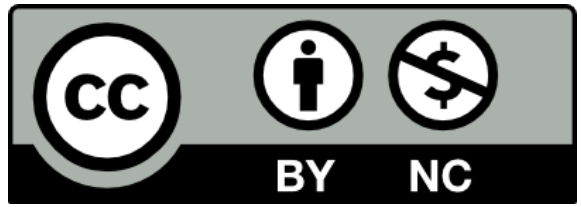
Business 4720 - Class 3

Relational Data

Joerg Evermann

Faculty of Business Administration
Memorial University of Newfoundland
jevermann@mun.ca



MEMORIAL UNIVERSITY

This Class

What You Will Learn:

- The relational data model
- Creating tables and constraints
- Querying data for descriptive analytics with SQL

MEMORIAL UNIVERSITY

The Relational Data Model

Basics

- Tables have columns/fields
- Fields are of a primitive data type

Columns Constraints

- **NOT NULL**: Field must not be empty
- **UNIQUE**: All values in columns must be different

Table Constraints

- **CHECK**: All values satisfy certain conditions
- **PRIMARY KEY**: Set of fields must uniquely identify a row
- **FOREIGN KEY**: Set of fields whose values must exist in the primary key of another table

MEMORIAL UNIVERSITY

Foreign Key Constraints

- Express relationships between tables
- May reference the same table (”*unary relationship*”) or another table (”*binary relationship*”)
- Represent ”1:1”, ”1:Many”, or ”Many:Many” relationships (”*cardinality*”)
- Together with NOT NULL contraints, can represent *optional* relationships

MEMORIAL UNIVERSITY

Basic SQL Commands

CREATE TABLE	Creates a new table with specified columns and constraints
DROP TABLE	Deletes a table and all its contents
INSERT	Inserts a row of data values into a table
UPDATE	Updates/modifies data values in a table
SELECT	Retrieves data values from one or more tables

MEMORIAL UNIVERSITY

Basic SQL Commands

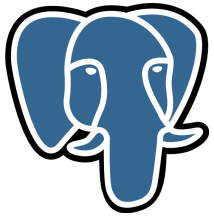
More Information

- This course covers only basic information
- Further information in PostgreSQL documentation

Data Definition	`https://www.postgresql.org/docs/current/ddl.html`
Data Types	`https://www.postgresql.org/docs/current/datatype.html`
Data Manipulation	`https://www.postgresql.org/docs/current/dml.html`
Data Queries	`https://www.postgresql.org/docs/current/queries.html`
Functions	`https://www.postgresql.org/docs/current/functions.html`

MEMORIAL UNIVERSITY

Let's Play with SQL



- PostgreSQL RDBMS installed in the course virtual machine
- Access options
  1. psql command line access
  2. DBeaver graphical user interface
  3. pgAdmin administration application

MEMORIAL UNIVERSITY

PostgreSQL Database Management System

- A DBMS is a background program without user interface
- Runs on one computer (”server”) or distributed across a cluster of multiple computers
  - *The name of the local computer is called ”localhost”*
- Administration tools allow basic manual interaction
  - pgAdmin offers easy tools for creating tables and querying data, but we will use the SQL language instead
- A DBMS can manage multiple databases
  - *Your database is called ”busi4720”*
- A database may have multiple schema
  - A schema is a grouping of tables and related info
  - *The default schema is called ”public”*
- A schema may contain multiple tables (and related information)

