Backend Development– W2S3 Postgresql

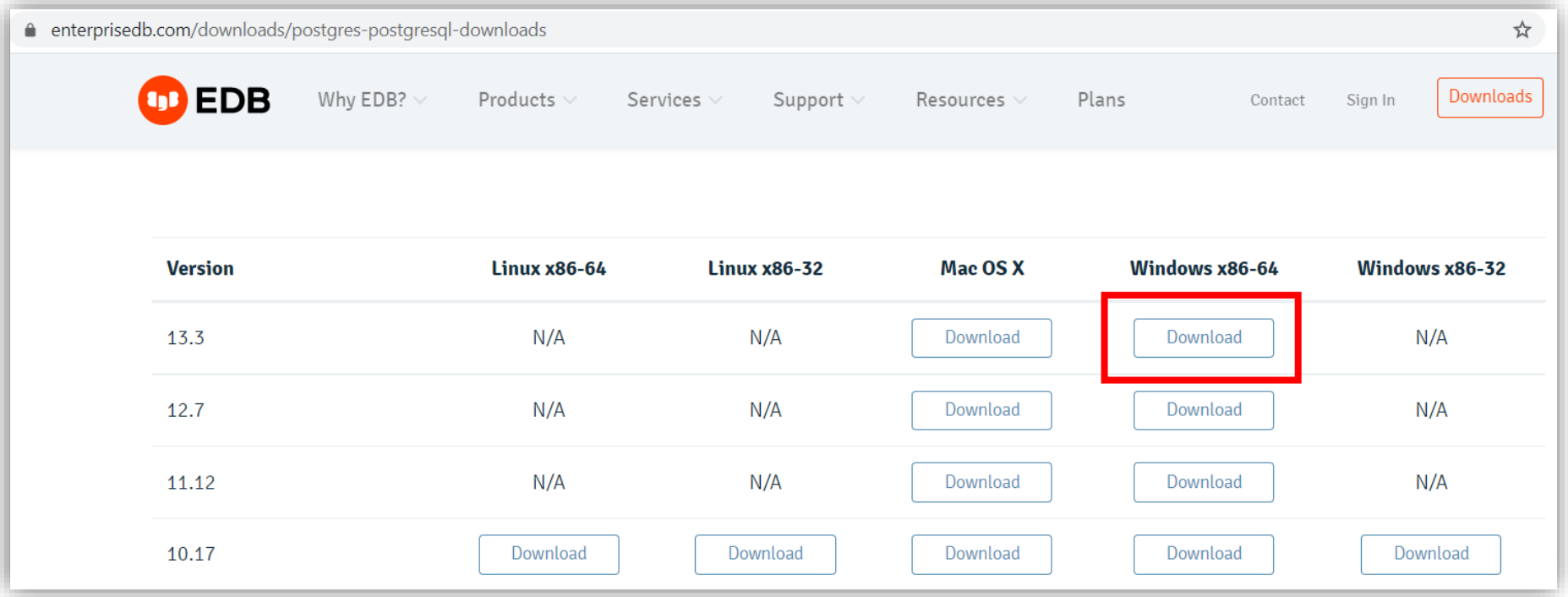
Cyrille Jegourel – Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Outline (Day 2, Session 3)

- Installation of Postgresql
- Using the SQL shell and postgresql
- Connect to and query a DB using Node.js

Postgresql installation



- Go to https://www.postgresql.org/download/
- Choose your OS family (e.g. Windows), download the installer of the last version, then follow the instructions (see next slide).

