TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Dr. Thomas Neumann

Exercises for *Foundations in Data Engineering*, WiSe 23/24

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)

http://db.in.tum.de/teaching/ws2324/foundationsde

Sheet Nr. 08

Exercise 1

In this exercise, we want to support fast roll-up and drill-down analytics of trade volumes in the `TPC-H SF 1` dataset with the help of the cube operator and precomputed result.

To be able to materialize precomputed results you need database instance with write privileges. As this is not supported by our WebInterfaces, we suggest you create a local instance of a Postgres database. This guide gives a nice introduction on how to set up Postgres on Ubuntu. Once you set up an instance, import the `TPC-H` dataset (schema) from `Sheet 02` with the commands:

```
cat schema.sql | psql
sed -i 's/|$//' *.tbl
// Execute this command for each table:
psql -c "\copy <tablename> from <tablename.tbl> delimiter '|'";
// For example for lineitem:
psql -c "\copy lineitem from lineitem.tbl delimiter '|'";
```

Then, create these two queries as examples of analytical workloads:

1. Create a query to determine the trade volume in the US automobile market. In this example, we use `l_quantity * l_extendedprice` as trade volume for one `lineitem`. The volume of the US automobile market is the `sum` of lineitem's trade volumes where the corresponding customer is from the `United States` and their market segment is `'AUTOMOBILE'`.
   Solution:

   ```
   SELECT sum(l_quantity * l_extendedprice)
   FROM lineitem,
        orders,
        customer,
        nation
   WHERE l_orderkey = o_orderkey
     and o_custkey = c_custkey
     and c_nationkey = n_nationkey
     and c_mktsegment = 'AUTOMOBILE'
     and n_name = 'UNITED STATES';
   ```

2. Create another query to find the trade volume of goods bought by customers on the national market. (Supplier's nation equals customer's nation)

Solution:

```
SELECT sum(l_quantity * l_extendedprice)
FROM lineitem,
     orders,
     customer,
     supplier
WHERE l_orderkey = o_orderkey
  and o_custkey = c_custkey
  and l_suppkey = s_suppkey
  and s_nationkey = c_nationkey;
```

To speed up the execution of this kind of queries, materialize precomputed aggregates with the help of the `cube` operator.

1. Materialize a data cube on the dimensions `c_mktsegment`, `c_nation`, `s_nation`. In each group, sum up `l_quantity * l_extendedprice`.

   Solution:

   ```
   CREATE TABLE volume_cube(
      volume bigint,
      c_mktsegment character(10),
      c_nationkey integer,
      s_nationkey integer);

   INSERT INTO volume_cube
   SELECT sum(l_quantity * l_extendedprice),
          c_mktsegment,
          c_nationkey,
          s_nationkey
   FROM lineitem,
        orders,
        customer,
        supplier
   WHERE l_orderkey = o_orderkey
     and o_custkey = c_custkey
     and l_suppkey = s_suppkey
   GROUP BY cube(c_mktsegment, c_nationkey, s_nationkey);
   ```

2. Reformulate the two above SQL queries to get their data from `volume_cube`. Compare the runtime of these queries to the queries that did not use the cube.

   Solution:

   ```
   -- us automobile volume
   SELECT volume
   FROM volume_cube, nation
   WHERE c_mktsegment = 'AUTOMOBILE'
     and c_nationkey = n_nationkey
     and n_name = 'UNITED STATES'
     and s_nationkey is null;
   ```

```
-- national volume
SELECT sum(volume)
FROM volume_cube
WHERE c_nationkey = s_nationkey
  and c_mktsegment is null;
```

Would it be sensible to create such a data cube for the dimensions `p_partkey`, `ps_partkey`, `o_orderkey`, and `c_custkey`?

Solution:
Number of unique elements in each table: $200k * 800k * 1500k * 150k = 3.6e22$ elements
This represents a lower bound on the number of entries the cube will have. With 4 Byte per Element: $1.44e23$ Byte = 192 Zebibyte No, not sensible.

Exercise 2 Six Degrees of Kevin Bacon

In a January 1994 Premiere magazine interview about the film The River Wild, the actor Kevin Bacon commented that he had worked with everybody in Hollywood or someone who's worked with them. A popular game revolving around the "six degrees of separation" concept and Kevin Bacon as the center of Hollywood evolved from this observation.

