Transactions
Stored Procedures
Triggers

Thomas Schwarz, SJ

Transactions

Transactions

- Databases have to process many operations in parallel
- This means some support for inter-process communication
  - Usually provided by locking
- DBMS differ in what they provide
  - Serializability:
    - All transactions appear to have been executed one after the other

ACID

- Atomicity: transactions is treated as a single, indivisible unit of work
- Consistency: If the DB is consistent before, it is consistent afterwards
- Isolation: Concurrent transactions do not interfere with each others
- Durability: Once a transaction is committed, its effects are permanent.

Transactions

- Atomicity
  - A single query is never interrupted:
    - Example:
      - A transfer of money from one account to another is executed completely or not at all
      - Both accounts have changed or none

Transactions

- Transaction
  - A group of SQL statements that are all processed in the order given or not at all
- SQL:
  - START TRANSACTION
  - either
  - COMMIT
  - ROLLBACK

Transactions

- Read only transactions
  - By declaring a transaction as read-only, SQL can usually perform it quicker
  - SET TRANSACTION READ ONLY;
  - SET TRANSACTION READ WRITE;

Transactions

- Dirty Reads:
    - Reading a record from an update that will be rolled-back
- Are dirty reads bad?
    - Depends
        - Sometimes, it does not matter, and we do not want the DBMS spend time on making sure that there are no dirty reads
        - Sometimes, a dirty read can absolutely mess up things
            - Selling the same commodity to two customers, …

Transactions

- SQL Isolation Levels:
  - Allow dirty reads:
    - SET TRANSACTION READ WRITE
    - SET ISOLATION LEVEL READ UNCOMMITTED

Transactions

- SQL Isolation Levels:
  - Allow reads only of committed data:
    - SET TRANSACTION READ WRITE
    - SET ISOLATION LEVEL READ COMMITTED

Transactions

- SQL Isolation Levels:
  - Disallow dirty reads, but insure that the reads are consistent:
    - SET TRANSACTION READ WRITE
    - SET ISOLATION LEVEL READ REPEATABLE READ

Transactions

- SQL Isolation Levels:
  - Serializability (default):
    - SET TRANSACTION READ WRITE
    - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

MySQL Transactions

- MySQL automatically wraps SQL statements into transactions
  - Control behavior with SET autocommit = OFF;
- Explicit transactions
    - START TRANSACTION
        - aliases: BEGIN, BEGIN WORK
    - COMMIT
    - ROLLBACK

MySQL Transactions

- Example:

```
CREATE DATABASE banks;

USE banks;

DROP TABLE IF EXISTS users;

CREATE TABLE users (
   id INT PRIMARY KEY AUTO_INCREMENT,
    name CHAR(30) NOT NULL,
    email CHAR(30)
    );

START TRANSACTION;
INSERT INTO users(name)
VALUES ('Thomas Schwarz');

UPDATE users
SET email = 'thomas.schwarz@marquette.edu';
```

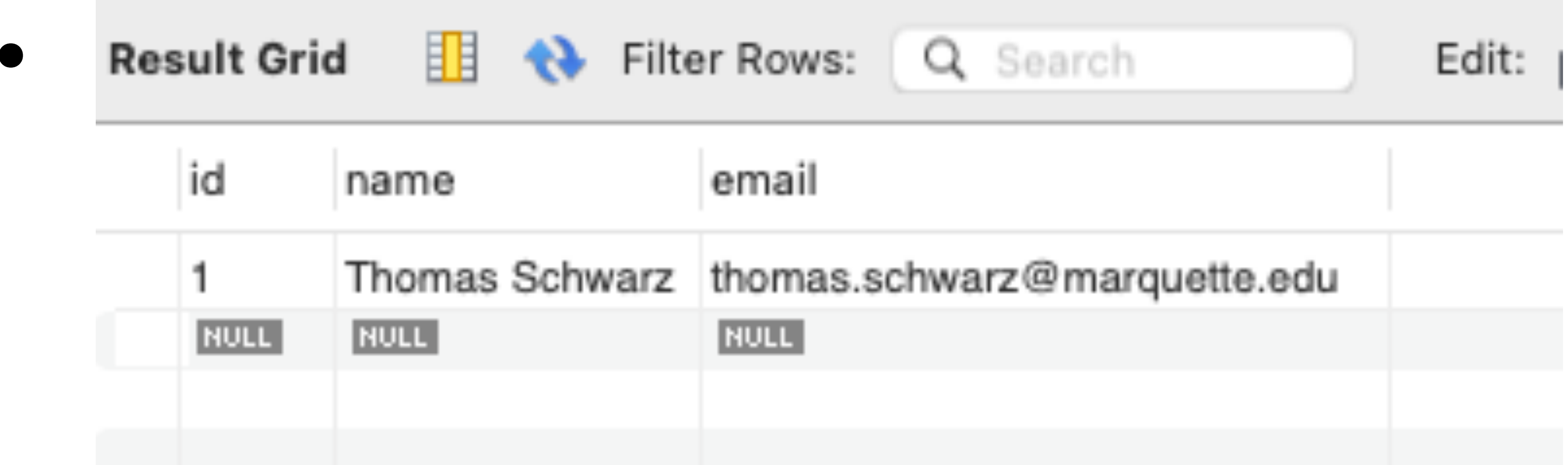
MySQL Transactions

- Open up a new window:

```
mysql> USE banks;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM users;
Empty set (0.00 sec)

mysql> 
```

- Transaction not committed, cannot see anything

MySQL Transactions

- In the first session:
  - `SELECT * FROM users;`
  - Shows the row:
  - 

id	name	email
1	Thomas Schwarz	thomas.schwarz@marquette.edu
NULL	NULL	NULL

MySQL Transactions

- In the first session:
  - `COMMIT;`
- Go back to the second session:

```
mysql> SELECT * FROM users;
Empty set (0.00 sec)

mysql> SELECT * FROM users;
+----+----------------+-------------------------------+
| id | name           | email                         |
+----+----------------+-------------------------------+
|  1 | Thomas Schwarz | thomas.schwarz@marquette.edu |
+----+----------------+-------------------------------+
1 row in set (0.00 sec)
```

MySQL Transactions

- If we had done a
  - `ROLLBACK;`
- the table would be empty

Stored Procedures

Thomas Schwarz, SJ

Stored Routines

- Stored Routine:
    - SQL statement or set of SQL statements that can be stored in the database server
        - Can be a function
        - Can be a stored procedure

