MLDS-413 Data Management and Information Processing

Homework 6: Advanced JOINs and set operations; PostgreSQL on large real-world databases

Name 1: Ayush Agarwal

NetID 1: scg1143

Name 2: Lowan (Sydney) Li

NetID 2: chr0390

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.

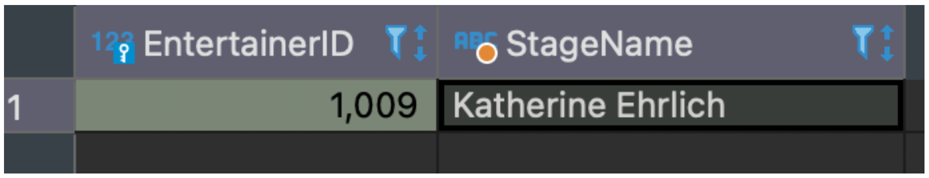
EntertainmentAgency.sqlite Database (60 points)

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 e.EntertainerID as EntertainerID , e.EntStageName as StageName
FROM
Entertainers e
EXCEPT
SELECT e.EntertainerID as EntertainerID , e.EntStageName as StageName
FROM
Entertainers e
inner join
Engagements e2
on e.EntertainerID = e2.EntertainerID
```

Output:

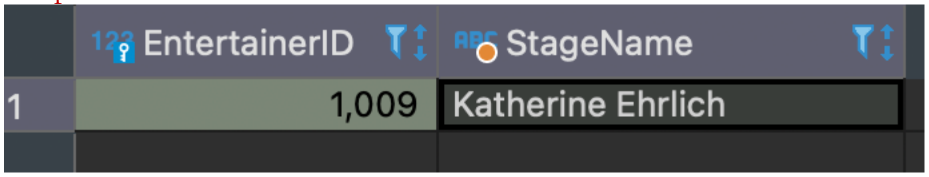


	EntertainerID	StageName
1	1,009	Katherine Ehrlich

2) **(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 e.EntertainerID as EntertainerID , e.EntStageName as StageName
FROM
Entertainers e
left join
Engagements e2
on e.EntertainerID = e2.EntertainerID
where
e2.EngagementNumber is NULL
```

Output:



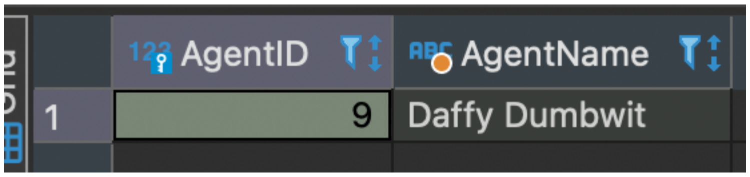
	EntertainerID	StageName
1	1,009	Katherine Ehrlich

3) **(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 as AgentID , a.AgtFirstName || ' ' || a.AgtLastName
FROM
Agents a
```

```
left join
Engagements e
on a.AgentID = e.AgentID
where e.EngagementNumber is NULL;
```

Output:

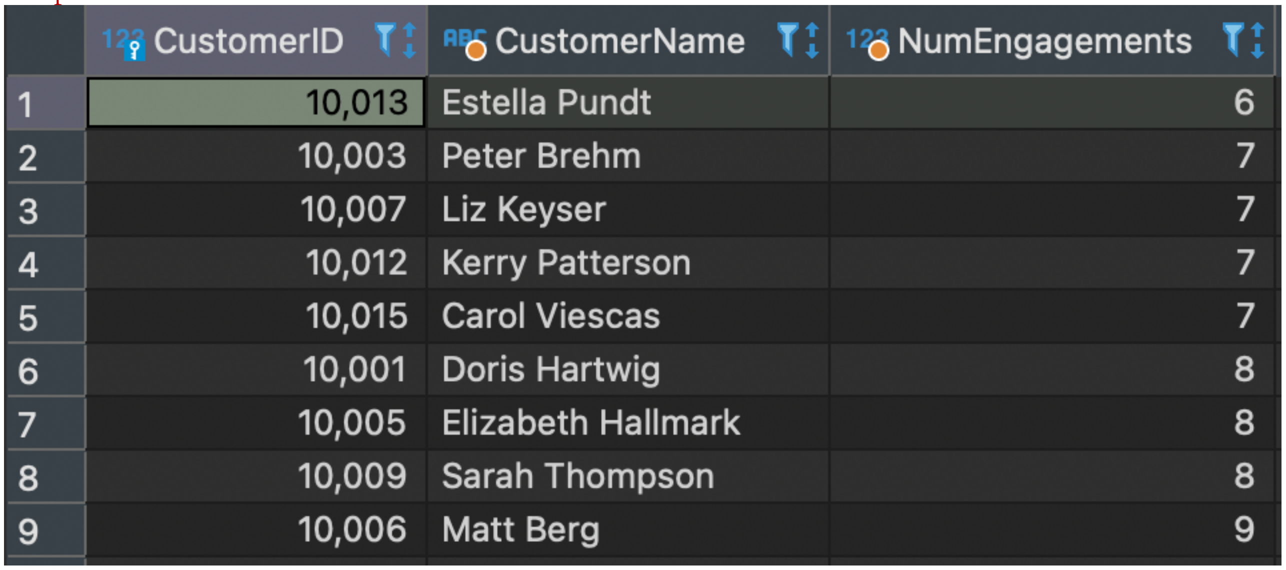


	AgentID	AgentName
1	9	Daffy Dumbwit

4) **(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 as CustomerID , c.CustFirstName || ' ' || c.CustLastName  as CustomerName, COUNT(e.EngagementNumber) as NumEngagements
FROM
Customers c
inner join
Engagements e
on c.CustomerID = e.CustomerID
group by
c.CustomerID
having
NumEngagements < 10
order by
NumEngagements ASC;
```

