R, SQL, and You: Adapting R to relational databases

Thomas Reynolds

Fall 2014

Contents

For the Statistical Computing for Biomedical Data Analytics course at the University of San Francisco

---

Goals of the Project

- Establish a PostgreSQL server on a remote system
- Create a database of meaningful population data
- Connect to the remote database in RStudio
- Perform analytics in R using the PostgreSQL database.

Why am I doing this?

PostgreSQL is the open-source relational database of choice for major industry and academia, as Zumel and Mount explain:

> In many production environments, the data you want lives in a relational or SQL database, not in files. Public data is often in files (as they are easier to share), but your most important client data is often in databases. Relational databases scale easily to the millions of records and supply important production features such as parallelism, consistency, transactions, logging, and audits.

> When you're working with transaction data, you're likely to find it already stored in a relational database, as relational databases excel at online transaction processing (OLTP).

So your data source is likely to be in this conceptual format, especially if from a public source such as the DHHS or FDA. What relational databases *don't* do well is data analysis, even in simple forms such as linear regression. This is where R shines — it's a programming language devoted to data analysis, moreso than the safe and comfortable Python you've likely used before now.

However, R has its own issues with large datasets; all analysis done in R is **in-memory**, which is not the ideal for those millions of records mentioned above. For example, publicly-available US Census data can top out at more than 43 GB in size—local analysis of such a large set in R would be arduous and inefficient.

So why not both? In this project, you'll be using an ideal paradigm of large dataset analysis, where the data itself resides on some other repository, and you are minimizing the amount of labor your client machine performs.

Establishing a PostgreSQL server and database

The first phase of the project is to configure the remote server that will be hosting your data. In a real-world setting this would be a remote server that you would connect to, hosted by a public agency like the CDC, or your research institution or company. The principles we will explore, though, will work just as easily on a virtual machine, with the added benefit that you will always have access to the "machine" hosting your data.

Create a virtual machine

If you have your own method of managing VMs, feel free to use it; in this assignment we will be using a combination of Vagrant, VirtualBox, and Fabric to create and manage our "remote" VM.

Vagrant and VirtualBox feature installation packages that can be installed from the host GUI (that is, your computer). **From here on out, assume all VM work is done in your CLI.**

Fabric is installed from the command line and requires pip, a Python-based package installer. To get pip type

```
python get-pip.py
```

into your CLI. After installation is complete, install Fabric with

```
pip install fabric
```

To create the VM itself, go to the directory you've chosen for this project, initialize a Vagrant Box with the operating system of your choice (the procedure below uses 'trusty tahr' Ubuntu 14.04, 64-bit), and start up the VM for the first time. Once the VM is up and running, halt the VM and open the Vagrantfile that is now in the chosen directory (any text editor in the CLI or GUI will do).

The VM needs dedicated port forwarding to communicate with the outside world (in this case, your host OS). PostgreSQL servers "listen" for outside connections at port 5432 by default. Search for ""forwarded_port"" in the file, and under any preexisting text enter

```
config.vm.network "forwarded_port", guest: 5432, host: 5432
```

Save and start the VM up again, and ssh into the VM using the 'vagrant' account (the password is also 'vagrant').

Install and configure PostgreSQL

Ubuntu is said to have PostgreSQL already installed, but this is not necessarily the case—nor is the instance on your VM necessarily up to date. At time of writing **9.3** is the most stable version. To install the full PostgreSQL implementation (not just the user client) and the PostGIS plugin (which we will discuss later), run the following:

```
sudo apt-get install postgresql postgresql-contrib
sudo apt-get install postgis
```

PostgreSQL expects many of its management tasks to be performed in a system-level user account called `postgres`, so set one up now. For security purposes you wouldn't give the postgres account administrator access on a production system, but doing so here is acceptable (and will save some time switching between accounts). Then, using

```
sudo su -u postgres
```

add the location of the PostgreSQL binaries to the postgres account's path. On the development machine this lesson was prepared on, the binaries are stored at `/usr/lib/psql/9.3/bin`. Being able to use these commands as vagrant is also necessary, so add this to the vagrant account path as well.