MEMORIAL UNIVERSITY

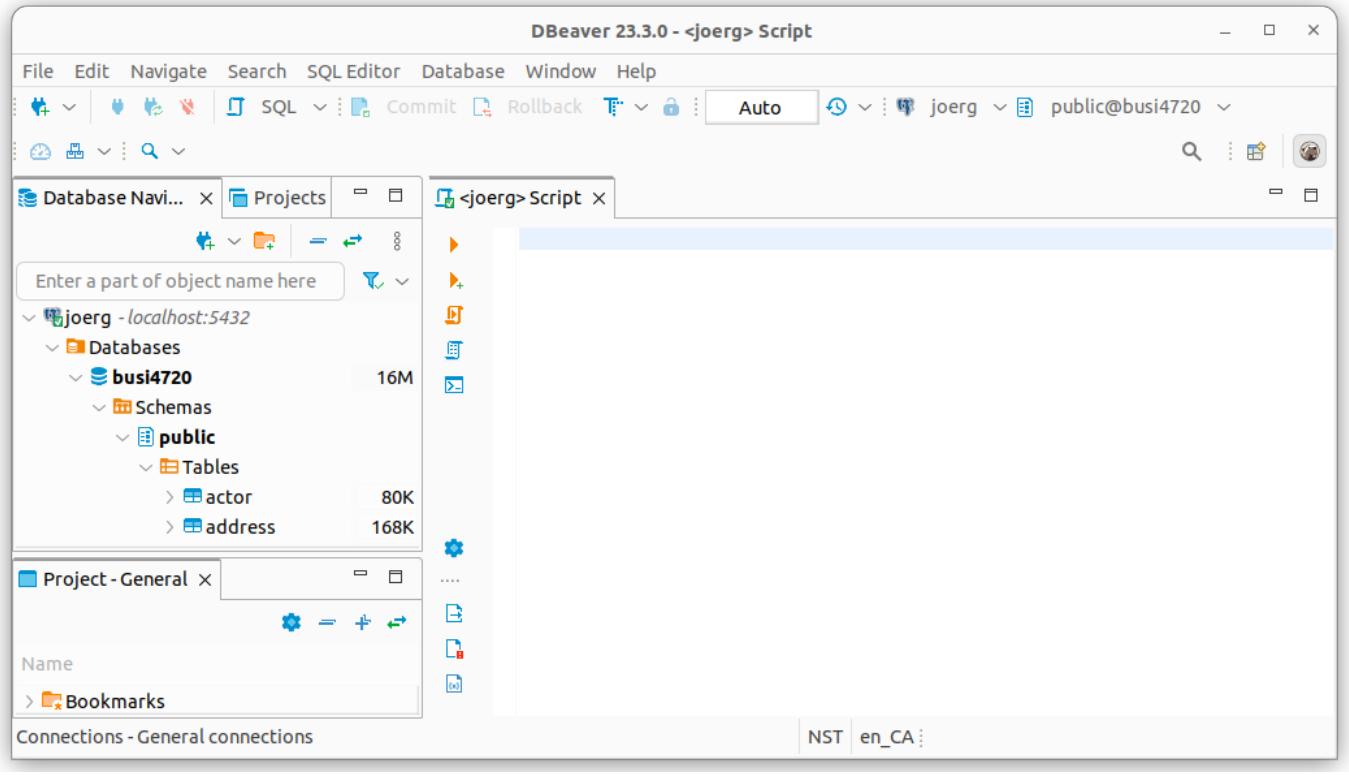
PSQL

- Type `psql` in a terminal window
- The psql command `\conninfo` shows connection info
- Quit psql using `\q`
- You should be connected to the ”busi4720” database
- **Tip**: Use a notepad application to assemble your SQL commands to paste into psql