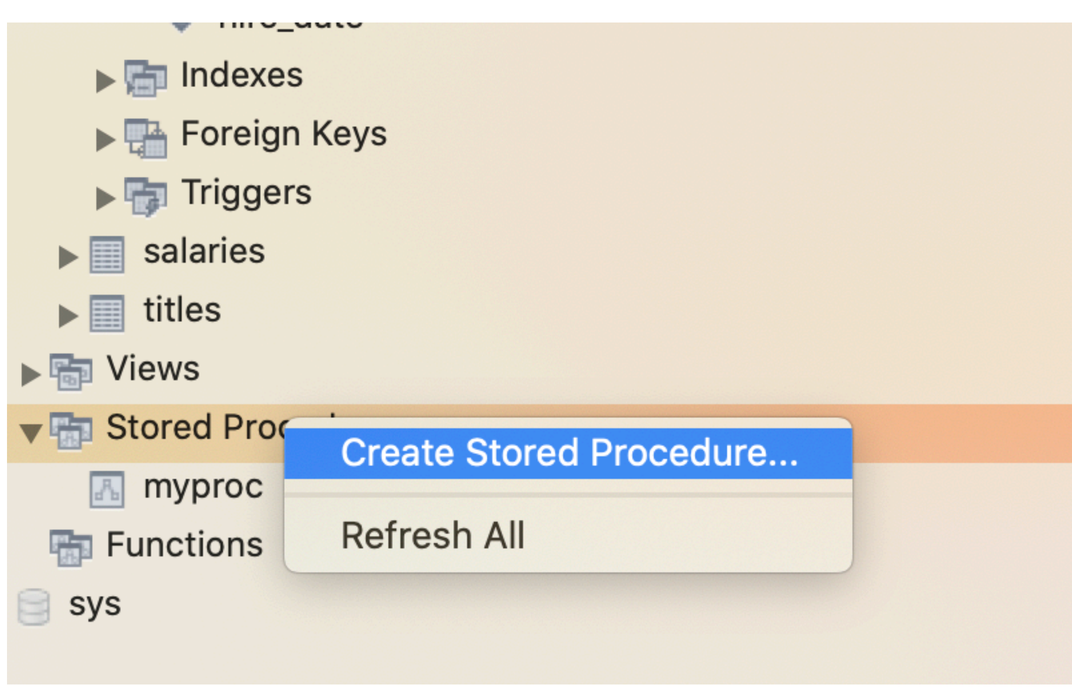
MySQL Stored Procedures

- Delimiters
  - Semicolon acts as a delimiter in SQL and Procedures
  - Need to change delimiter
    - Can be set to anything
      - E.g. double dollar sign
    - Afterwards reset it

```
DELIMITER $$

…

DELIMITER ;
```

MySQL Stored Procedures

- We can generate stored procedures from within MySQL workbench
  - Click on Stored Procedures at the end of the schema and select create stored procedure



MySQL Stored Procedures

- We can generate stored procedure using SQL

```
use employees;

DELIMITER $$

CREATE PROCEDURE myproc()
   BEGIN
      SELECT *
         FROM employees
         WHERE first_name LIKE 'th%';
   END
$$

DELIMITER ;
```

Keywords are CREATE PROCEDURE

MySQL Stored Procedures

- We can generate stored procedure using SQL

```
use employees;

DELIMITER $$

CREATE PROCEDURE myproc()
   BEGIN
      SELECT *
         FROM employees
         WHERE first_name LIKE 'th%';
   END
$$

DELIMITER ;
```

Need to give it a name

MySQL Stored Procedures

- We can generate stored procedure using SQL

```
use employees;

DELIMITER $$

CREATE PROCEDURE myproc()
   BEGIN
      SELECT *
         FROM employees
         WHERE first_name LIKE 'th%';
   END
$$

DELIMITER ;
```

Can have arguments

MySQL Stored Procedures

- We can generate stored procedure using SQL

```
use employees;

DELIMITER $$

CREATE PROCEDURE myproc()
   BEGIN
      SELECT *
         FROM employees
         WHERE first_name LIKE 'th%';
   END
$$

DELIMITER ;
```

BEGIN END encapsulate a SQL query terminated with ;