Postgresql installation

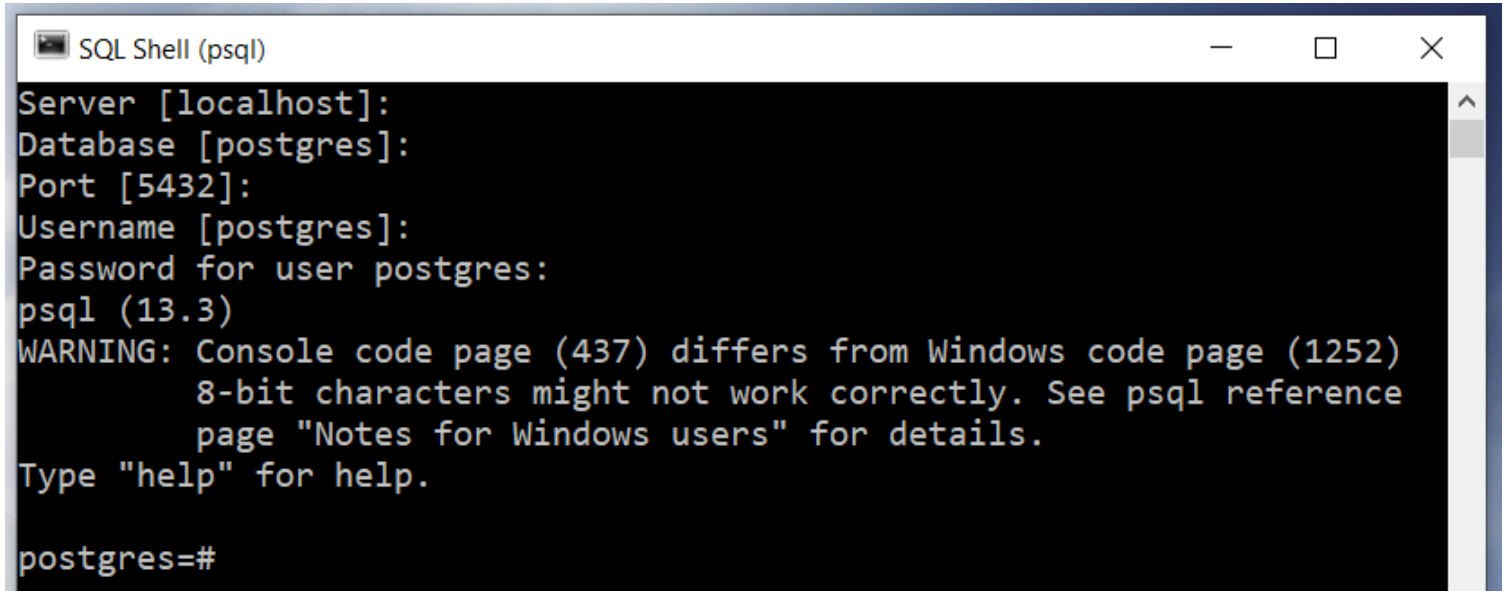
- Open the setup file.
- At some point, you are asked to provide a password for the database superuser. **Do not forget it.**
- Next, do not change the default values (port 5432 and path).
- At the end of the process, you can uncheck the “Stack Builder” checkbox.
- That’s it: pgAdmin and SQL shell (psql) are installed.

Connecting to a DB server

- Now that we have a DBMS,
  we need to connect to it.
- Using a GUI
- **Using a terminal**
- Using an application

