期末实验

期末实验

- **Similarity Joins** (Basic)
- **Block Nested Loop Joins** (additional)

Overview and Motivation

- Before starting, please read the entire assignment fully and carefully. For this assignment, you can (optionally) work in pairs.
- For this assignment, you will be implementing similarity joins. Similarity joins are useful for when the data is dirty or unreliable.

Overview and Motivation

- For example, say we have two relations A and B and we want to join them on the address field. We have a record in relation A:

Name	Address
Cheap but Delicious Restaurant	123 Durant Ave. Berkeley, CA

- And we have a record in relation B:

Address	Phone
123 Durant Avenue Berkeley, CA	(510) 123-4567

- and we wish to join them on the address field. We could try using "A.Address = B.Address" but that would never match, and we would never be able to match the two records.

Overview and Motivation

- For example, say we have two relations A and B and we want to join them on the address field. We have a record in relation A:

Name	Address
Cheap but Delicious Restaurant	123 Durant Ave. Berkeley, CA

- And we have a record in relation B:

Address	Phone
123 Durant Avenue Berkeley, CA	(510) 123-4567

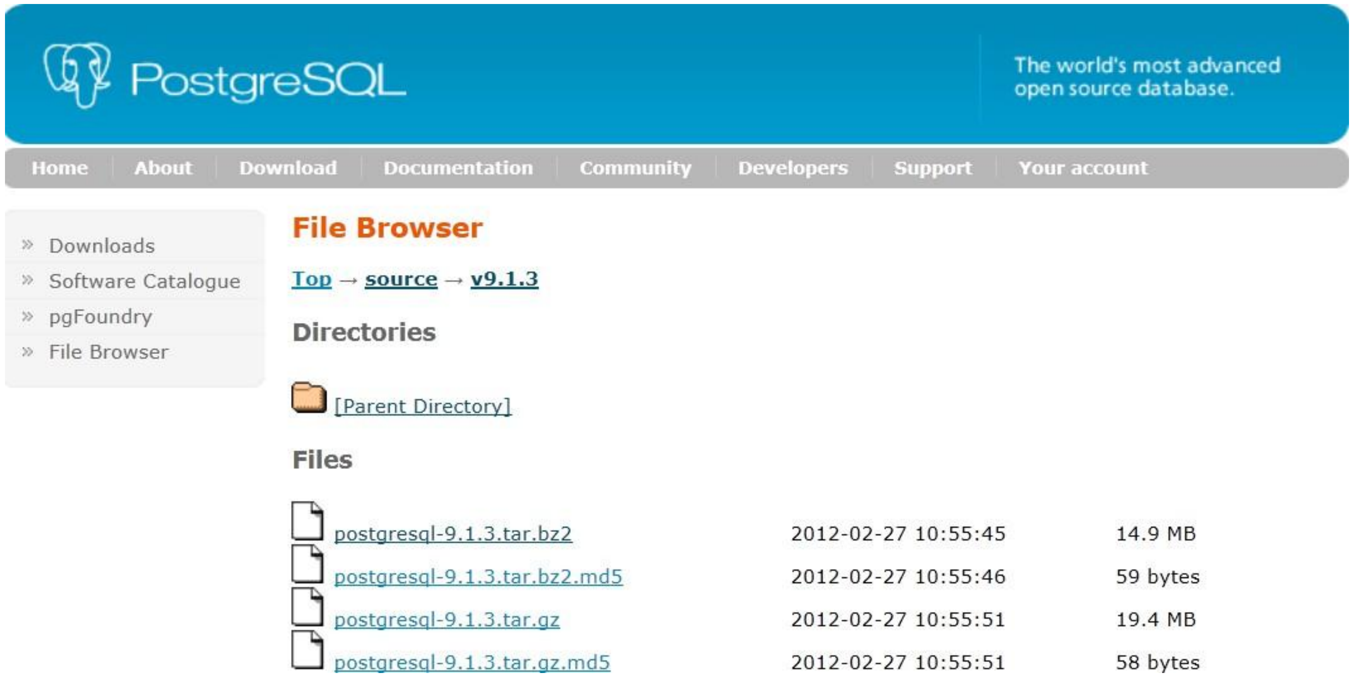
- However, if the records could be join if the addresses were "similar" enough, that would be useful. Similarity joins use similarity calculations to determine whether records should be matched. You will be implementing two different kinds of similarity metrics in Postgres: the Levenshtein distance calculation and the Jaccard index calculation.