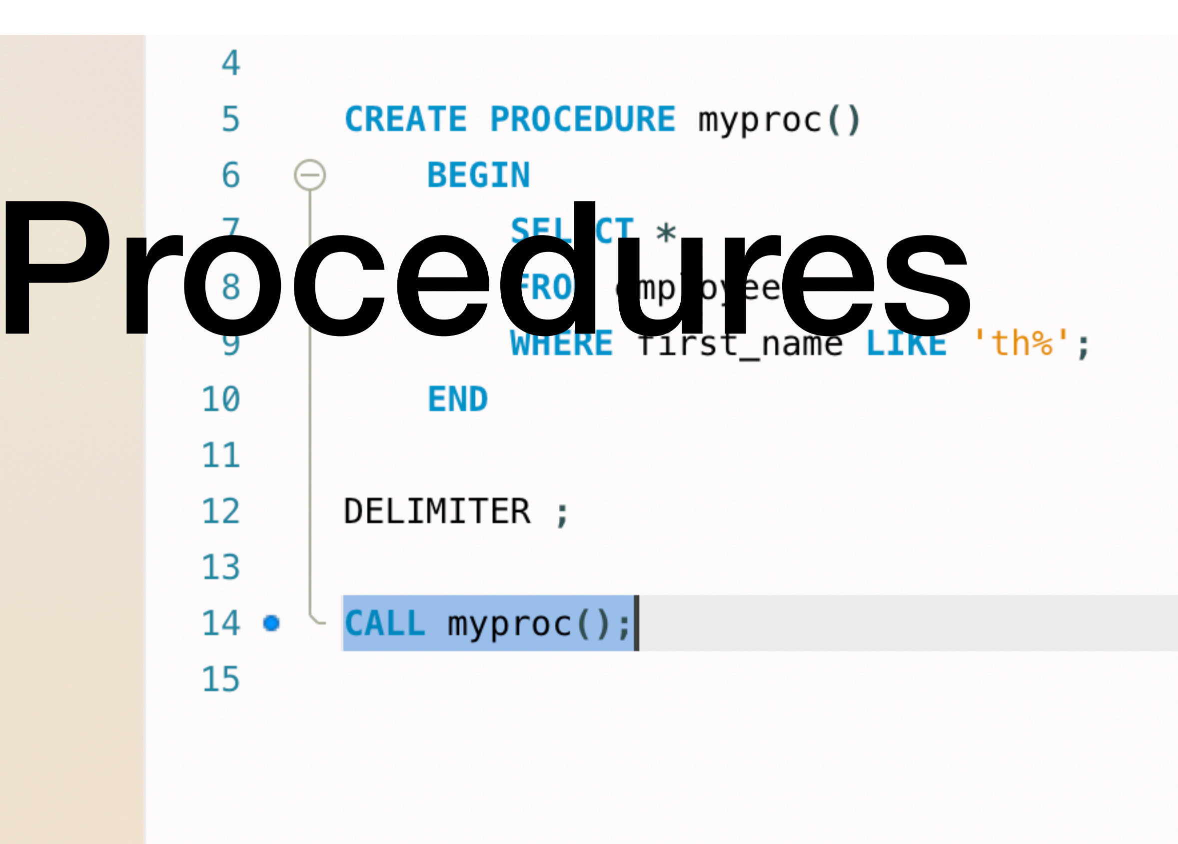
MySQL Stored Procedures

- We can generate stored procedure using SQL

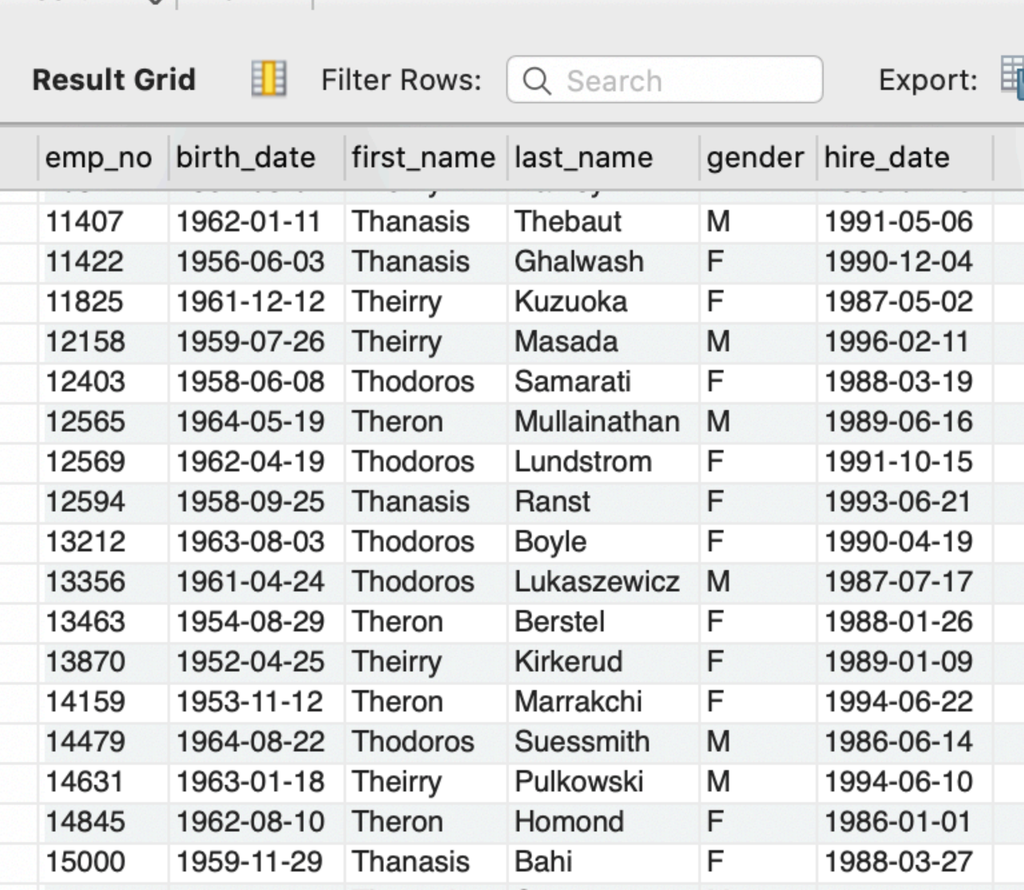
```
use employees;

DELIMITER $$

CREATE PROCEDURE myproc()
   BEGIN
      SELECT *
         FROM employees
         WHERE first_name LIKE 'th%';
   END
$$

DELIMITER ;
```

Don't forget to change the delimiter back to ;

MySQL Stored Procedures



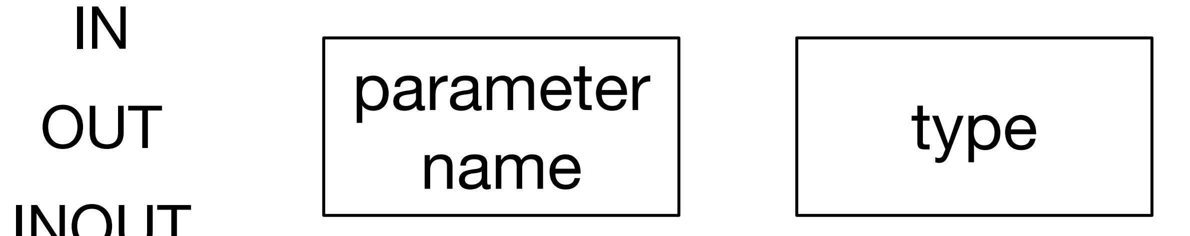
- We can call a stored procedure
  - From within the workbench, by clicking on it
  - Using CALL procedureName( )



emp_no	birth_date	first_name	last_name	gender	hire_date
11407	1962-01-11	Thanasis	Thebaut	M	1991-05-06
11422	1956-06-03	Thanasis	Ghalwash	F	1990-12-04
11825	1961-12-12	Theirry	Kuzuoka	F	1987-05-02
12158	1959-07-26	Theirry	Masada	M	1996-02-11
12403	1958-06-08	Thodoros	Samarati	F	1988-03-19
12565	1964-05-19	Theron	Mullainathan	M	1989-06-16
12569	1962-04-19	Thodoros	Lundstrom	F	1991-10-15
12594	1958-09-25	Thanasis	Ranst	F	1993-06-21
13212	1963-08-03	Thodoros	Boyle	F	1990-04-19
13356	1961-04-24	Thodoros	Lukaszewicz	M	1987-07-17
13463	1954-08-29	Theron	Berstel	F	1988-01-26
13870	1952-04-25	Theirry	Kirkerud	F	1989-01-09
14159	1953-11-12	Theron	Marrakchi	F	1994-06-22
14479	1964-08-22	Thodoros	Suessmith	M	1986-06-14
14631	1963-01-18	Theirry	Pulkowski	M	1994-06-10
14845	1962-08-10	Theron	Homond	F	1986-01-01
15000	1959-11-29	Thanasis	Bahi	F	1988-03-27

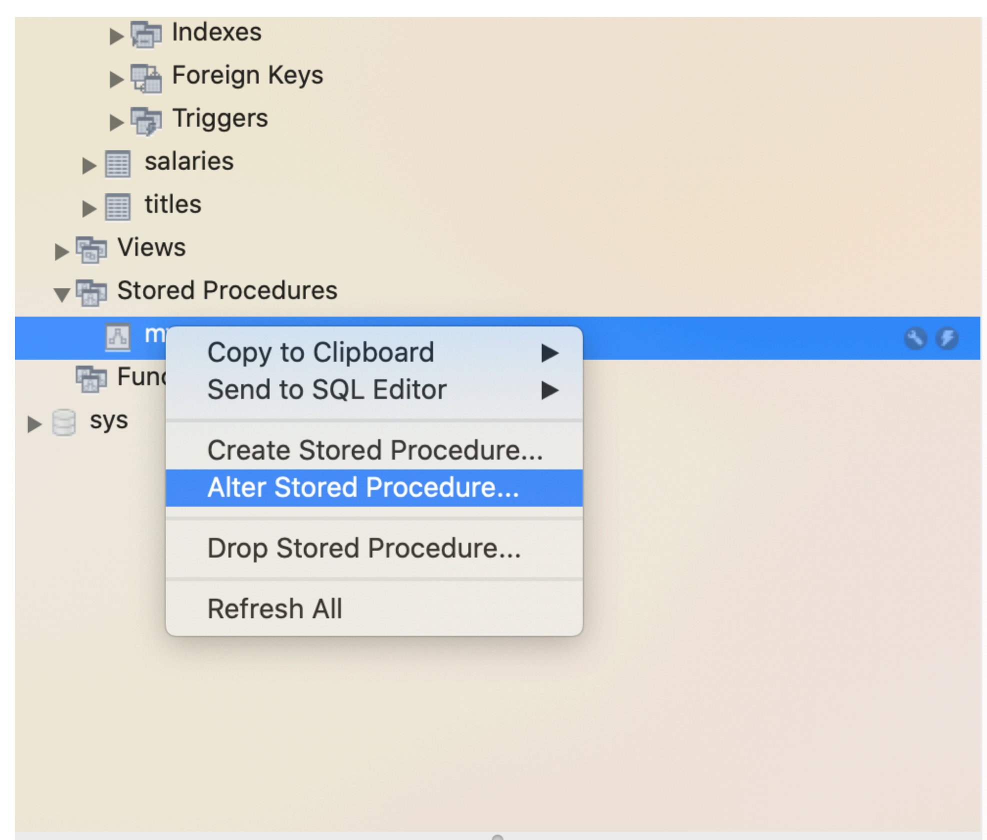
MySQL Stored Procedures

- Procedure parameters have three types
  - IN — input
  - OUT — output
  - INOUT — both input and output
- Procedure parameters have type
  - Definition of parameter:



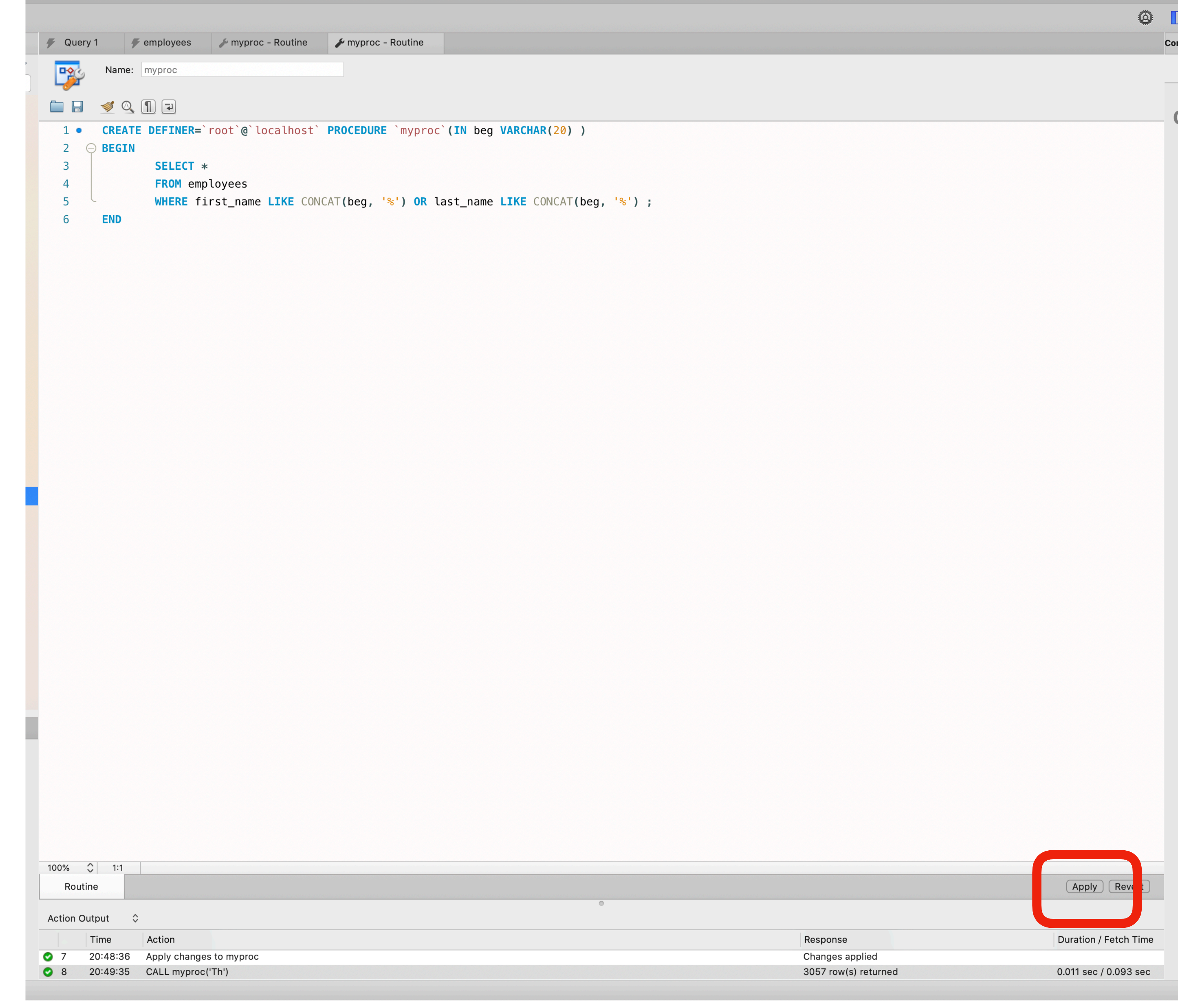
MySQL Stored Procedures

- Example:
    - Change a procedure
        - You can use workbench by selecting the name of the procedure and select 'alter procedure'



MySQL Stored Procedures

- Changing a procedure
  - After editing, click Apply



MySQL Stored Procedures

- Changing a procedure:
  - Combine with DROP and CREATE

MySQL Stored Procedures

Single Parameter

```
DELIMITER $$
CREATE PROCEDURE partial_name( IN beg VARCHAR(20) )
BEGIN
   SELECT first_name, last_name, gender
    FROM employees
    WHERE first_name LIKE CONCAT(beg, '%') OR last_name LIKE
CONCAT(beg, '%');
END

DELIMITER ;

CALL partial_name('dan');
```

MySQL Stored Procedures

```
DELIMITER $$
CREATE PROCEDURE partial_name( IN beg VARCHAR(20) )
BEGIN
   SELECT first_name, last_name, gender
    FROM employees
    WHERE first_name LIKE CONCAT(beg, '%') OR last_name LIKE
CONCAT(beg, '%');
END

DELIMITER ;

CALL partial_name('dan');
```

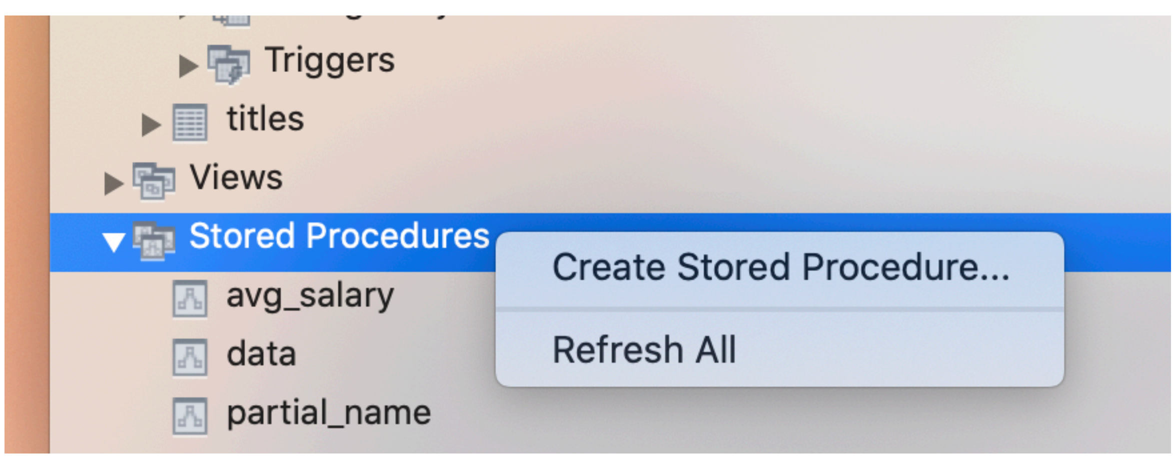
Have not yet discussed variables

MySQL Stored Procedures

- TASK
  - Write a stored procedure that takes as input the first name and the last name of a current employee.
  - It then returns:
    - The first and last name, gender, employee number, department, and last salary of the person (if it is in the database)