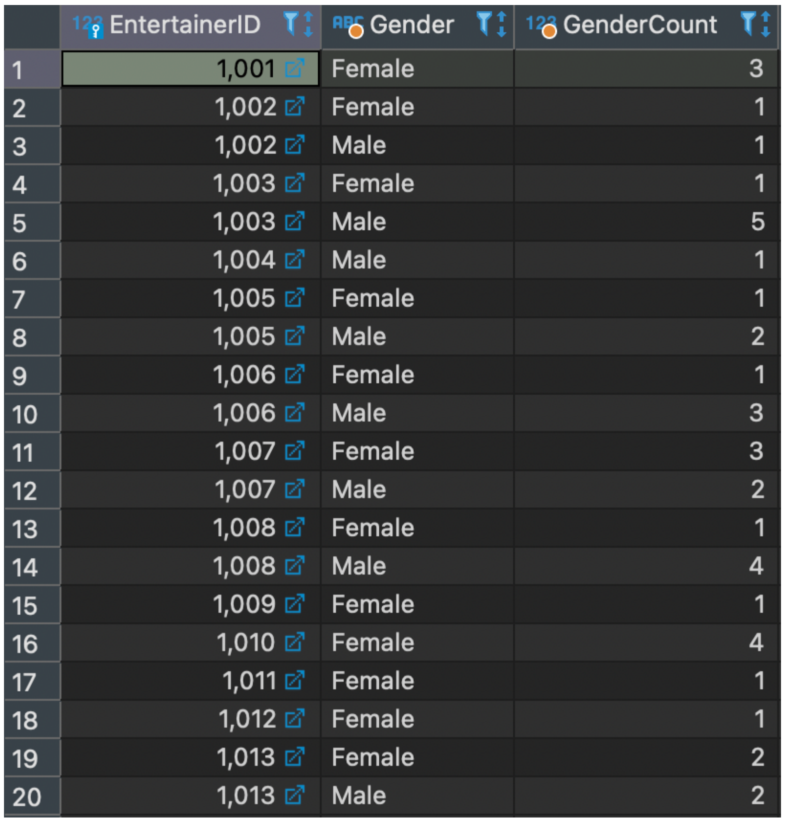
Output:



	CustomerID	CustomerName	NumEngagements
1	10,013	Estella Pundt	6
2	10,003	Peter Brehm	7
3	10,007	Liz Keyser	7
4	10,012	Kerry Patterson	7
5	10,015	Carol Viescas	7
6	10,001	Doris Hartwig	8
7	10,005	Elizabeth Hallmark	8
8	10,009	Sarah Thompson	8
9	10,006	Matt Berg	9

5) **(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 em.EntertainerID as EntertainerID , 'Male' as Gender , COUNT(*) as GenderCount
FROM
Entertainer_Members em
inner join
Members m
on
em.MemberID = m.MemberID
where m.Gender = 'M'
group by
em.EntertainerID
UNION
SELECT em.EntertainerID as EntertainerID , 'Female' as Gender , COUNT(*) as GenderCount
FROM
Entertainer_Members em
inner join
Members m
on
em.MemberID = m.MemberID
where m.Gender = 'F'
group by
em.EntertainerID ;
```



	EntertainerID	Gender	GenderCount
1	1,001	Female	3
2	1,002	Female	1
3	1,002	Male	1
4	1,003	Female	1
5	1,003	Male	5
6	1,004	Male	1
7	1,005	Female	1
8	1,005	Male	2
9	1,006	Female	1
10	1,006	Male	3
11	1,007	Female	3
12	1,007	Male	2
13	1,008	Female	1
14	1,008	Male	4
15	1,009	Female	1
16	1,010	Female	4
17	1,011	Female	1
18	1,012	Female	1
19	1,013	Female	2
20	1,013	Male	2

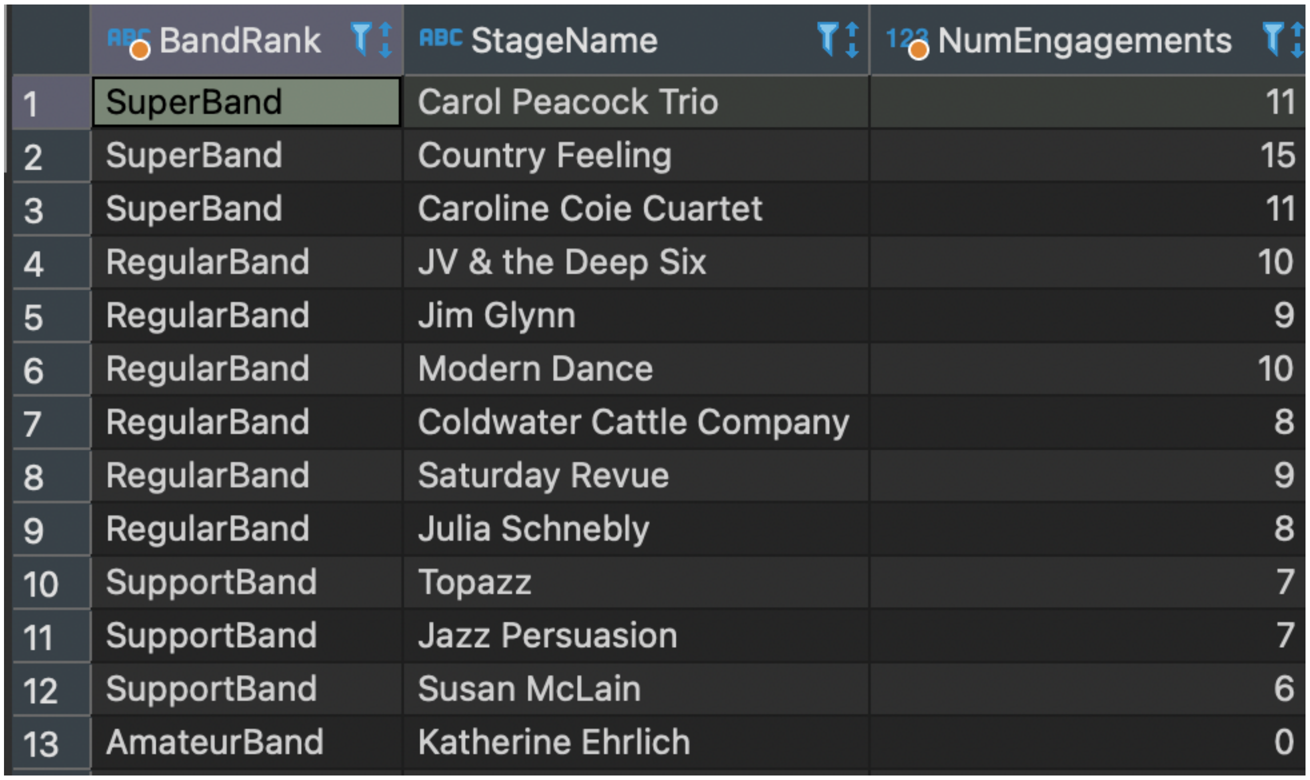
6) **(10 points)** You want to classify each entertainer as follows:

- 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(e2.EngagementNumber) > 10 then 'SuperBand'
                when COUNT(e2.EngagementNumber) > 7  then 'RegularBand'
                when COUNT(e2.EngagementNumber) >= 1 then 'SupportBand'
                else 'AmateurBand'
        END  as  BandRank,  e.EntStageName  as  StageName,  COUNT(e2.EngagementNumber)  as
NumEngagements
FROM
Entertainers e
left join
Engagements e2
on
e.EntertainerID = e2.EntertainerID
group by
e.EntertainerID
order by
        CASE
                when COUNT(e2.EngagementNumber) > 10 then 1
                when COUNT(e2.EngagementNumber) > 7  then 2
                when COUNT(e2.EngagementNumber) >= 1 then 3
                else 4
        END;
```