PostgreSQL needs to have a server, or "cluster", to hold your database; this is also the cluster that will accept connections from outside clients. The recommended directory to store the server information in is `/usr/local/pgsql/data`, which probably does not exist yet. Create this directory and initialize the cluster in the vagrant account with

```
sudo mkdir /usr/local/pgsql/data
sudo chown postgres /usr/local/pgsql/data
sudo su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

Refer to PostgreSQL's documentation on initialization if any errors are returned.

Open the `pg_hba.conf` file under `/etc/postgresql/9.3/cluster-name/`. You will need to use `sudo`. This file describes what types of connections, and from where, the PostgreSQL cluster will accept. The allowable connections are listed at the bottom of the file. The `IPv4 local connections` should look like this once you're done.

```
IPv4 local connections:
host    all             all             127.0.0.1/32            trust
host    all             all             10.0.0.0/8              password
```

Save and close, and now open the `postgresql.conf` file in the same directory. You will need to uncomment the `listen_addresses` and `port` settings so that the cluster will accept connections on the default port (5432). Also, set `listen_addresses = '*'` so that the cluster will listen for connections from any address. Save and close, and start up the server with

```
pg_ctl -D /usr/local/pgsql/data start
```

to make your databases available to the outside world (in this case, your host machine). *Tip: halting your VM will save the cluster in its running state, so you don't have to worry about starting the cluster again when the VM is restarted.*

Finally, because `postgres` is usually used only for administrative tasks, let's create another database user called `vagrant` (by default PostgreSQL will only allow `psql` logins from a system account with a matching name). To create another user for this cluster, enter at the system prompt

```
createuser -d -l -P vagrant
```

which will then immediately prompt you for a password (might as well use 'vagrant' for this instance; obviously you would use better security practices in a production setting). Finally, exit out of the `postgres` system account and enter the `psql` program as `vagrant` using

```
psql vagrant
```

which, if everything has worked, will take you to the following screen:

```
psql (9.3.5)
Type "help" for help.