HINT

- If you are working with MySQL Workbench, it is *less frustrating* to define and make changes in the Stored Procedures tab on the left.
    - The delimiter statements does not work too well



MySQL Stored Procedures

```
DELIMITER $$

CREATE PROCEDURE data(IN first VARCHAR(14), IN last VARCHAR(16))
BEGIN
   SELECT e.first_name, e.last_name, e.gender, d.dept_name,
          s.salary
   FROM employees e, departments d, salaries s, dept_emp de
   WHERE e.emp_no = de.emp_no AND de.dept_no = d.dept_no
       AND s.emp_no = e.emp_no AND s.to_date = '9999-01-01'
       AND e.first_name = first AND e.last_name = last;
END

DELIMITER ;
```

MySQL Stored Procedures

- TASK
  - Write a stored procedure that takes as input the first name and the last name of a current or past employee.
  - It then returns:
    - The first and last name, and average salary of the person

MySQL Stored Procedures

```
CREATE PROCEDURE `avg_salary`(
            IN first VARCHAR(12),
            last VARCHAR(16)
       )
BEGIN
    SELECT e.first_name, e.last_name, AVG(s.salary)
    FROM employees e, salaries s
    WHERE e.first_name = first
        AND e.last_name = last
        AND e.emp_no = s.emp_no;
END
```

MySQL Stored Procedures

- Using output variables
  - Let's change the previous procedure to return the average salary
  - We need to use the SELECT … INTO … construct

MySQL Stored Procedures

```
CREATE PROCEDURE avg_salary(
      IN first VARCHAR(12),
      last VARCHAR(16),
      out average_salary DECIMAL(10,2)
)
BEGIN
    SELECT AVG(s.salary)
    INTO average_salary
    FROM employees e, salaries s
    WHERE e.first_name = first
        AND e.last_name = last
        AND e.emp_no = s.emp_no;
END
```

This is our output

MySQL Stored Procedures

```
CREATE PROCEDURE avg_salary(
      IN first VARCHAR(12),
      last VARCHAR(16),
      out average_salary DECIMAL(10,2)
)
BEGIN
    SELECT AVG(s.salary)
     INTO average_salary
     FROM employees e, salaries s
     WHERE e.first_name = first
         AND e.last_name = last
         AND e.emp_no = s.emp_no;
END
```

Type is Decimal with two digits after comma

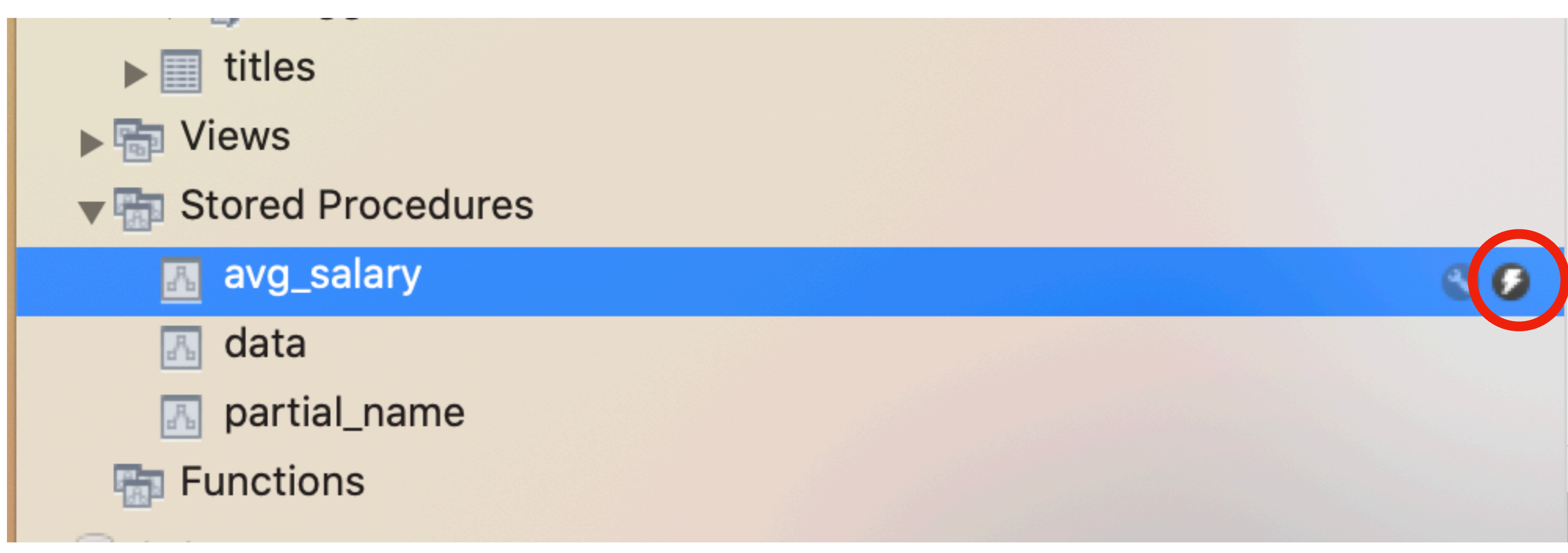
MySQL Stored Procedures

```
CREATE PROCEDURE avg_salary(
      IN first VARCHAR(12),
      last VARCHAR(16),
      out average_salary DECIMAL(10,2)
)
BEGIN
    SELECT AVG(s.salary)
    INTO average_salary
    FROM employees e, salaries s
    WHERE e.first_name = first
        AND e.last_name = last
        AND e.emp_no = s.emp_no;
END
```

