# MSiA-413 Data Management and Information Processing

### Homework 6: Advanced JOINs and set operations; Accessing large real-world databases

Name 1: Sam Swain

NetID 1: sms5736

Name 2: Yaasir Ahmed

NetID 2: yap80676

# Instructions

You should submit this homework assignment via Canvas. Acceptable formats are word files, text files, and pdf files. Paper submissions are not allowed and they will receive an automatic zero.

As explained during lecture and in the syllabus, assignments are done in groups. The groups have been created and assigned. Each group needs to submit only one assignment (i.e., there is no need for both partners to submit individually the same homework assignment).

Each group can submit solutions multiple times (for example, you may discover an error in your earlier submission and choose to submit a new solution set). We will grade only the last submission and ignore earlier ones.

Make sure you submit your solutions before the deadline. The policies governing academic integrity, tardiness and penalties are detailed in the syllabus.

**EntertainmentAgency.sqlite Database (60 points)**

This should be the original database, without the modifications you made in the previous assignment.

1. **(10 points)** Find the EntertainerID and stage name of the entertainers that have no engagements. You **must** use the EXCEPT set operator for full credit.

**select en.EntertainerID, EntStageName**

**from Entertainers en**

**LEFT join Engagements eg on en.EntertainerID=eg.EntertainerID**

**EXCEPT**

**select en.EntertainerID, EntStageName**

**from Entertainers en**

**LEFT join Engagements eg on en.EntertainerID=eg.EntertainerID**

**WHERE EngagementNumber NOT NULL**

**Output:**

|  |  |
| --- | --- |
| 1009 | Katherine Ehrlich |

1. **(10 points)** Find the EntertainerID **and stage name** of the entertainers that have no engagements. Your answer must be a single query with no subqueries. You **must not** directly use the result of question (1) above.

**select en.EntertainerID, EntStageName**

**from Entertainers en**

**left join Engagements eg on en.EntertainerID=eg.EntertainerID**

**where EngagementNumber is null**

**Output:**

|  |  |
| --- | --- |
| 1009 | Katherine Ehrlich |

1. **(10 points)** Find the agent ID and full name (first and last names concatenated, with a space in between) of the agents that procured no engagements. Your answer must be a single query with no subqueries.

**select a.AgentID, a.AgtFirstName ||" "|| a.AgtLastName Name**

**from Agents a**

**left join Engagements e on e.AgentID=a.AgentID**

**where EngagementNumber is NULL**

**Output:**

|  |  |
| --- | --- |
| 9 | Daffy Dumbwit |

1. **(10 points)** For all customers that have less than 10 engagements, list the customer ID, full name (single string containing the customer’s first and last name with a space in between), and number of engagements, in ascending order of number of engagements. Your answer must be a single query with no subqueries.

**select c.CustomerID, CustFirstName ||" "|| CustLastName, count(engagementnumber) eng\_count**

**from Customers c**

**left join Engagements e on c.CustomerID=e.CustomerID**

**group by c.CustomerID**

**having eng\_count < 10**

**order by eng\_count**

**Output:**

|  |  |  |
| --- | --- | --- |
| 10008 | Darren Gehring | 0 |
| 10011 | Joyce Bonnicksen | 0 |
| 10013 | Estella Pundt | 6 |
| 10003 | Peter Brehm | 7 |
| 10007 | Liz Keyser | 7 |
| 10012 | Kerry Patterson | 7 |
| 10015 | Carol Viescas | 7 |
| 10001 | Doris Hartwig | 8 |
| 10005 | Elizabeth Hallmark | 8 |
| 10009 | Sarah Thompson | 8 |
| 10006 | Matt Berg | 9 |

1. **(10 points)** Write the query to find the number of male and female members (separate counts for each gender) for each entertainer. The output table should have three columns named EntertainerID, Gender, and GenderCount. The query **must** use the UNION operator.

