Data Storage and Retrieval – Assignment 6

Aaron Niskin

October 7, 2016

For the first three problems, discuss how well the data supports answering the question and what assumptions and judgment calls you made.

1. **Rank movies by number of sequels.**

   So off the bat we can tell that there won't be any good solution to this due to the fact that there's no set schema for naming sequels. But we can get an estimate if we assume 2 (admittedly incorrect) things.

   1. That every sequel starts with the original movie name. i.e. Star Wars sequels will start with "Star Wars", Rocky sequels will start with "Rocky", etc. This may not be true in general, for instance, "Meet the Fockers" is the sequel of "Meet the Parents" (a counter example to this rule).
   2. That any movie that starts with the full name of another movie is a sequel to that movie. So, for instance, any movie starting with "Rocky" is a sequel of Rocky. Although this is true for Rocky in our databse, it's not true in general. Especially not if you include movies with such general titles as "It".

   But if we make these assumptions, we can accomplish somethings.

```
CREATE TEMPORARY TABLE movies_sequels AS
  SELECT m.title title, COUNT(*) - 1 sequels
    FROM movies m
    JOIN movies b
      ON (POSITION(m.title IN b.title) = 1)
      GROUP BY m.title;
SELECT * FROM movies_sequels ORDER BY sequels DESC LIMIT 10;
```

```
  title   | sequels
----------+---------
 M        |     164
 The Man  |      14
 Blondie  |      11
 Big      |       7
 Blue     |       6
 She      |       5
 Angel    |       5
 Heart    |       5
 Midnight |       5
 Rocky    |       4
(10 rows)
```

   So obviously movies like "M" don't have 164 sequels, but what can you do?

2. **Find all misspelled actor names in the movies database actors table.**

   This sounds like an exceedingly difficult problem. Especially since there are almost 5,000 actors and actresses in this database. However, this seems like a job for our good friend, "Dr. Levenshtein"! If we set a levenshtein distance of, say, 2, we can get a pretty good estimation, I suppose.

```
SELECT a.actor_id, a.name, b.name, b.actor_id
  FROM actors a
    JOIN actors b
    ON levenshtein(lower(a.name), lower(b.name)) < 2
      AND a.actor_id < b.actor_id
  LIMIT 10;
```

```
 actor_id |       name       |       name       | actor_id
----------+------------------+------------------+----------
      248 | Annette O'Toole  | Annette O`Toole  |      249
      823 | Claude Akins     | Claude Atkins    |      824
      824 | Claude Atkins    | Claude Atkuns    |      825
      940 | Dan O'Herlihy    | Dan O`Herlihy    |      942
     1002 | David Bowe       | David Bowie      |     1003
     1145 | Dick Van Dyke    | Dick van Dyke    |     1147
     1408 | Essie Davis      | Ossie Davis      |     3627
     1477 | Frances O'Connor | Frances O'Connor |     1478
     1522 | Frederic Forrest | Frederick Forrest |     1525
     1590 | George Coe       | George Cole      |     1591
(10 rows)
```

And sure enough, all of those seem like mispellings.

3. **Write a python program to maintain a table of Kevin Bacon numbers. Upon invocation, the program should update the table, only if it needs updating, and should then expose a prompt to the user to respond to queries. A query is an actor's name, possibly misspelled, and the response is a shortest chain of actors and movies to Kevin Bacon. For a challenge, produce a graphical representation of the chain of actors/movies from the actor to Kevin Bacon.**

```
SELECT * FROM actors
  WHERE levenshtein(lower(name), 'kevin bacon') < 4;
```

```
 actor_id |     name
----------+--------------
     2720 | Kevin Bacon
     2733 | Kevin Nealon
(2 rows)
```

So there are no missellings of Kevin Bacon's name (thank god). So let's start by building the initial graph (we'll use something like a lazy-man's Dijkstra's algorithm).

```
CREATE TABLE bacon_numbers AS
  SELECT actor_id, NULL bacon_number
    FROM actors;
```

We then create a temporary table to house new movies and a rule for inserting into the `movies_actors` table so that they also get inserted into our new table.

```
CREATE TABLE new_movies_for_bacon_numbers
  (movie_id INT REFERENCES movies,
  actor_id INT REFERENCES actors,
  PRIMARY KEY (movie_id, actor_id));
```

```
CREATE OR REPLACE RULE insert_into_new_movies
  AS ON INSERT TO movies_actors
    DO ALSO
      INSERT INTO new_movies_for_bacon_numbers VALUES (NEW.movie_id, NEW.actor_id);
```