MEMORIAL UNIVERSITY

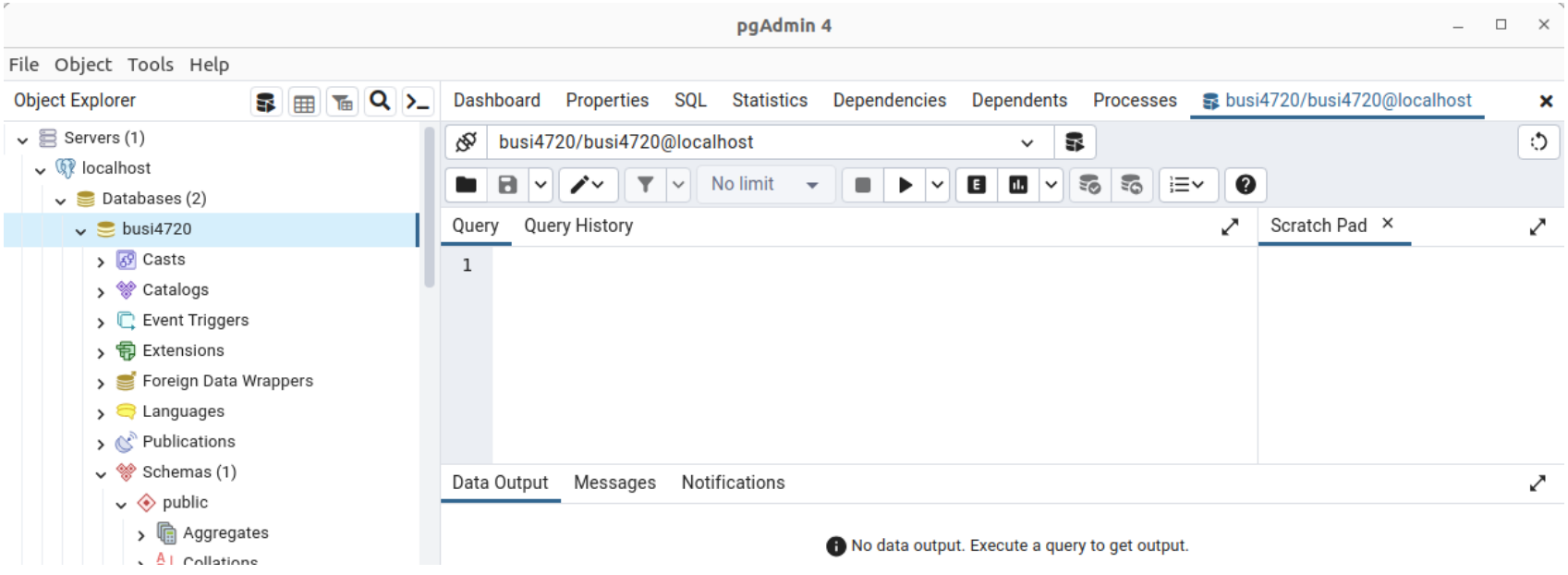
DBeaver

- Navigate to and select the ”busi4720” database in the navigator on the left
- In the toolbar, press the ”SQL” button



PostgreSQL Hands-On with pgAdmin

- Navigate to and select the ”busi4720” database in the object explorer on the left
- From the ”Tools” menu, select the ”Query tool”



PostgreSQL Hands-On

- Enter the following query:
  - In psql, just press RETURN to execute
  - In DBeaver, press CTRL-RETURN to execute
  - In pgadmin, press F5 to execute

```
CREATE TABLE products (
  pcode integer,
  name  varchar(100),
  price float4,
  PRIMARY KEY (pcode) );
```

Tips

- All SQL commands must end with a semicolon
- Capitilization does not matter

MEMORIAL UNIVERSITY

PostgreSQL Hands-On

- Create a table for the suppliers as well:

```
CREATE TABLE suppliers (
  scode integer,
  name  varchar(100),
  city  varchar(100),
  PRIMARY KEY (scode) );
```

- Insert some data into the tables:

```
INSERT INTO products VALUES (1, 'Hex Bolt', 1.99);
INSERT INTO products VALUES (2, 'Round Bolt', 2.99);

INSERT INTO suppliers VALUES (1, 'Bolts Inc', 'HVGB');
INSERT INTO suppliers
    VALUES (2, 'Hardware Co', 'Cartwright');
```

MEMORIAL UNIVERSITY

- Check that the data is there:

```
SELECT * FROM products;
SELECT * FROM suppliers;
```

- Can we capture which supplier supplies which product?
- We need a foreign key!
- Let's delete our "products" table and rework it

```
DROP TABLE products;
```

- **Assumption**: Suppliers supply many products, a product may be supplied by only one supplier

```
CREATE TABLE products (
  pcode    integer,
  name     varchar(100),
  price    float4,
  supplier integer,
  PRIMARY KEY (pcode),
  FOREIGN KEY (supplier) REFERENCES suppliers );
```

- Insert some product data:

```
INSERT INTO products VALUES(1, 'Hex Bolt', 1.99, 1);
INSERT INTO products VALUES(2, 'Round Bolt', 2.99, 1);
INSERT INTO products
    VALUES(3, 'Square Bolt', 3.99, NULL);
```