Overview and Motivation

- You will also be implementing the block nested loop join algorithm in Postgres. Block nested loop join is an optimization over simple nested loop join. Simple nested loop join scans the inner relation for every single outer relation tuple to find a match. This may be very slow if there are many outer relation tuples. Block nested loop join improves on that by scanning the inner relation for each "block" of outer relation tuples. This may greatly reduce the disk IOs performed.
- We suggest that you get started on this assignment with plenty of time to spare: although the actual code you will need to write for this assignment is relatively simple, understanding the existing code will take some time, as will debugging and testing your changes.

Preparation--Get source code

- URL: http://www.postgresql.org/ftp/source/v9.1.3/



File Browser

Top → source → v9.1.3

Directories

[Parent Directory]

Files

postgresql-9.1.3.tar.bz2	2012-02-27 10:55:45	14.9 MB
postgresql-9.1.3.tar.bz2.md5	2012-02-27 10:55:46	59 bytes
postgresql-9.1.3.tar.gz	2012-02-27 10:55:51	19.4 MB
postgresql-9.1.3.tar.gz.md5	2012-02-27 10:55:51	58 bytes

Preparation--Building Postgres

- After getting source code, configure, compile and install PostgreSQL. We will configure PostgreSQL to be installed in the "pgsql" subdirectory of your account(postgres):

```
cd postgresql-9.1.3
./configure --enable-depend --enable-cassert --enable-debug CFLAGS="-
O0" --prefix=$HOME/pgsql
make
make install
```

- Note that after you have modified PostgreSQL (as described below), you should recompile and reinstall the modified version of PostgreSQL using the "make" and "make install" commands, respectively.

Preparation--Building Postgres

- Some errors may occur in the building process

```
configure: error: readline library not found
solution:sudo apt install libreadline-dev

configure: error: zlib library not found
solution:sudo apt-get install zlib1g-dev

configure: error: C compiler cannot create executables
```

This error probably occurs when CFLAGS is wrong. The right CFLAGS is -O0 where the first character is Capital O, while the second character is number 0.

Preparation--Starting PostgreSQL Server

- After building and installing Postgres, and after you delete your old database, use the initdb command to initialize a local PostgreSQL database cluster. This probably only has to be done once.

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

```
/home/postgres/pgsql/bin/pg_ctl -D /home/postgres/pgsql/data -l logfile
start
```

- You can use “tail logfile” command to fileprint a few informational messages to the console:

```
server starting
postgres@z-ubuntu:~/postgresql-9.1.3$ tail logfile  LOG: database
system was shut down at 2022-02-27 05:33:51 CST  LOG: database
system is ready to accept connections
LOG: autovacuum launcher started
```

Preparation—Connect to the database

- Next, connect to PostgreSQL and run a few simple SQL queries. To start, **open a new terminal session** (e.g., open a new SSH session). Since PostgreSQL is still running in the foreground in your first terminal session, we'll let it continue running.
- In your new terminal session, connect to the "postgres" database using the "psql" command-line client:

```
$HOME/pgsql/bin/psql postgres
psql (9.1.3)
Type "help" for help.

postgres=#
```

Preparation—Connect to the database

- Psql is a simple command-line tool for executing SQL queries and examining query results -- it will be useful for testing the correctness of your changes. The "postgres=#" prompt indicates that psql is waiting for you to enter a command which will be executed in the "postgres" database.
- Next, execute a simple SQL command to create a new database table:

```
postgres=# create table test (a int, b int);
CREATE TABLE
```

- Remember that every SQL query must be terminated with a semi-colon (";").

Preparation--Stopping Postgres

- To exit psql, you can use "Ctrl+D" or the special command "\q".
- To shutdown the PostgreSQL server, you can also use the pg_ctl command:

```
/home/postgres/pgsql/bin/pg_ctl -D /home/postgres/pgsql/data stop
```

Preparation--Loading Test Data

- You can use the following commands to create a new database name and import the data with these commands:

```
$HOME/pgsql/bin/psql -p <port> postgres -c 'CREATE DATABASE
similarity;'
$HOME/pgsql/bin/psql -p <port> -d similarity -f
/home/postgres/similarity_data.sql
```

- Test Data: similarity_data.sql

Preparation--Source Code

- This assignment requires modifying the implementation of the PostgreSQL query executor. You should familiarize yourself with the following files to understand the implementation of the executor:
  - src/backend/executor/execMain.c
  - src/backend/executor/execProcnode.c
  - src/backend/executor/execScan.c
  - src/backend/executor/execTuples.c
  - src/backend/executor/nodeNestloop.c
  - src/backend/utils/fmgr/funcapi.c
  - src/include/catalog/pg_proc.h
  - src/include/executor/nodeNestloop.h
  - src/include/nodes/execnodes.h