This is how we set the value

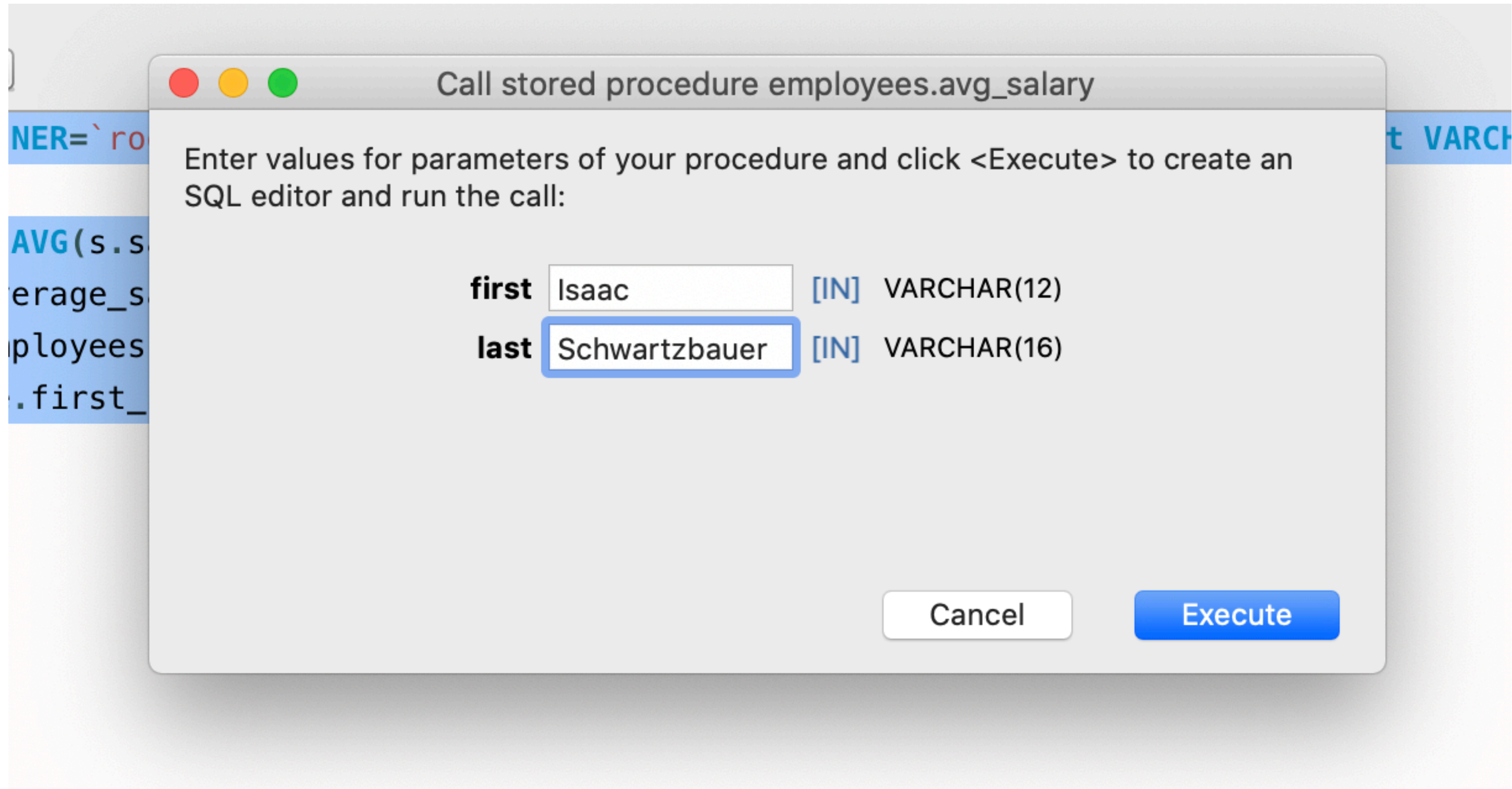
MySQL Stored Procedures

- How do we call this?
  - Easiest is using the lightning symbol in the MySQL work-bench



