期末 PJ 实验报告

Cr. R.W. & Qw.Y.

1. 系统与源码理解及安装

   a. **系统与源码理解**

   PostgreSQL 是一个基于 POSTGRES 的对象关系数据库管理系统(ORDBMS)。它具有跨平台，支持文本、图像、声音和视频，支持 SQL 许多功能等优点。

   PostgreSQL 由连接管理系统(系统逻辑控制作用)，编译执行系统(实现物理存储介质中数据的操作)，存储管理系统(提供对编译查询系统的支持)，事务系统(完成对操作请求的事务一致性支持)，系统表(元信息管理中心)五大部分组成。

   PostgreSQL 使用一种客户端/服务器的模型。一次 PostgreSQL 会话由一个服务器进程和那些需要执行数据库操作的用户的客户端应用组成。

   在一次查询中，首先建立一个从应用程序到 PostgreSQL 的服务器的连接，应用程序传递一个查询给服务器并等待接收由服务器传回的结果；在分析阶段，创建查询树对上一阶段传递的查询进行语法检查；接下来是重写系统阶段，它会得到分析阶段创建的查询树，它的一个应用是实现视图；而后，规划器/优化器接手重写过的查询树并创建一个将被作为执行器输入的查询计划；最后执行器递归地逐步通过计划树并按照计划表述的方式获取行。

   b. **安装**

   用的是 postgresql-10.4 版本，参考了 https://www.postgresql.org/docs/10/static/installation.html 上有关安装的教程，但没有完全按照这个版本来安装，过程中出了一点问题(如提示没有 zlib 这种)。具体代码见附录 Readme。

16.1. Short Version

```
./configure
make
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 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

之后建立数据库 similarity，用$HOME/pgsql/bin/psql similarity 进入到新建的数据库中，使用 select levenshtein_distance('sunday', 'saturday');和 select jaccard_index('sunday', 'saturday');就可以简单测试实现的函数。

2. 设计思路与实现方案

需要完成的两个函数在 src/backend/utils/fmgr/funcapi.c 中实现，分别为 levenshtein_distance()和 jaccard_index()。

此外，添加 c 函数以后，还需依据其参数个数和返回值类型等在 pg_proc.h 中为其添加命名空间。在 pg_proc.h 中已经有挺多别的函数的命名空间，参考这些格式和 https://doxygen.postgresql.org/pg__proc_8h.html 以及 https://www.postgresql.org/docs/10/catalog-pg-proc.html，就可以写出 levenshtein_distance()和 jaccard_index()的命名空间。

Functions

	CATALOG (pg_proc, 1255, ProcedureRelationId) **BKI_BOOTSTRAP BKI_ROWTYPE_OID**(81
Oid pronamespace	**BKI_DEFAULT** (PGNSP)
Oid proowner	**BKI_DEFAULT** (PGUID)
Oid prolang	**BKI_DEFAULT** (internal) **BKI_LOOKUP**(pg_language)
float4 procost	**BKI_DEFAULT** (1)
float4 prorows	**BKI_DEFAULT** (0)
char prokind	**BKI_DEFAULT** (f)
bool proisstrict	**BKI_DEFAULT** (t)
char provolatile	**BKI_DEFAULT** (i)
char proparallel	**BKI_DEFAULT** (s)
Oid prorettype	**BKI_LOOKUP** (pg_type)
ObjectAddress	**ProcedureCreate** (const char *procedureName, **Oid** procNamespace, **bool** replace, **bool** returnsSet, **Oid** returnType, **Oid** proowner, **Oid** languageObjectId, **Oid** languageValidator, const char *prosrc, const char *probin, char prokind, **bool** security_definer, **bool** isLeakProof, **bool** isStrict, char volatility, char parallel, **oidvector** *parameterTypes, **Datum** allParameterTypes, **Datum** parameterModes, **Datum** parameterNames, **List** *parameterDefaults, **Datum** trftypes, **Datum** proconfig, **Oid** prosupport, **float4** procost, **float4** prorows)
bool	**function_parse_error_transpose** (const char *prosrc)
List *	**oid_array_to_list** (**Datum** datum)

具体代码见附录 Readme。

此外一个需要注意的就是怎样获取入口和返回值。这可在文档

https://www.postgresql.org/docs/10/xfunc-c.html 中找到。

3. 关键代码说明、结果与简单优化

Levenshtein_distance：

求解两个字符串的最短编辑距离。使用的算法为动态规划。这是从前数据结构课程上机实验练习过的算法，因此写起来十分容易。即建立一个二维数组 d[M+1][N+1]，其中 d[i][j]表示第一个字符串的前 i 个元素构成的子串转化成第二个字符串的前 j 个元素构成的子串的最短编辑距离。这个算法的时空复杂度都是 O(MN)。关键代码如下：

