Homicides in Philadelphia Neighborhoods: Identifying Dangerous Neighborhoods with PostGIS

The data we will be using for this tutorial was compiled and cleaned by the Washington Post for an article and series of maps that visualize where in America homicides are going unsolved. The dataset consists of over 50,000 homicides entries spread over 50 US cities taking place from 2007 to 2017. Included in this dataset is information such as the victim's name, age, sex and race. The date that the homicide took place and the current status of the case whether it be open without an arrest, closed by arrest, or closed with no arrest. Also included is the city and state where the homicide took place as well as the latitude and longitude coordinates of the homicide itself. We will also be using a shapefile of Philadelphia neighborhoods created and maintained by Azevea which will allow us to do some spatial queries. Included in the shapefile are the geoms of the neighborhoods themselves as well as their names. Please see Appendix i for instruction on downloading the data, creating a PostGIS enabled database, and importing the data to that database.

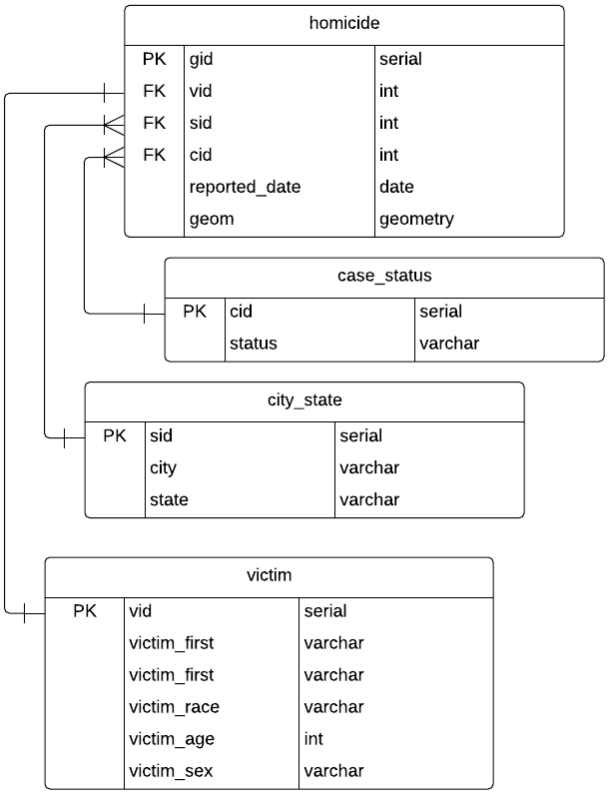
Normalization

Database normalization refers to the process of structuring a database in such a way that that it adheres to the normal forms to reduce the redundancy of data and improve data integrity. This is the homicide data as imported to the database.

Data Import		
PK	uid	varchar
	reported_date	varchar
	victim_last	varchar
	victim_first	varchar
	victim_race	varchar
	victim_age	varchar
	victim_age	varchar
	city	varchar
	state	varchar
	lat	varchar
	lon	varchar
	dispostion	varchar

To normalize this data, I had to break it out into a few different tables. First, I created a victim table to hold all the data about each victim which included victim first name, last name, age, sex and race. I then created a city and state look up table that held all 50 cities and their corresponding states. It

is worth noting that Tulsa appears twice in this list, once in Alabama and once in Oklahoma. It was for this reason that I had to concatenate the city and state when creating the sub-query that would reference the city_state table and create the foreign key. I ran into the same issue with the victim's table. Out of the 50,000 plus entries 4,147 of them share the same name, sex, age and race. To work around this, I was forced to bring in an existing unique identifier from the data import table and use that as a reference when assigning the foreign keys from the victim_table to the homicide table. This unique identifier is removed at the end of the ETL script. Next, I created the case_status table as a look up table for case dispositions. They are open not arrests, closed no arrests, and closed by arrest. Each table was given a serial primary key and was referenced as a foreign key in the homicide table. The homicide table itself held all the foreign keys, the reported date of the homicide, and the geom. This table structure allows for the insertion of new data in a far cleaner manner than having one large table. For instance, I could include a new case status like "cold case" which would be assigned a primary key of 4 and could then easily be referenced by new or existing cases in the main homicides table. The same is then true for a new city and state or a new victim record. This is the database after normalization.