- There are products that have no supplier (the ”square bolt”)
- There are suppliers that supply many products (supplier 1)
- There are suppliers that do not supply products (supplier 2)
- *We cannot record multiple suppliers for each product!*

MEMORIAL UNIVERSITY

- Make sure that every product has a supplier:

```
DROP TABLE IF EXISTS products;

CREATE TABLE products (
  pcode    integer,
  name     varchar(100),
  price    float4,
  supplier integer NOT NULL,
  PRIMARY KEY (pcode),
  FOREIGN KEY (supplier) REFERENCES suppliers );
```

- Can products be supplied by multiple suppliers?
- We need another table, representing the ”supplies” relationship, a ”Many:Many” relationship
- Let’s drop everything and start again!

```
DROP TABLE IF EXISTS products;
DROP TABLE IF EXISTS suppliers;
```

- *Tip*: Must drop tables in correct order because of foreign keys!

PostgreSQL Hand-On

```
CREATE TABLE products (
  pcode integer,
  name  varchar(100),
  PRIMARY KEY (pcode) );

CREATE TABLE suppliers (
  scode integer,
  name  varchar(100),
  city  varchar(100),
  PRIMARY KEY (scode) );

CREATE TABLE supplies (
  scode  integer NOT NULL,
  pcode  integer NOT NULL,
  price  float4 NOT NULL,
  PRIMARY KEY (scode, pcode),
  FOREIGN KEY (scode) REFERENCES suppliers,
  FOREIGN KEY (pcode) REFERENCES products );
```

MEMORIAL UNIVERSITY

```
INSERT INTO products VALUES (1, 'Hex Bolt');
INSERT INTO products VALUES (2, 'Round Bolt');

INSERT INTO suppliers VALUES (1, 'Bolts Inc', 'HVGB');
INSERT INTO suppliers
    VALUES (2, 'Hardware Co', 'Cartwright');

INSERT INTO supplies VALUES(1, 1, 1.99);
INSERT INTO supplies VALUES(1, 2, 2.49);
INSERT INTO supplies VALUES(2, 1, 2.99);
INSERT INTO supplies VALUES(2, 2, 1.79);
```

MEMORIAL UNIVERSITY

Foreign Key Summary

1:Many Relationship

- Requires a foreign key from the ”many” table into the ”one” table
- **Example**: A supplier supplies many products but a product has one supplier (or none, depending on NOT NULL constraint)

Many:Many Relationship

- Requires a table that represents the relationship
- Foreign keys from this table reference the ”main” tables
- **Example**: A supplier supplies many products and a product can be supplied by many suppliers
- Can be extended to relationships between three or more tables
- **Example**: A supplier supplies many products from multiple warehouses

Hands-On Exercise

1. Consider the following:
   - A book has an ISBN number and a title.
   - An author has a name and an address.
   - An author can write many books, and a book can be written by multiple authors. A book is written in a certain year.
2. Write the CREATE TABLE statements with the necessary FOREIGN KEY statements, and execute them on PostgreSQL
   - Use appropriate datatypes for the columns
   - Create an appropriate PRIMARY KEY for all tables
3. Use INSERT statements to create some example data
4. Use SELECT statements to ensure your data exists

MEMORIAL UNIVERSITY

Parts of the SELECT statement

- **SELECT**: which columns to query
- **FROM**: which tables to query from
- **JOIN**: how to combine data from multiple tables
- **WHERE**: conditions on field values
- **GROUP BY**: groups within which to aggregate data
- **HAVING**: conditions on group aggregate values
- **ORDER BY**: how to sort the result
- **LIMIT**: how many results to return

Querying – Simple Example

Set up tables with 1:Many relationship:

```
DROP TABLE IF EXISTS products;
DROP TABLE IF EXISTS suppliers;

CREATE TABLE suppliers (
scode integer,
name varchar(100),
city varchar(100),
PRIMARY KEY (scode) );

CREATE TABLE products (
pcode         integer,
name         varchar(100),
price         float4,
scode    integer,
PRIMARY KEY (pcode),
FOREIGN KEY (scode) REFERENCES suppliers );
```