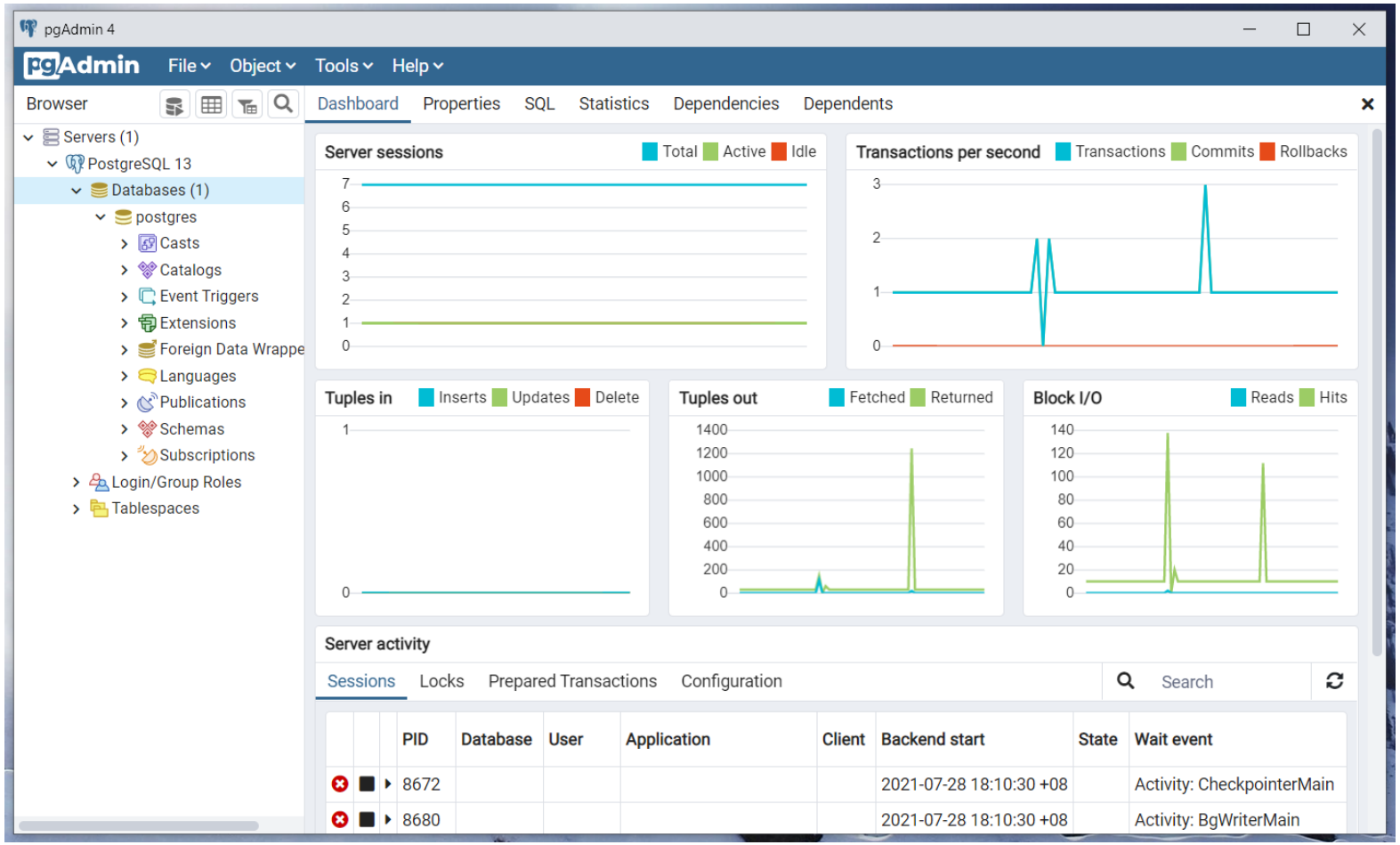
Setup Shell SQL

- Open psql and press enter.
- By default, you are connected to your local server. The default database is called postgres. Press enter.
- The default port is 5432. Press enter.
- The default username is postgres. Press enter.
- Enter your superuser password.
- Finally, you are in the psql mode (in fact, you are connected to the default ‘postgres’ database). ☺



Setup pgAdmin

- Open pgAdmin and enter the superuser password.
- Click on server on the left, then PostgreSQL and pass again the password for the user to connect the server PostgreSQL.
- Note that there is already the default database postgres on the left.



SQL shell (psql)

List of database

```
postgres=# \l
                                                List of databases
   Name    |  Owner   | Encoding |          Collate           |           Ctype            |   Access privileges
-----------+----------+----------+----------------------------+----------------------------+-----------------------
 postgres  | postgres | UTF8     | English_United States.1252 | English_United States.1252 |
 template0 | postgres | UTF8     | English_United States.1252 | English_United States.1252 | =c/postgres          +
           |          |          |                            |                            | postgres=CTc/postgres
 template1 | postgres | UTF8     | English_United States.1252 | English_United States.1252 | =c/postgres          +
           |          |          |                            |                            | postgres=CTc/postgres
(3 rows)
```

- Once you are in psql mode, using command **\l**, you can see the list of databases stored on your computer. Here, we have 3: postgres, template0 and template1.

Create database

- To create a database, just write CREATE DATABASE followed by a name and a semi-colon.
- Unlike JS scripts, the semi-colon is not optional as it indicates the end of the statement.
- You can write your commands in both in uppercase or lowercase characters.

```
postgres=# CREATE DATABASE test0;
CREATE DATABASE
postgres=# \l

   Name     |  Owner   | Encoding |
-----------+----------+----------+-
 postgres   | postgres | UTF8     |
 template0  | postgres | UTF8     |
            |          |          |
 template1  | postgres | UTF8     |
            |          |          |
 test0      | postgres | UTF8     |
(4 rows)
```

Connect to a database

```
Connection
  \c[onnect] {[DBNAME|- USER|- HOST|- PORT|-] | conninfo}
                          connect to new database (currently "postgres")
  \conninfo               display information about current connection
  \encoding [ENCODING]    show or set client encoding
  \password [USERNAME]    securely change the password for a user
```

```
postgres=# \c test0 postgres localhost 5432
You are now connected to database "test0" as user "postgres".
test0=#
```

- To connect to a database from the SQL shell, just execute the command \c followed by the database name, the user name (postgres by default), the host (usually an IP address but here localhost), and a port (5432 by default).
- Note: connecting to a non-existing database triggers a fatal error.
- Note 2: Anything after \c test0 was here optional.

Delete a database

- To delete a database, execute command “DROP DATABASE name;”
- However, you cannot delete a database currently open. You first need to connect to another database using command
  - “\c *another_database_name*”

```
test0=# DROP DATABASE test0;
ERROR:  cannot drop the currently open database
test0=#
test0=# \c postgres
You are now connected to database "postgres" as user "postgres".
postgres=# DROP DATABASE test0;
DROP DATABASE
```

```
   Name     |  Owner   | Encoding |
-----------+----------+----------+-
 postgres  | postgres | UTF8     |
 template0 | postgres | UTF8     |
           |          |          |
 template1 | postgres | UTF8     |
           |          |          |
(3 rows)
```

Create and drop a table

```
test0=# create table Payroll(
test0(# UserId integer,
test0(# Name varchar(100),
test0(# Job varchar(100),
test0(# Salary integer
test0(# );
CREATE TABLE
test0=#
```

- To create a table Payroll, you must write CREATE TABLE Payroll (attribute_name + type + constraints_if_there_are)
- Do not forget the semi-colon at the end to execute.
- \d shows you that test0 contains a table called Payroll.

```
test0=# \d
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
 public | payroll | table | postgres
(1 row)
```

- \d Payroll shows the columns of the table and their types.
- To drop a table (e.g. Payroll), execute **DROP TABLE Payroll;**

```
test0=# \d Payroll
                    Table "public.payroll"
 Column |          Type          | Collation | Nullable | Default
--------+------------------------+-----------+----------+---------
 userid | integer                |           |          |
 name   | character varying(100) |           |          |
 job    | character varying(100) |           |          |
 salary | integer                |           |          |
```