OUTPUT:



	BandRank	StageName	NumEngagements
1	SuperBand	Carol Peacock Trio	11
2	SuperBand	Country Feeling	15
3	SuperBand	Caroline Coie Cuartet	11
4	RegularBand	JV & the Deep Six	10
5	RegularBand	Jim Glynn	9
6	RegularBand	Modern Dance	10
7	RegularBand	Coldwater Cattle Company	8
8	RegularBand	Saturday Revue	9
9	RegularBand	Julia Schnebly	8
10	SupportBand	Topazz	7
11	SupportBand	Jazz Persuasion	7
12	SupportBand	Susan McLain	6
13	AmateurBand	Katherine Ehrlich	0

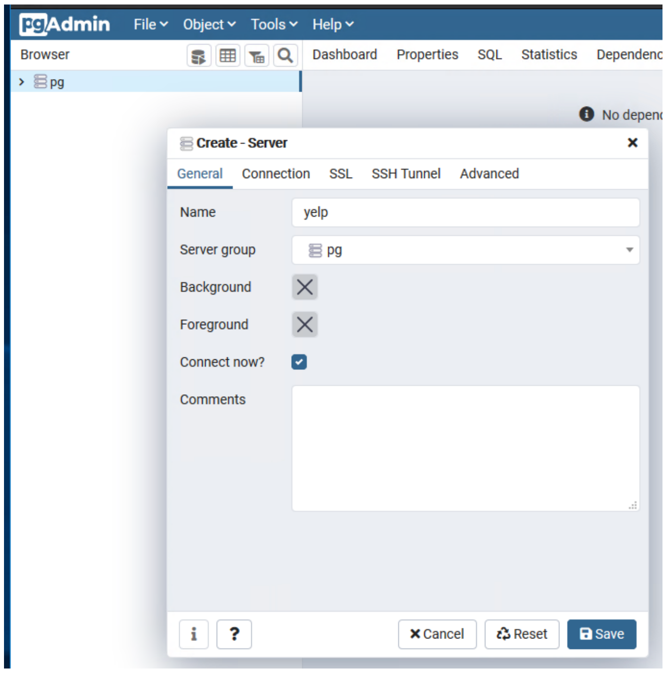
Yelp Database (yelp) - General Instructions for Postgres

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/.

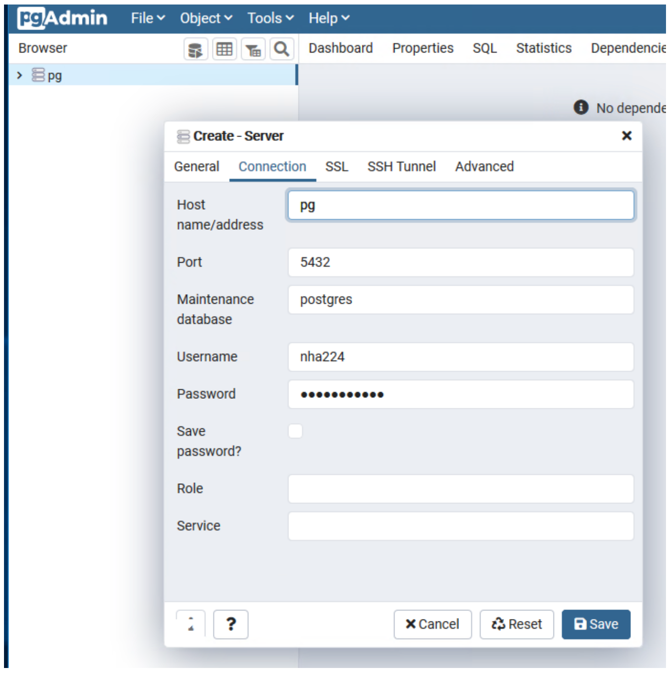
After you get on the NU network, open Remote Desktop and use your NetID credentials (or mcc\NetID) to connect to MSiA's remote desktop (e.g., `ts2.lab.analytics.northwestern.edu`). If you get a notice that the certificate cannot be verified, you can simply click "Continue" and proceed. Once you are in 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

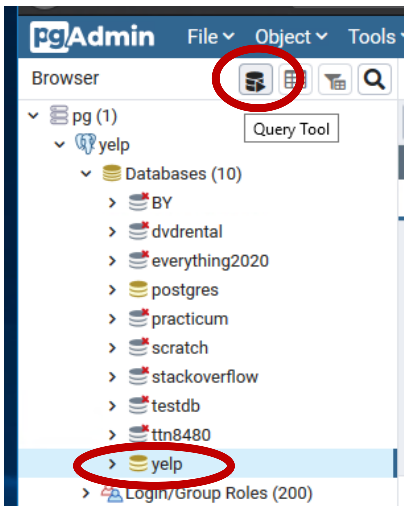
Start the "`pgAdmin 4 v4`" application and create a new server connection named `yelp` on the `pg` server group, provide `pg` as the host name and your NetID credentials, similar to the pictures below, and Save it.



