SQL Notes

**YouTube - SQL Tutorial - Full Database Course for Beginners**

**13/4/22**

Databases:

* Any collection of related info
* DBMS – Database Management System – software program that helps users create, maintain, and secure a database
* CRUD – create, read/retrieve, update, delete – 4 main operations to do with databases
* 2 main types of databases
  + Relational (SQL)
    - Organizes data into one or more tables
    - Each table has rows and columns
    - A unique key identifies each row
    - Use RDBMS, which mySQL is an example
      * Use Structured Query Language (SQL), the standardized language for interacting with RDBMS
  + Non-relational
    - Organize data in anything but traditional table
* Database query
  + Request made to the DBMS for specific info

Tables and Keys:

* Colum defines a single attribute
* Row is individual entry
* One special column called the primary key – uniquely defines the row (even if all other attributes between rows are the same)
  + Surrogate key (no mapping to real world) vs natural key (has meaning/purpose in real world, e.g. social security number).
  + Primary key can be made up of 2 columns (composite key) – because neither column alone can uniquely identify the row.
* Foreign key
  + Refers to the primary key in another table. The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.
  + Used to define relationships between tables.
  + Can have multiple in one table.
  + Foreign key can relate back to the same table.

SQL Basics:

* Hybrid language
  + Data Query Language (DQL)
  + Data Definition Language (DDL)
  + Data Control Language (DCL) – control access to data
  + Data Manipulation Language (DML) – inserting, updating, deleting data from database
* Query is set of instructions given to the RDMBS (written in SQL) that tells the RDBMS what info you want it to retrieve for you

MySQL Windows Installation:

* Process
  + Google Download MySQL Community Server
  + Click on Windows MySQL Installer MSI
  + Download
  + Accept license agreement, etc,
  + I did a custom installation, just installation server and shell.
  + Wait for things to be installed
  + Set up password for 'root' account
  + Once everything has configured, can click 'finish'
* Using MySQL:
  + Open MySQL Command Line Client, enter password
  + E.g. command: Can make a database with: create database databasename;
  + Here we do: create database giraffe;
* PopSQL - text editor installation process:
  + Google PopSQL
  + Had to sign in with google to download it
  + Now have PopSQL app
  + Went to 'manage connections', and then 'add new connection'
  + Put MySQL as the connection type, hostname is localhost (as long as select connect directly from my computer), database is giraffe (the one we made above), username is root, input password, and click connect
  + Now have mysql connection, and giraffe database showing
  + We have a text editor which is linked with our database

Creating Tables:

* Create tables inside of our database (populate the database to define our database schema)
* NB: Drag file from drafts into MyQueries to save?
* NB: Comment in SQL with /\* (text can span multiple lines) \*/ or -- (text cannot extend to new line)
* Common data types:
  + INT -- whole numbers
  + DECIMAL(M,N) -- decimal numbers. M is total number of digits, N is number you want to store after decimal point
  + VARCHAR(1) -- string of text of length 1
  + BLOB -- binary large object, stores large data
  + DATE -- 'YYYY-MM-DD'
  + TIMESTAMP -- 'YYYY-MM-DD HH:MM:SS'
* Click anywhere in a command and use run in PopSQL to run it – will say success if it has worked

/\* Creating database tables \*/

CREATE TABLE student (

    -- now create table attributes/columns - each has datatype

    student\_id INT PRIMARY KEY, -- tell mySQL this is the primary key for the table

    names VARCHAR(20),   -- allocate sufficient space to be able to store any name

    major VARCHAR(20) -- no comma

);

-- CREATE and TABLE are SQL reserve words, doesn't have to be all CAPS, but it is convention

-- Any command ends with semi-colon

-- could also define primary key at end with:

-- PRIMARY KEY(student\_id)

-- check whether table has been created correctly

-- decribe the table

DESCRIBE student;

-- to delete the table:

DROP TABLE student;

-- modify the table by adding another column called gpa with decimal data type

ALTER TABLE student ADD gpa DECIMAL(3,2);

-- or remove a column

ALTER TABLE student DROP COLUMN gpa;

Inserting Data:

/\* Inserting data \*/

-- get all (\*) info from table

SELECT \* FROM student;

INSERT INTO student VALUES(1, 'Jack', 'Biology');

-- put student id first, then name, then major

-- use quotes for strings

-- when run, it says we have affected 1 row

INSERT INTO student VALUES(2, 'Kate', 'Sociology');

-- what if student had no major? - Specify the attributes you do have

INSERT INTO student(student\_id, name) VALUES(3, 'Claire');

-- will insert NULL in major column

-- can't insert duplicate info with same primary key

4/5/22

NB – In PopSQL, had to manage connection and toggle the connect directly from my computer back to on.