Create a table with constraints

```
test0=# CREATE TABLE Payroll(
test0(# UserId BIGSERIAL NOT NULL PRIMARY KEY,
test0(# Name varchar(100) NOT NULL,
test0(# Job varchar(100) NOT NULL,
test0(# Salary integer NOT NULL);
CREATE TABLE
test0=# \d Payroll
                                    Table "public.payroll"
 Column |          Type          | Collation | Nullable |                 Default
--------+------------------------+-----------+----------+-----------------------------------------
 userid | bigint                 |           | not null | nextval('payroll_userid_seq'::regclass)
 name   | character varying(100) |           | not null |
 job    | character varying(100) |           | not null |
 salary | integer                |           | not null |
Indexes:
    "payroll_pkey" PRIMARY KEY, btree (userid)
```

- Add constraints by specifying PRIMARY KEY.
- NOT NULL avoids creating a row with a missing value
- BIGSERIAL is an auto-incremented bigint column taking 8 bytes. Behind the scenes, PostgreSQL will use a sequence generator to generate these column values upon inserting a new ROW.

Insert and visualize data

- To insert data, execute:
- INSERT INTO table_name(attribute_names) VALUES(respective_values);

```
test0=# INSERT INTO payroll(name, job, salary)
test0-# VALUES ('Anh', 'Prof', 10000);
INSERT 0 1
test0=# INSERT INTO payroll(name, job, salary)
test0-# VALUES ('Cyrille', 'Lecturer', 10000);
INSERT 0 1
```

- You can also insert several values at once. Each row must be enclosed within brackets and separated from each other by a comma.

```
test0=# INSERT INTO payroll(name, job, salary)
test0-# VALUES ('Dewi', 'TA', 9000), ('Stan', 'TA', '9000');
INSERT 0 2
```

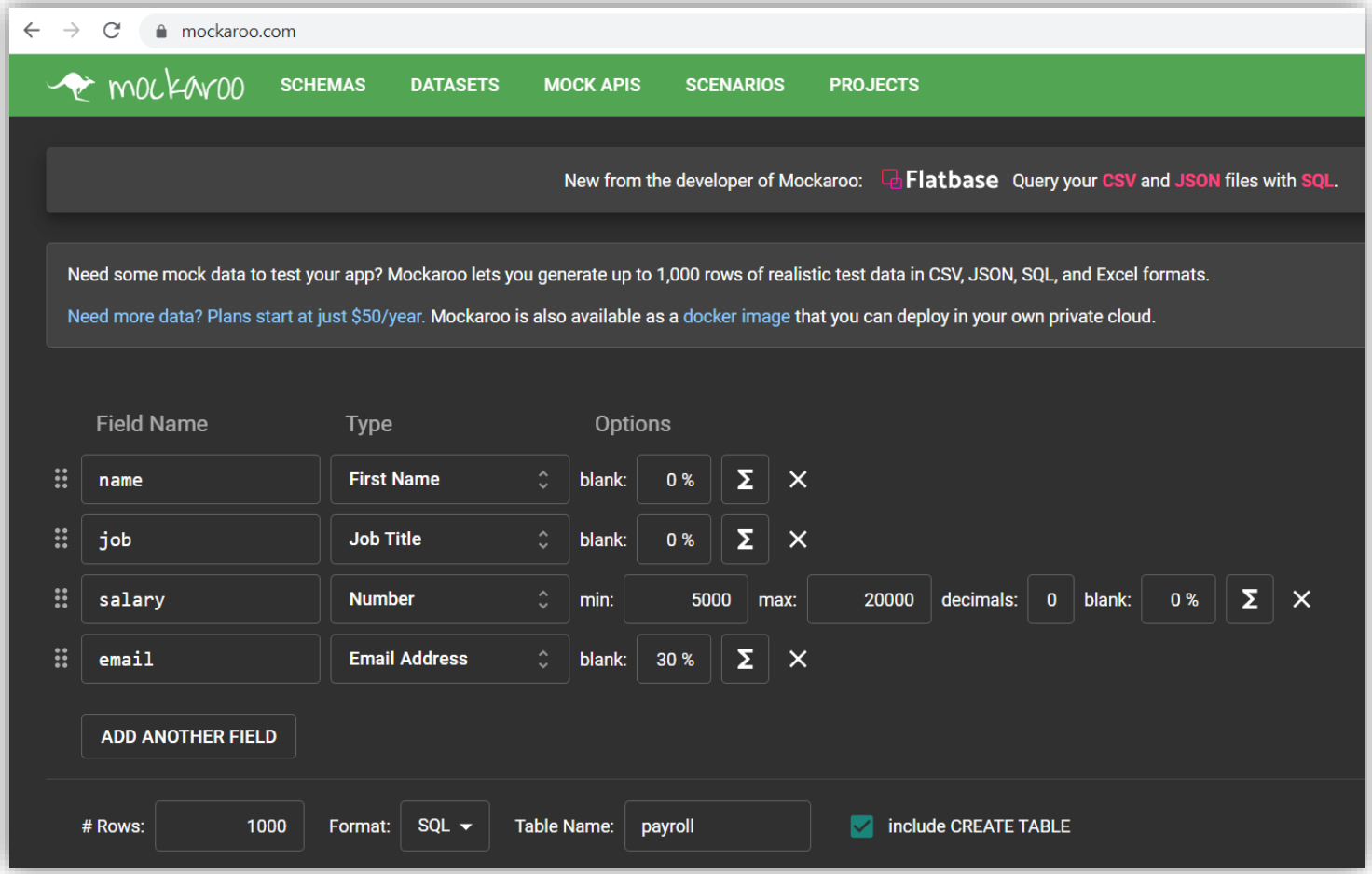
- To visualize the data, execute:
- SELECT * FROM payroll

```
test0=# SELECT * FROM payroll
test0-# ;
 userid |  name   |   job    | salary
--------+---------+----------+--------
      1 | Anh     | Prof     |  10000
      2 | Cyrille | Lecturer |  10000
      3 | Dewi    | TA       |   9000
      4 | Stan    | TA       |   9000
(4 rows)
```