vagrant=>
```

Exercise 1: Promoter sequence analysis

> *Necessary R packages: DBI, RPostgreSQL*

The NCBI RefSeq database proivdes a "non-redundant collection of sequences representing genomic data, transcripts and proteins. . . [with] data from over 2400 organisms and includes over one million proteins representing significant taxonomic diversity spanning prokaryotes." (Nucleic Acids Res. Jan 1, 2005; 33(Database issue): D501–D504. doi:10.1093/nar/gki025) In this exercise we will be using the `RPostgreSQL`, and `DBI` packages to push an analytical request to the PostgreSQL server, and receive results from the server in the R environment.

Halt the VM for now. From the Table Browser at UCSC's Genome Browser, download the complete hg19 refGene from the refSeq Genes track as a gzip archive (specify '.txt' in the filename), and move the enclosed flat text file to the directory with your VM's Vagrantfile. Start up the VM again and this file will appear in the VM's `/vagrant` directory. The schema in **Appendix 1** describes the database used for developing this exercise, and makes use of PostgreSQL's `CREATE MATERIALIZED VIEW` capability to generate subsets based on the strand the gene resides on. The column names are taken directly from the refSeq Genes track; keeping this naming convention is recommended so other users can immediately use the database.

Additionally, creating strand-based subsets allows us to designate a transcription start site (TSS) for each gene — on a "+"-strand gene this would be the `txStart` nucleotide, and the `txEnd` nucleotide on a "-"-strand gene. These TSSes only exist in the `posgene` and `neggene` views, and are not reflected in the original tables, but allows us to simplify our SQL queries.

Establish a connection to the database in R, and replicate the following results:

- Generate a table of promoter start site pairs within 1000 bp of each other and display a small sample of that list (~6 rows).

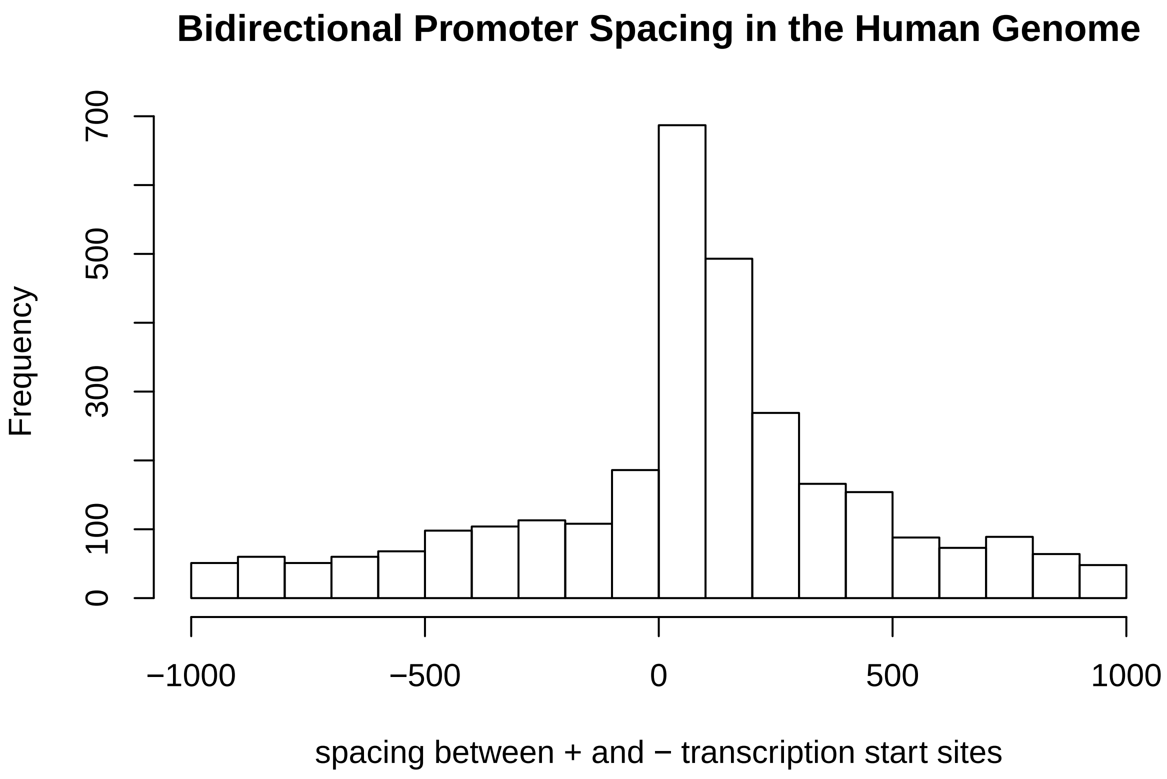
```r
library(DBI)
library(RPostgreSQL)
drv <- dbDriver("PostgreSQL")
host <- "127.0.0.1"
port <- "5432"
username <- "vagrant"
password <- "vagrant"
```

```r
conn <- dbConnect(drv, dbname = "hg19db", username, password, host, port)

allpairs <- dbGetQuery(conn, "SELECT pos.chrom as chrom, pos.tss as pos_tss, neg.tss as neg_tss, \n
head(allpairs)
```

```
##   chrom pos_tss neg_tss pos_symbol neg_symbol spacing
## 1  chr1  762970  762902  LINC01128  LINC00115      68
## 2  chr1  763177  762902  LINC01128  LINC00115     275
## 3  chr1 1167628 1167447    B3GALT6       SDF4     181
## 4  chr1 1243993 1243269      PUSL1      ACAP3     724
## 5  chr1 1260142 1260067       CPTP     CPSF3L      75
## 6  chr1 1334909 1334718  LOC148413      CCNL2     191
```

- Generate a histogram of spacing between pairs of promoters on opposite strands. In our definition, bidirectional promoters cannot have negative spacing between transcription start sites.

```r
hist(allpairs$spacing, breaks = 21, main = "Bidirectional Promoter Spacing in the Human Genome",
    xlab = "spacing between + and - transcription start sites")
```



- Estimate the number of human promoters that are bidrectional.

