Homework #6
CS486/586
Prof. Charles Winstead
CRN: 40876

Jiacheng Zhao
Jordan Witte
Resheet Schultz
Due: 02-27-2019

1 Embedded SQL (50 pts)

1) *Using your favorite language, develop a simple application that connects to the class database, executes a query that produces multiple rows and multiple columns, prints out the results of the query (attribute names and values), closes your connection(s) and frees your resources. You will need to execute this program on a system that is part of the university network to be able to access the classdb server. If you follow the cat instructions for remote access, you can also execute your program from your laptop if you have tunneled into the university network using VPN.*

(A) Program Description:

We provide two apps built with Node.js : the first is a simple program that connects to and queries the class database, and publishes the results in html to display in a web browser. The second program is an expanded version that provides the additional features listed for extra credit, and is described in more detail below.

The first program uses a hard-coded query:

```
queryStr = 'SELECT * FROM SPY.AGENT;';
```

Which obtains all rows and columns in the agents table. After the app completes the database query it closes the pg connection to free the database resources.

(B) SQL Library Description:

Node uses the **pg** library to create and use connections to PostgreSQL. Like much of Node, the library functions run asynchronously by default and so we use callbacks to enforce linear execution of the database connection and subsequent query.

Like many embedded sql tools, The pg library initiates a connection using some given credentials for a particular database, and returns a connection object which is then used to make subsequent queries to the database. The pg library allows a few different ways to provide connection information to the db client. This program declares the database info and user credentials as an object literal inside the program. Another approach is to define the required credentials as a set of environment variables, which get read by Node when the connection is initialized.

Since the database query makes a single select statement that returns all records in the table, it makes a single round-trip to the db server.

(C) Code Listing

spy_access_simple.js

```
const { Client } = require('pg');
var express = require('express');
var client = null;
var queryStr = 'SELECT * FROM SPY.AGENT;';
var app = express();

app.get('/', function(req, get_res) {
    client = new Client({
    user: 'w19wdb45',
    host: 'dbclass.cs.pdx.edu',
```

```
    database: 'w19wdb45',
    password: 'q25?kxEkxm' ,
    port: 5432,
    });

  client.connect( function (err) {

      agents_query = {
        text: queryStr,
        rowMode: 'array',
      };

      client.query(agents_query, function (err, res) {
        if (err) { console.error(err) }
      var outStr = "<html><body>" + '<form action = "http://127.0.0.1:8081/" method =
          "POST">' +
      "<TABLE><tr>" + res.fields.map(x => "<th>" + x.name+"</th>").join("") + "</tr>";
        for (var i = 0; i<res.rows.length; i++) {
          outStr += "<tr>" + Object.values(res.rows[i]).map(x => "<td>" +
              x+"</td>\n").join("") + "</tr>";
        }
        outStr += "</TABLE></body></html>";
        get_res.end(outStr);
        client.end();

    })
  })
})

var server = app.listen(8081, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log("listening at %s:%s", host, port);
})
```

(D) The first several rows of the resulting program output (total of 662 rows):