Background Story

On April 7, 1994, a lengthy newsgroup thread headed "Kevin Bacon is the Center of the Universe" appeared. The game was created in early 1994 by three Albright College students, Craig Fass, Brian Turtle, and Mike Ginelli. According to an interview with the three in the spring 1999 issue of the college's magazine, The Albright Reporter, they were watching Footloose during a heavy snowstorm. When the film was followed by The Air Up There, they began to speculate on how many movies Bacon had been in and the number of people he had worked with. In the interview, Brian Turtle said, "It became one of our stupid party tricks I guess. People would throw names at us and we'd connect them to Kevin Bacon." Wikipedia: Six Degrees of Kevin Bacon

Today, *Six Degrees of Kevin Bacon* is a parlour game based on the "six degrees of separation" concept, which posits that any two people on Earth are six or fewer acquaintance links apart.

Task

In this exercise, we want to investigate the question, whether all the actors listed in the Internet Movie Database are separated from Kevin Bacon by less than a degree of 6. To that end, we assign each actor a Bacon Number. Those actors that appeared in a movie in which also Mr. Bacon appeared, receive Bacon number 1. Actors that did not work with Kevin Bacon, but with someone of Bacon number 1 are assigned number 2 and so on. The question then is: Are there actors or actresses with a Bacon number of 7 or higher? Tutorial: Install & Use PostgreSQL

Preparing the Data Set

1. Download the IMDb actors file (actors.list.xz)
2. Prepare the data so that it can be loaded into a Postgres database table with the name `playedin_text`, with the schema `(actor::text, movie::text)`. Take special care to remove everything from the movie titles that is not the title itself and the year of appearance. This will allow to find other actors which worked on the same movies. If you decide to use `gnu utils` for data preparation, you may need to convert the actors file from Latin-1 character set to UTF-8 with the `iconv` command.

3. Load the data into Postgres, preferably using the command `\copy`. Check that the number of distinct actors is ca. 2 million, the number of movies is ca. 1 million.
4. At this point, you may want to tweak your Postgres configuration a little for better analytical query performance. For example: Allow the database to use more main memory for queries by setting `work_mem` to up to two thirds of the available memory. Tell the query planner that Postgres can use all available main memory by setting `effective_cache_size` to your physical main memory size minus 1 GB. On Ubuntu 20.10, these configuration settings are in file `/etc/postgresql/12/main/postgresql.conf`. For changes to this file to take effect, Postgres needs a restart with the command

   ```
   sudo service postgresql restart
   ```

Solution:

```
# Trim start and end of actors.list into actorsTrimmed.list
cat actors.list | tail -n +240 | head -n -272 >> actorsTrimmed.
   list
# convert to utf-8
iconv -f latin1 -t utf-8 actorsTrimmed.list > actorsUtf8.list

sed -r -e 's/\t+/|/g' -e 's/((\s*)|(\{.*\})|(\[.*\])|(<.*>)|\([a
   -zA-Z].*\))*$//g' actorsUtf8.list | awk -F'|' '{if($1 != ""){
   actor=$1}; if ($2) print actor "|" $2}' > actors.csv

sudo -u postgres createuser <your user>
sudo -u postgres createdb imdb
psql -d imdb
CREATE TABLE playedin_text (
    actor_name text not null,
    movie_name text not null);
\copy playedin_text from actors.csv delimiter '|';
```

Formulating the Query

1. Let's first find out some basic questions: In how many movies has Kevin Bacon starred? With how many actors has he worked together?

   Solution:

   ```
   -- basic1.sql
   -- #movies Kevin Bacon starred in
   SELECT count(distinct movie_name)
   FROM playedin_text
   WHERE actor_name = 'Bacon, Kevin';

   -- #actors he worked with
   SELECT count(distinct p2.actor_name)
   FROM playedin_text p1,
        playedin_text p2
   WHERE p1.actor_name = 'Bacon, Kevin'
     and p1.movie_name = p2.movie_name;
   ```

2. Formulate a query with a recursive view, which finds the number of actors that have a Bacon Number $<= c$ where c is a given constant.

Solution:

```
--bacon_text.sql
with recursive baconnr (actor_name, nr) as (
        SELECT 'Bacon, Kevin', 0
        UNION
        SELECT p2.actor_name,
               baconnr.nr + 1
        FROM baconnr,
             playedin_text p1,
             playedin_text p2
        WHERE baconnr.actor_name = p1.actor_name
          and p1.movie_name = p2.movie_name
          and baconnr.nr < c
)
SELECT count(distinct actor_name)
FROM baconnr;

-- To only get actors with Bacon Number equal to c, you can
   use:
with recursive baconnr (actor_name, nr) as (
... see above ...
)
SELECT count(distinct actor_name)
FROM baconnr;
WHERE nr = c;
```

3. Run the recursive query, so that it counts all actors with Bacon number $<= 1$. Enable the psql `\timing` feature to keep track of execution times.
4. Now, try to determine all actors with Bacon number up to 2. Stop the query if it does not finish within one minute.