**select EntertainerID, Gender, count(Gender) gender\_count**

**from Members**

**NATURAL join Entertainer\_Members**

**where gender == "M"**

**group by EntertainerID**

**union**

**select EntertainerID, Gender, count(Gender) gender\_count**

**from Members**

**NATURAL join Entertainer\_Members**

**where gender == "F"**

**group by EntertainerID**

**Output:**

|  |  |  |
| --- | --- | --- |
| 1001 | F | 3 |
| 1002 | F | 1 |
| 1002 | M | 1 |
| 1003 | F | 1 |
| 1003 | M | 5 |
| 1005 | F | 1 |
| 1005 | M | 2 |
| 1006 | F | 1 |
| 1006 | M | 3 |
| 1007 | F | 3 |
| 1007 | M | 2 |
| 1008 | F | 1 |
| 1008 | M | 4 |
| 1009 | F | 1 |
| 1010 | F | 4 |
| 1011 | F | 1 |
| 1012 | F | 1 |
| 1013 | F | 2 |
| 1013 | M | 2 |

1. **(10 points)** You want to classify each entertainer as follows:

* Super Band (if it has more than 10 engagements)
* Regular Band (if it has more than 7 but no more than 10 engagements)
* Support Band (if it has at least one engagement, but no more than 7), and
* Amateur Band (if it has no engagements)

Write the query that makes this classification and returns the class of the entertainer (on an output column named BandRank), the entertainer’s stage name, and the number of engagements, with the entertainers appearing in descending rank (i.e., super bands first, followed by regular bands, then support bands, and amateurs at the bottom). Your answer must be a single query with no subqueries.

**select**

**case**

**when count(eg.EngagementNumber) > 10 then "Super Band"**

**when count(eg.EngagementNumber) > 7 and count(eg.EngagementNumber) <= 10 then "Regular Band"**

**when count(eg.EngagementNumber) > 0 and count(eg.EngagementNumber) <= 7 then "Support Band"**

**else "Amateur Band"**

**end BandRank,**

**EntStageName,**

**count(eg.EngagementNumber) NumEng**

**from Entertainers en**

**left join Engagements eg on en.EntertainerID=eg.EntertainerID**

**group by en.EntertainerID**

**order by NumEng desc**

**Output:**

|  |  |  |
| --- | --- | --- |
| Super Band | Country Feeling | 15 |
| Super Band | Carol Peacock Trio | 11 |
| Super Band | Caroline Coie Cuartet | 11 |
| Regular Band | JV & the Deep Six | 10 |
| Regular Band | Modern Dance | 10 |
| Regular Band | Jim Glynn | 9 |
| Regular Band | Saturday Revue | 9 |
| Regular Band | Coldwater Cattle Company | 8 |
| Regular Band | Julia Schnebly | 8 |
| Support Band | Topazz | 7 |
| Support Band | Jazz Persuasion | 7 |
| Support Band | Susan McLain | 6 |
| Amateur Band | Katherine Ehrlich | 0 |

# **Yelp Database (yelp) - General Instructions for Postgres**

In this part of the assignment you will write queries on a large, real-world dataset stored in a Postgres database server. To connect to this server you will use your MSiA credentials. If you have trouble logging into the database server, please contact the MSiA systems administrator via Slack or email and cc the instructor. To connect you'll have to be on the main Northwestern network, i.e., either be on campus or connect through the NU VPN. You can find instructions for setting up the NU VPN at: <http://www.it.northwestern.edu/oncampus/vpn/>.

After you get on the NU network, open Remote Desktop and use your NetID credentials (or mcc\NetID) to connect to MSiA’s remote desktop (e.g., ts2.lab.analytics.northwestern.edu). If you get a notice that the certificate cannot be verified, you can simply click “Continue” and proceed. Once you are in Remote Desktop, you can connect to the yelp database on the Postgres server either through a graphical user interface, or a command-line terminal.

# Option A: Graphical User Interface

Start the “pgAdmin 4 v4” application and create a new server connection named yelp on the pg server group, provide pg as the host name and your NetID credentials, similar to the pictures below, and Save it.