Preparation--Source Code

- Of these files, nodeNestloop.c and funcapi.c are the primary files you will need to modify (although modifications to other files are also necessary).
- The following functions are worth particular attention:
  - levenshtein_distance() in funcapi.c: This is the function that is called to compute the Levenshtein distance of two strings. Code already exists to get the two PostgreSQL strings to C strings. You will have to implement this function and return the correct Levenshtein distance value.
  - jaccard_index() in funcapi.c : This is the function that is called to compute the Jaccard index of two strings. Code already exists to get the two PostgreSQL strings to C strings. You will have to implement this function and return the correct Jaccard index value.
  - ExecNestLoop() in nodeNestloop.c: This is the main loop for the nested loop join. You will have to modify this function to implement block nested loop join.

Preparation--Source Code

- Understanding the existing code is intended to be part of the assignment. You can add and manage any new data structures that you need, and you can add elog() statements at any point to output a message to the log file, using the same syntax as printf():

```
elog(LOG, "Two plus three is %d", 2+3);
```

Preparation—Some modificatons

- In the file “src/backend/utils/fmgr/funcapi.c” add the following two functions:
- And the main work of this project(Part I) is to implement these two functions

```
Datum levenshtein_distance(PG_FUNCTION_ARGS)
{
        text    * str_01 = PG_GETARG_DATUM(0);
        text    *txt_02 = PG_GETARG_DATUM(1);
        int32 result=1;
        PG_RETURN_INT32(result);
}
Datum jaccard_index (PG_FUNCTION_ARGS)
{
        text    *str_01 = PG_GETARG_DATUM(0);
        text    *txt_02 = PG_GETARG_DATUM(1);
        float4 result=1;
        PG_RETURN_FLOAT4(result);
}
```

Preparation—Some modificatons

- In the file “src/include/catalog/pg_proc.h” add the following two sentences:

- DATA(insert OID = 4375 (levenshtein_distance PGNSP PGUID 12 1 0 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 = 4376 (jaccard_index PGNSP PGUID 12 1 0 0 f f f t f i 2 0 700 "25 25" _null_ _null_ _null_ _null_ jaccard_index _null_ _null_ _null_));
- DESCR(" jaccard index ");

