪枝的动态规划	4521	5083	6555

Jaccard distance

时间（ms）	第一样例	第二样例	第三样例
基本算法	4521	5083	6555
缩减字符表I	3291	4027	4635
缩减字符表II	3088	3682	4349
缩减字符表I+哈希	3600	3987	4645

5.5 其他

事实上，还有一些其他的策略可供使用，用以针对一些特殊性的查询。例如在使用Jaccard方法时，由于算法的原因，导致每一次都需要计算一次第一个字符串的哈希，这个代价是不必要的。哈希可以在计算之后保留下来，在每一个对象计算结束之后再放掉。这样可以有效地避免重复计算。以及对于Levenshtein距离很高的代价是维护一个距离表的初始化。而这个列表不需要对于每一个查询都初始化，而是作为一个全局变量，每一次在进行查询结束时自动恢复该表，如此可以节约大量的初始化以及请求空间的时间。

6 致谢

感谢张作柏同学就实验过程中出现的一系列问题给出帮助，王辰浩同学就实现过程中的算法问题给予大量细致的指导。