After being normalized this dataset is still relatively large at 50,000 plus entries that are located all over the United States. However, I am only interested in the records that fall within Philadelphia. In the homicide table the sid that references Philadelphia is 47 so I can write an index that speeds up my queries a little bit.

```
create index philly_idx on homicide (sid);
```

Spatial Queries

Before we begin trying out spatial queries, we need to make sure we can connect to our database in QGIS so that we can create views for our queries. Right click the PostGIS icon in the browser on the left-hand side of QGIS and create a new connection. The name is arbitrary, service can be left blank, host is "localhost", port is most likely 5432, and the database is "homicides". From the drop down under the PostGIS icon you can now select a PostGIS layer to add to your map. There is one view I would like to create right off the bat. A view of the neighborhoods of Philadelphia, this will give us a base map to display our homicide points and neighborhood selections against.

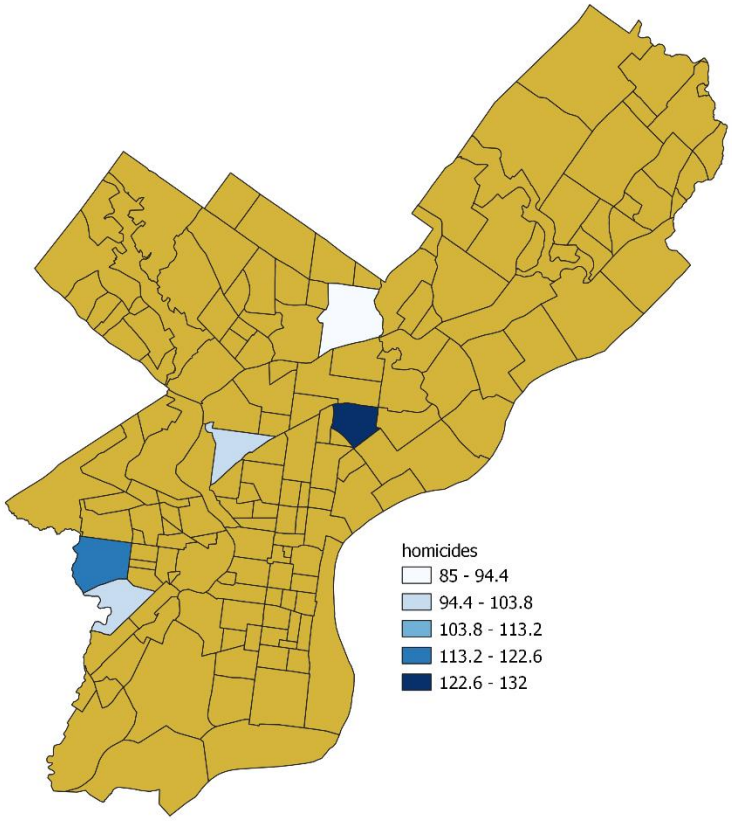
```
create or replace view neighborhoods as select name, geom from nhoods;
```

The first spatial query we will be doing allows us to total the number of homicides in each Philadelphia neighborhood. We will be ordering the results in descending order and limiting it to 5 results, showing us the 5 "most dangerous" neighborhoods in Philadelphia.

```
create or replace view "most dangerous" as
select n.mapname as neighborhood, count(s.geom) as homicides, n.geom
from homicide s JOIN nhoods n
on st_contains(n.geom, st_transform(s.geom, 2272))
where s.sid = 47
group by n.mapname, n.geom
order by count(s.geom) desc
limit 5;
```

The neighborhood name, homicide count, and geom are returned to us after joining the homicide and nhoods tables using the st_contains function which finds all the homicide points within each neighborhood polygon. While the neighborhoods table is already in Pennsylvania state plane south the geom of the homicides must be transformed on the fly. This will be true for all the queries we do in this tutorial. The where clause makes sure that we are only dealing with homicide records that reference the sid of 47 which is Philadelphia, and we group by the neighborhood name and geom. Lastly, we get the list of neighborhoods in order from greatest to least homicides and limit the return to 5.