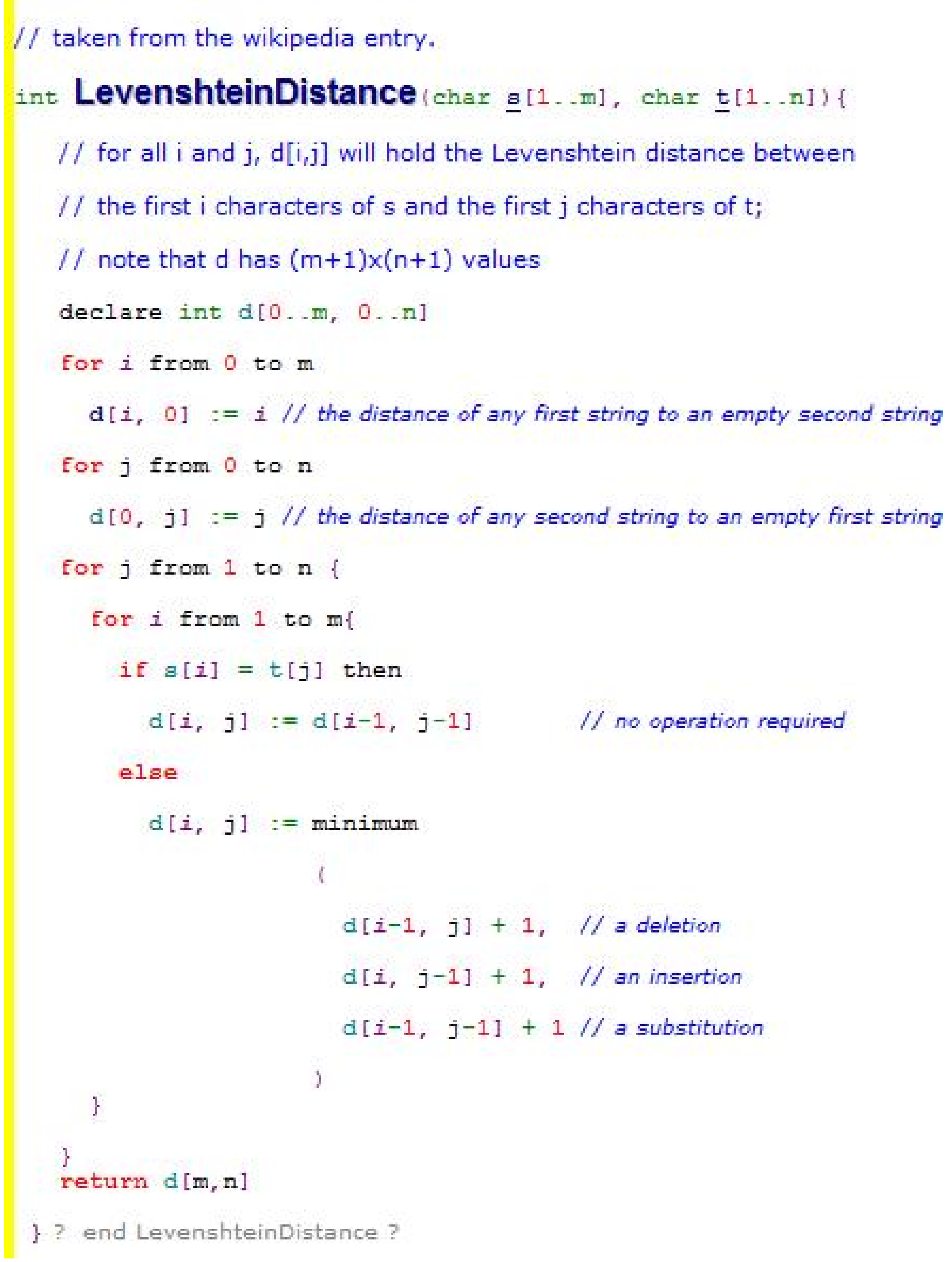
Part 1: Similarity Join Assignment Description

- In this assignment, you will modify PostgreSQL to add two similarity calculations for string fields: **Levenshtein Distance** and **Jaccard Index**. Code has already been added to PostgreSQL which added two functions, levenshtein_distance(s1, s2) and jaccard_index(s1, s2). The function bodies are empty and you will implement the two functions.

- **Note:** For both comparisons, make case INSENSITIVE comparisons.
- **Note:** You may assume all strings are less than 100 characters long, and contain only ASCII characters.

Levenshtein Distance

- The Levenshtein distance of two strings is the minimum number of edits required to transform one string to another. An "edit" can be an insertion of a single character, a deletion of a single character, and a substitution of a single character. This notion of an "edit" distance can be used to compare the similarity of two strings. If two strings have a small Levenshtein distance, then they are probably similar.
- The most common way to implement this distance value of two strings is to use dynamic programming. The pseudocode for the algorithm is:



```
// taken from the wikipedia entry.
int LevenshteinDistance(char s[1..m], char t[1..n]){
   // for all i and j, d[i,j] will hold the Levenshtein distance between
   // the first i characters of s and the first j characters of t;
   // note that d has (m+1)x(n+1) values
   declare int d[0..m, 0..n]
   for i from 0 to m
     d[i, 0] := i // the distance of any first string to an empty second string
   for j from 0 to n
     d[0, j] := j // the distance of any second string to an empty first string
   for j from 1 to n {
     for i from 1 to m{
       if s[i] = t[j] then
         d[i, j] := d[i-1, j-1]       // no operation required
       else
         d[i, j] := minimum
                    (
                      d[i-1, j] + 1,  // a deletion
                      d[i, j-1] + 1,  // an insertion
                      d[i-1, j-1] + 1 // a substitution
                    )
     }
   }
   return d[m,n]
 } ? end LevenshteinDistance ?
```

Levenshtein Distance

- Here are examples of some Levenshtein distance values. You can check your results with these.

```
similarity=# select levenshtein_distance('sunday', 'sunday');
levenshtein_distance
----------------------
0
(1 row)
similarity=# select levenshtein_distance('sunday', 'Monday');
levenshtein_distance
----------------------
2
(1 row)
similarity=# select levenshtein_distance('sunday', 'saturday');
levenshtein_distance
----------------------
3
(1 row)
```

Jaccard Index

- The Jaccard index is another way to represent the similarity of two strings. Specifically, the Jaccard index is computed over the two sets of bigrams of the two strings. The exact equation for the Jaccard index is:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

- Given two sets A and B, the Jaccard index is the size of the set intersection of A and B, divided by the size of the set union of A and B. Remember, a set does not have duplicates.