Graphical user interface, text

Description automatically generated Graphical user interface

Description automatically generated

Graphical user interface, application

Description automatically generatedNow, you can connect to the yelp database on Postgres and start issuing queries to it. Selecting the yelp database and clicking on the Query Tool (figure at right) opens a SQL editor in which you can write SQL queries.

To issue queries, you first need to be able to examine the schema, table constraints, and indexes that have been created on the database. The navigational panel on the left side of pgAdmin together with the Dependents and Properties tabs can provide this information. An example is shown in the picture in the next page.

In this example, selecting the yelp 🡪 Schemas 🡪 public 🡪 Tables options on the left-side navigational panel provides the list of all tables in the yelp database. Further navigating through a table provides a list of all columns in that table, e.g., Tables 🡪 user 🡪 Columns. The Dependents tab for a particular column shows the constraints that have been defined for that column, e.g., for the case of user.id, the Dependents tab reports that it is a primary key for table user, and a foreign key in tables elite\_years, friend, review, and tip.

Graphical user interface, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Similarly, the Properties tab shows the definition of a particular column (e.g., data type and definition—e.g., user.id is a varchar(22), whether the column is a primary or foreign key, whether NULL is allowed, and the default value. An example is shown at the picture on the right.

All this information can also be retrieved through SQL queries issued against the tables that Postgres implements. In particular, one can retrieve the list of relations in the yelp database schema by issuing the query below. A snapshot of pgAdmin after executing this query appears in the next page:

SELECT table\_name, table\_type

FROM information\_schema.tables

WHERE table\_schema = 'public';



To execute the query you can press the Run button ( ), highlighted by the red arrow in the picture in the next page. The results of the query appear in the “Data Output” tab. Postgres also allows the user to measure the time it took to execute a query. This can be done by selecting “Timing” in the pull-down menu next to the Explain/Analyze key (see the second red arrow at the picture below). The timing results and all other messages or errors appear in the “Messages” tab.

Graphical user interface, application

Description automatically generated

To retrieve the schema of a table, the user can also issue the query below (e.g., for the user table):

SELECT column\_name, data\_type, character\_maximum\_length, is\_nullable, column\_default

FROM information\_schema.columns

WHERE table\_name = 'user';

Table

Description automatically generated

Similarly, one can retrieve the indexes that have been defined on tables of the schema:

SELECT tablename, indexname, indexdef

FROM pg\_indexes

WHERE schemaname = 'public'

ORDER BY tablename,indexname;

Graphical user interface, text, application

Description automatically generated

And finally, one can retrieve the table constraints for all tables in the schema:

SELECT conrelid::regclass AS table\_from, conname, pg\_get\_constraintdef(oid)

FROM pg\_constraint

WHERE contype IN ('f', 'p') AND connamespace = 'public'::regnamespace

ORDER BY conrelid::regclass::text, contype DESC;

Graphical user interface, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Note that Postgres implements a stricter form of GROUP BY (all columns you SELECT must also appear in the GROUP BY clause). Also note that table “user” must be accessed as public.user because user is a reserved keyword in Postgres.

# Option B: Command-Line Interface

Alternatively, you can use the command line, provided that you have login permissions to the server. Once you are in Remote Desktop, start the “Putty” application, provide as the host name pg.analytics.northwestern.edu (see picture on the right) and click “Open”. If you get an alert that the server’s key is not cached in the registry, click “Yes” to trust the host and store its key in the registry.