```
if(toupper(s[i-1])==toupper(t[j-1]))
        temp=0;
else
        temp=1;
min=d[i-1][j-1]+temp;
int d1=d[i-1][j]+1;
int d2=d[i][j-1]+1;
if(d1<min)
        min=d1;
if(d2<min)
        min=d2;
d[i][j]=min;
```

一开始 temp=0 的条件仅设置为 s[i-1]==t[j-1]，结果运行出来发现结果总比答案少了 10，然而反复检查代码后又觉得没错，于是猜想可能是大小写的判断上与答案不同。改过以后就跑出了和答案相同的结果。如下：

```
qiaowenyang@qiaowenyang-virtual-machine:~/postgresql-10.4$ $HOME/pgsql/bin/psql
 similarity
psql (10.4)
Type "help" for help.

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantaddress ra, addressphone ap where l
evenshtein_distance(ra.address, ap.address) < 4;
 count
-------
  2592
(1 row)

Time: 85847.351 ms (01:25.847)
```

然而耗时很长，于是考虑了以下优化。

简单的优化算法

在相似度查询中，所要求的仅是字符串编辑距离小于 4 或 3 的表项，而字符串编辑距离大于等于 4 的，因其参考度不高，可直接过滤掉。据此优化 dp 算法如下：

```
if(abs(len1-len2)>=4)
        PG_RETURN_INT32(4);

int32 result=1;
int i,j,temp,min;
int d[len1+1][len2+1];

for(i=0;i<=len1;i++)
        d[i][0]=i;

for(j=0;j<=len2;j++)
        d[0][j]=j;

for(i=1;i<=len1;i++){
        for(j=1;j<=len2;j++){
                if(abs(i-j)>=4){
                        d[i][j]=4;
                        continue;
                }
                if(toupper(s[i-1])==toupper(t[j-1]))
                        temp=0;
                else
                        temp=1;
                min=d[i-1][j-1]+temp;
                int d1=d[i-1][j]+1;
                int d2=d[i][j-1]+1;
                if(d1<min)
                        min=d1;
                if(d2<min)
                        min=d2;
                d[i][j]=min;
        }
}
```

就算法整体来说，还是复杂度为 O(MN)的，但由于对字符串长度的限制，很多时候复杂度可以达到常数级或线性，因此平均下来运行效率就有了明显的提升，如下：

```
qiaowenyang@qiaowenyang-virtual-machine:~/postgresql-10.4$ $HOME/pgsql/bin/psql
similarity
psql (10.4)
Type "help" for help.

similarity=# \timing
Timing is on.
similarity=# select count(*) from restaurantaddress ra, addressphone ap where le
venshtein_distance(ra.address, ap.address) < 4;
 count
-------
  2592
(1 row)

Time: 18633.133 ms (00:18.633)
similarity=#
```

Jaccard_index：

求两个字符串的 Jaccard_index，普通算法使用两层循环，长度为 M+N 的 bitmap 记录 bigram 是否重读, isec 记录交集，usec 记录|s|+|t|，并集元素个数为 usec-isec。时间复杂度为 O(MN)，空间复杂度为 O(c)。运行时间平均 8s。