Constraints:

-- Constraints ###########################################

-- 1.

DROP TABLE student;

CREATE TABLE student (

    student\_id INT,

    name VARCHAR(20) NOT NULL, -- This column cannot be NULL, has to have a name

    major VARCHAR(20) UNIQUE, -- each row for this column has to be unique (can't have same major)

    PRIMARY KEY(student\_id)

);

SELECT \* FROM student;

-- populate table

INSERT INTO student VALUES(1, 'Jack', 'Biology');

INSERT INTO student VALUES(2, 'Kate', 'Sociology');

INSERT INTO student VALUES(3, NULL, 'Chemistry'); -- Won't work because can't be NULL

INSERT INTO student VALUES(4, 'Jack', 'Biology'); -- Won't work because can't have duplicate major entry

INSERT INTO student VALUES(5, 'Mike', 'Computer Science');

-- NOT NULL and UNIQUE are 'constraints'

-- 2.

DROP TABLE student;

CREATE TABLE student (

    student\_id INT,

    name VARCHAR(20),

    major VARCHAR(20) DEFAULT 'Undecided', -- if no entry given, fill in with this default

    PRIMARY KEY(student\_id)

);

SELECT \* FROM student;

-- populate table

INSERT INTO student(student\_id, name) VALUES(1, 'Jack');

INSERT INTO student VALUES(2, 'Kate', 'Sociology');

INSERT INTO student VALUES(3, NULL, 'Chemistry');

INSERT INTO student VALUES(4, 'Jack', 'Biology');

INSERT INTO student VALUES(5, 'Mike', 'Computer Science');

 -- 3.

DROP TABLE student;

CREATE TABLE student (

    student\_id INT AUTO\_INCREMENT, -- specify that data should be automatically incremented every time we add one in

    name VARCHAR(20),

    major VARCHAR(20),

    PRIMARY KEY(student\_id)

);

SELECT \* FROM student;

-- populate table

INSERT INTO student(name, major) VALUES('Jack', 'Biology');

INSERT INTO student(name, major) VALUES('Kate', 'Sociology');

-- has automatically put in student\_id primary keys

Update and delete:

-- Update and delete ############################################

DROP TABLE student;

CREATE TABLE student (

    student\_id INT,

    name VARCHAR(20),

    major VARCHAR(20),

    PRIMARY KEY(student\_id)

);

-- populate table

INSERT INTO student VALUES(1, 'Jack', 'Biology');

INSERT INTO student VALUES(2, 'Kate', 'Sociology');

INSERT INTO student VALUES(3, 'Claire', 'Chemistry');

INSERT INTO student VALUES(4, 'Jack', 'Biology');

INSERT INTO student VALUES(5, 'Mike', 'Computer Science');

SELECT \* FROM student;

-- change name of one of the majors - the following 3 lines are 1 SQL statement/query

UPDATE student

SET major = 'Bio'

WHERE major = 'Biology';

UPDATE student

SET major = 'Comp Sci'

WHERE major = 'Computer Science';

UPDATE student

SET major = 'Comp Sci'

WHERE student\_id = 4;

UPDATE student

SET major = 'Biochemistry'

WHERE major = 'Bio' OR major = 'Chemistry'; -- do multiple conditions with OR

UPDATE student

SET name = 'Tom', major = 'undecided' -- change multiple columns in same query

WHERE student\_id = 1;

UPDATE student

SET name = 'Tom', major = 'undecided' -- change multiple columns in same query

WHERE student\_id = 1;

UPDATE student

SET major = 'undecided'; -- apply to every row of table

SELECT \* FROM student;

DELETE FROM student

WHERE student\_id = 5;

DELETE FROM student

WHERE name = 'Tom' AND major = 'undecided';