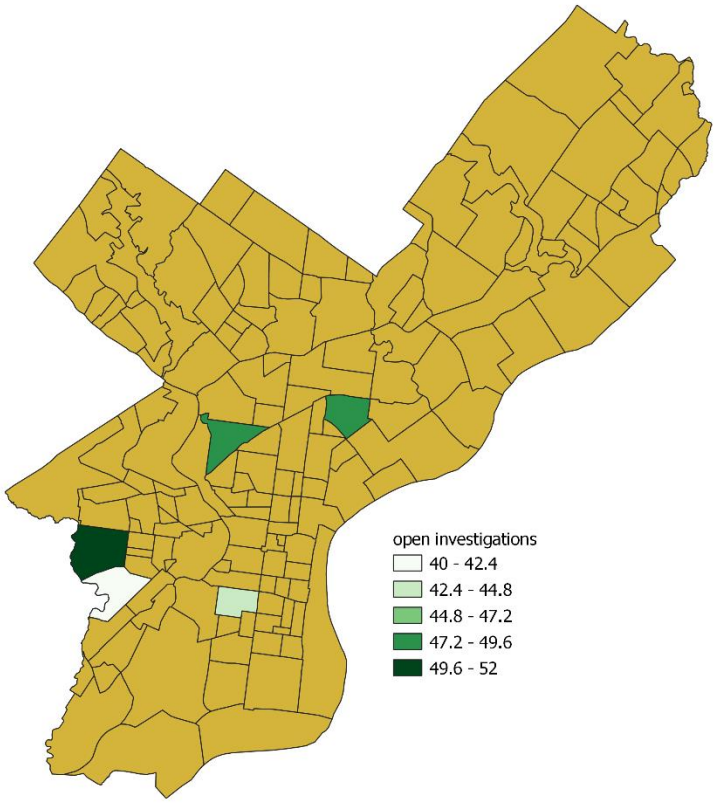
neighborhood	homicides
Upper Kensington	132
Cobbs Creek	115
Strawberry Mansion	102
Kingsessing	95
Olney	85



By adding to the where clause we can find the neighborhoods that have to most open homicide investigations. A cid of 3 means that the investigation is open as opposed to closed with arrests or closed without arrests.

```
create or replace view "most open investigations" as

select n.mapname as neighborhood, count(s.geom) as open investigations,
n.geom

from homicide s JOIN nhoods n

on st_contains(n.geom, st_transform(s.geom, 2272))

where s.sid = 47 and s.cid = 3

group by n.mapname, n.geom

order by count(s.geom) desc

limit 5;
```

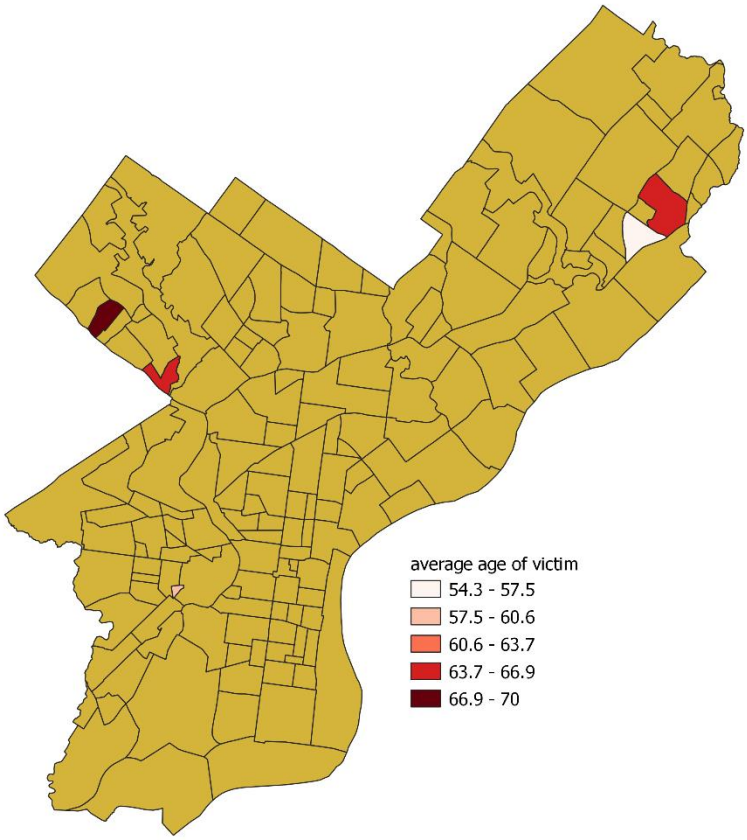
neighborhood	open investigations
Cobbs Creek	52
Strawberry Mansion	49
Upper Kenington	48
Point Breeze	43
Kingsessing	40