Name: yelp
Server group: pg
Background
Foreground
Connect now? ✓
Comments



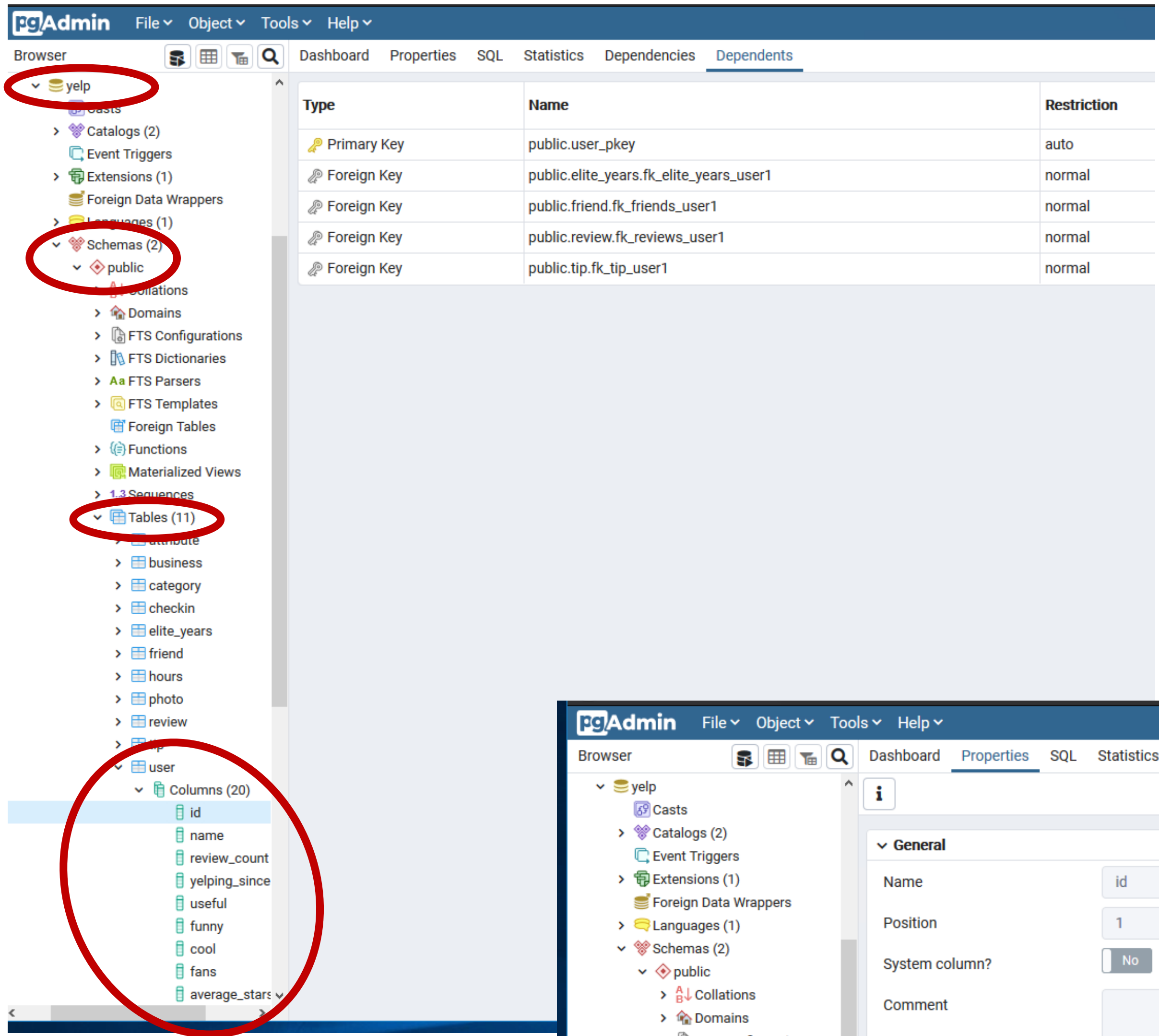
Host name/address: pg
Port: 5432
Maintenance database: postgres
Username: nha224
Password: ●●●●●●●●●●●
Save password?
Role
Service



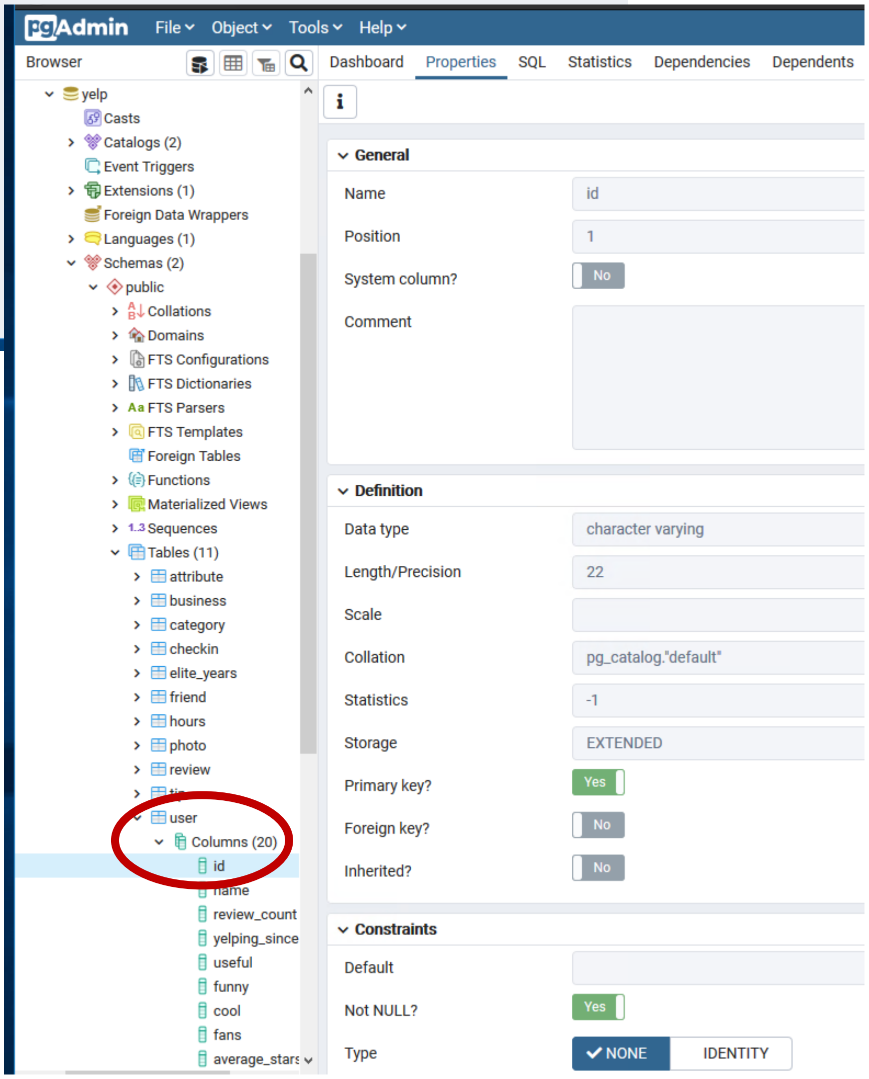
Now, you can connect to the `yelp` database on Postgres and start issuing queries to it. Selecting the `yelp` database and clicking on the `Query Tool` (figure at right) opens a SQL editor in which you can write SQL queries.