```r
bidir_pair <-(allpairs[allpairs$spacing > 0,])
num_bidir_prom <- nrow(as.table(unique(bidir_pair$pos_tss))) + nrow(as.table(unique(bidir_pair$neg_tss)
total_tss <- nrow(dbGetQuery(conn, "SELECT tss from posgene")) + nrow(dbGetQuery(conn, "SELECT tss from

bidir_percent  <- (num_bidir_prom / total_tss) * 100
invisible(dbDisconnect(conn))
```

If we define a bidirectional promoter pair as two promoters on complementary strands within 1000 bp of each others' TSSes, then there are 3784 such pairs in hg19. 11.3066603% of hg19 refGene promoters are bidirectional, according to our analysis.

Exercise 2: Population data

Necessary R packages: DBI, RPostgreSQL, dplyr, maps, maptools, RColorBrewer

The Vaccine Adverse Event Reporting System (VAERS) is a database maintained by the CDC which, as the name suggests, collects reports submitted by health practitioners and patients on pathological incidents that occur contemporarily with vaccine administrations *with no assertion that the two are causally related*. The CDC reports these events in three separate tables, all keyed to a unique VAERS_ID:

- Date stamped observations and outcomes,
- Categorized symptoms,
- Information on the vaccine used.

Because events are usually characterized with the state they take place in, we can use VAERS to generate geographic maps. Specifically, the VAERS data will be applied to a *choropleth*, which is a map "that uses graded differences in shading or color or the placing of symbols inside defined areas on the map in order to indicate the average values of some property or quantity in those areas." (From The Free Dictionary)

This exercise also demonstrates the `dplyr` package, which allows for radically different interaction with the database than before. In Exercise 1, we used the `DBI` package to send explicit SQL requests to the gene database, which worked well since we were performing relatively simple relational selections, However, these are immediate requests to the cluster, which take time to perform and return. `dplyr`, on the other hand, is lazy — none of the objects created in R using dplyr actually exist as data frames or tables on either computer, until R receives a `collect` call. `dplyr` then translates and composes each command into a single query to the SQL database. Additionally `dplyr` commands can be chained with the `%>%` operator, which allows sequential functions to be scripted in a more intuitive manner than nesting functions.

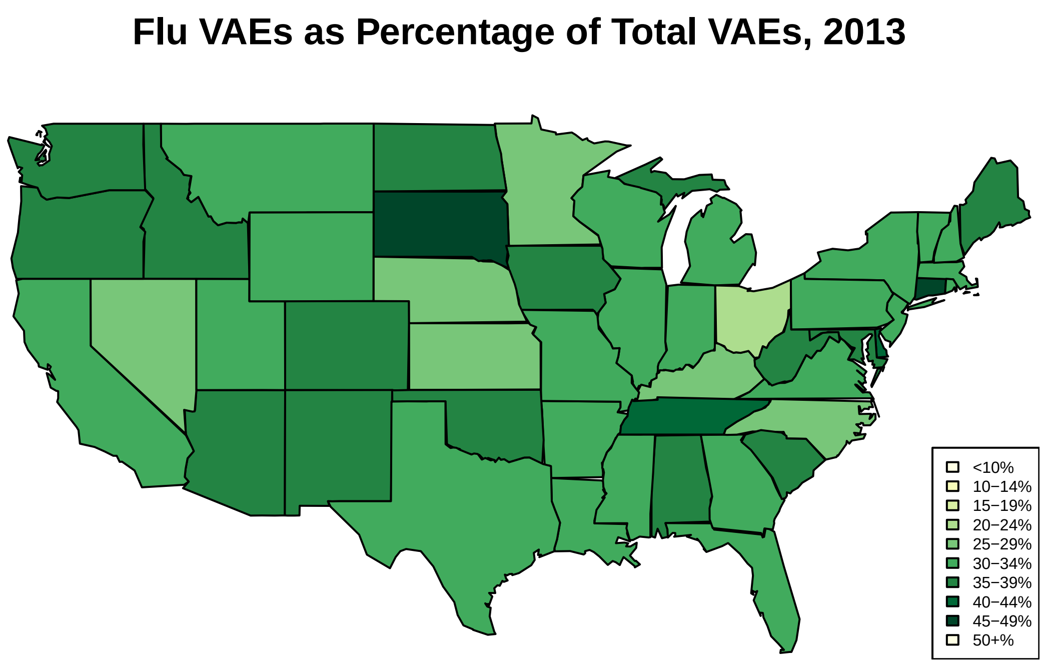
The `RColorBrewer` package is not absolutely necessary, but it makes generating a spectrum of colors for graphical output much easier.

Create a database on your VM with the 2013 VAERS dataset and replicate the following results:

- Generate a map of the contiguous United States, colored by the percent of influenza-related VAEs per total reported VAEs in each state.