The second spatial query we will be preforming will be the same general syntax, but it will involve joining a third table, the victim table. In this query we want to find the average age of homicide victims in each neighborhood. At first, we will sort by descending to show the 5 neighborhoods with the highest average age.

```
create or replace view "average_homicide_age_high" as
select n.mapname as neighborhood, avg(v.age)as "average age", n.geom
from homicide s join victim v using (vid) JOIN nhoods n
on st_contains(n.geom, st_transform(s.geom, 2272))
where s.sid = 47
group by n.mapname, n.geom
order by avg(v.age) desc
limit 5;
```

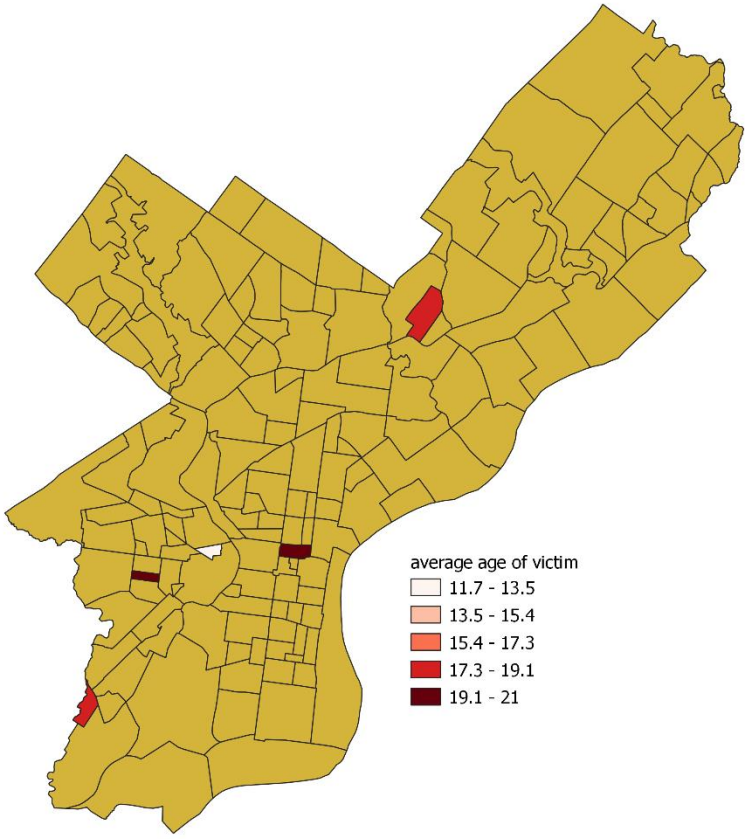
neighborhood	average age
Roxborough Park	70.0
Morrell Park	64.0
Wissahickon	64.0
Woodland Terrace	60.0
Academy Gardens	54.333



Now we change the order by clause to sort by ascending to find the neighborhoods with the lowest average age.

```
create or replace view "average_homicide_age_low" as
select n.mapname as neighborhood, avg(v.age)as "average age", n.geom
from homicide s join victim v using (vid) JOIN nhoods n
on st_contains(n.geom, st_transform(s.geom, 2272))
where s.sid = 47
group by n.mapname, n.geom
order by avg(v.age) asc
limit 5;
```