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`.



Type	Name	Restriction
Primary Key	public.user_pkey	auto
Foreign Key	public.elite_years.fk_elite_years_user1	normal
Foreign Key	public.friend.fk_friends_user1	normal
Foreign Key	public.review.fk_reviews_user1	normal
Foreign Key	public.tip.fk_tip_user1	normal



General

Name	id
Position	1
System column?	No
Comment	

Definition

Data type	character varying
Length/Precision	22
Scale	
Collation	pg_catalog."default"
Statistics	-1
Storage	EXTENDED
Primary key?	Yes
Foreign key?	No
Inherited?	No

Constraints

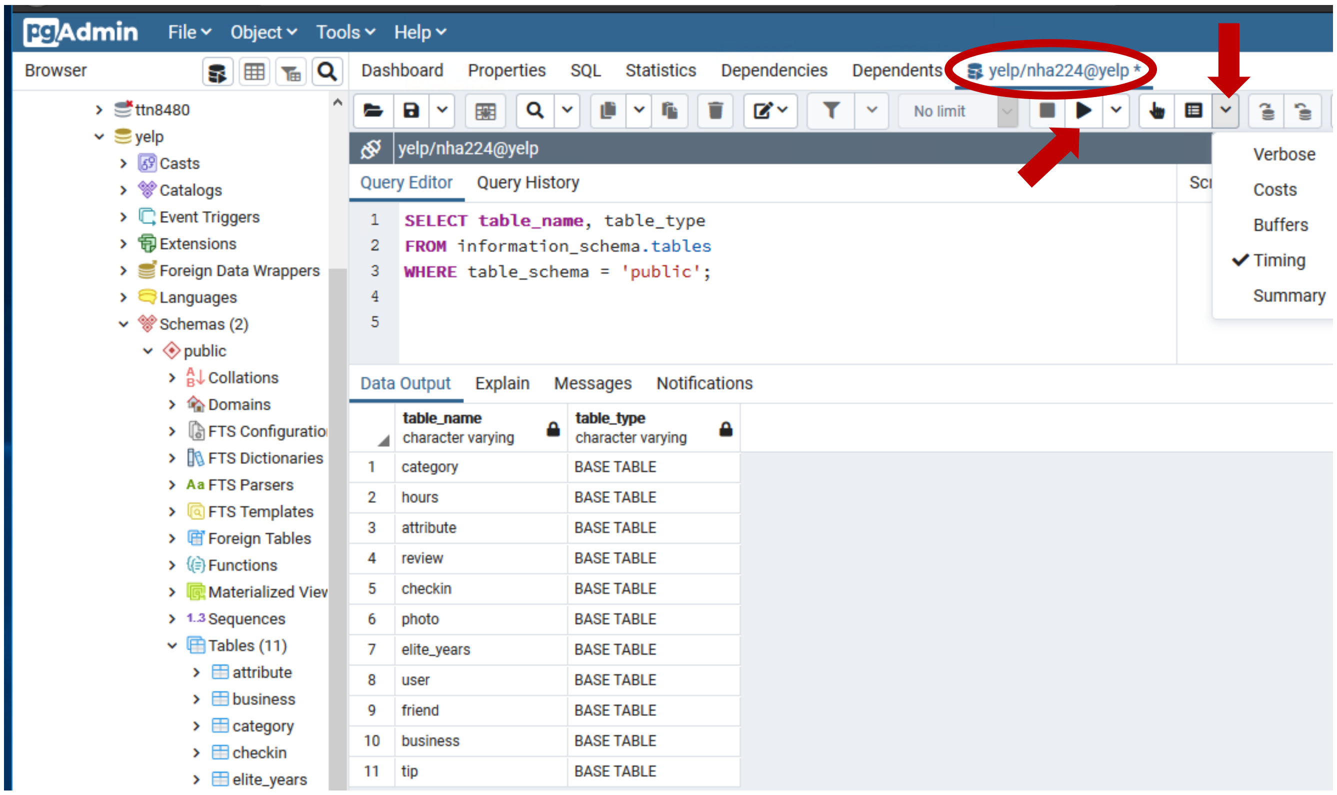
Default	
Not NULL?	Yes
Type	[✓ NONE] IDENTITY

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.

