How to use Tern Data Creator (TDC)

github

Tern Data Creator (TDC) is a python code that creates a database of site information when given a city name. The database has 27 columns: card id, place id, geometry (coordinates), business status, name of the palace, address, google map url, tags, last updated, user id of the card creator (Tern's id for the default data), created time, rating, number of ratings, likes, dislikes, must dos, popularity, top popularity, relevant links, website, opening hours, phone number, photo reference code, reviews, top card id for the place, photo, and description.

TDC uses two types of requests from Google places API: place search (nearby search) and place details. Nearby search takes a location in latitude and longitude as a required input, and keyword (such as "sights in SF"), radius of the search area, and next page token as optional input. The response returns a lot of information about the sites. In fact, about half of the data in the final database comes from this response, including place id. We can then use the place id to request for place details. Place details return additional information about the place, including the rating, phone number, and opening hours.

In order to use the Google places API, you need to get an API key. You can get the API key by following the instructions in this link. Also, you would need to install Python, Pip, and install all the python libraries needed in the code, including googlemaps, openpyexl, and pandas. You can install them with the command- "pip install xxx".

The main function of Tern Data Creator is create_table function. The function takes in a city name and an index as arguments. City name should be in string and index should be an integer, where the number represents the index where the new data should be stored at. For instance, if the last index of an existing dataset is 59, then 60 should be used for the index of the new dataset of a city. The python for loop in the script automates the process of updating index, and iterates through a list of cities. The for loop appends each pandas dataframe for a city to a list (frames), and pd.concat() allows merging all the dataframes into one with a continuous indexing.

The steps above will generate all the data except the description for the places. The descriptions are retrieved from the Wikipedia or from Google search results if not in Wikipedia. Function **inWikipedia** takes in a wikipedia url as an argument, and returns true if the wikipedia page exists and false otherwise. Inside the create_table function, 'add descriptions section' first creates either a wikipedia link or a google search link if there is no wikipedia page for it. Then, using the link, the function operates HTTP requests and parses the response using the Beautiful soup library.

To connect to the Azure virtual machine, start the virtual machine 'test-vm-01'. Once it starts running, you can connect to the virtual machine using RDP, SSH, or Bastion (I tried RDP and Bastion and they both worked). For authentication, the username is 'adminuser' and the password is 'Tern2022!!@@' (make sure to capitalize T).

It took about an hour to create data for all the cities in California with a population greater than 200,000, which was 23 cities and 1371 sites. The runtime has reduced a lot from the first time where it took 2.5 hours, but it might still need to get faster to scrape larger data. The file is in both excel and csv form with names 'California_final.xlsx' and 'California_final.csv', respectively.

I did not yet add the data to the SQL database, but you can follow the instructions below to create an Azure Database with the scraped data. This worked well for our initial data with 60 rows, but there could be some issues with the new data due to its huge size. In those cases, you might want to change some of the constraints in SQL or check if there are any exceptional cases in the data and edit the python code.

Steps to create Azure Database:

1. Follow the steps in the link, just select the ‘Development’ workload type in Step 4 to get free access upto a certain limit.
2. In Step 6, select ‘Public Access’ and set IPs (0.0.0.0 - 255.255.255.255) for experimentation. Use Private VNet for secure access, but it requires a paid subscription.
3. To connect to the database, use steps this link. The database can be accessed from the Azure CLI in the Azure Portal or a 3rd-party software like the pgAdmin. For pgAdmin use this link to connect.
4. To add PostGIS extension to Azure PostgreSQL Database, watch this video from 13:15.

Creating tables:

1. Cards Table

```
CREATE TABLE IF NOT EXISTS public."Cards"
(
    card_guid integer NOT NULL DEFAULT nextval('"Cards_card_guid_seq"'::regclass),
    place_id_guid integer,
    name character varying(2100) COLLATE pg_catalog."default" NOT NULL,
    description character varying(3000) COLLATE pg_catalog."default",
    photos bytea,
    tags character varying(2100) COLLATE pg_catalog."default",
    created_at date NOT NULL,
    created_by character varying(2100) COLLATE pg_catalog."default" NOT NULL,
    likes integer,
    dislikes integer,
    must_dos integer,
    popularity real,
    links character varying(2100)[] COLLATE pg_catalog."default",
    place_id character varying(30) COLLATE pg_catalog."default",
    CONSTRAINT "Cards_pkey" PRIMARY KEY (card_guid),
    CONSTRAINT "Photos" UNIQUE (photos)
        INCLUDE(photos)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public."Cards"
    OWNER to ternadmin;
```

2. Unique_Sites

```
-- Table: public.Unique_Sites
-- DROP TABLE IF EXISTS public."Unique_Sites";

CREATE TABLE IF NOT EXISTS public."Unique_Sites"
(
    place_id_guid integer NOT NULL DEFAULT
nextval('"Unique_Sites_place_id_guid_seq"'::regclass),
    place_id character varying(30) COLLATE pg_catalog."default" NOT NULL,
    business_status integer,
    address character varying(2100) COLLATE pg_catalog."default",
    google_maps_url character varying(2100) COLLATE pg_catalog."default" NOT
NULL,
    open_hours character varying(500) COLLATE pg_catalog."default",
    website character varying(2100) COLLATE pg_catalog."default",
    phone_number character varying(25) COLLATE pg_catalog."default",
    ratings real,
    user_ratings_total integer,
    top_popularity real NOT NULL,
    top_card integer,
    geometry double precision[] NOT NULL,
    reviews text COLLATE pg_catalog."default",
    geom geometry(Point,4326),
    CONSTRAINT "Unique_Sites_pkey" PRIMARY KEY (place_id_guid),
    CONSTRAINT "Top Card" UNIQUE (top_card)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public."Unique_Sites"
    OWNER to ternadmin;
-- Index: idx_unique_sites

-- DROP INDEX IF EXISTS public.idx_unique_sites;

CREATE INDEX IF NOT EXISTS idx_unique_sites
    ON public."Unique_Sites" USING gist
    (geom)
    TABLESPACE pg_default;
```

Executing Distance-based Queries (Modified version of Anna's email):

To run proximity queries, lat and lon are not enough, we need a geometry field to speed things up (geography may be a little too much according to docs but they're being vague). geometry fields can be built based on lat, lon or imported, but i wanted to create mine after importing the lat, lon only. Populate the field and create an index on it

```
SELECT AddGeometryColumn('Unique_Sites', 'geom', 4326, 'POINT', 2);   //the  2 and 4326 are important, need to be hardcoded
```

```
UPDATE Unique_Sites SET geom = ST_SetSRID(ST_MakePoint(geometry[2], geometry[1]), 4326);
```

```
CREATE INDEX idx_unique_sites ON Unique_SItes USING gist(geom);
```

here's how you can run a query on that data using some random location in New York. Note that the first point in ST_MakePoint is the LON , the second one is LAT

```
SELECT *
FROM Unique_Sites
WHERE ST_DWithin(geom, ST_MakePoint(-74.0060, 40.7128)::geography, 1000);
```

This query searches within **1000 meters of a latitude and longitude provided** . It's important to cast the results of ST_MakePoint function to geography , otherwise it will search 1000 degrees instead of meters.