MEMORIAL UNIVERSITY

Querying – Simple Example

Insert some data:

```
INSERT INTO SUPPLIERS VALUES (1, 'Bolts Inc', 'HVGB');
INSERT INTO SUPPLIERS VALUES (2, 'Hardware Co', 'Cartwright');

INSERT INTO products VALUES(1, 'Hex Bolt A', 1.99, 1);
INSERT INTO products VALUES(2, 'Round Bolt 1', 2.99, 1);
INSERT INTO products VALUES(3, 'Square Bolt B', 3.99, 1);
INSERT INTO products VALUES(4, 'Hex Bolt B', 2.99, 1);
INSERT INTO products VALUES(5, 'Round Bolt 2', 1.99, 2);
INSERT INTO products VALUES(6, 'Square Bolt 7', 3.49, 2);
```

MEMORIAL UNIVERSITY

Querying – Simple Example

Select all columns from products table:

```
SELECT * FROM products;
```

Select all columns from products table for supplier 1:

```
SELECT * FROM products WHERE scode = 1;
```

Select some columns from products table with price < 3:

```
SELECT name, supplier FROM products WHERE price < 3;
```

Select all columns from products, ordered by ascending price and then descending name:

```
SELECT * FROM products ORDER BY price ASC, name DESC;
```

Select the names of the 3 cheapest products:

```
SELECT name FROM products ORDER BY price ASC LIMIT 3;
```

Querying – Join Tables

Select products and the names of their suppliers:

```
SELECT * FROM products INNER JOIN suppliers USING (scode);

-- (Almost) Equivalent:
SELECT * FROM products
    INNER JOIN suppliers ON products.scode=suppliers.scode;

SELECT * FROM products, suppliers
    WHERE products.scode=suppliers.scode;
```

Rename columns for clarity:

```
SELECT products.name AS "Product",
       price AS "Price",
       suppliers.name AS "Supplier",
       city AS "Location"
    FROM products INNER JOIN suppliers USING (scode);
```

MEMORIAL UNIVERSITY

Types of Joins

Types of joins between tables T1 and T2:

INNER	Selects only rows from T1 that have corresponding row in T2 and vice versa.
LEFT OUTER	Selects all rows from T1, if T2 does not have a corresponding row, NULL values are inserted.
RIGHT OUTER	Select all rows from T2, if T1 does not have a corresponding row, NULL values are inserted.
FULL OUTER	Selects all rows from T1, T2. If the other table is missing corresponding row, NULL values are inserted.

MEMORIAL UNIVERSITY

Querying – Groups and Aggregates

Select the number of products for each supplier:

```
SELECT scode, count(*) from products GROUP BY scode;
```

Add the name of suppliers by joining suppliers table:

```
SELECT suppliers.name, count(*)
    FROM suppliers INNER JOIN products USING (scode)
    GROUP BY scode;
```

Mean price of products for each supplier, ordered descending:

```
SELECT suppliers.name, AVG(price)
    FROM suppliers INNER JOIN products USING (scode)
    GROUP BY scode
    ORDER BY AVG(price) DESC;
```

Find only suppliers for which average price less than 2.80:

```
SELECT suppliers.name, AVG(price)
    FROM suppliers INNER JOIN products USING (scode)
    GROUP BY scode
    HAVING AVG(price) < 2.8;
```

Frequently Used Aggregation Functions

Standard SQL:

MIN()	MAX()	COUNT()	SUM()	AVG()

PostgreSQL (excerpt):

CORR()	STDDEV()	VARIANCE()
MODE()	PERCENTILE()	REGR_SLOPE()
REGR_R2()	RANK()	DENSE_RANK()