Jaccard Index

- In this assignment, each string will be represented as a set of bigrams. A bigram of a string is a sequence of two consecutive characters from the string. In order to distinguish the bigrams at the beginning and end of the string, the '$' character will pad the string at the beginning and the end.
- To compute the Jaccard index of two strings, you first compute the sets of bigrams for each string. Then, you compute the set intersection and set union of the two sets. The Jaccard index is the ratio between the size of the intersection and size of the union.
- For example, the set of bigrams of the string "apple" is:

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

Jaccard Index

- And the set of bigrams of the string "apply" is:

B = {a, ap, pp, pl, ly, y}

- The size of the set intersection of the two sets is:

|{$a, ap, pp, pl}| = 4

- And the size of the set union of the two sets is:

|{a, ap, pp, pl, le, e, ly, y$}| = 8

- Therefore, the Jaccard index is:

J(A, B) = |{$a, ap, pp, pl}|/|{$a, ap, pp, pl, le, e$, ly, y$}| = 4/8 = 1/2

Implementation and Testing

- A better way to join on this field would be to use a similarity metric. Using either levenshtein_distance() or jaccard_index() will allow the join condition to match more strings, and produce more results. The following queries are expected to return more results than the query above:

```
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) >  .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;
```

Part 2: Block Nested Loop Join Description (additional)

- In this assignment, you will also modify PostgreSQL to add the block nested loop optimization for nested loop joins. The pseudocode for nested loop join is:

```
for (each tuple i in outer relation)
  {  for (each tuple j in inner relation)
  {
    if (join_condition(tuple i, tuple j) is true)
      emit (tuple i, tuple j)
    else
      continue
  }
}
```

Part 2: Block Nested Loop Join Description

- However, this algorithm will scan the inner table as many times as there are rows in the outer tuple. Block nested loop join can improve this by scanning the inner table for a block of outer tuples. By scanning the inner table for each BLOCK of outer tuples, fewer scans will be performed. The pseudocode for block nested loop join is:

```
for (each block of tuples B in outer relation)
  {  for (each tuple j in inner relation) {
    for (each tuple i in block B) {
      if (join_condition(tuple i, tuple j) is true)
        emit (tuple i, tuple j)
      else
        continue
    }
  }
}
```

Part 2: Block Nested Loop Join Description

- You will implement the block nested loop join algorithm for this assignment. You will add functionality to read a block of outer relation tuples, and join with the inner relation.
- Your implementation of block nested loop join should not change the results of the join for different block sizes. It is only an optimization which does not alter the semantics of the join. However, rows in the result may be out of order, and you should take that into account when testing your code.

Tuple Tables

- The PostgreSQL executor often has to pass tuples around between operators. To make this more efficient, the executor uses something called a TupleTable. The tuple table consists of a collection of slots; each slot has type TupleTableSlot.   This allows any tuple to be fully materialized only once, and the query operators can just pass along TupleTableSlots (which can be thought of as pointers). This makes the execution more efficient since you don't have to always copy tuples to create a new result tuple.

Tuple Tables

- In order to implement block nested loop join, you will have to create extra TupleTableSlots for the block of outer tuples. During the execution of the join, you will also have to copy tuples for block nested loop join, since the simple nested loop join never had to re-examine a tuple once it was read.
- You should read execTuples.c carefully to figure out which functions you will need in order to implement block nested loop join. In particular, you should pay attention to ExecInit*TupleSlot() and ExecCopySlot().

Rebuilding Postgres with your new code

- Use the following command to rebuild and install a modified version of postgres that uses your new modified code:

```
make
make install
```

Experiments--Similarity Join Experiments

- The three experimental similarity queries we are interested in are:

levenshtein

```
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;
```

jaccard

```
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;
```

Experiments--Similarity Join Experiments

- The three experimental similarity queries we are interested in are:

combined

```
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;
```

Experiments--Similarity Join Experiments

- To generate the three result files (levenshtein.txt, jaccard.txt, combined.txt), run the following three commands from the command prompt:
  - **levenshtein.txt:**

```
$HOME/pgsql/bin/psql -p <port> 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
```

  - **Use the similar commands to generate jaccard.txt and combined.txt files.**

Experiments--Similarity Join Experiments

- We are also interested in the performance of the different similarity computations.
- To measure the timing of the similarity comparisons, you need to toggle on the timing option in psql:

```
similarity-# \timing
Timing is on.
```