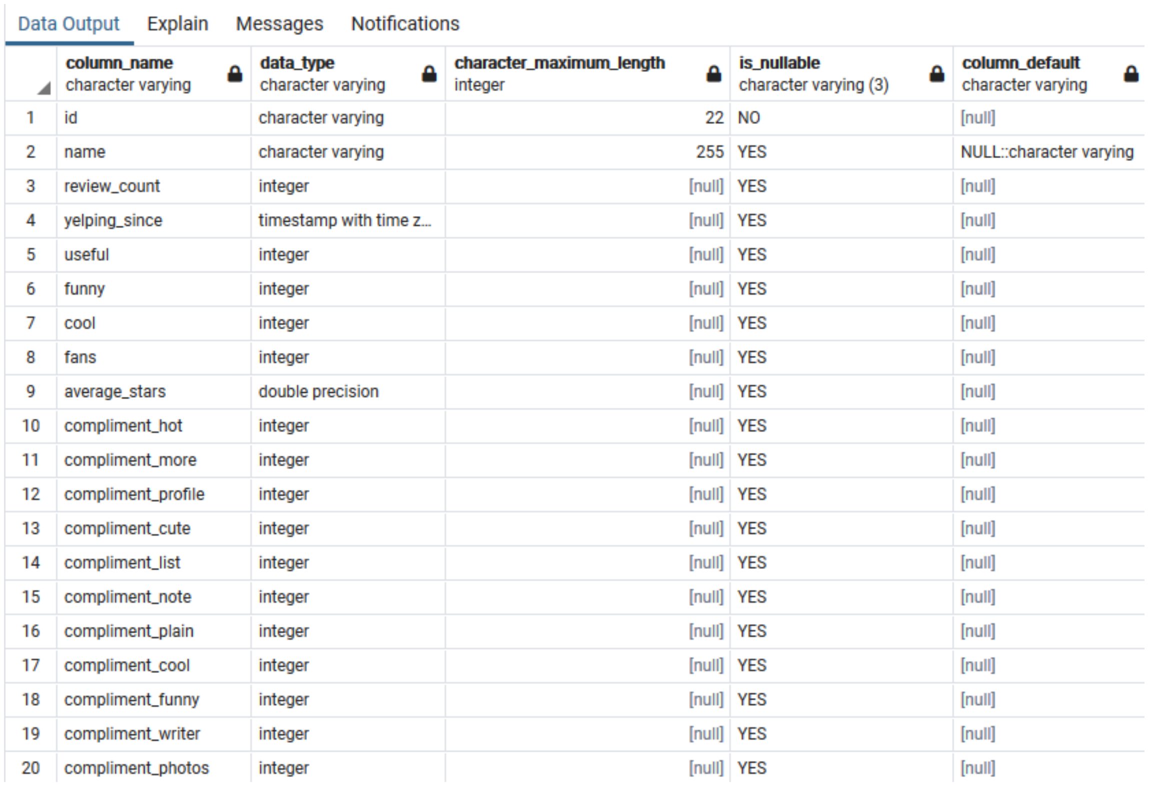


```
SELECT table_name, table_type
FROM information_schema.tables
WHERE table_schema = 'public';
```

	table_name character varying	table_type character varying
1	category	BASE TABLE
2	hours	BASE TABLE
3	attribute	BASE TABLE
4	review	BASE TABLE
5	checkin	BASE TABLE
6	photo	BASE TABLE
7	elite_years	BASE TABLE
8	user	BASE TABLE
9	friend	BASE TABLE
10	business	BASE TABLE
11	tip	BASE TABLE

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

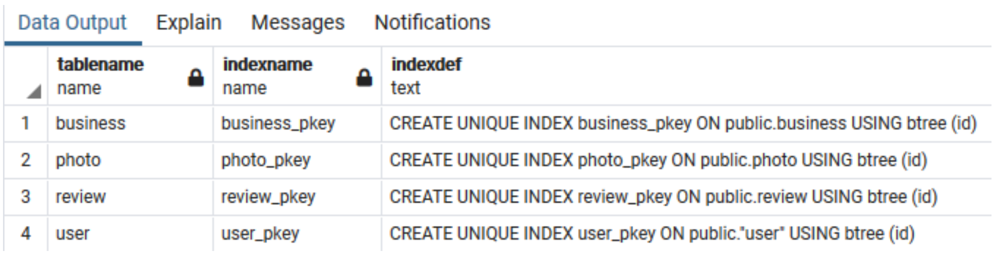
```
SELECT column_name, data_type, character_maximum_length, is_nullable, column_default
FROM   information_schema.columns
WHERE  table_name = 'user';
```



	column_name character varying	data_type character varying	character_maximum_length integer	is_nullable character varying (3)	column_default character varying
1	id	character varying	22	NO	[null]
2	name	character varying	255	YES	NULL::character varying
3	review_count	integer	[null]	YES	[null]
4	yelping_since	timestamp with time z...	[null]	YES	[null]
5	useful	integer	[null]	YES	[null]
6	funny	integer	[null]	YES	[null]
7	cool	integer	[null]	YES	[null]
8	fans	integer	[null]	YES	[null]
9	average_stars	double precision	[null]	YES	[null]
10	compliment_hot	integer	[null]	YES	[null]
11	compliment_more	integer	[null]	YES	[null]
12	compliment_profile	integer	[null]	YES	[null]
13	compliment_cute	integer	[null]	YES	[null]
14	compliment_list	integer	[null]	YES	[null]
15	compliment_note	integer	[null]	YES	[null]
16	compliment_plain	integer	[null]	YES	[null]
17	compliment_cool	integer	[null]	YES	[null]
18	compliment_funny	integer	[null]	YES	[null]
19	compliment_writer	integer	[null]	YES	[null]
20	compliment_photos	integer	[null]	YES	[null]

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;
```

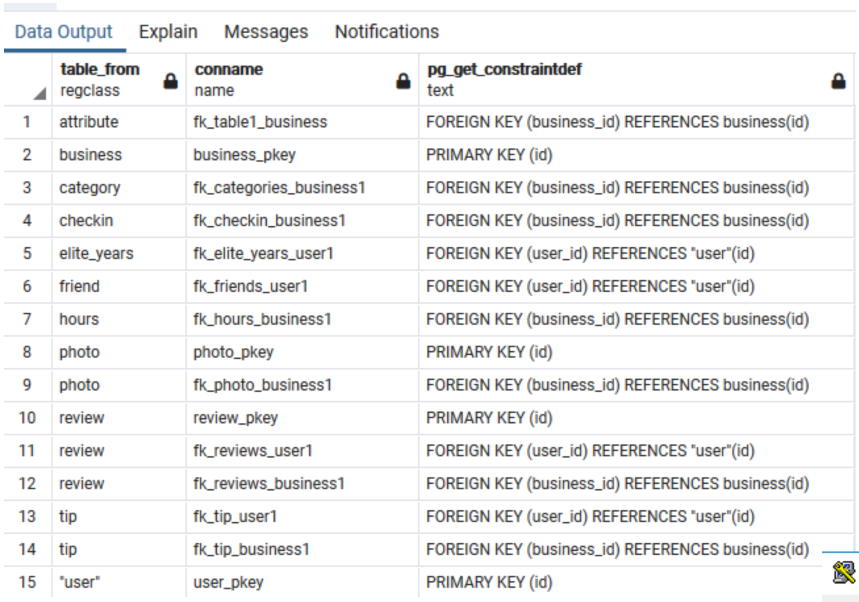


Data Output
Explain
Messages
Notifications

	tablename name	indexname name	indexdef text
1	business	business_pkey	CREATE UNIQUE INDEX business_pkey ON public.business USING btree (id)
2	photo	photo_pkey	CREATE UNIQUE INDEX photo_pkey ON public.photo USING btree (id)
3	review	review_pkey	CREATE UNIQUE INDEX review_pkey ON public.review USING btree (id)
4	user	user_pkey	CREATE UNIQUE INDEX user_pkey ON public."user" USING btree (id)

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;
```