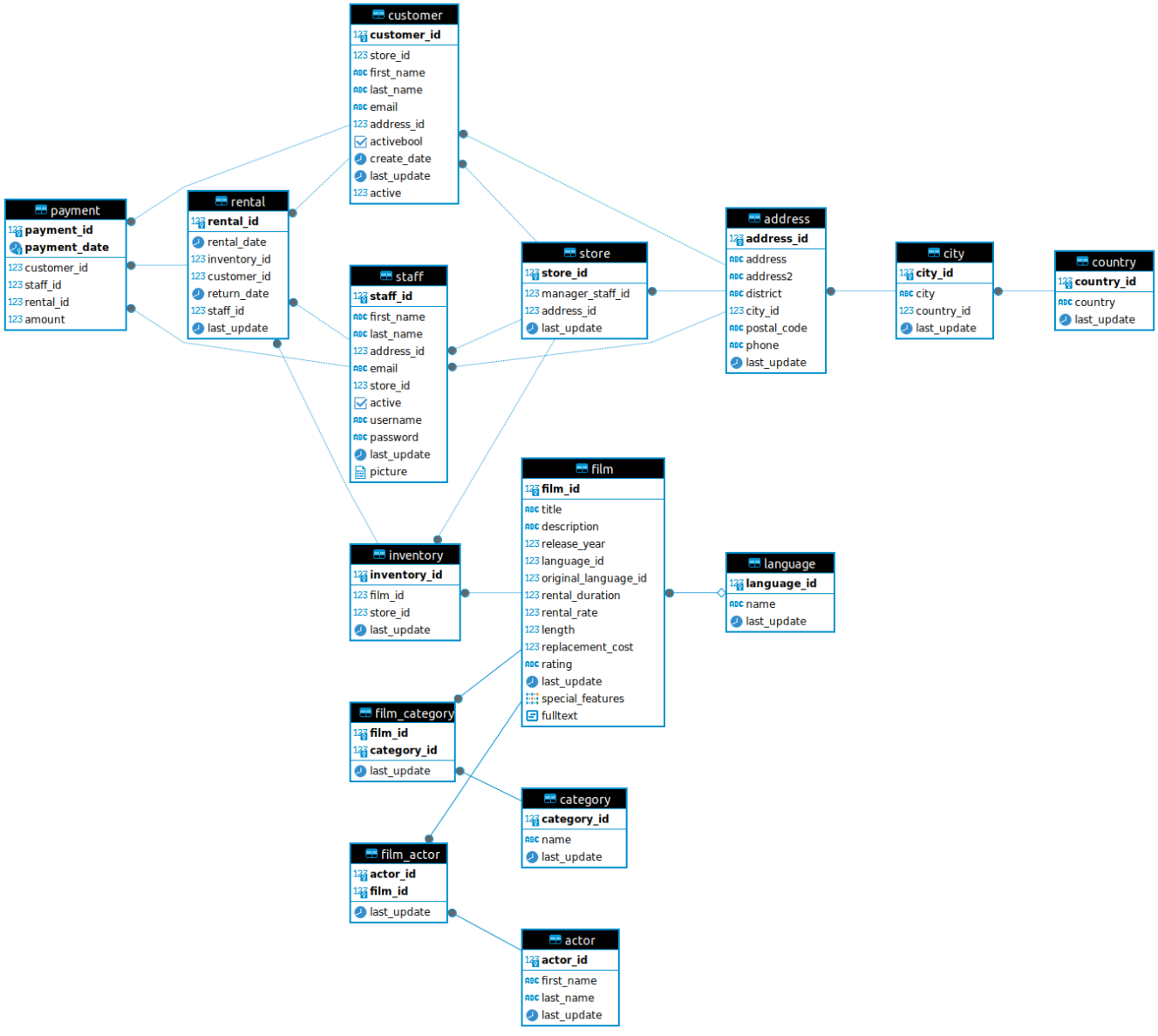
MEMORIAL UNIVERSITY

Relational Diagrams

- Graphically show the relationships in a database model
- Often called "*Entity-Relationship diagram*" (but that is not quite correct)
- Useful for understanding the structure of the data
- Useful for writing queries with many JOINs
- Can be automatically created from an existing database
  - In the **pgAdmin** Object Explorer, right-click on the "busi4720" database, select "ERD for Database"
  - In the **DBeaver** Navigator, select the "busi4720", then the "public" schema, then right click and select "View Diagram"

