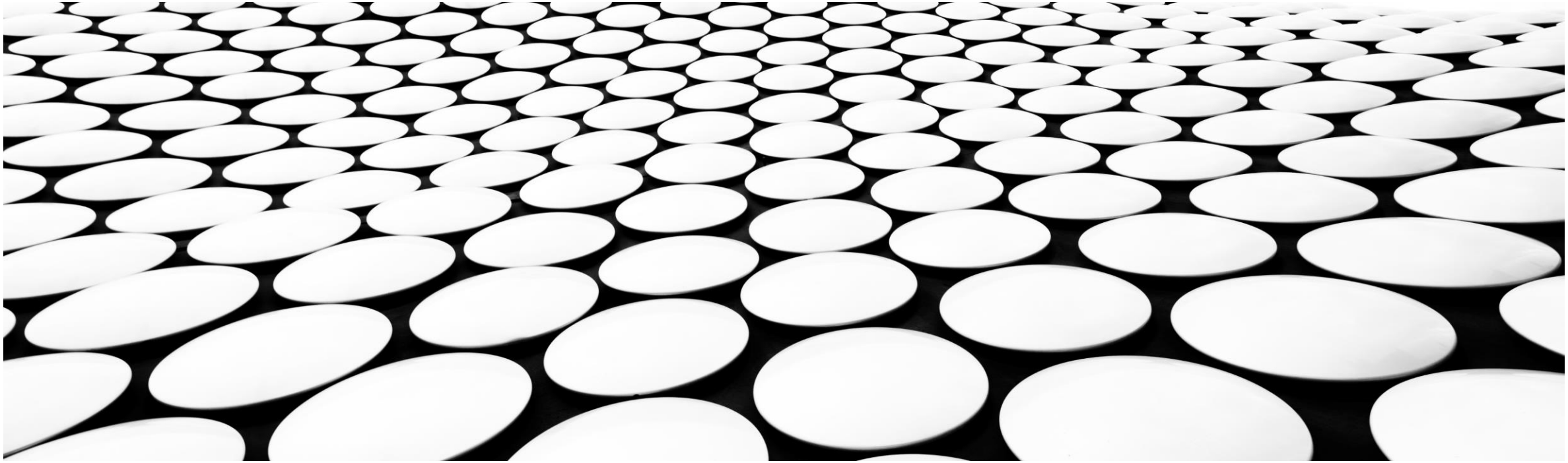
The Postgres relational database management system and Application Programming Interface

COEN-317: Distributed Systems
Robert Bruce
Department of Computer Science and Engineering
Santa Clara University



Introduction to Postgres

Postgres is an open source, ANSI SQL compliant relational database management system.

Additional information about Postgres is at the web address http://www.postgresql.org/about/

Postgres: command line admin interface

This command creates a new database called coen317:

```
create database coen317
```

Postgres tables: structure management

Postgres provides standard SQL commands for tables:

create table: creates a new table.

alter table: updates the structure of a table.

drop table: deletes the structure of a table (any data in the table is also deleted).

Postgres: columns types we will use

Postgres contains a variety of fields (columns). We'll be using the following types:

bigserial: auto-incrementing eight-byte integer.

integer: integer data type.

varchar: variable length character.

timestamp: date and time.

Postgres tables: data management

Postgres provides standard SQL commands for managing the data stored in tables. These commands do not change the structure of the tables.

select from *table*: retrieves a row of data from *table*.

insert into *table*: adds a new row of data to *table*.

update *table*: updates data in *table*.

delete from *table*: deletes data from *table*.

Primary keys and foreign keys

STUDENTS	
student_id	bigserial
first_name	varchar(100)
last_name	varchar(100)

GRADES	
student_id	bigint
course_id	bigint
grade	varchar(2)

COURSES	
course_id	bigserial
course_name	varchar(255)
course_description	varchar(1024)

Primary keys and foreign keys

STUDENTS	
student_id	bigserial
first_name	varchar(100)
last_name	varchar(100)

GRADES	
student_id	bigint
course_id	bigint
grade	varchar(2)

COURSES	
course_id	bigserial
course_name	varchar(255)
course_description	varchar(1024)

Primary keys and foreign keys

STUDENTS	
student_id	bigserial
first_name	varchar(100)
last_name	varchar(100)

GRADES	
student_id	bigint
course_id	bigint
grade	varchar(2)

Primary key

COURSES	
course_id	bigserial
course_name	varchar(255)
course_description	varchar(1024)

Primary keys and foreign keys

STUDENTS	
student_id	bigserial
first_name	varchar(100)
last_name	varchar(100)

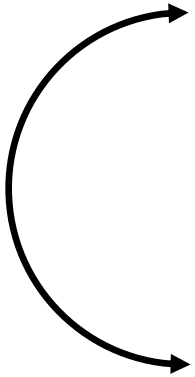
GRADES	
student_id	bigint
course_id	bigint
grade	varchar(2)

COURSES	
course_id	bigserial
course_name	varchar(255)
course_description	varchar(1024)

Primary key 

Primary keys and foreign keys

The yellow highlighted columns (fields) form a relation. This relation ties the **STUDENTS** and **GRADES** tables together.