DELETE FROM student; -- delete all rows

Basic Queries:

* Asking the DBMS for a particular piece of info

DROP TABLE student;

CREATE TABLE student (

    student\_id INT,

    name VARCHAR(20),

    major VARCHAR(20),

    PRIMARY KEY(student\_id)

);

-- populate table

INSERT INTO student VALUES(1, 'Jack', 'Biology');

INSERT INTO student VALUES(2, 'Kate', 'Sociology');

INSERT INTO student VALUES(3, 'Claire', 'Chemistry');

INSERT INTO student VALUES(4, 'Jack', 'Biology');

INSERT INTO student VALUES(5, 'Mike', 'Computer Science');

-- SELECT tells DBMS that we want to get some info from it

SELECT \* FROM student; -- selects everything/all

SELECT name

FROM student; -- just get names

SELECT name, major

FROM student;

SELECT student.name, student.major -- does same thing as above but may be useful when things get more complicated

FROM student;

SELECT student.name, student.major

FROM student

ORDER BY name; -- in alphabetical order based off name

SELECT student.name, student.major

FROM student

ORDER BY name DESC; -- in backwards alphabetical order based off name

SELECT student.name, student.major

FROM student

ORDER BY student\_id DESC; -- even though not returning student\_id, can still order by it

SELECT \*

FROM student

ORDER BY student\_id ASC;

SELECT \*

FROM student

ORDER BY major, student\_id; -- order by major first, and if any have same major, then order by student\_id

SELECT \*

FROM student

LIMIT 2; -- limit number of rows you get back to 2

SELECT \*

FROM student

ORDER BY student\_id DESC

LIMIT 2;

SELECT \*

FROM student

WHERE major = 'Biology'; -- has to meet certain condition

SELECT name, major

FROM student

WHERE major = 'Chemistry' OR major = 'Biology';

-- other condition comparators: <, >, <=, >=, =, <> (not equal), AND, OR

SELECT name, major

FROM student

WHERE major <> 'Chemistry';

SELECT name, major

FROM student

WHERE student\_id < 3;

SELECT \*

FROM student

WHERE student\_id <= 3 AND name <> 'Jack';

-- IN keyword

SELECT \*

FROM student

WHERE name IN ('Claire', 'Kate', 'Mike'); -- name has to be one of these

SELECT \*

FROM student

WHERE major IN ('Biology', 'Chemistry') AND student\_id > 2;

Company Database Intro:

-- make the company database

CREATE TABLE employee (

    emp\_id INT PRIMARY KEY,

    first\_name VARCHAR(40),

    last\_name VARCHAR(40),

    birth\_day DATE,

    sex VARCHAR(1),

    salary INT,

    super\_id INT, -- This will be a foreign key but can't define this yet as table it will point to doesn't exist yet

    branch\_id INT -- This will also be a foreign key

);

CREATE TABLE branch (

  branch\_id INT PRIMARY KEY,

  branch\_name VARCHAR(40),

  mgr\_id INT, -- This will become a foreign key which will point to the employee table

  mgr\_start\_date DATE,

  FOREIGN KEY(mgr\_id) REFERENCES employee(emp\_id) ON DELETE SET NULL

);

-- So the foreign key is mgr\_id which points to emp\_id in employee table. Put ON DELETE SET NULL after, which will be discussed later on

-- Now make super\_id and branch\_id in employee table foreign keys

ALTER TABLE employee

ADD FOREIGN KEY(branch\_id) -- make branch\_id a foreign key

REFERENCES branch(branch\_id) -- points to branch\_id in branch table

ON DELETE SET NULL;

ALTER TABLE employee

ADD FOREIGN KEY(super\_id)

REFERENCES employee(emp\_id)

ON DELETE SET NULL;

-- make client table

CREATE TABLE client (

  client\_id INT PRIMARY KEY,

  client\_name VARCHAR(40),

  branch\_id INT,

  FOREIGN KEY(branch\_id) REFERENCES branch(branch\_id) ON DELETE SET NULL

);

-- make works\_with table, which has composite primary key

-- Composite primary key made of emp\_id and client\_id, which are both also foreign keys