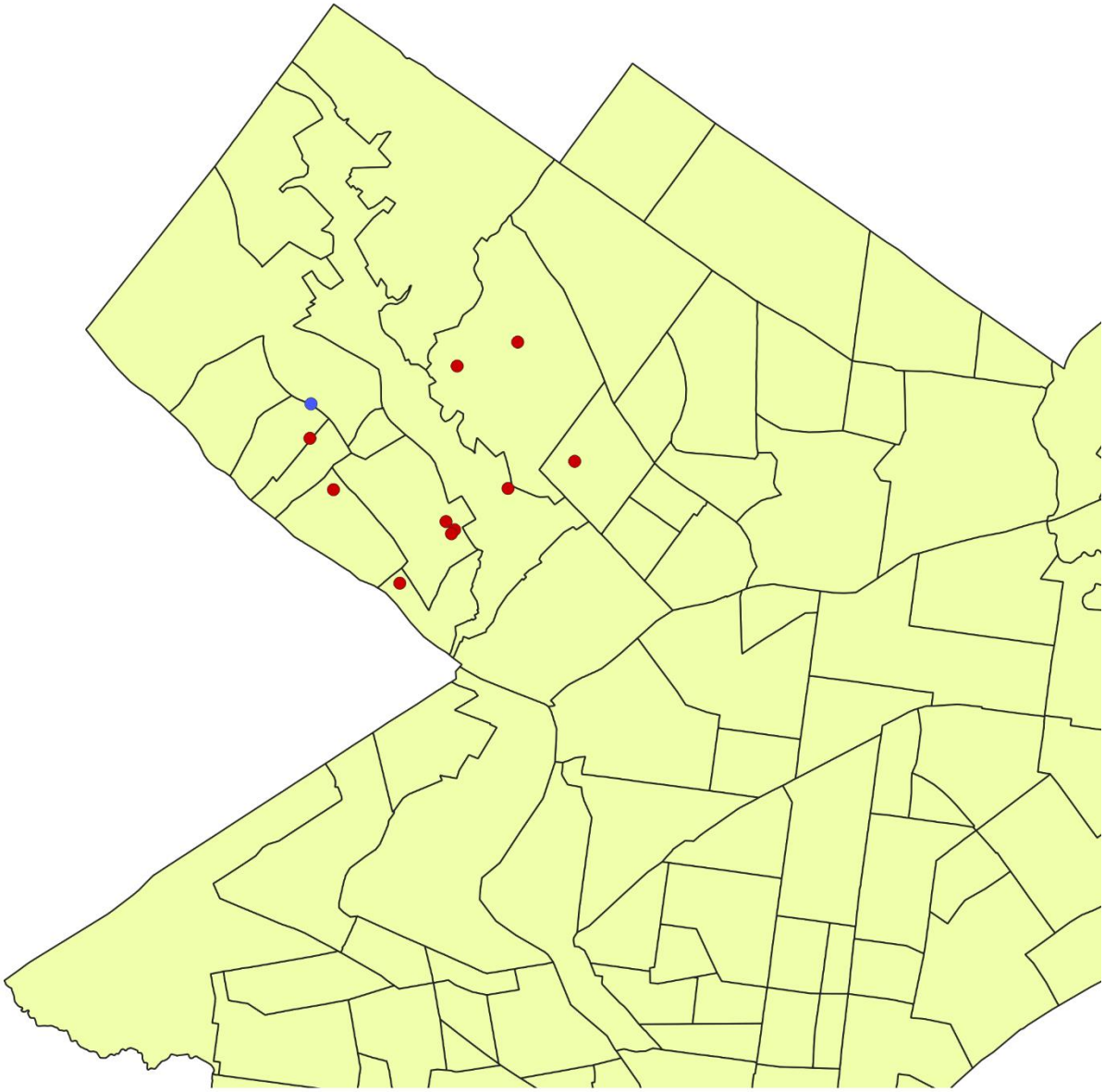
neighborhood	average age
Powelton	11.666
Crescentville	17.333
Clearview	19.0
Callowhill	20.5
Garden Court	21.0



The last spatial query we will do uses a KNN bounding-box centroid distance operator. It will return the distance between the centroids of the geoms you enter but, in this case, it will be a user defined point and the geom from the homicide table limited to 10 returns giving us the 10 closest homicides to out user defined point. Out table will consist of the first name, last name, and age of the homicide victim.

```
create or replace view "closest to me" as

select v.first_name, v.last_name, v.age, h.geom

from homicide h join victim v using (vid)

order by st_transform(st_setsrid(st_point(-75.229028, 40.044249),
4326),2272) <-> st_transform(h.geom, 2272)

limit 10;
```

first_name	last_name	age
KATHLEEN	MCEWAN	70
JOHN	MURRAY	51
TOY	CHARDA BRYANT	26
JASON	LARKIN	3
CINDY	CLOTHIER	35
TERRELL	BRUCE	33
LYNDA	MITROS	64
DANIEL	COOK	27
JOHN	FULLARD	36
RONALD	WILLIAMS	64



The blue point denotes out user defined point while the red points represent the nearest homicides.

Appendix i

The data used in this tutorial was compiled by the Washington Post and includes data about solved and unsolved homicides in 50 different US cities totaling over 50,000 records from 2007 to 2017.

The Washington Post's map can be found here.

The data in CSV format can be downloaded here.