```r
library(DBI)
library(RPostgreSQL)
library(dplyr)
library(maps)
library(maptools)
library(RColorBrewer)

vaers <- src_postgres(dbname = "vaccinedb", host = "127.0.0.1", port = "5432",
    user = "vagrant", password = "vagrant")
vaersdata <- tbl(vaers, "vaersdata")
vaersvax <- tbl(vaers, "vaersvax")
vaerssymptoms <- tbl(vaers, "vaerssymptoms")
```

```
stateevents <- vaersdata %>% select(state, vaers_id)
statetotals <- stateevents %>% count(state) %>% arrange(state)

statefluevents <- vaersvax %>% group_by(vax_type) %>% filter(vax_type ~ "FLU*") %>%
    select(vaers_id, vax_type) %>% inner_join(select(vaersdata, vaers_id, state)) %>%
    select(state, vaers_id)
stateflutotals <- statefluevents %>% count(state) %>% arrange(state)

abb <- cbind(state.abb)
sname <- tolower(cbind(state.name))
abbtoname <- cbind(abb, sname)
abbtoname <- as.data.frame(as.matrix(abbtoname))

flupercent <- inner_join(stateflutotals, statetotals, by = "state")
flupercplot <- collect(flupercent %>% mutate(percent = (n.x * 100)/n.y))

colors <- brewer.pal(9, "YlGn")
flupercplot$colorBuckets <- as.numeric(cut(flupercplot$percent, c(10, 15, 20,
    25, 30, 35, 40, 45, 50, 100)))
fluleg.txt <- c("<10%", "10-14%", "15-19%", "20-24%", "25-29%", "30-34%", "35-39%",
    "40-44%", "45-49%", "50+%")

states.matched <- abbtoname$state.abb[match(map("state", plot = FALSE)$names,
    abbtoname$state.name)]
colorsmatched <- flupercplot$colorBuckets[match(states.matched, flupercplot$state)]
map("state", col = colors[na.omit(colorsmatched)], fill = TRUE)
legend("bottomright", fluleg.txt, horiz = FALSE, fill = colors, cex = 0.5)
title("Flu VAEs as Percentage of Total VAEs, 2013")
```



- Examine the composition of `statefluevents`, and `flupercent` with `dplyr`'s `explain()` function.

```r
explain(statefluevents)
```

```
## <SQL>
## SELECT "vax_type" AS "vax_type", "state" AS "state", "vaers_id" AS "vaers_id"
## FROM (SELECT * FROM (SELECT "vaers_id" AS "vaers_id", "vax_type" AS "vax_type"
## FROM "vaersvax"
## WHERE "vax_type" ~ 'FLU*') AS "_W2"
##
## INNER JOIN
##
## (SELECT "vaers_id" AS "vaers_id", "state" AS "state"
## FROM "vaersdata") AS "_W3"
##
## USING ("vaers_id")) AS "_W4"
##
##
## <PLAN>
## Hash Join  (cost=2712.90..4146.61 rows=9237 width=12)
##   Hash Cond: (vaersvax.vaers_id = vaersdata.vaers_id)
##   ->  Seq Scan on vaersvax  (cost=0.00..1063.06 rows=9237 width=9)
##         Filter: (vax_type ~ 'FLU*'::text)
##   ->  Hash  (cost=2223.18..2223.18 rows=29818 width=7)
##         ->  Seq Scan on vaersdata  (cost=0.00..2223.18 rows=29818 width=7)
```

```r
explain(flupercent)
```

```
## <SQL>
## SELECT "state", "n.x", "n.y"
## FROM (SELECT * FROM (SELECT "state" AS "state", "n" AS "n.x"
## FROM (SELECT "state", count(*) AS "n"
## FROM (SELECT * FROM (SELECT "vaers_id" AS "vaers_id", "vax_type" AS "vax_type"
## FROM "vaersvax"
## WHERE "vax_type" ~ 'FLU*') AS "_W2"
##
## INNER JOIN
##
## (SELECT "vaers_id" AS "vaers_id", "state" AS "state"
## FROM "vaersdata") AS "_W3"
##
## USING ("vaers_id")) AS "_W4"
## GROUP BY "state") AS "_W5"
## ORDER BY "state") AS "_W6"
##
## INNER JOIN
##
## (SELECT "state" AS "state", "n" AS "n.y"
## FROM (SELECT "state", count(*) AS "n"
## FROM "vaersdata"
## GROUP BY "state") AS "_W1"
## ORDER BY "state") AS "_W7"
##
## USING ("state")) AS "_W8"
```