 CREATE TABLE works\_with (

  emp\_id INT,

  client\_id INT,

  total\_sales INT,

  PRIMARY KEY(emp\_id, client\_id),

  FOREIGN KEY(emp\_id) REFERENCES employee(emp\_id) ON DELETE CASCADE,

  FOREIGN KEY(client\_id) REFERENCES client(client\_id) ON DELETE CASCADE

);

-- ON DELETE CASCADE will be discussed later

-- make branch\_supplier table

CREATE TABLE branch\_supplier (

  branch\_id INT,

  supplier\_name VARCHAR(40),

  supply\_type VARCHAR(40),

  PRIMARY KEY(branch\_id, supplier\_name),

  FOREIGN KEY(branch\_id) REFERENCES branch(branch\_id) ON DELETE CASCADE

);

-- Now need to insert data into these tables

-- Corporate branch

INSERT INTO employee VALUES(100, 'David', 'Wallace', '1967-11-17', 'M', 250000, NULL, NULL);

-- have to put branch id as NULL to start with because the corporate branch (1) hasn't been created yet

-- now make the corporate branch in the branch table

INSERT INTO branch VALUES(1, 'Corporate', 100, '2006-02-09');

-- so can now update David Wallace entry to say that he works for the corporate branch

-- have to do it this way due to the circular relationship between the 2 tables

UPDATE employee

SET branch\_id = 1

WHERE emp\_id = 100;

-- insert other entry in the corporate branch

INSERT INTO employee VALUES(101, 'Jan', 'Levinson', '1961-05-11', 'F', 110000, 100, 1);

-- Scranton branch

INSERT INTO employee VALUES(102, 'Michael', 'Scott', '1964-03-15', 'M', 75000, 100, NULL);

INSERT INTO branch VALUES(2, 'Scranton', 102, '1992-04-06');

UPDATE employee

SET branch\_id = 2

WHERE emp\_id = 102;

INSERT INTO employee VALUES(103, 'Angela', 'Martin', '1971-06-25', 'F', 63000, 102, 2);

INSERT INTO employee VALUES(104, 'Kelly', 'Kapoor', '1980-02-05', 'F', 55000, 102, 2);

INSERT INTO employee VALUES(105, 'Stanley', 'Hudson', '1958-02-19', 'M', 69000, 102, 2);

-- Stamford branch

INSERT INTO employee VALUES(106, 'Josh', 'Porter', '1969-09-05', 'M', 78000, 100, NULL);

INSERT INTO branch VALUES(3, 'Stamford', 106, '1998-02-13');

UPDATE employee

SET branch\_id = 3

WHERE emp\_id = 106;

INSERT INTO employee VALUES(107, 'Andy', 'Bernard', '1973-07-22', 'M', 65000, 106, 3);

INSERT INTO employee VALUES(108, 'Jim', 'Halpert', '1978-10-01', 'M', 71000, 106, 3);

-- BRANCH SUPPLIER

INSERT INTO branch\_supplier VALUES(2, 'Hammer Mill', 'Paper');

INSERT INTO branch\_supplier VALUES(2, 'Uni-ball', 'Writing Utensils');

INSERT INTO branch\_supplier VALUES(3, 'Patriot Paper', 'Paper');

INSERT INTO branch\_supplier VALUES(2, 'J.T. Forms & Labels', 'Custom Forms');

INSERT INTO branch\_supplier VALUES(3, 'Uni-ball', 'Writing Utensils');

INSERT INTO branch\_supplier VALUES(3, 'Hammer Mill', 'Paper');

INSERT INTO branch\_supplier VALUES(3, 'Stamford Lables', 'Custom Forms');

-- CLIENT

INSERT INTO client VALUES(400, 'Dunmore Highschool', 2);

INSERT INTO client VALUES(401, 'Lackawana Country', 2);

INSERT INTO client VALUES(402, 'FedEx', 3);

INSERT INTO client VALUES(403, 'John Daly Law, LLC', 3);

INSERT INTO client VALUES(404, 'Scranton Whitepages', 2);

INSERT INTO client VALUES(405, 'Times Newspaper', 3);

INSERT INTO client VALUES(406, 'FedEx', 2);

-- WORKS\_WITH

INSERT INTO works\_with VALUES(105, 400, 55000);

INSERT INTO works\_with VALUES(102, 401, 267000);