Tweaking the Query

1. How come this simple degree 2 query does not stop? (*Hint:* Have a look at the size of intermediate results in the query. Try to determine the size by appropriate SQL queries.)

   Solution:

   ```
   -- bacon_degree2_cardinality.sql
   SELECT count(p2.actor_name)
   FROM playedin_text p1,
        playedin_text p2
   WHERE 'Bacon, Kevin' = p1.actor_name
     and p1.movie_name = p2.movie_name;
   -- compare this to distinct count
   ```

2. Is there any mean to reduce the size of intermediate results and thus speed up the query evaluation? (*Hint:* Try to remove duplicates after every join. With this technique, the query should finish within reasonable time for $c <= 2$.)

Solution:

```
-- bacon_unique_text.sql
WITH recursive baconnr (actor_name, nr) as (
        SELECT 'Bacon, Kevin', 0
UNION ALL
        SELECT distinct p2.actor_name,
           movies.nr + 1
        FROM (SELECT distinct p1.movie_name,
                     baconnr.nr
          FROM baconnr,
               playedin_text p1
          WHERE baconnr.actor_name = p1.actor_name
          ) movies,
          playedin_text p2
        WHERE movies.movie_name = p2.movie_name
      and movies.nr < 2
)
SELECT count(distinct actor_name)
FROM baconnr;
```

3. Working on strings is expensive. We can reduce the size of the `playedin_text` table by using integers instead of text to express the same information. Create a table `actors` (`id::integer`, `name::text`), a table `movies` (`id::integer`, `name::text`) and a table `playedin` (`actor::integer`, `movies::integer`). Now fill the `actors` table with a string containing the actor's name and a unique id, so that each actor is in the `actors` table once. Do the same with the `movies` table. Once this is done, fill the `playedin` table so that it contains a row r for every row t in `playedin_text`. In t the integer for actor should reference the name in `actors` that is equal to the actor name in r. Likewise for the movie names.

Solution:

```
-- played_in_normalize.sql
CREATE TABLE actors (
        id serial primary key,
        name text not null);

CREATE TABLE movies (
        id serial primary key,
        name text not null);

CREATE TABLE playedin (
        movie integer references movies (id),
        actor integer references actors (id));

INSERT INTO actors (name)
        SELECT distinct actor_name
        FROM playedin_text;

INSERT INTO movies (name)
        SELECT distinct movie_name
        FROM playedin_text;

-- read from all
INSERT INTO playedin (actor, movie)
        SELECT actors.id,
               movies.id
        FROM actors,
             movies,
             playedin_text p
        WHERE p.actor_name = actors.name
          and p.movie_name = movies.name;

CREATE INDEX ON playedin (actor);
CREATE INDEX ON playedin (movie);
```

4. Run the Bacon Number query again on the `playedin` table for values up to $c <= 5$. Does it run faster than before?

Solution:

```
-- bacon_unique.sql
WITH recursive baconnr (actor_id, nr) as (
    SELECT id, 0
    FROM actors
    WHERE name = 'Bacon, Kevin'
UNION ALL
    SELECT distinct p2.actor,
           movies.nr + 1
    FROM (SELECT distinct p1.movie,
                 baconnr.nr
          FROM baconnr,
               playedin p1
          WHERE baconnr.actor_id = p1.actor
         ) movies,
         playedin p2
    WHERE movies.movie = p2.movie
      and movies.nr < 5
)
SELECT count(distinct actor_id)
FROM baconnr;
```

5. How many actors are there with a Bacon Number <= 6?

   Solution:
   See above and adapt parameter.

6. How many actors are there overall?

   Solution:

   ```
   SELECT count(*)
   FROM actors;
   ```

7. Concluding from this information, are there actors which are separated from Kevin Bacon by a degree of more than 6?

   Solution:
   Yes there are. Even there are people with Bacon Number 7. Just run the above query for $c <= 7$ ($=movies.nr < 7$) and compute the difference in count to $c <= 6$.
   You can also take the overall number of actors and compare it to the count of $c <= 6$.