- * is a wildcard for “all”

Create Mock data

- If you need to create mock data, go on www.mockaroo.com
- Create your fields and choose a type among the many existing ones. For numbers, you can choose min, max, number of decimals.
- You can also set up a percentage of blank values. Here, a new row has 30% chances of email emptiness.



- Then, select a number of rows, set up the format to SQL, name the table, and check the checkbox CREATE TABLE. Finally, download the data.

Create Mock data

C: > Users > User > Downloads > payroll.sql

```sql
create table payroll (
    name VARCHAR(50),
    job VARCHAR(50),
    salary INT,
    email VARCHAR(50)
);
insert into payroll (name, job, salary, email) values ('Sayers', 'Biostatistician II', 11804, 'soverall0@google.it')
insert into payroll (name, job, salary, email) values ('Martie', 'Technical Writer', 19142, 'mmcaulay1@ask.com');
insert into payroll (name, job, salary, email) values ('Codie', 'Recruiting Manager', 9876, 'csoitoux2@who.int');
```

- You can visualize the file in VS Code and update it accordingly (e.g., by adding NOT NULL constraints or by increasing the number of characters in the email VARCHAR).
- To import it into psql, first drop your current payroll table. Then, execute \i filepath.
- Note that on Windows, you might need to enclose the path by single quotes and to replace the backslashes by /.

```
test0=# \i 'C:/Users/User/Downloads/payroll.sql'
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

Clear the shell

- To clear the shell, **CTRL + L** should work on Linux and Mac (?).
- On Windows 10, you have to execute **\! cls**

Filter data

```
test0=# SELECT name, salary FROM payroll
test0-# WHERE (salary > 10000) AND (name = 'Alix' OR name = 'Garv');
 name | salary
------+--------
 Garv |  19975
 Alix |  10636
(2 rows)
```

- To filter data, execute the command SELECT followed by a list of attributes separated by comma (or * if you want everything) FROM the current table WHERE some constraints must be verified.
- These constraints evaluate to Booleans and support arithmetic expressions, Boolean operators (AND, OR, NOT), comparison operators (=, <>, >=, etc…).

Output control: sort data

- You can display sorted data using SELECT with the following option:
- “ORDER BY attribute_name DESC” (for sorting in descending order, ASC in ascending order)
- LIMIT number; is an option to limit the numbers of rows to display in the shell.

- If you order by an attribute of string type instead of numbers, rows will be ordered alphabetically.

```
test0=# SELECT userid, name, salary FROM payroll
test0-# ORDER BY salary DESC
test0-# LIMIT 5;
 userid |   name   | salary
--------+----------+--------
    479 | Warden   |  19995
    931 | Lorinda  |  19981
    613 | Veronika |  19975
      9 | Garv     |  19975
    848 | Seka     |  19966
(5 rows)
```

```
test0=# SELECT userid, name, salary FROM payroll
test0-# ORDER BY name ASC
test0-# LIMIT 5;
 userid |  name    | salary
--------+---------+--------
    125 | Abagail |  18685
    377 | Abbey   |  12943
    323 | Abbot   |  11632
    150 | Ad      |  13803
    339 | Adamo   |  16341
(5 rows)
```

Distinct values and offset

- There are 2 ‘Alix’ in the table.
- Assume that we only want to display each name once and only once (e.g., to count the diversity of names in our population).
- Add the keyword DISTINCT after SELECT.

```
test0=# SELECT name FROM payroll
test0-# WHERE name = 'Alix';
 name
------
 Alix
 Alix
(2 rows)
```

```
test0=# SELECT DISTINCT name FROM payroll
test0-# WHERE name = 'Alix';
 name
------
 Alix
(1 row)
```

- In order to display some rows from a particular offset, e.g. 5, add the option OFFSET 5.

```
test0=# SELECT userid, name FROM payroll
test0-# OFFSET 5
test0-# LIMIT 5;
 userid |  name
--------+---------
      6 | Lotty
      7 | Maurine
      8 | Sara
      9 | Garv
     10 | Imogen
(5 rows)
```

Aggregation

- GROUP BY is really useful to count or to perform some operations over a group of rows.
- In the 1st example, we are counting the number of individuals in the table in each job category.