INSERT INTO works\_with VALUES(108, 402, 22500);

INSERT INTO works\_with VALUES(107, 403, 5000);

INSERT INTO works\_with VALUES(108, 403, 12000);

INSERT INTO works\_with VALUES(105, 404, 33000);

INSERT INTO works\_with VALUES(107, 405, 26000);

INSERT INTO works\_with VALUES(102, 406, 15000);

INSERT INTO works\_with VALUES(105, 406, 130000);

-- check what tables look like

select \* from employee;

select \* from works\_with;

-- More basic queries

-- find all employees

SELECT \*

FROM employee;

-- find all clients

SELECT \*

FROM client;

-- find all employees ordered by salary

SELECT \*

FROM employee

ORDER BY salary DESC;

-- find all employees ordered by sex then name

SELECT \*

FROM employee

ORDER BY sex, first\_name, last\_name;

-- find the first 5 employees in the table

SELECT \*

FROM employee

LIMIT 5;

-- find the forename and surnames of all employees (shows different column names using AS keyword)

SELECT first\_name AS forename, last\_name AS surname

FROM employee;

-- find out all the different genders using distinct keyword

SELECT DISTINCT sex

FROM employee;

Functions:

-- Functions based on company database

-- find the number of employees

SELECT COUNT(emp\_id)

FROM employee;

-- find number of employees that have a supervisor

SELECT COUNT(super\_id)

FROM employee;

-- find the number of female employees born after 1970

SELECT COUNT(emp\_id)

FROM employee

WHERE sex = 'F' AND birth\_day > '1970-01-01';

-- find the average of all employee's salaries

SELECT AVG(salary)

FROM employee;

-- find the average male salary

SELECT AVG(salary)

FROM employee

WHERE sex = 'M';

Wildcards:

-- % means any number of characters

-- \_ means any single character

-- find any clients who are a LLC

SELECT \*

FROM client

WHERE client\_name LIKE '%LLC'; -- name has to end in LLC

-- LIKE is special SQL keyword which we can use with wildcards

-- find any branch supplier who are in the label business

SELECT \*

FROM branch\_supplier

WHERE supplier\_name LIKE '% Label%';

-- find any employee born in OCT

SELECT \*

FROM employee

WHERE birth\_day LIKE '\_\_\_\_-10%';

-- find any clients who are schools

SELECT \*

FROM client

WHERE client\_name LIKE '%school%';

Unions:

-- Combine results of multiple SELECT statements into one

-- find list of employee and branch names

SELECT first\_name AS name\_branch\_client -- change name of comment

FROM employee

UNION

SELECT branch\_name

FROM branch

UNION

SELECT client\_name

FROM client;

-- have to have same number of columns

-- have to have similar data-type

-- Find list of all clients and branch suppliers' names

SELECT client\_name, client.branch\_id -- prefix table to column to make it more readable

FROM client

UNION

SELECT supplier\_name, branch\_supplier.branch\_id

FROM branch\_supplier;

-- find list of all money spent or earned by the company

SELECT salary

FROM employee

UNION

SELECT total\_sales

FROM works\_with;

Joins:

-- Combine rows from 2 or more tables based on a related column between them

-- Make new branch (no manager id or start date)

INSERT INTO branch VALUES(4, 'Buffalo', NULL, NULL);

SELECT \*

FROM branch;

-- find all branches and the names of their managers

-- branch table only has manger id, but not their name

-- mgr\_id in branch table and emp\_id in employee table are the shared column

SELECT employee.emp\_id, employee.first\_name, branch.branch\_name

FROM employee

JOIN branch

ON employee.emp\_id = branch.mgr\_id; -- column in common

-- There are 4 basic types of join

-- JOIN is an INNER JOIN - whenever mgr\_id is ame as emp\_id, this will be returned to us

-- LEFT JOIN -- all employees will be returned (not just managers)

-- all rows from the left table (i.e. employee table i.e. one specified in FROM statement) are included

SELECT employee.emp\_id, employee.first\_name, branch.branch\_name

FROM employee

LEFT JOIN branch

ON employee.emp\_id = branch.mgr\_id;

-- RIGHT JOIN -- includes all rows from branch table (right table)