agent_id	first	middle	last	address	city	country	salary	clearance_id
1	Nick	Jim	Black	44 1st Avenue	Athens	USA	50553	5
2	Bill	null	Bundt	34 2nd Avenue	Paris	France	50955	6
3	Mathew	null	Cohen	45 3rd Avenue	New York	USA	55920	5
4	Jim	null	Cowan	1 4th Avenue	Athens	USA	66554	6
5	George	null	Fairley	17 5th Avenue	New York	USA	76396	5
7	Bill	null	Heeman	54 6th Avenue	San Francisco	USA	51564	4
8	Andrew	null	James	3 7th Avenue	Paris	France	53357	3
12	Kristin	null	Delcambre	2-6 8th Avenue	Athens	USA	50503	5
14	John	null	Johnston	8 9th Avenue	Seattle	USA	54479	4
20	George	null	Jones	8 10th Avenue	Paris	France	50171	6
21	Jim	null	Kieburtz	55 11th Avenue	Baghdad	Iraq	54492	6
22	George	null	Launchbury	44 12th Avenue	Hong Kong	China	54453	2
24	Chris	null	Leen	7 13th Avenue	Athens	USA	56719	2
25	Jim	null	Maier	92-94 14th Avenue	Hong Kong	China	50662	5
27	George	null	McNamee	44 15th Avenue	Warsaw	Poland	54453	2
30	Kristin	null	Moody	34 16th Avenue	Milan	Italy	54803	5
31	Lois	null	Oviat	33 17th Avenue	Seattle	USA	54802	3
33	Mathew	null	Pu	18 18th Avenue	Athens	USA	54266	3
35	Jonathan	null	Sheard	24 19th Avenue	Warsaw	Poland	52297	2
36	Nick	null	Steere	15 20th Avenue	San Francisco	USA	56702	5
37	John	null	Walpole	4 21st Avenue	New York	USA	54519	6
39	Nicolas	null	Barnard	17 22nd Avenue	Seattle	USA	55622	2
43	Jim	null	Novick	33 23rd Avenue	Athens	USA	54803	6
45	Pete	null	Consel	42 24th Avenue	Athens	USA	53612	5
48	Bill	null	Bellegarde	27 25th Avenue	Seattle	USA	54512	6
49	Jonathan	null	Hammerstrom	89 26th Avenue	Paris	France	58864	5
50	Helen	null	Hermansky	74 27th Avenue	Athens	USA	57574	5
55	John	null	House	35 28th Avenue	Seattle	USA	54803	5
56	Nicolas	null	Macon	27 29th Avenue	San Francisco	USA	55261	5
58	Jason	null	Pavel	17A 30th Avenue	Baghdad	Iraq	54264	5
59	David	null	Shapiro	15 31st Avenue	Sydney	Austrailia	52302	6
61	Matt	null	Song	5 32nd Avenue	Sydney	Austrailia	82018	2
62	Tim	null	Tolmach	52 33rd Avenue	Athens	USA	55151	3
64	George	null	van Santen	40 34th Avenue	Paris	France	55263	4
65	Jonh	null	Wan	32 35th Avenue	San Francisco	USA	56853	6
69	Nick	null	Yan	15 36th Avenue	Seattle	USA	56702	5

(E) Since the database query makes a single select statement that returns all records in the table, it makes a single round-trip to the db server.

Additional program features

The second version of the app adds the following features

- accepts user input via an HTML form to apply a basic search filter to the returned rows
- uses a SQL stored function to fetch results
- sequentially fetches chunks of data using a Cursor
- adds a little CSS styling to make the generated table more readable.

  The user input form in HTML accepts a string pattern to filter rows in the agent table by first name.

  The program uses a stored function (as opposed to a procedure, because it fetches table contents without making changes to the database), which is defined inside the database as follows:

```
CREATE FUNCTION get_agent_data_by_name_pattern(
  IN PATT VARCHAR(40))
  RETURNS TABLE (AGENT_ID INTEGER, FIRST VARCHAR(40), LAST VARCHAR(40)) AS
    $$
    SELECT (AGENT_ID, FIRST, LAST ) FROM AGENT where FIRST LIKE PATT;
    $$
  LANGUAGE SQL
```

This function fetches a subset of the columns of the agent table, supposing we wanted a way for a user to fetch some of the columns from the agent table, while hiding sensitive information such as agent salary. The Node app calls this function in the database call using the query:

```
queryStr = 'SELECT * FROM get_agent_data_by_name_pattern($1);
```

The program uses the node-postgres api for passing a user-defined input string to the query in a secure way. Furthermore it uses a Cursor object to fetch results in batches of 10 at a time, and accumulates an output string before returning the document to the web browser.

Program code:

spy_access_expanded.js

```
const { Client } = require('pg');
const Cursor = require('pg-cursor')
var express = require('express');
var bodyParser = require('body-parser');

var urlencodedParser = bodyParser.urlencoded({ extended: false })

var client = null;

var queryStr = 'SELECT * FROM get_agent_data_by_name_pattern($1);';

var style_text = "body {padding:100px; } th, td {border: 1px solid black; } table{
    border-collapse: collapse; width: 100%} form {padding:20px;}";
var name_filter_pattern = '%';

var app = express();

app.get('/', function(req, get_res) {

    var num_db_hits = 0;
```

```
  var num_results = 0;

  var outStr = "<html><head><style>" + style_text + " </style></head><body>" + '<form
      action = "http://127.0.0.1:8081/" method = "POST">' +
          'First Name Filter: <input type = "text" name = "name_filter"> <input type =
              "submit" value = "Apply"></form>' +
          "<TABLE>";

  client = new Client({
  user: 'w19wdb45',
  host: 'dbclass.cs.pdx.edu',
  database: 'w19wdb45',
  password: 'q25?kxEkxm' ,
  port: 5432,
  });

  var got_header = false;
client.connect( function (err) {

      if (err) { console.error('cursor read error: ', err); }

      var agents_query = {
        text: queryStr,
        values: [name_filter_pattern],
        rowMode: 'array',
      };

      var cursor = client.query( new Cursor(queryStr,[name_filter_pattern], 'array' ));

      cursor.read(10, function query_callback(err, rows, res) {
          num_db_hits++;
          if (err) { console.error('cursor read error: ', err) }
          if (! got_header) {
              outStr += "<tr>" + cursor._result.fields.map(x => "<th>" +
                  x.name+"</th>").join("") + "</tr>";
              got_header = true;
          }

          if (! res || res.rows.length==0) {
              //cursor returned nothing; hit end of result
              outStr += "</TABLE></body></html>";
              console.log('db read done, num fetches:', num_db_hits);
              console.log('num_results:', num_results);
              get_res.end(outStr);
              client.end() // close db conenction.

          } else {
              //recursively fetch next result from the db cursor
              cursor.read(10, query_callback)
              for (var i = 0; i<res.rows.length; i++) {
```

```
                outStr += "<tr>" + Object.values(res.rows[i]).map(x => "<td>" +
                    x+"</td>\n").join("") + "</tr>";
                num_results++;
                }
            }
        })
    })
})
app.post('/', urlencodedParser, function(req, post_res) {

    if ( req.body.name_filter == '') {
      name_filter_pattern = '%';
    } else {
      name_filter_pattern = req.body.name_filter;
    }

  post_res.redirect('/');

})

var server = app.listen(8081, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log("listening at %s:%s", host, port);
})
```

The capture of this program's output shows the fetch results using the input string pattern:

```
 '%d%'
```

which fetches all agents whose first names contain a lowercase "d".

This program fetches up to 10 results from the cursor with each call to the database. The query in this case returned 55 rows, requiring 6 round-trips to the database.

Here are the first several rows of this query's output: (total of 55 rows):

First Name Filter: Apply

agent_id	first	last
1	Nick	Black
2	Bill	Bundt
3	Mathew	Cohen
4	Jim	Cowan
5	George	Fairley
7	Bill	Heeman
8	Andrew	James
12	Kristin	Delcambre
14	John	Johnston
20	George	Jones
21	Jim	Kieburtz
22	George	Launchbury
24	Chris	Leen
25	Jim	Maier
27	George	McNamee
30	Kristin	Moody
31	Lois	Oviat
33	Mathew	Pu
35	Jonathan	Sheard
36	Nick	Steere
37	John	Walpole
39	Nicolas	Barnard
43	Jim	Novick
45	Pete	Consel
48	Bill	Bellegarde
49	Jonathan	Hammerstrom
50	Helen	Hermansky
55	John	House

2 Disk Access

Consider a hard disk with the following characteristics:

Block size: 8192 bytes

Block transfer time: 0.06ms

Track-to-track seek time: 1ms

Rotational speed: 7200 rpm

Average seek time: 6ms

Bytes per sector: 512 bytes

Sectors per track: 2048

Platters: 4 two-sided platters

2.1) *Estimate the time to read 300 blocks from disk, where the blocks are randomly distributed across the disk. How many megabytes of data are read?*

Read time is the sum of seek time, transfer time, and rotational delay per block, for 300 blocks.

Rotational delay $= \frac{1}{7200\ \text{rpm}} * 60$ seconds $* \frac{1}{2} = 0.0041\bar{6}$ seconds = $4.1\bar{6}$ ms

7200 rpm

120 rotations per second

$\frac{1}{120}$ seconds per rotation

$\frac{1}{2} * \frac{1}{120} = \frac{1}{240}$ seconds per half-rotation

$= 0.0041\bar{6}$ seconds per half-rotation

$= 4.1\bar{6}$ ms rotational delay

Average seek time $= 6$ ms

transfer time = 0.06 ms

Total read time = blocks * (rotational delay + avg seek time + transfer time)

Total read time (ms) = $300 * (4.1\bar{6} + 0.06 + 6) = 3068$ ms ≈ 3.1 seconds

Size of data = $300 * 8192 = 2457600$ bytes $= 2.34375$ MB

2.2) *Estimate the time to read 300 blocks from disk, where the blocks are all on a single cylinder, but randomly distributed within that cylinder.*

Having all of our target blocks on a single cylinder means that average seek time is zero.