```
        for(j = i+1; (!flag)&&(j < len1-1) ; j++)
        {
                if(s[i] == s[j] && s[i+1] == s[j+1])
                {
                        ss[j] = 1;
                }
        }
        if(!flag)
                        usec++;
        flag = 0;
}
flag = 0;
for(i = 0; i<len2-1; i++)
{
        if(tt[i] == 1)
                flag = 1;
        for(j = i+1; (!flag)&&(j<len2-1); j++)
                if(t[i] == t[j] && t[i+1] == t[j+1])
                        tt[j] = 1;
        if(!flag)
        {
                usec++;
                for(j = 0; j < len1-1; j++)
                        if(!ss[j]&&t[i]==s[j]&&t[i+1]==s[j+1])
                                isec++;
        }
        flag = 0;
}
```

容易想到使用哈希算法改进，参考 BKDRhash 函数的建立方法，考虑到 ASCII 不包括控制字符和空格有 94 个字符，取偏移量为 128，建立哈希函数 h(ab) = a<<7+b。时间复杂度为 O(N)，空间复杂度为 O(c)，使用了两个 bool 类长度为 127*127 的哈希表。运行时间平均为 6.6s。

```
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: 6604.002 ms (00:06.604)
```

```
if(!ss[s[len1-1]]){
        usec++;
        ss[s[len1-1]] = 1;
}
for(i = 0; i < len1-1; i++)
{
        h = (s[i]<<7)+s[i+1];
        if(!ss[h]){
                ss[h] = 1;
                usec++;
                }
}
tt[t[0]] = 1;
if(ss[t[0]])
        isec++;
if(!tt[t[len2-1]]){
        tt[t[len2-1]] = 1;
        usec++;
        if(ss[t[len2-1]])
                isec++;
}
for(i = 0; i<len2-1; i++)
{
        h = (t[i]<<7)+t[i+1];
        if(!tt[h]){
                tt[h] = 1;
                usec++;
                if(ss[h])
                        isec++;
        }
}
```

4. 进一步优化

利用 q-gram 优化编辑距离

1）Jokinen 和 Ukkonen 提出了过滤算法中最早的过滤标准: Count Filtering。s, t 为比较字符串，τ 为编辑距离下界。两字符串最少共有 LBs;t = (max(|s|, |t|) − q + 1) − q · τ 个相同的 q-gram。原理：最坏情况下每个编辑操作破坏 q 个 q-gram，原字符串有 length-(q-1)个 q-gram。由于我们希望在计算 matching q-gram 的时候使用优化了的 hash 算法，使用了集合运算不考虑字符串内重复的 q-gram，因此将下界调整为：max(|union_of_q_gram(s)|, | union_of_q_gram(t)|) − q · τ，意义为最坏情况下每个编辑操作破坏 q 个不同的 q-gram。

2）Prefix Filtering: 两个字符串$(q \cdot \tau + 1)$长度前缀里包含的 q-gram 中至少有一个是匹配的。因为需要考虑到重复的 q-gram，$q \cdot \tau + 1$ 比较小，因此使用普通的两层循环，时间复杂度是常数级别 O（square$(q \cdot \tau)$）。

实验：取 $q = 2, \tau = 3$。

```
bool prefix_filter(char*s, char*t){
        int q = 2, count = 0, botom = 3, min = q*botom+1, isec = 0, i, h=0,j;
        for(i = 0; i<min; i++){
                for(j=0; j<min; j++){
                        if(s[i]==t[j]&&s[i+1]==t[j+1])
                                return 0;
                }
        }
        return(isec==0);
}
bool count_filter(char*s, char*t,int len1,int len2){
        int q = 2, count = 0, botom = 3,usec1 = 0,usec2 = 0;//distance<=3
        int isec = 0;//matching bigram>=count
        int i;
        bool ss[16129]={0}, tt[16129]={0};
        int h = 0;
        if(prefix_filter(s,t)) return 1;
        for(i = 0; i < len1-1; i++)
        {
                h = (s[i]<<7)+s[i+1];
                        if(!ss[h]){
                                ss[h] = 1;
                                usec1++;
                        }
        }
        for(i = 0; i<len2-1; i++)
        {
                h = (t[i]<<7)+t[i+1];
                if(!tt[h]){
                        tt[h] = 1;
                        usec2++;
                        if(ss[h])
                                isec++;
                }
        }
        count = ((usec1<usec2)?usec2:usec1)-q-botom*q;
        return(isec<count);
}
```

单独使用 Length Filtering 和 Count Filtering，平均运行时间：4.5s。

加入 Prefix Filtering，平均运行时间：3.5s，改善不多，可能是因为前缀筛选条件比较弱，筛选效果不是非常好。