- When the timing is turned on, at the bottom of every query, you will see timing information like:

```
Time: 1234.033 ms
```

Experiments--Similarity Join Experiments

- We are also interested in the performance of the different similarity computations.
- To measure the timing of the similarity comparisons, you need to toggle on the timing option in psql:

```
similarity-# \timing
Timing is on.
```

- When the timing is turned on, at the bottom of every query, you will see timing information like:

```
Time: 1234.033 ms
```

Experiments--Block Nested Loop Join Description (additional)

- We are interested in the performance gain of block nested loop join. If the query timing information is not on, turn it on with:

```
similarity-# \timing
Timing is on.
```

- You will be running the experiments with several different nested loop join block sizes. The 6 different block sizes you will experiment with are: 1, 2, 8, 64, 128, 1024
- You should report the performance of PostgreSQL using different block sizes for the block nested loop join. Run the following:

Experiments--Block Nested Loop Join Description (additional)

- 1.Run this query TWICE:

```
SELECT count(*) FROM restaurantaddress ra, restaurantphone rp WHERE
ra.name = rp.name;
```

- 2.Record the timing information of the SECOND run:

```
Time: 1234.033 ms
```

- 3.Record the number of shared blocks read of the SECOND run (from the statistics in the log file) and briefly explain/justify the performance numbers you saw from the different block sizes.

```
1 100.11
2 90.22                                        block_performance.txt
8 80.33
```

These performance numbers make sense because ...

实验流程

- Part I
  - 准备Linux开发环境，你可以在Windows下的虚拟机（ virtualbox ）里安装，也可以直接安装Linux系统。进行必要的配置。
  - 下载Postgres安装文件源码，安装前面的介绍进行编译，安装。
  - 理解Postgres的源码，对于Part I来说，你主要的工作就是实现那两个函数。（可以利用附录介绍的Source Insight查看源码）
  - 修改源码，并调试（可利用GDB）。
  - 完成后重新编译，安装Postgres。
  - 运行实验测试例子，记录运行花费时间。
  - 撰写实验报告。
- Part II类似，但Part II要求较高， 需要对Postgres内部实现流程有较深入理解，请同学依据自身情况考虑是否要实现Part II。

实验说明

- 本次实验可以由1~2人一组完成。
- 实验截止时间：6月19日
- 实验报告提交到elearning
- 实验完成后请在上机课上给助教现场检查。
- Part I每组都必须完成， Part 2为可选任务，完成Part 2的同学将给予加分。从往届同学的完成情况来看，绝大部分同学只完成了Part I。
- 实验一定要本组独立完成。
- 评分标准：
  - 现场运行情况，代码质量，实验报告。

实验报告

- 本次实验提交内容包括：
- 代码
  - 实验中修改的代码文件（不用整个工程）。
  - 运行脚本：安装Postgres，实验测试时的命令脚本。（以自身情况而定）。
  - 结果文件： levenshtein.txt, jaccard.txt, combined.txt（Part I）；
- 实验报告：
  - 1. 对实验修改的源代码进行解释。
  - 2. 对PostgreSQL数据库内部的实现过程进行说明。（依自己的理解程度而定）
  - 3. Part II 中实现过程说明及实验结果性能比较。（可选）

实验参考

- 本次实验主要参考UC Berkeley数据库课程
- http://www-inst.eecs.berkeley.edu/~cs186/archives.html
- https://sites.google.com/a/cs.berkeley.edu/cs186-s12/ （要翻墙）
- https://github.com/cs186-fa12/fa12

Helpful Material

Debugging Postgres With GDB

- To debug PostgreSQL using gdb, first start psql. Next, in a separate shell window, you first to determine the PID of the backend process.

```
Postgres=#select pg_backend_pid();
pg_backend_pid
------------------------
             15681
1 (row)
```

- Next, attach to the backend "postgres" process using gdb:

```
gdb ~/pgsql/bin/postgres 15681
```