SELECT employee.emp\_id, employee.first\_name, branch.branch\_name

FROM employee

RIGHT JOIN branch

ON employee.emp\_id = branch.mgr\_id;

-- FULL OUTER JOIN - CAN'T do in MySQL - basically left and right join combined

Nested Queries:

-- Using multiple select statements to get specific peice of info

-- one query informing another query

-- Find names of employees who have sold over 30,000 to single client

-- Use emp\_id in client table to get name from employee table

SELECT employee.first\_name, employee.last\_name

FROM employee

WHERE employee.emp\_id IN ( -- check if emp\_id is in the following result

    SELECT works\_with.emp\_id

    FROM works\_with

    WHERE works\_with.total\_sales > 30000

);

-- find all clients who are handled by the branch that michael scott manages

-- assume you know Michael's id

-- manager\_id maps us to branch\_id in branch table, and branch\_id maps us to clients in client table

SELECT client.client\_name -- get the clients

FROM client

WHERE client.branch\_id = ( -- where branch\_id is met by the following

    SELECT branch.branch\_id

    FROM branch

    WHERE branch.mgr\_id = 102 -- this gets us the branch

    LIMIT 1 -- make sure only get 1 manager

);

On delete:

/\*

CREATE TABLE branch (

  branch\_id INT PRIMARY KEY,

  branch\_name VARCHAR(40),

  mgr\_id INT,

  mgr\_start\_date DATE,

  FOREIGN KEY(mgr\_id) REFERENCES employee(emp\_id) ON DELETE SET NULL

  -- i.e. if the employee\_id in the employee table gets deleted, we set the mgr\_id in the branch table to NULL

);

\*/

-- If we delete Michael Scott from the employee table, then 102 mgr\_id in the branch table no longer exists

-- The mgr\_id in the branch table is supposed to be linking us to the employee in the employee table

-- On delete set null means if we delete an employee, the mgr\_id in the branch table will be set to NULL

-- On delete cascade with delete the entire row in the branch table

DELETE FROM employee

WHERE emp\_id = 102;

SELECT \*

FROM employee; -- no michael scott. Lots of employees have NULL as their super\_id

SELECT \*

FROM branch; -- NULL in mgr\_id for Scranton

/\*

CREATE TABLE branch\_supplier (

  branch\_id INT,

  supplier\_name VARCHAR(40),

  supply\_type VARCHAR(40),

  PRIMARY KEY(branch\_id, supplier\_name),

  FOREIGN KEY(branch\_id) REFERENCES branch(branch\_id) ON DELETE CASCADE

);

\*/

-- if branch is removed in branch table, any branch\_suppliers for that branch will be deleted (entire row)

DELETE FROM branch

WHERE branch\_id = 2;

SELECT \*

FROM branch;

SELECT \*

FROM branch\_supplier;

-- branch\_id in branch\_supplier table is part of the primary key so can't set it to NULL as primary key can't have NULL value

Triggers:

-- Defines a certain action to happen when something gets performed on the database

CREATE TABLE trigger\_test (

    message VARCHAR(100)

);

-- Have to define triggers in the command line as can't configure the DELIMITER in PopSQL

-- Do the following in command line client:

/\*

use giraffe

DELIMITER $$ -- usually the delimiter is a ;, we can change it to $$

-- BUT because ; is used to end the INSERT INTO, we can't use it to end the trigger

CREATE

    TRIGGER my\_trigger BEFORE INSERT -- before something gets inserted

    ON employee -- into the employee table

    FOR EACH ROW BEGIN

        INSERT INTO trigger\_test VALUES('added new employee'); -- insert this phrase into the trigger table

    END$$

DELIMITER ; -- change delimiter back at end

\*/

-- test by adding another employee

SELECT \*

FROM trigger\_test; -- no entries

INSERT INTO employee

VALUES(109, 'Oscar', 'Martinez', '1968-02-19', 'M', 69000, 106, 3);

SELECT \*

FROM trigger\_test; -- now see the message in this table

-- make another trigger in command line client – give it a different name

/\*

DELIMITER $$ -- usually the delimiter is a ;, we can change it to $$

-- BUT because ; is used to end the INSERT INTO, we can't use it to end the trigger