```
similarity=# select count(*) from restaurantaddress ra, addressphon
e ap where levenshtein_distance(ra.address, ap.address) < 4;
 count
-------
  2592
(1 row)

Time: 4535.789 ms (00:04.536)
```

```
similarity=# select count(*) from restaurantaddress ra, addressphon
e ap where levenshtein_distance(ra.address, ap.address) < 4;
 count
-------
  2592
(1 row)

Time: 3542.932 ms (00:03.543)
```

其他可能的优化策略：

1、考虑 q-gram 不同的 q 取值，q 增大的时候需要考察的匹配次数减少，但是哈希函数计算步骤增加，相对应的下界成 τ 倍减小，就筛选条件来说 q 越小越严格。

2、建立倒排索引，和索引的 BKDRHash 表。建立索引可以考虑扩展 gin 索引的操作符类。(对 jaccard_index 也有效，其他可用的索引类型有：GIST，距离树等)

3、Position Filtering: 两个字符串至少有$(\max(|s|,|t|)-q+1)-q\cdot\tau$ 个串中位置相同的 q-gram。

4、Location-based Mismatch Filtering & Content-based Mismatch Filtering: 用贪心算法（计算间距大于 q-1 的 mismatch q-gram）求出消除 mismatch q-gram 的编辑距离，得到编辑距离下界，可以与 τ 比较。内容方面考虑滑动窗格内的字符串的频率向量的 L1 距离不大于 2τ。

5、从 join 实现方法考虑，还可以改进为 block-nested loop-join 等花费更少的连接方法。

完整测试运行结果：

```
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: 18917.428 ms (00:18.917)
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where levenshtein_distance(ra.name, rp.name) < 3;
 count
-------
  2130
(1 row)

Time: 42137.329 ms (00:42.137)
similarity=# select count(*) from restaurantaddress ra, addressphone ap where levenshtein_distance(ra.address, ap.address) < 4;
 count
-------
  2592
(1 row)

Time: 89515.512 ms (01:29.516)
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
 count
-------
  1653
(1 row)

Time: 9086.724 ms (00:09.087)
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
 count
-------
  2347
(1 row)

Time: 23367.719 ms (00:23.368)
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
 count
-------
  2199
(1 row)

Time: 37160.318 ms (00:37.160)
similarity=#
```

简单优化后：

```
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: 8755.911 ms (00:08.756)
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where levenshtein_distance(ra.name, rp.name) < 3;
 count
-------
  2130
(1 row)

Time: 4904.516 ms (00:04.905)
similarity=# select count(*) from restaurantaddress ra, addressphone ap where levenshtein_distance(ra.address, ap.address) < 4;
 count
-------
  2592
(1 row)

Time: 11785.448 ms (00:11.785)
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
 count
-------
  1653
(1 row)

Time: 6283.174 ms (00:06.283)
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
 count
-------
  2347
(1 row)

Time: 6481.373 ms (00:06.481)
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
 count
-------
  2199
(1 row)

Time: 7047.343 ms (00:07.047)
similarity=#
```

进一步优化：

```
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: 5719.404 ms (00:05.719)
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where levenshtein_distance(ra.name, rp.name) < 3;
 count
-------
  2099
(1 row)

Time: 2255.333 ms (00:02.255)
similarity=# select count(*) from restaurantaddress ra, addressphone ap where levenshtein_distance(ra.address, ap.address) < 4;
 count
-------
  2592
(1 row)

Time: 2787.742 ms (00:02.788)
similarity=# select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;
 count
-------
  1653
(1 row)

Time: 5320.440 ms (00:05.320)
similarity=# select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;
 count
-------
  2347
(1 row)

Time: 5818.580 ms (00:05.819)
similarity=# select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
 count
-------
  2199
(1 row)

Time: 6936.315 ms (00:06.936)
similarity=#
```

参考文献

[1] A. Andoni, M. Deza, A. Gupta, P. Indyk, and S. Raskhod- nikova. Lower bounds for embedding edit distance into normed spaces. In SODA, pages 523–526, 2003.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In VLDB, 2006.

[3] Xiao, Chuan & Wang, Yi & Lin, Xuemin. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. PVLDB. 1. 933-944. 10.14778/1453856.1453957, 2008.