STUDENTS	
student_id	bigserial
first_name	varchar(100)
last_name	varchar(100)

GRADES	
student_id	bigint
course_id	bigint
grade	varchar(2)

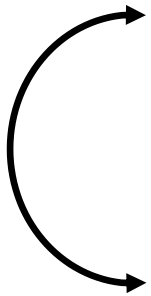
COURSES	
course_id	bigserial
course_name	varchar(255)
course_description	varchar(1024)

Primary keys and foreign keys

STUDENTS	
student_id	bigserial
first_name	varchar(100)
last_name	varchar(100)

GRADES	
student_id	bigint
course_id	bigint
grade	varchar(2)

COURSES	
course_id	bigserial
course_name	varchar(255)
course_description	varchar(1024)



The yellow highlighted columns (fields) form a relation. This relation ties the **GRADES** and **COURSES** tables together.

SQL WHERE clause

The WHERE clause is a powerful SQL command for defining a condition. This condition must be satisfied (evaluated to Boolean TRUE) before the appropriate action is taken. The WHERE clause can be applied to SELECT, INSERT, UPDATE, and DELETE commands. Examples:

```
SELECT student_id FROM students WHERE last_name = "BRUCE";
```

```
SELECT first_name, last_name FROM students, grades WHERE student.student_id = grades.student_id
AND grades.grade = "A";
```

```
SELECT course_id, course_description FROM courses;
```

```
UPDATE courses SET course_description = "Distributed Systems" WHERE course_id = 317;
```

SQL ORDER clause

The ORDER clause follows a WHERE clause. It only applies to SELECT statements. The ORDER clause provides a means of sorting the output in ASCending or DESCending order based on the columns SELECTed. Examples:

```
SELECT student_id FROM students WHERE last_name = "BRUCE" ORDER BY student_id ASC;
```

```
SELECT first_name, last_name FROM students, grades WHERE student.student_id = grades.student_id
AND grades.grade = "A" ORDER BY last_name DESC;
```

```
SELECT course_id, course_description FROM courses ORDER BY course_id ASC;
```

SQL DISTINCT clause

The DISTINCT clause is used in SELECT statements. It removes duplicate rows of data. Each row returned is thus unique or DISTINCT from each other. Example:

```
SELECT DISTINCT last_name FROM students ORDER BY last_name;
```

Retrieving auto incremented bigserial column from last INSERT

The BIGSERIAL datatype is an unsigned integer datatype that automatically increments its column value atomically when a new row of data is INSERTed. This is a very useful operation when the bigserial column needs to be unique and be the primary key. *Primary keys, by definition, must be unique!*

Since this operation is atomic, no race conditions can occur. This is very important when developing applications in highly scaled environments with many concurrent users!

Retrieving the last INSERTed serial number for an INSERT operation is easy. Here's how to do so:

```
INSERT INTO students (first_name, last_name) VALUES ("ROB", "BRUCE") RETURNING student_id;
```

Postgres C application programming interface

Step 1: Connect to the Postgres database management system with the **Pqconnectdb()** function. If authentication is successful and a connection is made to the Postgres database, **Pqconnectdb()** function will return a handle to the connection. In the example below, that handle is **db_connection**.

```
PGconn   *db_connection;
PGresult *db_result;

db_connection = PQconnectdb("host = 'localhost' dbname = 'coen317' user = 'postgres' password =
'PASSWORD_TO_ACCESS_THIS_DATABASE'");
if (PQstatus(db_connection) != CONNECTION_OK)
{
  printf ("Connection to database failed: %s", PQerrorMessage(db_connection));
  PQfinish (db_connection);
  exit (EXIT_FAILURE);
}
```

Postgres C application programming interface (API)

Step 2: Now that a Postgres database handle has been established, we can issue standard SQL commands through the handle, **db_connection**. Let's issue a standard SELECT command:

```
SELECT students.student_id, students.first_name, students.last_name FROM students ORDER BY students.student_id ASC
```

Postgres C application programming interface (API)

```
strncpy (&db_statement[0], "SELECT students.student_id, students.first_name, students.last_name FROM students
ORDER BY students.student_id ASC", MAX_DB_STATEMENT_BUFFER_LENGTH);

db_result = PQexec (db_connection, &db_statement[0]);
if (PQresultStatus(db_result) == PGRES_TUPLES_OK)
{
  num_rows = Pqntuples(db_result);
  if (num_rows == 0)
  {
    printf ("database is empty");
  }
  else
  {
    for (row = 0; row < num_rows; row++)
    {
      printf ("%s %s %s\n", PQgetvalue (db_result, row, 0), PQgetvalue (db_result, row, 1), PQgetvalue
(db_result, row, 2));
    }
  }
}
```