Debugging with GDB

- 一、用 break 命令来设置断点,介绍常用方法：
  - break <function>： 在进入指定函数时停住。
    - (gdb) break main
      Breakpoint 1 at 0x8050718: file gdb_sample_1.c, line 8.
  - break <linenum>： 在指定行号停住。
    - (gdb) break 19
    - Breakpoint 2 at 0x8050773: file gdb_sample_1.c, line 19.
  - break ... if <condition>：可以是上述的参数， condition 表示条件，在条件成立时停住。
    - 比如在循环境体中，可以设置 break if i=100 ，表示当 i 为 100 时停住程序。
- 二、用delete 命令删除断点
  - (gdb) delete 2
- 三、用info break命令来查看当前所有断点
  - (gdb) info break
  - Num Type Disp Enb Address What
  - 1 breakpoint keep y 0x08050718 in main at gdb_sample_2.c:8
  - breakpoint already hit 1 time

Debugging with GDB

- 一、使用run命令从程序开始
  - (gdb) run
  - Starting program: /home/hw/gdb_sample_1

- 二、使用next命令从断点开始按步调试程序
  - (gdb) n
  - 9 int y = 3;
  - (gdb) n
  - 10 int z = x + y;

- 三、使用step进入当前被调用的函数
  - (gdb) s
  - f (a=2, b=3, c=5) at gdb_sample_1.c:28
  - 28 a = a * 2;

Debugging with GDB

- 四、finish 结束执行当前函数，显示其返回值（如果有的话）。
  - (gdb) finish
  - Run till exit from #0 f (a=2, b=3, c=5) at gdb_sample_1.c:28
  - 0x08050748 in main () at gdb_sample_1.c:11
  - 11   int w = f(x, y, z);
  - Value returned is $2 = 17
- 五、continue 从断点开始继续执行

Debugging with GDB

- 一、backtrace打印当前的函数调用栈的所有信息。如：
  - (gdb) backtrace
  - #0 0x08050783 in g (p=0x0) at gdb_sample_1.c:32
  - #1 0x0805076d in f (p=0x0) at gdb_sample_1.c:25
  - #2 0x08050755 in main () at gdb_sample_1.c:17
- 二、如果要查看某一层的信息，需要在切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前栈，如果要查看栈下面层的详细信息，首先要做的是切换当前栈。
  - frame <n>：n 是一个从 0 开始的整数，是栈中的层编号。比如： frame 0，表示栈顶， frame 1，表示栈的第二层。
    - (gdb) frame 2
    - #2 0x08050755 in main () at gdb_sample_1.c:17
    - 17   f(px); /* Should cause a run-time error */
  - up <n>：表示向栈的上面移动 n 层，如果不打 n，这时表示向上移动一层。
  - down <n>：表示向栈的下面移动 n 层，如果不打 n，这时表示向下移动一层。

Debugging with GDB

- 一、quit退出调试
- 二、ps 进程查看命令
  - ps
  - PID TTY   TIME CMD
  - 27894 pts/19   0:00 ps
  - 25028 pts/19   0:00 bash
  - 27874 pts/19   0:06 gdb_samp

Postgres 简要介绍--PostgreSQL的**Backend**目录

- Bootstrap-通过initdb创建最初的数据库模板
- Main-将控制转到postmaster或postgres
  - 检查进程名(argv[0])和各种标志，然后将控制转到postmaster或postgres
- Postmaster-控制postgres服务器程序启动/终止
  - 创建共享内存，然后进入一个循环等待连接请求。当一个连接请求到达时，启动一个postgres后台服务进程，将连接转给它
- Libpg-后台服务器libpg库函数
  - 处理与客户进程间的通讯
- Tcop-将请求分派到合适的模块
  - 这是 postgres后台服务进程的主要处理部分，它调用 parse, optimizer, executor, commands 中的函数。
- Parse-将SQL查询转化为查询树
  - 将来自 libpg的SQL查询转换 命令形式的结构供 optimizer/executor 或 commands 使用。首先对 SQL 语句进词法分析，转换为关键字，标识符和常量，然后进行语法分析。语法分析生成命令形式结构来表示查询的组成。然后这个命令形式结构被分离、检查和传送给 commands中的处理函数，或者转换结点链表供 optimizer 和 executor处理
- Optimizer-创建查询路径和查询计划
  - 使用parser 的输出来为executor生成优化了的查询计划。