A command-line terminal to pg will appear. Login with your NetID credentials, as in the picture below:

Text

Description automatically generated

Then, you can login to the Yelp database by issuing the command “psql -d yelp”. For example:

[nha224@pg ~]$ psql -d yelp

psql (10.5)

Type "help" for help.

yelp=>

You may find the following Postgres commands useful: “\d” presents the list of the relations in the database:

yelp=> \d

List of relations

Schema | Name | Type | Owner

--------+-------------+-------+--------

public | attribute | table | nha224

public | business | table | nha224

public | category | table | nha224

public | checkin | table | nha224

public | elite\_years | table | nha224

public | friend | table | nha224

public | hours | table | nha224

public | photo | table | nha224

public | review | table | nha224

public | tip | table | nha224

public | user | table | nha224

The command “\d tableName” presents a description of table with name tableName, including the column names and types, default values, foreign keys, and available indexes:

yelp=> \d user

Table "public.user"

Column | Type | Collation | Nullable | Default

--------------------+------------------------+-----------+----------+-------------------------

id | character varying(22) | | not null |

name | character varying(255) | | | NULL::character varying

review\_count | integer | | |

yelping\_since | time without time zone | | |

useful | integer | | |

funny | integer | | |

cool | integer | | |

fans | integer | | |

average\_stars | double precision | | |

compliment\_hot | integer | | |

compliment\_more | integer | | |

compliment\_profile | integer | | |

compliment\_cute | integer | | |

compliment\_list | integer | | |

compliment\_note | integer | | |

compliment\_plain | integer | | |

compliment\_cool | integer | | |

compliment\_funny | integer | | |

compliment\_writer | integer | | |

compliment\_photos | integer | | |

**Indexes:**

**"user\_pkey" PRIMARY KEY, btree (id)**

Referenced by:

TABLE "elite\_years" CONSTRAINT "fk\_elite\_years\_user1" FOREIGN KEY (user\_id) REFERENCES "user"(id)

TABLE "friend" CONSTRAINT "fk\_friends\_user1" FOREIGN KEY (user\_id) REFERENCES "user"(id)

TABLE "review" CONSTRAINT "fk\_reviews\_user1" FOREIGN KEY (user\_id) REFERENCES "user"(id)

TABLE "tip" CONSTRAINT "fk\_tip\_user1" FOREIGN KEY (user\_id) REFERENCES "user"(id)

Note that the Yelp database has a table named “user”. However, “user” is a reserved keyword in the SQL standard, and thus also in PostgreSQL (as Postgres’ variant of SQL is known). To access the table named “user” in the Yelp database you need to use “public.user” instead, as in the example below:

SELECT id FROM public.user ORDER BY id LIMIT 10;

Postgres provides a facility to time the execution of queries. Turn on execution timing by issuing the command “\timing”. The database server will reply with “Timing is on.” to notify you that your command succeeded. If you issue the command “\timing” again, timing will turn off.

yelp=> \timing

Timing is on.

yelp=> \timing

Timing is off.

PostgreSQL follows the SQL standard more strictly than many other systems. For example, when a SQL statement contains a GROUP BY clause, each column that is projected in a SELECT clause should have the same value among all rows within each group. For example, the two queries below are correct and will return results:

SELECT **id, name**, COUNT(\*) FROM public.user GROUP BY **id, name LIMIT 10**;

SELECT **id, name**, COUNT(\*) FROM public.user GROUP BY **id LIMIT 10**;

However, the following query will fail with an error, because each group includes many different “id” values:

SELECT **id, name**, COUNT(\*) FROM public.user GROUP BY **name**;

You can exit from the database by typing “\q” or by pressing CONTROL-d.

yelp=> \q

# **Yelp Database (yelp) (40 points)**

The database “yelp” has data from the Yelp business review app (<http://yelp.com/>). It provides access to data from 12 metropolitan areas and 4.7 million reviews of 156,639 businesses. It also includes data on 1.1 million users and 1 million “tips” from these users. The database is about 10.8 GiB. This makes the yelp dataset a medium-sized database: large enough to require a well-engineered database server, but a dwarf compared to truly big data.

The database schema is provided below:

Diagram

Description automatically generated

Note that the position of the linking lines does not directly indicate which columns are linked; there is no such requirement or standard for ER diagrams. You will need to infer which columns are the ones linking the tables.

You will use this database to answer the following questions. Unless otherwise noted, for each question provide:

* **The query you constructed**
* **The output of that query**
* **Any other information requested by the question (e.g., timing results)**

1. **(10 points)** Which state has the most businesses, and how many businesses are there?