```
test0=# SELECT job, COUNT(*) FROM payroll GROUP BY job;
                 job                  | count
--------------------------------------+-------
 Recruiting Manager                   |     9
 Budget/Accounting Analyst II         |     1
 Financial Advisor                    |     8
 Social Worker                        |     8
 Human Resources Manager              |    12
 Statistician III                     |     2
```

- Note that in the 2nd example, we are calculating the average salary per job and we name this column using **as** keyword followed by the new name.

```
test0=# SELECT job, AVG(salary) as AvgPerJob
test0-# FROM payroll GROUP BY job;
                 job                  |       avgperjob
--------------------------------------+-----------------------
 Recruiting Manager                   |     11142.444444444444
 Budget/Accounting Analyst II         | 17216.0000000000000000
 Financial Advisor                    |     12651.625000000000
 Social Worker                        |     13572.875000000000
 Human Resources Manager              |     12124.666666666667
 Statistician III                     |  8393.0000000000000000
```

Filtered Aggregation

- In the 2nd example, we are calculating the average salary per job and we name this column using **as** keyword followed by the new name.
- In the 3rd example, we add another filtering after we perform the aggregation, using the keyword HAVING followed by a Boolean expression.
- The Boolean expression often has the form of an arithmetic expression and might make use of basic operations (including modulo, exponents, bitwise, etc.) and mathematical functions (avg, min, max, sum, count, round, etc…).

```
test0=# SELECT job, AVG(salary) as AvgPerJob
test0-# FROM payroll GROUP BY job;
                  job                   |        avgperjob
----------------------------------------+------------------------
 Recruiting Manager                     |     11142.444444444444
 Budget/Accounting Analyst II           | 17216.0000000000000000
 Financial Advisor                      |     12651.625000000000
 Social Worker                          |     13572.875000000000
 Human Resources Manager                |     12124.666666666667
 Statistician III                       |  8393.0000000000000000
```

```
test0=# SELECT job, AVG(salary) as AvgPerJob
test0-# FROM payroll GROUP BY job
test0-# HAVING AVG(salary) > 13000;
                job                     |        avgperjob
----------------------------------------+------------------------
 Budget/Accounting Analyst II           | 17216.0000000000000000
 Social Worker                          |     13572.875000000000
 Quality Engineer                       |     13898.800000000000
 Database Administrator II              | 14999.5000000000000000
 Research Assistant II                  | 17975.0000000000000000
```

Primary keys (drop)

- Recall: primary keys uniquely defines a record in a table.
- Drop a primary key:
  - ALTER TABLE payroll DROP CONSTRAINT *table_name*_pkey;
- Now, you can duplicate rows or add a row with same userid in our example.

```
test0=# INSERT INTO payroll(userid, name, job, salary)
test0-# VALUES(1, 'Willy', 'Lecturer', 8000);
INSERT 0 1
test0=# SELECT * FROM payroll
test0-# WHERE userid = 1;
 userid |  name  |        job         | salary |        email
--------+--------+--------------------+--------+---------------------
      1 | Sayers | Biostatistician II |  11804 | soverall0@google.it
      1 | Willy  | Lecturer           |   8000 |
(2 rows)
```

```
test0=# \d payroll
                                    Table "public.payroll"
 Column |          Type          | Collation | Nullable |
--------+------------------------+-----------+----------+----
 userid | bigint                 |           | not null | next
 name   | character varying(50)  |           | not null |
 job    | character varying(50)  |           | not null |
 salary | integer                |           | not null |
 email  | character varying(150) |           |          |
Indexes:
    "payroll_pkey" PRIMARY KEY, btree (userid)


test0=# ALTER TABLE payroll DROP CONSTRAINT payroll_pkey;
ALTER TABLE
test0=# \d payroll
                                    Table "public.payroll"
 Column |          Type          | Collation | Nullable |
--------+------------------------+-----------+----------+----
 userid | bigint                 |           | not null | next
 name   | character varying(50)  |           | not null |
 job    | character varying(50)  |           | not null |
 salary | integer                |           | not null |
 email  | character varying(150) |           |          |


test0=#
```

Delete a record

- Add a primary key will fail if there are duplicates or rows with identical values for a potential primary key.
- You might need to delete some records.

```
test0=# DELETE FROM payroll WHERE userid = 1 AND name = 'Willy';
DELETE 1
test0=# SELECT * FROM payroll
test0-# WHERE userid = 1;
 userid |  name  |        job         | salary |        email
--------+--------+--------------------+--------+---------------------
      1 | Sayers | Biostatistician II |  11804 | soverall0@google.it
(1 row)
```

- Note that we can delete more than one record at once. Be careful using DELETE. ☺

Primary keys (add)

- Recall: primary keys uniquely defines a record in a table.
- Add a primary key:
  - ALTER TABLE payroll ADD PRIMARY KEY (*attribute*);

```
test0=# ALTER TABLE payroll ADD PRIMARY KEY (userid);
ALTER TABLE
test0=# \d payroll
                                    Table "public.payroll"
 Column |          Type          | Collation | Nullable |
--------+------------------------+-----------+----------+-----
 userid | bigint                 |           | not null | next
 name   | character varying(50)  |           | not null |
 job    | character varying(50)  |           | not null |
 salary | integer                |           | not null |
 email  | character varying(150) |           |          |
Indexes:
    "payroll_pkey" PRIMARY KEY, btree (userid)
```

Update a record