MEMORIAL UNIVERSITY

Relational Diagram for the Pagila Database



MEMORIAL UNIVERSITY

The Pagila Demo Database

- Based on the Sakila database[1], ported to PostgreSQL[2]
- Pre-installed in the course virtual machine
- DVD rental case with main tables:
  - **Film**: has many categories, is in inventory at many stores
  - **Store**: has many films in inventory, has a staff manager
  - **Customer**: may have many rentals, associated with a store
  - **Actor**: may appear in many films, films have many actors
  - **Inventory**: stores have many films in inventory, and film may be in inventory in many stores
  - **Rental**: Inventory is rented by customer through staff
  - **Staff**: Staff are associated with a store
  - **Address**: Stores, customers, and staff have an addresses

[1] `https://dev.mysql.com/doc/sakila/en/`, `https://dev.mysql.com/doc/sakila/en/sakila-license.html`

[2] `https://github.com/devrimgunduz/pagila`, `https://github.com/devrimgunduz/pagila/blob/master/LICENSE.txt`

MEMORIAL UNIVERSITY

Querying Data with SQL

Find all actors and the films they appeared in, ordered by film category and year, for those films that are rated PG:

```
SELECT concat(left(actor.first_name, 1),
              '. ', actor.last_name) AS Actor,
       category.name AS Category,
       film.title,
       film.release_year
  FROM film_actor
  INNER JOIN actor USING (actor_id)
  INNER JOIN film USING (film_id)
  INNER JOIN film_category USING (film_id)
  INNER JOIN category USING (category_id)
  WHERE film.rating = 'PG'
  ORDER BY actor.last_name,
           actor.first_name,
           category.name ASC,
           film.release_year DESC,
           film.title ASC;
```

MEMORIAL UNIVERSITY

Equivalent alternative without using JOIN:

```
SELECT concat(left(actor.first_name, 1),
              '. ', actor.last_name) AS Actor,
       category.name AS Category,
       film.title,
       film.release_year
  FROM film_actor, film, actor, film_category, category
  WHERE actor.actor_id = film_actor.actor_id AND
        film.film_id = film_actor.film_id AND
        film_category.film_id = film.film_id AND
        category.category_id = film_category.category_id AND
        film.rating = 'PG'
  ORDER BY actor.last_name,
           actor.first_name,
           category.name ASC,
           film.release_year DESC,
           film.title ASC;
```

Querying Data with SQL

Find the most popular actors in the rentals in each city:

```
SELECT city.city,
       concat(actor.first_name, '. ',
           actor.last_name) AS actor_name,
       count(concat(actor.first_name, '. ',
           actor.last_name)) AS Number_Rentals
  FROM rental
  INNER JOIN inventory USING (inventory_id)
  INNER JOIN store USING (store_id)
  INNER JOIN address USING (address_id)
  INNER JOIN city USING (city_id)
  INNER JOIN film USING (film_id)
  INNER JOIN film_actor USING (film_id)
  INNER JOIN actor USING (actor_id)
  GROUP BY city.city, actor.actor_id
  HAVING count(rental.rental_id) >= 300
  ORDER BY city ASC,
           Number_Rentals DESC,
           actor_name ASC;
```

MEMORIAL UNIVERSITY

Querying Data with SQL

Find the customers who spent the most, with their phone numbers and cities, the cities their store is in, and the number of rentals with the highest total rental payments for each category grouped by city of the rental store:

```
SELECT category.name AS category_name,
       store_city.city AS store_city,
       customer.customer_id,
       concat(customer.first_name, ' ',
          customer.last_name) AS customer_name,
       cust_city.city AS customer_city,
       cust_address.phone AS customer_phone,
       count(rental.rental_id) AS num_rentals,
       sum(amount) AS total_amount
 FROM city AS cust_city, city AS store_city,
      address AS cust_address, address AS store_address,
      store, rental
 INNER JOIN payment USING (customer_id)
 INNER JOIN customer USING (customer_id)
 INNER JOIN inventory USING (inventory_id)
 INNER JOIN film USING (film_id)
 INNER JOIN film_category USING (film_id)
 INNER JOIN category USING (category_id)
```