Postgres C application programming interface (API)

```
strncpy (&db_statement[0], "SELECT students.student_id, students.first_name, students.last_name FROM students
ORDER BY students.student_id ASC", MAX_DB_STATEMENT_BUFFER_LENGTH);

db_result = PQexec (db_connection, &db_statement[0]);
if (PQresultStatus(db_result) == PGRES_TUPLES_OK)
{
  num_rows = Pqntuples(db_result);
  if (num_rows == 0)
  {
    printf ("database is empty");
  }
  else
  {
    for (row = 0; row < num_rows; row++)
    {
      printf ("%s %s %s\n", PQgetvalue (db_result, row, 0), PQgetvalue (db_result, row, 1), PQgetvalue
(db_result, row, 2));
    }
  }
}
```

PQexec() executes the SQL command we entered.

Postgres C application programming interface (API)

```
strncpy (&db_statement[0], "SELECT students.student_id, students.first_name, students.last_name FROM students
ORDER BY students.student_id ASC", MAX_DB_STATEMENT_BUFFER_LENGTH);

db_result = PQexec (db_connection, &db_statement[0]);
if (PQresultStatus(db_result) == PGRES_TUPLES_OK)
{
  num_rows = Pqntuples(db_result);
  if (num_rows == 0)
  {
    print
  }
  else
  {
    for (row = 0; row < num_rows; row++)
    {
      printf ("%s %s %s\n", PQgetvalue (db_result, row, 0), PQgetvalue (db_result, row, 1), PQgetvalue
(db_result, row, 2));
    }
  }
}
```

PQresultStatus() informs our program if the SQL command executed successfully (i.e. there were no syntax errors in our query).

Postgres C application programming interface (API)

```
strncpy (&db_statement[0], "SELECT students.student_id, students.first_name, students.last_name FROM students
ORDER BY students.student_id ASC", MAX_DB_STATEMENT_BUFFER_LENGTH);

db_result = PQexec (db_connection, &db_statement[0]);
if (PQresultStatus(db_result) == PGRES_TUPLES_OK)
{
  num_rows = Pqntuples(db_result);
  if (num_rows == 0)
  {
    printf ("database is em
  }
  else
  {
    for (row = 0; row < num_rows; row++)
    {
      printf ("%s %s %s\n", PQgetvalue (db_result, row, 0), PQgetvalue (db_result, row, 1), PQgetvalue
(db_result, row, 2));
    }
  }
}
```

Pqntuples() returns the number of results (rows) from the SQL command.

Postgres C application programming interface (API)

```
strncpy (&db_statement[0], "SELECT students.student_id, students.first_name, students.last_name FROM students
ORDER BY students.student_id ASC", MAX_DB_STATEMENT_BUFFER_LENGTH);

db_result = PQexec (db_connection, &db_statement[0]);
if (PQresultStatus(db_result) == PGRES_TUPLES_OK)
{
  num_rows = Pqntuples(db_result);
  if (num_rows == 0)
  {
    printf ("database is e
  }
  else
  {
    for (row = 0; row < num_rows; row++)
    {
      printf ("%s %s %s\n", PQgetvalue (db_result, row, 0), PQgetvalue (db_result, row, 1), PQgetvalue
(db_result, row, 2));
    }
  }
}
```

PQgetvalue() returns a row and column of data from the results of the SQL query.

Postgres C application programming interface

Step 3: Before a program exits, the Postgres database handle must be released and the connection to the database closed. This is accomplished by issuing the following commands:

```
db_result = PQexec (db_connection, "CLOSE myportal");
PQclear (db_result);
PQfinish (db_connection);
```

Programming project hint

Use Postgres to create a unique video identification number (video_id) to associate with each uploaded video for processing.

Example:

I upload an mpeg4 video for my project. It is a ten second video encoded at thirty frames per second. Postgres creates a video_id of 12. I would therefore save this video on my server as:
12.mp4

When I convert the video to 300 still images (for computer vision processing), the filenames would be:
12.1.png, 12.2.png, 12.3.png, … 12.298.png, 12.299.png, 12.300.png

When I draw a face mesh on each of these images, the filenames would be:
mesh-12.1.png, mesh-12.2.png, mesh-12.3.png, … mesh-12.298.png, mesh-12.299.png, mesh-12.300.png

Lastly, combining these still images with meshes into an mpeg movie, the resultant video would be:
mesh-12.mp4

For further reading

Postgres source code and documentation:
http://www.postgresql.org/

A handy list of Postgres commands (if you are using the command line interface):
https://www.postgresqltutorial.com/postgresql-cheat-sheet/

How to install and configure Postgres on Ubuntu 22.04:
https://www.digitalocean.com/community/tutorials/how-to-install-postgresql-on-ubuntu-22-04-quickstart

How to connect Postgres from Python:
https://www.postgresqltutorial.com/postgresql-python/

My example C program on Camino (under files folder : example programs : postgres example):
db_menu.c