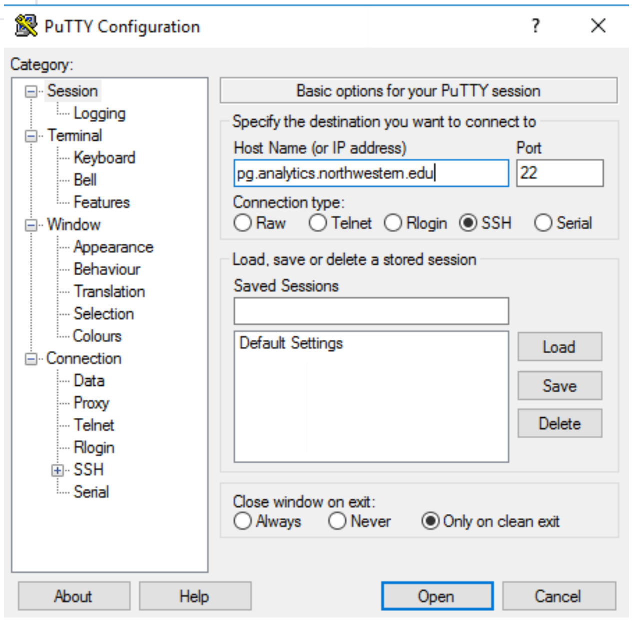
Data Output
Explain
Messages
Notifications

	table_from regclass	conname name	pg_get_constraintdef text
1	attribute	fk_table1_business	FOREIGN KEY (business_id) REFERENCES business(id)
2	business	business_pkey	PRIMARY KEY (id)
3	category	fk_categories_business1	FOREIGN KEY (business_id) REFERENCES business(id)
4	checkin	fk_checkin_business1	FOREIGN KEY (business_id) REFERENCES business(id)
5	elite_years	fk_elite_years_user1	FOREIGN KEY (user_id) REFERENCES "user"(id)
6	friend	fk_friends_user1	FOREIGN KEY (user_id) REFERENCES "user"(id)
7	hours	fk_hours_business1	FOREIGN KEY (business_id) REFERENCES business(id)
8	photo	photo_pkey	PRIMARY KEY (id)
9	photo	fk_photo_business1	FOREIGN KEY (business_id) REFERENCES business(id)
10	review	review_pkey	PRIMARY KEY (id)
11	review	fk_reviews_user1	FOREIGN KEY (user_id) REFERENCES "user"(id)
12	review	fk_reviews_business1	FOREIGN KEY (business_id) REFERENCES business(id)
13	tip	fk_tip_user1	FOREIGN KEY (user_id) REFERENCES "user"(id)
14	tip	fk_tip_business1	FOREIGN KEY (business_id) REFERENCES business(id)
15	"user"	user_pkey	PRIMARY KEY (id)

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.



Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): pg.analytics.northwestern.edu Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

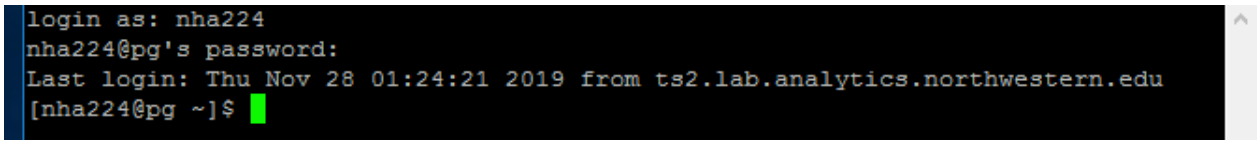
Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

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



```
login as: nha224
nha224@pg's password:
Last login: Thu Nov 28 01:24:21 2019 from ts2.lab.analytics.northwestern.edu
[nha224@pg ~]$
```