The Philadelphia neighborhoods polygon we used to help run our spatial queries and create the base map for our views can be found on Azevea's github here: Philadelphia Neighborhood Shp

Open Dbeaver or your preferred database managing software and create a new PostGIS enabled database with the following commands.

```
create database homicides;

create extension postgis;
```

After downloading the data from the Washington Post github and the Azevea github open a OSGeoW4 shell and navigate to the appropriate directory then run the following ogr2ogr commands.

```
ogr2ogr -f PostgreSQL PG:"host=localhost port=5432 dbname=homicides
user=postgres password=password" -lco SCHEMA=hdata -lco PRECISION=NO -
oo EMPTY_STRING_AS_NULL=YES homicide_data.csv -nln "data_import"
```

This command moves the homicide data to the database we just created and places it in a schema called hdata and into a table named data_import.

```
ogr2ogr -f PostgreSQL PG:"host=localhost port=5432 dbname=homicides
user=postgres password=password" -lco SCHEMA=hdata -nlt
PROMOTE_TO_MULTI -lco PRECISION=NO Neighborhoods_Philadelphia.shp -nln
"nhoods"
```

This command moves the Philadelphia neighborhoods data into our database and into table named nhoods.

Congratulations you are now ready to move onto the ETL process.

Appendix ii

This appendix documents the ETL script itself used to normalize the homicide data we loaded into our database.

```
set search_path to hdata, public;


--delete rows with empty lat and lon coords (60 of 51000 plus deleted)

delete from data_import

where lon is null or trim(lon) = '';


/*creates victim table

 * I included the uid from the data_import table becuase there are 4147 identical victims that share first name, last name, age, sex and race.

 * Using a subquery to return one result and assign a serial primary key result proved difficult without the uid column

 *

 * select count(*) from victim a

 * where (select count(*) from victim b

 * where a.first_name = b.first_name and a.last_name = b.last_name and a.age = b.age and a.sex = b.sex and a.race = b.race) > 1;

 * count: 4147

 *

 * uid is deleted later

 */

drop table if exists victim cascade;

create table victim (

      vid serial primary key,

      uid varchar,

      last_name varchar,

      first_name varchar,

      age int,

      sex varchar,

      race varchar);
```

```
--populates victim table
insert into victim (uid, last_name, first_name, age, sex, race)
select uid, victim_last, victim_first, case when victim_age = 'Unknown'
then null else victim_age end::int, victim_sex, victim_race
from data_import;


--creates city_state table
drop table if exists city_state cascade;
create table city_state (
      sid serial primary key,
      city varchar,
      state varchar);


--populates city_state table
insert into city_state(city, state)
select distinct city, state
from data_import;


--creates case_status table
drop table if exists case_status cascade;
create table case_status (
      cid serial primary key,
      status varchar);


--populates case_status table
insert into case_status(status)
select distinct disposition
from data_import;
```

```
--creates homicide table
drop table if exists homicide cascade;
create table homicide (
	gid serial primary key,
	vid int references victim(vid),
	sid int references city_state(sid),
	cid int references case_status(cid),
	reported_date date,
	geom geometry);


--populates homicide table
insert into homicide(vid, sid, cid, reported_date, geom)
select (select vid from victim where victim.uid = data_import.uid),
	(select sid from city_state where (city_state.city ||
city_state.state) = (data_import.city || data_import.state)),(select
cid from case_status where case_status.status =
data_import.disposition),
	reported_date::date,
	st_transform(st_setsrid(st_point(lon::double precision,
lat::double precision), 4326), 4326) --cast to 2272 for philly in
analysis
from data_import;


--uid column used to assign primary key is redundant and deleted here
alter table victim
drop column uid cascade;
```

I also made a small change to the neighborhoods data that we brought in changing the name of the geometry column from wkb_geometry to just geom.

```
alter table nhoods
rename column wkb_geometry to geom;
```

Appendix iii

Indexes:

```
create index philly_idx on homicide (sid)
```

Appendix iiii

Data Dictionary

victim:

vid - serial - unique victim id

last_name - varchar - victim last name

first_name - varchar - victim first name

age - int - victim age

sex - varchar - victim sex

race varchar - victim race

city_state:

sid - serial - unique city and state id

city - varchar - city of homicide

state - varchar - state abbreviation

case_status:

cid - serial - unique id of case disposition

status - varchar - current status of homicide

case_status:

cid - serial - unique id of case disposition

status - varchar - current status of homicide

homicide:

gid - serial - unique homicide id

vid - int - unique victim id (references victim table)

sid - int - unique city and state id (references city_state table)

cid - int - unique case is (references case_status table)

reported date - date - reported date of homicide

geom - geometry - point location of homicide (WGS84)