```
##
##
## <PLAN>
## Merge Join  (cost=6552.34..6554.63 rows=54 width=19)
##   Merge Cond: (vaersdata.state = vaersdata_1.state)
##   ->  Sort  (cost=4177.43..4177.57 rows=54 width=11)
##         Sort Key: vaersdata.state
##         ->  HashAggregate  (cost=4174.80..4175.34 rows=54 width=3)
##               ->  Hash Join  (cost=2712.90..4128.61 rows=9237 width=3)
##                     Hash Cond: (vaersvax.vaers_id = vaersdata.vaers_id)
##                     ->  Seq Scan on vaersvax  (cost=0.00..1063.06 rows=9237 width=4)
##                           Filter: (vax_type ~ 'FLU*'::text)
##                     ->  Hash  (cost=2223.18..2223.18 rows=29818 width=7)
##                           ->  Seq Scan on vaersdata  (cost=0.00..2223.18 rows=29818 width=7)
##   ->  Materialize  (cost=2374.90..2375.71 rows=54 width=11)
##         ->  Sort  (cost=2374.90..2375.04 rows=54 width=11)
##               Sort Key: vaersdata_1.state
##               ->  HashAggregate  (cost=2372.27..2372.81 rows=54 width=3)
##                     ->  Seq Scan on vaersdata vaersdata_1  (cost=0.00..2223.18 rows=29818 width=3)
```

Works Referenced

Exercise 1 is adapted from the "Human Bi-Directional Promoters" exercise by Dr. Robert Horton.

For further information, consult *Practical Data Science with R* by Nina Zumel and John Mount from Manning Publications (ISBN 9781617291562).

Appendix I: Database schemata and commands for exercises

Schema for Excerise 1

```
--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;

--
-- Name: plpgsql; Type: EXTENSION; Schema: -; Owner:
--

CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;


--
-- Name: EXTENSION plpgsql; Type: COMMENT; Schema: -; Owner:
```

```
--

 COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';


 SET search_path = public, pg_catalog;

 SET default_tablespace = '';

 SET default_with_oids = false;

--
-- Name: hg19_refgene; Type: TABLE; Schema: public; Owner: vagrant; Tablespace:
--

 CREATE TABLE hg19_refgene (
     bin integer,
     name text,
     chrom text,
     strand text,
     txstart integer,
     txend integer,
     cdsstart integer,
     cdsend integer,
     exoncount integer,
     exonstarts text,
     exonends text,
     score integer,
     name2 text,
     cdsstartstat text,
     cdsendstat text,
     exonframes text
 );


 ALTER TABLE public.hg19_refgene OWNER TO vagrant;

--
-- Name: neggene; Type: VIEW; Schema: public; Owner: vagrant
--

 CREATE VIEW neggene AS
  SELECT array_to_string(array_agg(DISTINCT hg19_refgene.name), ','::text) AS name,
     array_to_string(array_agg(DISTINCT hg19_refgene.name2), ','::text) AS symbol,
     hg19_refgene.chrom,
     hg19_refgene.strand,
     hg19_refgene.txend AS tss
    FROM hg19_refgene
   WHERE ((hg19_refgene.strand = '-'::text) AND (hg19_refgene.chrom !~~ '%ctg%'::text))
   GROUP BY hg19_refgene.chrom, hg19_refgene.strand, hg19_refgene.txend;


 ALTER TABLE public.neggene OWNER TO vagrant;
```

```
--
-- Name: posgene; Type: VIEW; Schema: public; Owner: vagrant
--

CREATE VIEW posgene AS
 SELECT array_to_string(array_agg(DISTINCT hg19_refgene.name), ','::text) AS name,
    array_to_string(array_agg(DISTINCT hg19_refgene.name2), ','::text) AS symbol,
    hg19_refgene.chrom,
    hg19_refgene.strand,
    hg19_refgene.txstart AS tss
   FROM hg19_refgene
  WHERE ((hg19_refgene.strand = '+'::text) AND (hg19_refgene.chrom !~~ '%ctg%'::text))
  GROUP BY hg19_refgene.chrom, hg19_refgene.strand, hg19_refgene.txstart;