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;
```

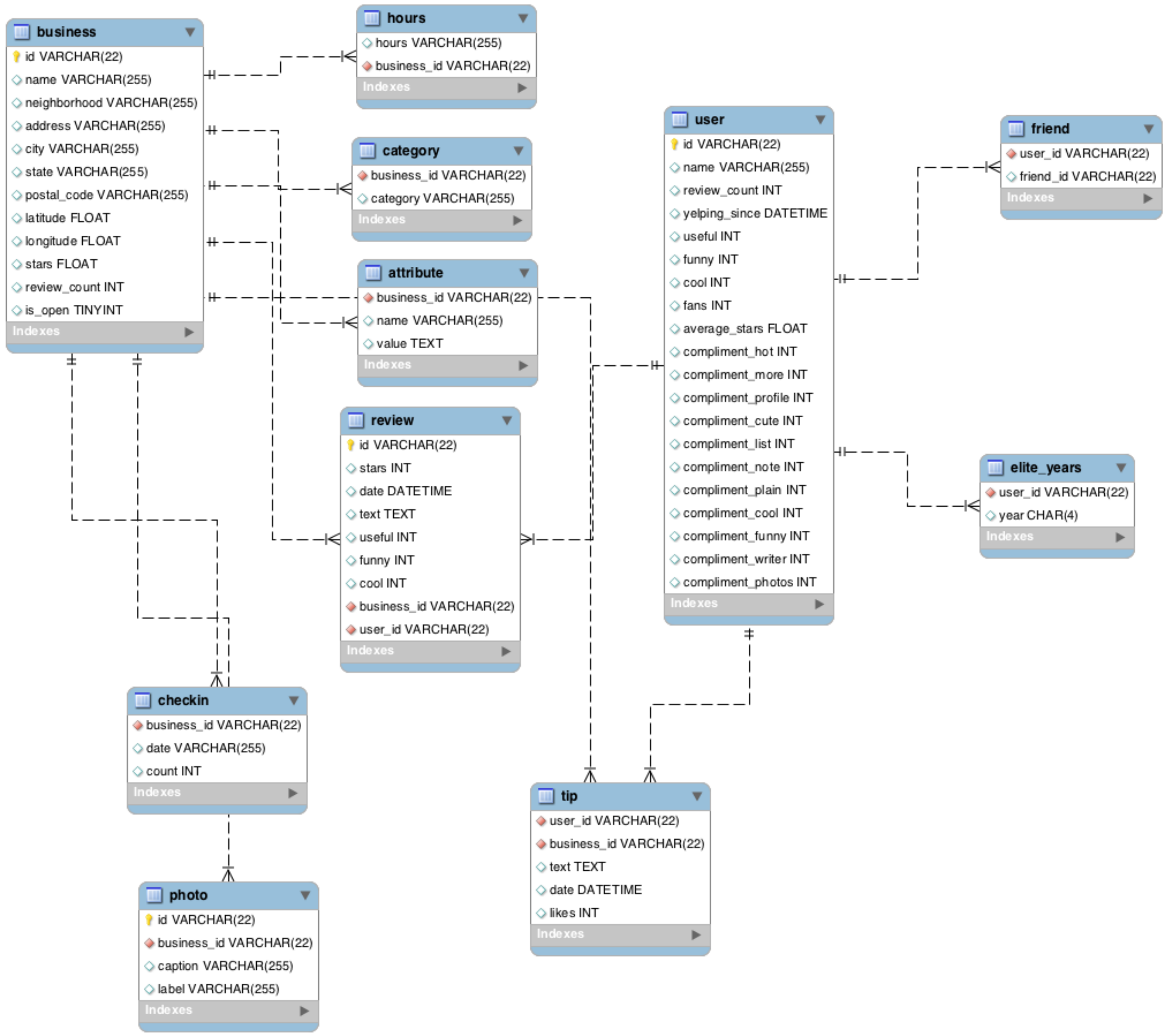
You can exit from the database by typing "`\q`" or by pressing `CONTROL-d`.

```
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:



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.

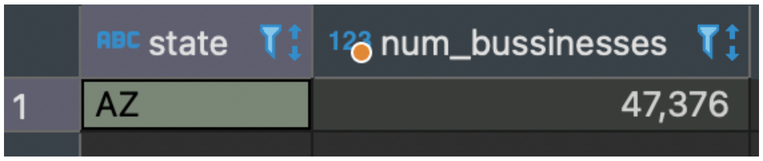
You will use this database to answer the following questions. Unless otherwise noted, for each question provide:

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

7) **(10 points)** Which state has the most businesses, and how many businesses are there?

```
select b.state , COUNT(b.id) num_bussinesses
from
public.business b
group by
b.state
order by
num_bussinesses
desc limit 1;
```

Output:

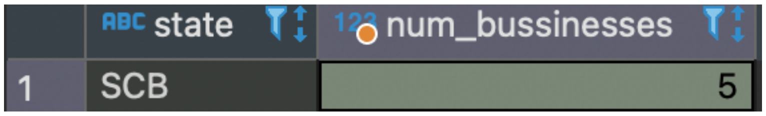


	state	num_bussinesses
1	AZ	47,376

8) **(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 b.state , COUNT(b.id) num_bussinesses
from
public.business b
group by
b.state
order by
num_bussinesses
desc
limit 1 offset (select (COUNT(distinct b2.state) / 2) from public.business b2);
```

OUTPUT:

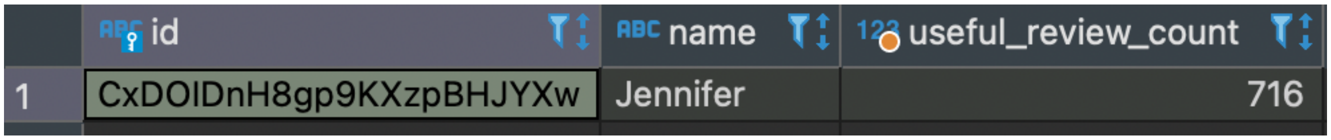


	state	num_bussinesses
1	SCB	5

9) **(10 points)** Find the name of the user that has given the largest number of useful reviews to closed businesses. Print both the user name and the number of such reviews the user has given.

```
select u.id , u."name" , COUNT(*) AS useful_review_count
from
public."user" u
inner join
public.review r
on u.id = r.user_id
inner join
public.business b
on r.business_id = b.id
where b.is_open = 0 and r.useful > 0
group by u.id
```

order by useful_review_count desc limit 1;

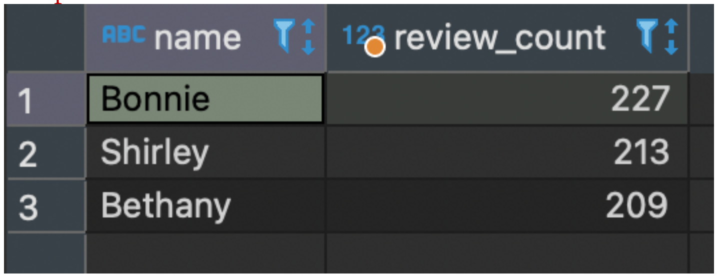


	id	name	useful_review_count
1	CxDOIDnH8gp9KXzpBHJYXw	Jennifer	716

10) **(10 points)** Find the top 3 users that have provided the largest number of reviews of businesses within a range of 0.1 degrees from latitude 36.0 and longitude -115.0. For each one of these users, provide in your answer the name and the number of reviews that user provided for businesses within that range. *Hint:* You can use the Pythagorean theorem to find businesses within the requested range. For example, locations within d degrees from latitude X and longitude Y satisfy the formula `sqrt(power(longitude-Y, 2.0) + power(latitude-X, 2.0)) ≤ d`.

```
SELECT u.name, COUNT(*) AS review_count
FROM public.user AS u
JOIN public.review AS r ON u.id = r.user_id
JOIN public.business AS b ON r.business_id = b.id
WHERE sqrt(power(b.longitude - (-115.0), 2.0) + power(b.latitude - 36.0, 2.0)) <= 0.1
GROUP BY u.id
ORDER BY review_count DESC
LIMIT 3;
```

Output:



	name	review_count
1	Bonnie	227
2	Shirley	213
3	Bethany	209