```
test0=# SELECT * FROM payroll LIMIT 1;
 userid |  name  |        job         | salary |        email
--------+--------+--------------------+--------+---------------------
      1 | Sayers | Biostatistician II |  11804 | soverall0@google.it
(1 row)
```

- To update a record, use:
  - UPDATE table_name SET attribute = whatever WHERE …;

```
test0=# UPDATE payroll SET email = 'sayers@google.com' WHERE userid = 1;
UPDATE 1
```

- Alike the DELETE function, Do not forget the WHERE clause. Otherwise, you will update/delete the entire table.

```
test0=# SELECT * FROM payroll WHERE userid = 1;
 userid |  name  |        job         | salary |       email
--------+--------+--------------------+--------+------------------
      1 | Sayers | Biostatistician II |  11804 | sayers@google.com
(1 row)
```

- You can also update several attributes after SET at once by separating each update by a comma.

Foreign keys

- Recall: in short, a foreign key is an attribute of table A that references a primary key in a table B. This foreign key links the tables A and B together.
- Note: the types of the foreign key must be the same in both tables.
- Remember that a primary key can be composite.
- To create a foreign key in a table B, use:
  - FOREIGN KEY (*attribute*) REFERENCES *table_A (attribute)*

```
test0=# CREATE TABLE regist(
test0(# userid bigint,
test0(# car varchar(50),
test0(# PRIMARY KEY (userid, car),
test0(# FOREIGN KEY (userid) REFERENCES payroll (userid)
test0(# );
CREATE TABLE
```

```
test0=# \d regist
                    Table "public.regist"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 userid | bigint                |           | not null |
 car    | character varying(50) |           | not null |
Indexes:
    "regist_pkey" PRIMARY KEY, btree (userid, car)
Foreign-key constraints:
    "regist_userid_fkey" FOREIGN KEY (userid) REFERENCES payroll(userid)
```

Foreign keys

- We can insert values in the regist table.
- Note that the key of the regist table must be composite since a user could have several cars.
- However, you can not add a row with a foreign key value that does not exist into the referenced table (there is no user in the payroll table with an id equals to 1001).

```
test0=# SELECT * FROM regist;
 userid |    car
--------+----------
      2 | Honda
      3 | Mercedes
      4 | Mercedes
      5 | Ford
      7 | Kya
      2 | Kya
(6 rows)
```

```
test0=# INSERT INTO regist VALUES (1001, 'Kya');
ERROR:  insert or update on table "regist" violates foreign key constraint "regist_userid_fkey"
DETAIL:  Key (userid)=(1001) is not present in table "payroll".
```

Inner join

- To join tables, use the JOIN keyword with a ON option to define the joint constraint.
- For the inner join, you need to join on the foreign key, common to both tables.

```
test0=# SELECT * FROM payroll JOIN regist
test0-# ON payroll.userid = regist.userid;
 userid |  name   |            job             | salary |          email          | userid |   car
--------+---------+----------------------------+--------+-------------------------+--------+----------
      2 | Martie  | Technical Writer           |  19142 | mmcaulay1@ask.com       |      2 | Honda
      3 | Codie   | Recruiting Manager         |   9876 | csoitoux2@who.int       |      3 | Mercedes
      4 | Hebert  | Administrative Assistant I |   5222 | hdegoey3@microsoft.com  |      4 | Mercedes
      5 | Farrell | GIS Technical Architect    |  18128 | fbucky4@reuters.com     |      5 | Ford
      7 | Maurine | Editor                     |  19308 | mmatteucci6@nytimes.com |      7 | Kya
      2 | Martie  | Technical Writer           |  19142 | mmcaulay1@ask.com       |      2 | Kya
(6 rows)
```

Inner join

- It seemed that we had two columns in the previous table named userid. In fact, one of them was payroll.userid while the other was regist.userid.
- If you only want to visualize some columns, you will need to indicate from which table you are extracting data.

```
test0=# SELECT payroll.userid, payroll.name, regist.car FROM payroll JOIN regist
test0-# ON payroll.userid = regist.userid;
 userid |  name   |   car
--------+---------+----------
      2 | Martie  | Honda
      3 | Codie   | Mercedes
      4 | Hebert  | Mercedes
      5 | Farrell | Ford
      7 | Maurine | Kya
      2 | Martie  | Kya
(6 rows)
```

Expanded mode

- Tables with a lot of attributes might be messy to visualize.
- \x turns on or off the expanded mode.

```
test0=# \x
Expanded display is off.
test0=# \x
Expanded display is on.
test0=# SELECT * FROM payroll JOIN regist
test0-# ON payroll.userid = regist.userid;
-[ RECORD 1 ]----------------------
userid | 2
name   | Martie
job    | Technical Writer
salary | 19142
email  | mmcaulay1@ask.com
userid | 2
car    | Honda
-[ RECORD 2 ]----------------------
userid | 3
name   | Codie
job    | Recruiting Manager
salary | 9876
email  | csoitoux2@who.int
userid | 3
car    | Mercedes
```

Left join

- A left join of A and B includes the rows in A and B joined by the foreign key of B referencing A **and** rows of A undefined in B.
- In the example, the values in regist.car for users 1, 6, 8 and 9 will be null since they have no registered car.