CREATE

    TRIGGER my\_trigger2 BEFORE INSERT -- before something gets inserted

    ON employee -- into the employee table

    FOR EACH ROW BEGIN

        INSERT INTO trigger\_test VALUES(NEW.first\_name); -- get the name of the new employee inserted

    END$$

DELIMITER ; -- change delimiter back at end

\*/

INSERT INTO employee

VALUES(110, 'Kevin', 'Malone', '1978-02-19', 'M', 69000, 106, 3);

SELECT \*

FROM trigger\_test; -- now see 2 extra messages in this table

-- make another trigger that uses conditionals

/\*

DELIMITER $$ -- usually the delimiter is a ;, we can change it to $$

-- BUT because ; is used to end the INSERT INTO, we can't use it to end the trigger

CREATE

    TRIGGER my\_trigger3 BEFORE INSERT -- before something gets inserted

    ON employee -- into the employee table

    FOR EACH ROW BEGIN

        IF NEW.sex = 'M' THEN

            INSERT INTO trigger.test VALUES('added male employee');

        ELSEIF NEW.sex = 'F' THEN

            INSERT INTO trigger\_test VALUES('added female employee');

        ELSE

            INSERT INTO trigger\_test VALUES('added other employee');

        END IF;

    END$$

DELIMITER ; -- change delimiter back at end

\*/

INSERT INTO employee

VALUES(111, 'Pam', 'Beesly', '1988-02-19', 'F', 69000, 106, 3);

SELECT \*

FROM trigger\_test; -- now see 2 extra messages in this table

-- Could also do AFTER rather than BEFORE

-- Or could do BEFORE/AFTER DELETE rather than BEFORE/AFTER INSERT

-- Can drop triggers in terminal using

-- DROP TRIGGER my\_trigger;

ER diagrams:

* When designing database, one of the most important things is defining the database schema
* Schema = all different tables and their attributes.
* Use ER diagram (entity relationship) to take your requirements and convert them into a database schema
* Entity = an object we want to model and store information about e.g. student (square)
* Attribute = specific pieces of info about an entity e.g. name, grade, gpa (oval)
* Primary key = attribute that uniquely identifies an entry e.g. student\_id (underlined)
* Composite attributes – an attribute that can be broken up into sub-attributes
  + E.g. name can be made up of first name and last name
* Multi-valued – attribute = an attribute that can have more than 1 value (double ring / 2 circles)
  + E.g. clubs
* Derived attribute = an attribute that can be derived from the other attributes (dashed circle)
  + E.g. has\_honors – could be derived from gpa – e.g. anyone who has gpa greater than certain value could have honors
  + E.g. age from birth\_date
* Can have another entity e.g. Class
  + Could have primary key class\_id
* Want to define relationships between the entities (diamond)
  + E.g. could denote that a student takes a class i.e. the student is related to the class
  + Can define participation
    - Student connected to the relationship using single line, meaning partial participation. Not all students need to take a class.
    - Class connected to the relationship using double line, meaning total participation. All the classes need to be taken by at least a single student. All classes must participate in the relationship.
  + Relationship attribute – an attribute about the relationship
    - E.g. grade as an attribute for the relationship between a student and a class. Student takes the class and gets a grade for it
  + Relationship cardinality – the number of instances of an entity from a relation that can be associated with the relation
    - E.g. student can take N number of classes
    - E.g. class can be taken by M number of students
    - N:M cardinality relationship
    - Could also be 1:1 – student can take 1 class, and class can be taken by 1 student
    - Or 1:N, or N:1 etc.
* Weak entity – an entity that cannot be uniquely identified by its attributes alone
  + Double square
  + E.g. exam
  + Relation between class and exam – class can have an exam (double diamond)
  + Exam can have exam\_id attribute
  + BUT exam cannot exist on its own without a class
  + Weak entity has to have total participation in relation (double line) – all exams must have a class

Diagram

Description automatically generated

Convert ER diagram to a database schema:

* For each regular entity type, create a table – the columns of this table will be the attributes we defined for that entity
* For weak entity type, create a table with all its attributes. The primary key of this will be the partial key of the weak identity plus the primary key of its owner
* Mapping of binary relationships – use foreign keys
  + Which table you put the foreign key in is based on the type of relation e.g. 1:1, 1:N, N:M.