Postgres 简要介绍--PostgreSQL的**Backend**目录

- Optimizer/path-使用parser的输出创建查询路径
- Optimizer/plan-优化path输出
- Optimizer/prep-处理特殊查询计划
- Optimizer/util-优化器支持函数
- Executor-执行来自optimizer 的复杂的节点形式查询计划
  - 处理select, insert, update, delete 语句。处理这些语句的操作包括堆扫描、索引扫描、排序、连接表、分组、计算集函数和唯一性处理
- Commands- 不需要executor执行的命令
  - 执行不需要复杂处理的SQL命令，包括vacuum, copy, alter, create table, create type 等命令。一般对命令先做一些处理，再调用catalog目录下的底层函数来做实际的工作。
- Catalog-系统目录操作
  - 包含用于操作系统表和系统目录的函数、表、索引、过程、运算符、类型和集函数的创建和操纵函数在这里可以找到。它们都是底层函数，通常由上层将用户请求格式化 为预定义格式的函数调用
- Storage-管理各种类型存储系统
  - 支持服务器以统一 方式存取资源
- Access-各种存取方法
  - 支持堆、索引和事务对数据的存取

Postgres 简要介绍--PostgreSQL的**Backend**目录

- Nodes-创建操纵节点和链表
  - postgreSQL将SQL查询存储到称为节点的结构中。节点是通用的容器，它包含一个类型字段和一个与类型有关的数据字段，节点通常串成链表。节点和链表广泛用于parser,optimizer和executor中用于请求数据。

- 常用的数据类型/SQL语句的解释和执行
  - Basic nodes definition (src/include/nodes/nodes.h)
  - SQL parsed struct (src/include/nodes/nodes.h)
  - List定义 (src/include/nodes/pg_list.h)
  - List实现 (src/backend/nodes/list.c)
  - Postgres 运行入口文件 (src/backend/tcop/postgres.c)
  - Utility Stmt 运行文件 (src/backend/tcop/utility.c)
  - SQL analyze and rewrite (src/backend/parser/analyze.c)

Postgres 简要介绍—数据类型

- PostgreSQL 数据用一种非常简单的形式实现 类似 C++的继承， 在nodes.h这样定义：
  - typedef struct Node
  - {
  - NodeTag type;
  - } Node;
- 一般SQL语句经过在parsenodes.h文件中的parser解析后有一个对应的struct来跟踪：
  - 假设有一个create talbe 的sql (create table test (name varchar(20), passw varchar(20)))
  - 它会被解析到如下的structure:

```
–   typedef struct CreateStmt
–   {
–       NodeTag          type;
–       RangeVar          *relation;          /* relation to create */
–       List             *tableElts;          /* column definitions (list of ColumnDef) */
–       List             *inhRelations;         /* relations to inherit from (list of inhRelation) */
–       List             *constraints;         /* constraints (list of Constraint nodes) */
–       List             *options;            /* options from WITH clause */
–       OnCommitAction oncommit;              /* what do we do at COMMIT? */
–       char             *tablespacename;      /* table space to use, or NULL */
–   } CreateStmt;
```

PG的很多struct的第一个元素都是NodeTag,这样它在传递指针变量时，就可以简地把参数设置成：Node*。其实就是所有的struct都继承了Node这个struct。在nodes.h 中, CreateStmt的type 的值就是T_CreateStmt。

Postgres 简要介绍--数据类型

- PG中还大量使用了链表类型List，在pg_list.h中定义：
  - typedef struct List
  - {
  - NodeTag type; /* T_List, T_IntList, or T_OidList */
  - int length;
  - ListCell *head;
  - ListCell *tail;
  - } List;

- 可以看到，List 的定义是基于基类Node 来进行的。常用的List操作函数有：
  - 取List第一个元素 > ListLCell * list_head(List *l)
  - 得到List 的元素个数 > int list_length(List *l)
  - 遍历List > foreach(cell,l){… …}
  - 其它操作参考pg_list.h 和 list.c

Postgres 简要介绍--SQL语句的解释和执行

- 所有的SQL都会先解析成与之相对就的struct： select 会解析到SelectStmt。具体过程如下：
  - 在postgres.c 的839行可以看到调用这样一个函数，parsetree_list = pg_parse_query(query_string)，它只是做了基本的分析。
  - 在930行调用函数querytree_list = pg_analyze_and_rewrite(parsetree, query_string,NULL, 0)， 对SQL进行重写，这些rewrite 比较复杂，是由一系列的transform 函数完成的,具体可以查看analyze.c
  - 把所有的执行语句都转换成Query结构体，再交给优化器进行优化和路径选择。
  - 然后进入PortalRun 入口执行查询计划。

- 在这个过程中也可以使用日志Log的方法把执行过程记录下来
  - ereport(LOG, (errmsg(“begin create: %s ”,((CreateStmt *)parsetree) ->relation->relname)));
  - 当创建Table test时，就会在日志文件中记录：LOG: begin create:test