ALTER TABLE public.posgene OWNER TO vagrant;

--
-- Name: public; Type: ACL; Schema: -; Owner: postgres
--

REVOKE ALL ON SCHEMA public FROM PUBLIC;
REVOKE ALL ON SCHEMA public FROM postgres;
GRANT ALL ON SCHEMA public TO postgres;
GRANT ALL ON SCHEMA public TO PUBLIC;
```

Schema for Exercise 2

```
--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;

--
-- Name: plpgsql; Type: EXTENSION; Schema: -; Owner:
--

CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;


--
-- Name: EXTENSION plpgsql; Type: COMMENT; Schema: -; Owner:
--

COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';
```

```
SET search_path = public, pg_catalog;

SET default_tablespace = '';

SET default_with_oids = false;

--
-- Name: vaersdata; Type: TABLE; Schema: public; Owner: vagrant; Tablespace:
--

CREATE TABLE vaersdata (
    vaers_id real NOT NULL,
    recvdate date,
    state character(2),
    age_yrs real,
    cage_yr real,
    cage_mo real,
    sex character(1),
    rpt_date date,
    symptom_text character varying(2048),
    died character(1),
    datedied date,
    l_threat character(1),
    er_visit character(1),
    hospital character(1),
    hospdays real,
    x_stay character(1),
    disable character(1),
    recovd character(1),
    vax_date date,
    onset_date date,
    numdays real,
    lab_data character varying(750),
    v_adminby character(3),
    v_fundby character(3),
    other_meds character varying(750),
    cur_ill character varying(500),
    history character varying(750),
    prior_vax character varying(256),
    splttype character varying(25)
);


ALTER TABLE public.vaersdata OWNER TO vagrant;

--
-- Name: vaerssymptoms; Type: TABLE; Schema: public; Owner: vagrant; Tablespace:
--

CREATE TABLE vaerssymptoms (
    vaers_id real,
    symptom1 text,
    symptomversion1 real,
    symptom2 text,
    symptomversion2 real,
    symptom3 text,
    symptomversion3 real,
    symptom4 text,
    symptomversion4 real,
    symptom5 text,
    symptomversion5 real
);


ALTER TABLE public.vaerssymptoms OWNER TO vagrant;

--
-- Name: vaersvax; Type: TABLE; Schema: public; Owner: vagrant; Tablespace:
--

CREATE TABLE vaersvax (
    vaers_id real,
    vax_type text,
    vax_manu text,
    vax_lot text,
    vax_dose text,
    vax_route text,
    vax_site text,
    vax_name text
);


ALTER TABLE public.vaersvax OWNER TO vagrant;


--
-- Name: vaersdata_pkey; Type: CONSTRAINT; Schema: public; Owner: vagrant; Tablespace:
--

ALTER TABLE ONLY vaersdata
    ADD CONSTRAINT vaersdata_pkey PRIMARY KEY (vaers_id);


--
-- Name: vaerssymptoms_vaers_id_fkey; Type: FK CONSTRAINT; Schema: public; Owner: vagrant
--

ALTER TABLE ONLY vaerssymptoms
    ADD CONSTRAINT vaerssymptoms_vaers_id_fkey FOREIGN KEY (vaers_id) REFERENCES vaersdata(vaers_id);


--
-- Name: vaersvax_vaers_id_fkey; Type: FK CONSTRAINT; Schema: public; Owner: vagrant
--

ALTER TABLE ONLY vaersvax
    ADD CONSTRAINT vaersvax_vaers_id_fkey FOREIGN KEY (vaers_id) REFERENCES vaersdata(vaers_id);
```

```
--
-- Name: public; Type: ACL; Schema: -; Owner: postgres
--

 REVOKE ALL ON SCHEMA public FROM PUBLIC;
 REVOKE ALL ON SCHEMA public FROM postgres;
 GRANT ALL ON SCHEMA public TO postgres;
 GRANT ALL ON SCHEMA public TO PUBLIC;


--
-- PostgreSQL database dump complete
--
```