Cont'd

```
WHERE store.store_id = inventory.store_id
  AND store_address.address_id = store.address_id
  AND store_city.city_id = store_address.city_id
  AND cust_address.address_id = customer.address_id
  AND cust_city.city_id = cust_address.city_id
GROUP BY category.name, customer.customer_id,
         cust_address.address_id, cust_city.city,
         store_city.city
HAVING sum(amount) IN (
 SELECT sum(amount) AS maxamount
  FROM store, address, city AS inner_city, rental
  INNER JOIN payment USING (customer_id)
  INNER JOIN customer USING (customer_id)
  INNER JOIN inventory USING (inventory_id)
  INNER JOIN film USING (film_id)
  INNER JOIN film_category USING (film_id)
  INNER JOIN category AS inner_category USING (category_id)
```

MEMORIAL UNIVERSITY

Cont'd

```
  WHERE inner_category.name = category.name AND
        inner_city.city = store_city.city AND
        store.store_id = inventory.store_id AND
        address.address_id = store.address_id AND
        inner_city.city_id = address.city_id
  GROUP BY inner_category.name,
           inner_city.city,
           customer.customer_id
  ORDER BY inner_category.name ASC,
           inner_city.city,
           maxamount DESC
  LIMIT 1 )
ORDER BY category.name ASC, store_city ASC;
```

Querying Data with SQL

Get the total rental revenue and number of rentals for each store by month:

```
SELECT city.city,
       extract(year from payment_date) AS year,
       extract(month from payment_date) AS month,
       sum(amount) as dollars,
       count(rental.rental_id) as rentals
FROM payment, rental, inventory, store, address, city
WHERE payment.rental_id = rental.rental_id AND
      rental.inventory_id = inventory.inventory_id AND
      inventory.store_id = store.store_id AND
      store.address_id = address.address_id AND
      address.city_id = city.city_id
GROUP BY city.city,
         extract(year from payment_date),
         extract(month from payment_date)
ORDER BY city.city,
         extract(year from payment_date),
         extract(month from payment_date);
```

MEMORIAL UNIVERSITY

Querying Data with SQL

Get the top 5 and the bottom 5 grossing customers by year

```
( SELECT concat(customer.first_name, ' ',
               customer.last_name) AS customer_name,
        extract(year from payment_date) AS year,
        sum(amount) as dollars,
        'Top-5' as note
FROM payment, customer
WHERE payment.customer_id = customer.customer_id
GROUP BY extract(year from payment_date),customer.customer_id
ORDER BY dollars DESC LIMIT 5
) UNION (
SELECT concat(customer.first_name, ' ',
             customer.last_name) AS customer_name,
        extract(year from payment_date) AS year,
        sum(amount) as dollars,
        'Bottom-5' as note
FROM payment, customer
WHERE payment.customer_id = customer.customer_id
GROUP BY extract(year from payment_date),customer.customer_id
ORDER BY dollars ASC LIMIT 5 )
ORDER BY dollars DESC;
```

Tips on Querying Data with SQL

- Subqueries are intuitive but not efficient
- Use set operations to combine results from multiple queries (UNION, INTERSECT, EXCEPT)
- Use DISTINCT to query unique values
- Postgres can import/export from/to CSV and JSON files

MEMORIAL UNIVERSITY

Hands-On Exercises

1. Find the names and the rental numbers of the top 5 customers who rented the most films
   - **Tip**: Join tables ”rental”, ”customer”, use ”group by” and the ”count()” function
2. Calculate the rental revenue per customer. Who are the top 5? Bottom 5?
   - **Tip**: Join tables ”rental”, ”customer”, ”payment”, use ”group by” and the ”sum()” function
3. Calculate the average rental revenue per customer for each store
   - **Tip**: Join tables ”rental”, ”customer”, ”payment”, ”inventory”, use ”group by” and the ”avg()” function

MEMORIAL UNIVERSITY

Hands-On Exercises [cont'd]

4. Calculate the rental counts for each country of customer. Are there countries with no rentals?
   - **Tip**: Join tables "rental", "customer", "address", "city", "country", use "group by" and the "count()" function
5. Find all films with a single actor
   - **Tip**: Join tables "film", "film_actor", use "group by" and the "count()" function with a HAVING clause
6. Create tables to represent a part-of hiearchy. For example, a product may be a part of another product, and prodcuts may have multiple parts.
   - **Tip**: You need only one table

MEMORIAL UNIVERSITY