Altering the above computation to:

$300 * (0 + 0.06 + 4.1\bar{6}) = 1268$ ms ≈ 1.3 seconds

2.3) *Estimate the time to read a list of 300 blocks from disk, where the blocks are arranged consecutively in a tracks of a single cylinder, and there's no head-to-head delay.*

With consecutive blocks on a track, rotational delay becomes zero, meaning the only contributor to read time is transfer time.

Total read time $= 300 * 0.06 = 18$ ms

2.4) *Suppose we dont care about the order in which the 300 blocks are read. Suggest how to speed up the times for Questions 2.1 and 2.2 above.*

If order of the blocks that we need to read from disk doesnt matter, then we might reduce read time through the following strategies:

1. Read all of the target blocks that are located on a given track, before moving to the next track. This might reduce the number of occurrences of average seek time in the computation of total read time.
2. Read from the relevant tracks in order of their arrangement on the disk. This might reduce the value of average seek time in the computation, bringing it slightly closer to track-to-track seek time.

Considering the setup in Q2.1, the reduction in time from strategy 1 depends on the number of times that more than one block of interest occurs on a given track. Assuming a large number of tracks on the disk, this number of occurrences is statistically near 0. Though strategy 2 might reduce track seek time to a value closer to the track-to-track seek time and might provide some savings in read time.

Considering the setup in Q2.2, average seek time is already zero. But strategy 1 might reduce total read time because it could mean that we replace replace the average rotational delay for all 300 reads with time required with a single, full rotation.

These changes in read strategy assume that deciding the new order of block reads is actually possible without adding new overhead, which may be a bad assumption.

2.5) *If your system can transfer data from Intel Optane persistent memory at 5 GiB/s, estimate the time it would take to read this amount of data from it. How much faster is it to access this data from persistent memory than from disk, both for the best disk io (question 2.3) and the worst disk io (question 2.1)?*

Read time for 2457600 bytes at 5 GiB/s is $= 2457600/(5 * 1024 * 1024 * 1024) \approx 0.458$ ms.

This is $3068/0.458 =$ **6698 times faster** than the read time from Q2.1.

and $18/0.458 =$ **39.3 times faster** than the read time from Q2.3.

Database Indexes

Exercise 3.1

Short Answer/True/False Questions. Answer the following questions, and provide a brief explanation why

A) *True/False: Unclustered B+ Tree Indices themselves are stored in order of their search key value.*

True. The leaves of the B+ Tree contain index values which are ordered to allow log-speed lookup time.

B) *True/False: Indexes will consume disk space to store additional data, and can slow down inserts and updates to the table.*

True. The index itself contains a redundant set of the index values which consumes a small amount of additional disk space. Inserts and Update operations on a table using an index must also update the index, which consumes time especially when, for example, needing to update and rebalance a B+ Tree index.

C) *True/False: Columns with many duplicate values are good candidates for indexes.*

False. The index lookup returns all the rows that match a given index value; if many rows are returned then additional search (and possibly linear search, if lacking another index) is required to located an exact record.

D) *True/False: The order of columns in an index affects the usefulness of that index.*

True. The order of columns in the index determines the order of attributes that the index filters on. If the first column has a low rate of duplicate values then that index operates faster because the application of subsequent index values needs to run on a smaller subset of the data.

E) *True/False: You can create many clustered indexes for a single table to speed up different types of queries.*

False. A clustered index has index values in the same order as the records themselves in the table, so any two clustered indexes on a table will have identical lookup behavior.

F) *True/False: Indexing a foreign key is often a good idea because it can be used to improve join performance.*

True. Since an inner join needs to locate records by matching up corresponding values in a column – typically a foreign key – , having faster lookup on that key improves the speed of the join.

Exercise 3.2

Using the spy db, give 2 examples of a query and an index that would speed up that query.

The query

```
'SELECT * FROM SPY.AGENT WHERE ADDRESS = '44 1st Avenue';
```

(which returns a single record) would likely run slightly faster if the agent table had an index on the address column. This seems supported by the fact that a similar single-record lookup on spy.agent using agent_id (which does have an associated index) runs slightly faster than the above. An index on address would likely be effective because most (516 /of 662) of the address values are unique.

As a second example, the query

```
'SELECT * FROM SPY.MISSION WHERE NAME = 'Third Age';
```

Would be sped up by an index on the name column.

Exercise 3.3

What is a covering index? Give an example of one and a query that uses it.

A covering index for a given query is one that contains all columns that are specified in the query's search conditions (the WHERE clause). For example, for the query,

```
'SELECT * FROM SPY.AGENT where agent_id=4';
```

a covering index would be just an index on the agent_id column.