So after messing with this a bit, I realized that I would essentially have to redo everything anyway (or store the intermediaries in an array along with a few other hacks) any time a movie were inserted. There's probably a better way to do this, but I haven't thought of it yet. Here's my python script for now though:

```
import csv
import psycopg2
import datetime

#conn = psycopg2.connect("dbname=movies_aaron user = aniskin")
conn = psycopg2.connect("dbname=movies user = amniskin")
cur = conn.cursor()
cur.execute("SELECT * FROM bacon_numbers;")
bacon_numbers = cur.fetchall()
if len(bacon_numbers) == 0:
    # First to set Kevin Bacon's ID number. This could be done by checking every time, but since we
    bacon = 2720
    cur.execute("SELECT * FROM movies_actors;")
    movies_actors = cur.fetchall()
    cur.execute("DELETE FROM new_movies_for_bacon_numbers;")
    cur.execute("SELECT * FROM actors;")
    actors = cur.fetchall()
    bacon_numbers = [(actor,None,None) for actor,i in actors]
    ###  Here, I'm making an array of actors. Each actor will have three attributes:
    ######  0. actor ID (which is also the actor's index + 1 in the actors array)
    ######  1. current minimal distance
    ######  2. The closest intermediate through which the shortest distance is.
    bacon_numbers[bacon - 1] = (bacon_numbers[bacon-1][0], 0, bacon)
    new_movies = [x for (x,y) in movies_actors if y == bacon]
    vis_movies = set(new_movies)
    vis_actors = [bacon]
    new_edges = [(bacon, y) for (x,y) in movies_actors if x in new_movies and y != bacon]
    count = 0
    while len(new_edges) > 0:
        count = count + 1
        new_actors = []
        for (old, new) in new_edges:
            bacon_numbers[new - 1] = (bacon_numbers[new - 1][0], count, old)
            vis_actors.append(new)
            new_actors.append(new)
        new_movies = [(mov, actor) for (mov,actor) in movies_actors if mov not in vis_movies and ac
        new_edges = []
        for (mov, old) in new_movies:
            if mov not in vis_movies:
                tmp = [(old, new) for (m, new) in movies_actors if m == mov and new not in vis_acto
                for old,new in tmp:
                    vis_actors.append(new)
                new_edges.extend(tmp)
                vis_movies.add(mov)
```

```
    with open('bacon_numbers.csv', 'w', newline='') as bn:
        a = csv.writer(bn, delimiter=',')
        data = [('actor_id', 'bacon_number', 'intermediary')]
        data.extend(bacon_numbers)
        a.writerows(data)
    # Commit the changes to the database and close the cursor and connection to the DB
    cur.executemany("INSERT INTO bacon_numbers VALUES(%s, %s, %s)", bacon_numbers)
    conn.commit()

cur.close()
conn.close()
```

4. a. **Find the Postgres page size (aka block size) in one of the system tables. (The system tables are named pg_*.)**

```
SELECT * FROM pg_settings WHERE name LIKE '%block%' LIMIT 10;
```

```
      name      | setting | unit |    category    |                  short_desc
----------------+---------+------+----------------+---------------------------------------------
 block_size     | 8192    |      | Preset Options | Shows the size of a disk block.
 wal_block_size | 8192    |      | Preset Options | Shows the block size in the write ahead log
(2 rows)
```

b. **The pg_stats table gives the average number of bytes per value (avg_width) of each column of a table. How many bytes are in the average row of the stories table?**

```
SELECT avg_width, attname FROM pg_stats
  WHERE tablename = 'movies';
```

```
 avg_width | attname
-----------+----------
         4 | movie_id
        15 | title
       152 | genre
```

So, about 171 byes per row (on average).

c. **A database page contains a header, row pointers, and row data, as described here. How many disk page fetches are needed for the fanfiction query**

```
SELECT title FROM stories WHERE title = 'Die, Harry Potter, Die';
```

i. **if there's no index (compare your answer to explain's cost)?**
So at first I figured (if postgres were intelligent) this would be as simple as recognizing that you're requesting the very thing you specified and hence no search needed. But this is not true becauseif there is no such movie, we expect this query to return with an empty table. So this is not as simple as I'd thought.
As far as how many pages this query would take, on average $\frac{n}{2}$. Due to the fact that records are stored in heaps on pages without any regard to an order, we are left to simply search page after page until we find it.

```
EXPLAIN SELECT title FROM stories_orig WHERE title = 'Die, Harry Potter, Die';
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on stories_orig  (cost=0.00..73541.31 rows=4 width=19)
   Filter: ((title)::text = 'Die, Harry Potter, Die'::text)
(2 rows)
```

So, for some reason it's only fetching about 74k pages when it's not indexed. This suggests to me that postgresql is doing something intelligent here with the storage. Each record should be about a page (some even more) due to the length of our fields. I think postgresql may have split up the table into separate columns with regard to pagination, which messes with our results.

ii. **if there's a B-tree index (compare your answer to explain's cost)?**
This should be about log base (branching factor) of the number of records to find the page. Then it depends on how they store the titles. But since the titles cant' get too long, they can probably store quite a few rows on a page, making it much cheaper.