**select state, count(id) businesses**

**from business b**

**where state in ('AK', 'AL', 'AR', 'AZ', 'CA', 'CO', 'CT', 'DC', 'DE', 'FL', 'GA',**

**'HI', 'IA', 'ID', 'IL', 'IN', 'KS', 'KY', 'LA', 'MA', 'MD', 'ME',**

**'MI', 'MN', 'MO', 'MS', 'MT', 'NC', 'ND', 'NE', 'NH', 'NJ', 'NM',**

**'NV', 'NY', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC', 'SD', 'TN', 'TX',**

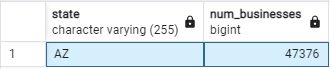
**'UT', 'VA', 'VT', 'WA', 'WI', 'WV', 'WY')**

**group by state**

**order by businesses desc**

**limit 1;**

**Output:**



1. **(10 points)** What is the median number of businesses per state and which state has it? Note: given an ordered set of items, you can consider the median item to be the one at location #items / 2. You do not have to follow the strict mathematical definition that treats odd #items differently from even ones.

**select state, count(state) businesses**

**from business**

**where state in ('AK', 'AL', 'AR', 'AZ', 'CA', 'CO', 'CT', 'DC', 'DE', 'FL', 'GA',**

**'HI', 'IA', 'ID', 'IL', 'IN', 'KS', 'KY', 'LA', 'MA', 'MD', 'ME',**

**'MI', 'MN', 'MO', 'MS', 'MT', 'NC', 'ND', 'NE', 'NH', 'NJ', 'NM',**

**'NV', 'NY', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC', 'SD', 'TN', 'TX',**

**'UT', 'VA', 'VT', 'WA', 'WI', 'WV', 'WY')**

**group by state**

**order by businesses desc**

**LIMIT 1 OFFSET (select count(distinct state)/2**

**from business**

**where state in**

**('AK', 'AL', 'AR', 'AZ', 'CA', 'CO', 'CT', 'DC', 'DE', 'FL', 'GA',**

**'HI', 'IA', 'ID', 'IL', 'IN', 'KS', 'KY', 'LA', 'MA', 'MD', 'ME',**

**'MI', 'MN', 'MO', 'MS', 'MT', 'NC', 'ND', 'NE', 'NH', 'NJ', 'NM',**

**'NV', 'NY', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC', 'SD', 'TN', 'TX',**

**'UT', 'VA', 'VT', 'WA', 'WI', 'WV', 'WY'));**

**Output:**



1. **(10 points)** Find the absolute number and percentage of businesses that have photos in the database, and businesses without any photo. *Hint*: to obtain a floating-point result in SQL arithmetic operations, at least one of the arithmetic operands must be a floating point number.

**select (**

**select count(distinct business\_id) as absolute**

**from photo) absolute\_photos,**

**cast((select count(distinct business\_id)**

**from photo) as float) / count(id) \* 100 percent\_photos,**

**count(distinct id) - (select count(distinct business\_id) as absolute**

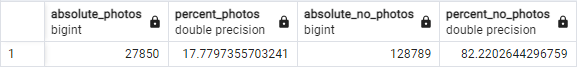
**from photo) absolute\_no\_photos,**

**(1 - cast((select count(distinct business\_id)**

**from photo) as float) / count(id)) \* 100 as percent\_no\_photos**

**from business**

**Output:**



1. **(10 points)** Some businesses are open fewer days of the week than others. Use a common table expression to find airports that are open only once a week and report their business id, name, and hours of operation.

**with**

**airport\_1\_day\_ids as**

**(**

**select c.business\_id**

**from category c**

**join hours h on c.business\_id=h.business\_id**

**where category in ('Airports')**

**group by c.business\_id**

**having count(c.business\_id) = 1**

**)**

**select**

**a.business\_id,**

**name,**

**hours**

**from airport\_1\_day\_ids a**

**join business b on a.business\_id=b.id**

**join hours h on a.business\_id=h.business\_id**

**Output:**