```
test0=# SELECT payroll.userid, payroll.name, regist.car
test0-# FROM payroll LEFT JOIN regist
test0-# ON payroll.userid = regist.userid
test0-# LIMIT 10;
 userid |  name   |   car
--------+---------+----------
      1 | Sayers  |
      2 | Martie  | Honda
      2 | Martie  | Kya
      3 | Codie   | Mercedes
      4 | Hebert  | Mercedes
      5 | Farrell | Ford
      6 | Lotty   |
      7 | Maurine | Kya
      8 | Sara    |
      9 | Garv    |
(10 rows)
```

Activity 3: SQL Queries

Given payroll and regist tables,

- Write a SQL query to return the top ten jobs by rounded average salary. Rename the columns ‘job’ and ‘avg_salary’.
- Write a SQL query to return the number of users having an email ending by ‘.net’.
- Write a SQL query to return the number of users who do not have a registered car.
- Write a SQL query to return the users “not having an email” or “having a car and a salary greater than 13000”.

Activity 4: More SQL Queries

Given the following relation that captures topics discussed by various users

Discussion(**user1, user2, topic**)

where `user1` always precedes `user2` in alphabetical order.

- Write a SQL query that returns all topics discussed by `Alice` and `Bob`, but not discussed by `Alice` and `Chuck`. (Hint: you may want to use the `not in` syntax in SQL, i.e., `select a from R where a not in (select a from S)`)

- Given the `Discussion` relation in the previous question. Write a SQL query that returns the number of topics discussed by more than 10 pairs of users. (Hint: you may want to create a temporary table.)

https://www.postgresqltutorial.com/postgresql-temporary-table/

Node.js and Postgresql interoperability

Using the pg module in Node.js

- How can you query a database in a Node.js program?
- First, install the pg module using node.js package manager.
- From what follows, create a file pg_example.js in your project folder.
- …

```
PS C:\Users\User\Desktop\Test> npm install pg

added 20 packages, and audited 21 packages in 2s

1 package is looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\User\Desktop\Test>
```

Using the pg module in Node.js

- …
- The node.js program on the right performs a query on the payroll database and displays the result on the console as an array of objects.
- Let’s have a closer look to this program in the next slides.

Test > JS pg_example.js > ...

```javascript
const {Pool} = require('pg');
const connectionInfo = `postgres://postgres:super@localhost:5432/test0`;
const pool = new Pool({connectionString: connectionInfo});

pool.query(
    `SELECT userid, name FROM payroll
    LIMIT 5;`,
    [],
    function (err, result) {
        if (err) {
            console.log(err);
            process.exit(1);
        }
        console.log(result.rows);
        process.exit(0);
    }
)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Test> node .\pg_example.js
[
  { userid: '2', name: 'Martie' },
  { userid: '3', name: 'Codie' },
  { userid: '4', name: 'Hebert' },
  { userid: '5', name: 'Farrell' },
  { userid: '6', name: 'Lotty' }
]
PS C:\Users\User\Desktop\Test>
```

Connection information

- First, we need the pg module.
- Then, we set up a ‘pool’ using the connection information to query the database and return any results.
- The connection info is a string that points to the Postgresql database server and provides username, password, host, port and database name.

```
const {Pool} = require('pg');
const connectionInfo = `postgres://postgres:super@localhost:5432/test0`;
const pool = new Pool({connectionString: connectionInfo});
```

- Note that in practice, it is not a good idea to have the superuser query your database from your webapp…
- And the password should not be hardcoded… ☺

Query the database

- With everything set up, pool.query() provides a sql query enclosed in literal quotes (1st argument), query parameters in an array (2nd argument, by default, an empty array) and a function to run when the query completes.
- Good practice: this function has an error argument

```
pool.query(
    `SELECT userid, name FROM payroll
    LIMIT 5;`,
    [],
    function (err, result) {
        if (err) {
            console.log(err);
            process.exit(1);
        }
        console.log(result.rows);
        process.exit(0);
    }
)
```

Activity 5: A simple MVC

- Create a simple database with userid, login, password, and a few other attributes of your choice. You can use mockaroo.com if necessary.
- Make sure that two different users have different logins.
- Create a simple HTML document with a form asking for login and password.
- After submission, these credentials must be passed to a nodejs web server which returns an error if login and/or password are empty, are not alphanumeric or are too long.
- If the credentials do not trigger an error, the web server send a SQL query to the database requesting the information about the user.
- If the user exists, the user should be able to view his details in his/her browser.
- If the user does not exist, an error message should be sent back.
- Optional: you can add other features of your choice. For example, you can allow a new user to create an account, update or to delete his/her record.

Summary

- Installation and setup of Postgresql
- How to use the SQL shell
- How to connect to postgresql using Node.js