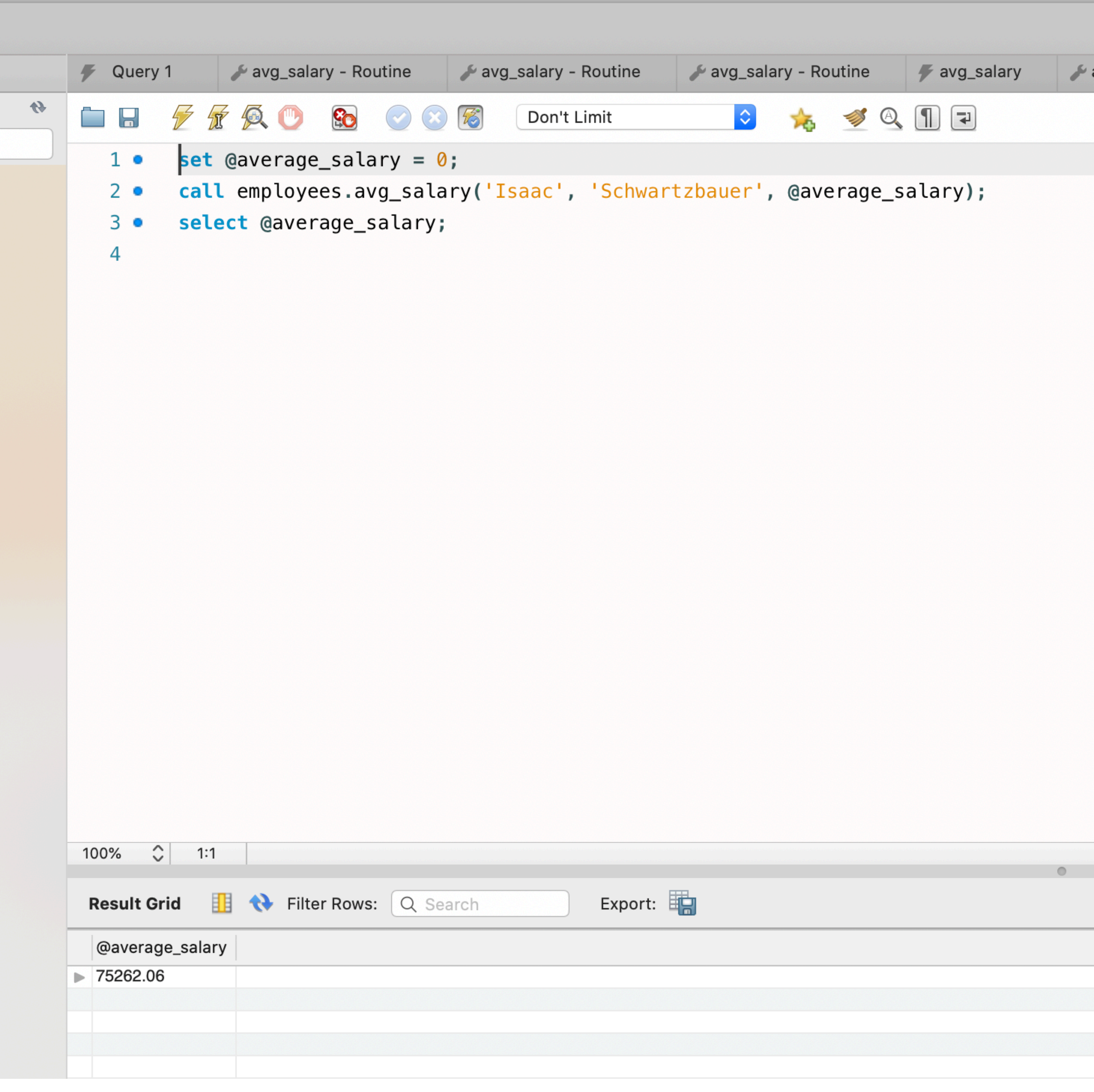
MySQL Stored Procedures

- This gives you an interactive window



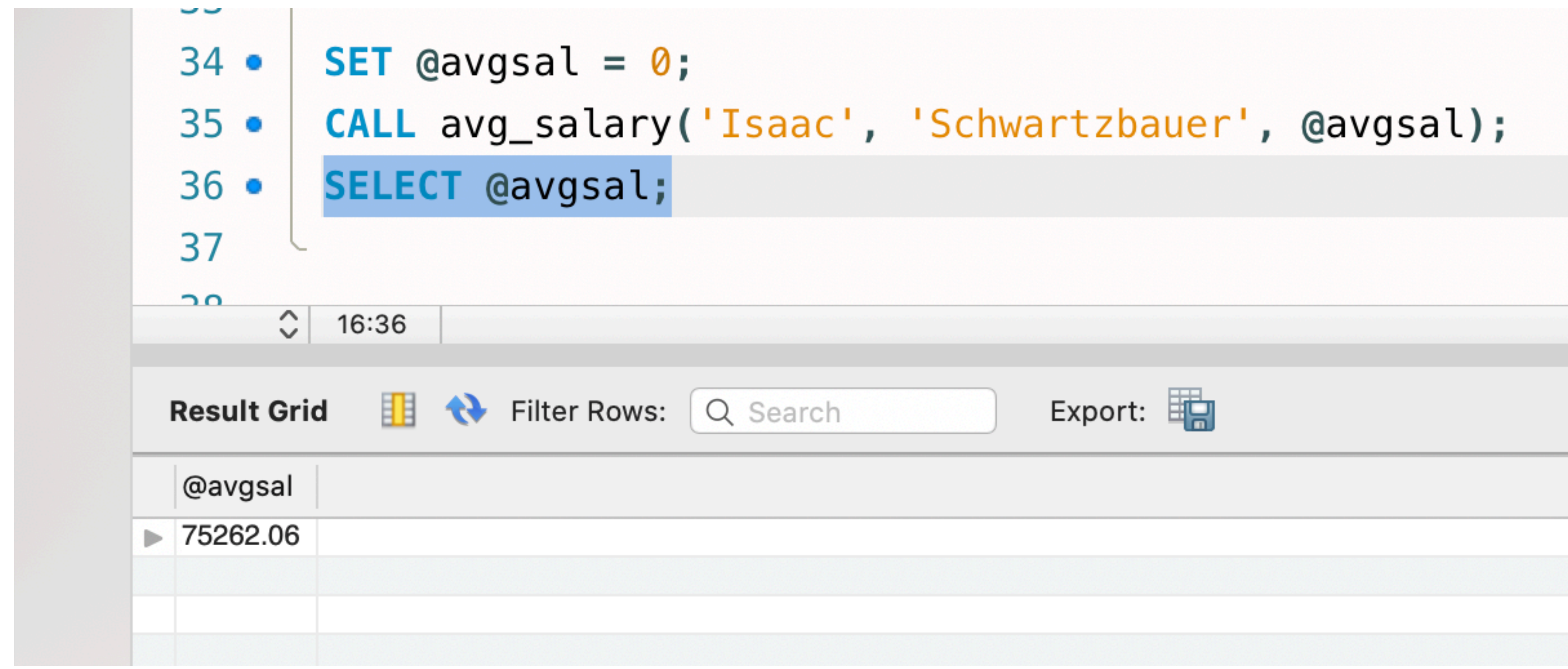
MySQL Stored Procedures

- Result grid shows the value
  - $75262.06
- But you can also see how to call the procedure as well.
-



MySQL Stored Procedures

- First, we need to define a variable



MySQL Stored Procedures

- Defining variables
  - Variables start with an ampersand @myvar
    - You can enclose the name with '   ' or "   " to use other characters than alpha-numeric and '$'
  - Initialized with SET
    - Assignment is with integer, decimal, floating-point, binary or non-binary string, or NULL
    - Can use CAST if necessary

MySQL Stored Procedures

- We call the function with

```
SET @avgsal = 0;
CALL avg_salary('Isaac', 'Schwartzbauer', @avgsal);
SELECT @avgsal;
```

- Notice how
  - we include the output variable in the call
  - we use SELECT to access the value of the variable

MySQL Stored Procedures

- TASK
  - Write a stored procedure that takes as input the first name and the last name of a current or past employee.
  - It then returns:
    - The employee number
  - Call it from the query tab

MySQL Stored Procedures

- Solution

```
CREATE DEFINER=`root`@`localhost` PROCEDURE
`employee_number`(
   IN first VARCHAR(12),
   IN last VARCHAR(16),
   OUT empl_numb INT
  )
BEGIN
   SELECT emp_no
     INTO empl_numb
     FROM employees
     WHERE first_name = first
        AND last_name = last;
END
```

MySQL Stored Procedures

- Solution

```
set @empl_numb = 0;
call employees.employee_number(
    'Isaac',
    'Schwartzbauer',
    @empl_numb
);
select @empl_numb;
```

MySQL Functions

- A procedure can have more than one output
- Functions have none or one output
- MySQL functions return exactly one value

MySQL Functions

- We can use the MySQL workbench by clicking on Functions in the scheme pane
  - Just below stored procedures
  - Or can use a query
  - Function values are obtained through a SELECT statement
    - `select employees.f_avg(101010);`

MySQL Functions

- Example:

Definition of a function

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
   DECLARE v_avg_salary DECIMAL(10,2);
   SELECT AVG(salary)
      INTO v_avg_salary
      FROM salaries s
      WHERE s.emp_no = p_emp_no;
   RETURN v_avg_salary;
END
```

MySQL Functions

- Example:

Name

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
   DECLARE v_avg_salary DECIMAL(10,2);
   SELECT AVG(salary)
      INTO v_avg_salary
      FROM salaries s
      WHERE s.emp_no = p_emp_no;
   RETURN v_avg_salary;
END
```

MySQL Functions

Name and type of input variable

- Example:

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
   DECLARE v_avg_salary DECIMAL(10,2);
   SELECT AVG(salary)
      INTO v_avg_salary
      FROM salaries s
      WHERE s.emp_no = p_emp_no;
   RETURN v_avg_salary;
END
```

MySQL Functions

- Example:

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
    DECLARE v_avg_salary DECIMAL(10,2);
    SELECT AVG(salary)
        INTO v_avg_salary
        FROM salaries s
        WHERE s.emp_no = p_emp_no;
    RETURN v_avg_salary;
END
```

Keyword Returns is needed plus specification of return value type

MySQL Functions

Describes behavior of the function

- Example:

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
    DECLARE v_avg_salary DECIMAL(10,2);
    SELECT AVG(salary)
        INTO v_avg_salary
        FROM salaries s
        WHERE s.emp_no = p_emp_no;
    RETURN v_avg_salary;
END
```

MySQL Functions

- DETERMINISTIC:
  - always produces the same result for the same parameter
- NO SQL:
  - function has no SQL statements
- READS SQL DATA
  - contains statements that read via SQL, but does not modify the database
- MODIFIES SQL DATA
  - contains statements that write (e.g. inserts)

MySQL Functions

- Example:

Body of the function

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
   DECLARE v_avg_salary DECIMAL(10,2);
   SELECT AVG(salary)
      INTO v_avg_salary
      FROM salaries s
      WHERE s.emp_no = p_emp_no;
   RETURN v_avg_salary;
END
```

MySQL Functions

- Example:

Here we define a variable to be used in several statements

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
   DECLARE v_avg_salary DECIMAL(10,2);
   SELECT AVG(salary)
      INTO v_avg_salary
      FROM salaries s
      WHERE s.emp_no = p_emp_no;
   RETURN v_avg_salary;
END
```

MySQL Functions

- Example:

The variable has type Decimal(10,2) to represent currency values

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
   DECLARE v_avg_salary DECIMAL(10,2);
   SELECT AVG(salary)
      INTO v_avg_salary
      FROM salaries s
      WHERE s.emp_no = p_emp_no;
   RETURN v_avg_salary;
END
```

MySQL Functions

- Example:

The SELECT … INTO … clause updates the value of v_avg_salary

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
    DECLARE v_avg_salary DECIMAL(10,2);
    SELECT AVG(salary)
        INTO v_avg_salary
        FROM salaries s
        WHERE s.emp_no = p_emp_no;
    RETURN v_avg_salary;
END
```

MySQL Functions

- Example:

A function has to return a value

```
CREATE FUNCTION f_avg (p_emp_no INTEGER)
RETURNS decimal(10,2)
READS SQL DATA
BEGIN
   DECLARE v_avg_salary DECIMAL(10,2);
   SELECT AVG(salary)
      INTO v_avg_salary
      FROM salaries s
      WHERE s.emp_no = p_emp_no;
   RETURN v_avg_salary;
END
```

MySQL Functions

- Functions and procedures can have more than one select value
  - Find the last salary of an employee given by first and last name
    - Query 1: Find the last from_date in salaries for the employee
      - We store it in a variable
    - Query 2: Return the corresponding salary

```
CREATE FUNCTION f_latest_salary(p_first_name VARCHAR(12), p_last_name VARCHAR(16))
    RETURNS decimal(10,2)
    READS SQL DATA
BEGIN
    DECLARE v_max_from_date  date;
    DECLARE v_last_salary DECIMAL(10,2);

    SELECT
        MAX(from_date)
    INTO v_max_from_date
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name;

    SELECT
        s.salary
    INTO v_last_salary
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name
        AND s.from_date = v_max_from_date;

RETURN v_last_salary;
END
```

```
CREATE FUNCTION f_latest_salary(p_first_name VARCHAR(12), p_last_name VARCHAR(16))
    RETURNS decimal(10,2)
    READS SQL DATA
BEGIN
    DECLARE v_max_from_date  date;
    DECLARE v_last_salary DECIMAL(10,2);

    SELECT
        MAX(from_date)
    INTO v_max_from_date
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name;

    SELECT
        s.salary
    INTO v_last_salary
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name
        AND s.from_date = v_max_from_date;

RETURN v_last_salary;
END
```

Declare a variable of type date to contain the last contract to-date

```
CREATE FUNCTION f_latest_salary(p_first_name VARCHAR(12), p_last_name VARCHAR(16))
    RETURNS decimal(10,2)
    READS SQL DATA
BEGIN
    DECLARE v_max_from_date  date;
    DECLARE v_last_salary DECIMAL(10,2);

    SELECT MAX(from_date)
    INTO v_max_from_date
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name;

    SELECT
        s.salary
    INTO v_last_salary
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name
        AND s.from_date = v_max_from_date;

RETURN v_last_salary;
END
```

Declare a variable of type Decimal for the salary

```
CREATE FUNCTION f_latest_salary(p_first_name VARCHAR(12), p_last_name VARCHAR(16))
    RETURNS decimal(10,2)
    READS SQL DATA
BEGIN
    DECLARE v_max_from_date  date;
    DECLARE v_last_salary DECIMAL(10,2);

    SELECT MAX(from_date)
    INTO v_max_from_date
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name;

    SELECT
        s.salary
    INTO v_last_salary
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name
        AND s.from_date = v_max_from_date;

RETURN v_last_salary;
END
```

Determine the last contract from_date for an employee with given first and last name

```
CREATE FUNCTION f_latest_salary(p_first_name VARCHAR(12), p_last_name VARCHAR(16))
    RETURNS decimal(10,2)
    READS SQL DATA
BEGIN
    DECLARE v_max_from_date  date;
    DECLARE v_last_salary DECIMAL(10,2);

    SELECT MAX(from_date)
    INTO v_max_from_date
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name;

    SELECT
        s.salary
    INTO v_last_salary
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name
        AND s.from_date = v_max_from_date;

RETURN v_last_salary;
END
```

and load it into the
variable
v_max_from_date

```
CREATE FUNCTION f_latest_salary(p_first_name VARCHAR(12), p_last_name VARCHAR(16))
    RETURNS decimal(10,2)
    READS SQL DATA
BEGIN
    DECLARE v_max_from_date  date;
    DECLARE v_last_salary DECIMAL(10,2);

    SELECT MAX(from_date)
    INTO v_max_from_date
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name;

    SELECT
        s.salary
    INTO v_last_salary
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name
        AND s.from_date = v_max_from_date;

RETURN v_last_salary;
END
```

Second query:

Find the corresponding amount

```
CREATE FUNCTION f_latest_salary(p_first_name VARCHAR(12), p_last_name VARCHAR(16))
    RETURNS decimal(10,2)
    READS SQL DATA
BEGIN
    DECLARE v_max_from_date  date;
    DECLARE v_last_salary DECIMAL(10,2);

    SELECT MAX(from_date)
    INTO v_max_from_date
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name;

    SELECT
        s.salary
    INTO v_last_salary
    FROM employees e
    JOIN salaries s ON e.emp_no = s.emp_no
    WHERE e.first_name = p_first_name
        AND e.last_name = p_last_name
        AND s.from_date = v_max_from_date;

RETURN v_last_salary;
END
```

And finally return this value

MySQL Functions

- Why do we need functions?
  - Procedures can not be embedded in a select statement
  - But functions can

MySQL Variables

- MySQL has three types of variables
  - Local: Scope is in a BEGIN … END block
  - Use DECLARE and no ampersand
- Session
  - Starts and ends with making a connection to the database
  - One session per current user
  - Use SET @variable = NULL
- Global

MySQL Variables

- Global variables
  - Survives disconnection
  - Needs to be system variables
    - E.g. .max_connections( ), .max_join_size( )
    - `SET GLOBAL max_connections = 1000;`
    - `SET @@global.max_connections = 1000;`

MySQL Variables

- Try it out
  - Set max_connections to 1
  - Then try another connection in MySQL workbench

Conditions

- MySQL has an IF statement
- MySQL has a CASE statement
- My SQL has a CASE expression
  - which is the easiest

```
CASE  attribute
   WHEN …  THEN …
   ELSE …
END
```

CASE Statement

- Example
  - Expanding gender

```
SELECT first_name, last_name, CASE
      WHEN 'M' THEN 'male'
      ELSE 'female'
      END AS gender
FROM employees
WHERE emp_id = p_emp_id;
```

CASE Statement

- Example:
  - Select employee data, including whether they are managers or not
  - Information is in employees and in dept_manager table
  - One possibility: Use a LEFT JOIN
    - Also: limit queries to a range of emp_no chosen to have managers and no-managers in it

CASE Statement

```
SELECT e.emp_no,
       e.first_name,
       e.last_name,
       dm.emp_no
FROM employees e LEFT JOIN dept_manager dm
           ON e.emp_no = dm.emp_no
WHERE e.emp_no BETWEEN 109990 AND 111000;
```

CASE Statement

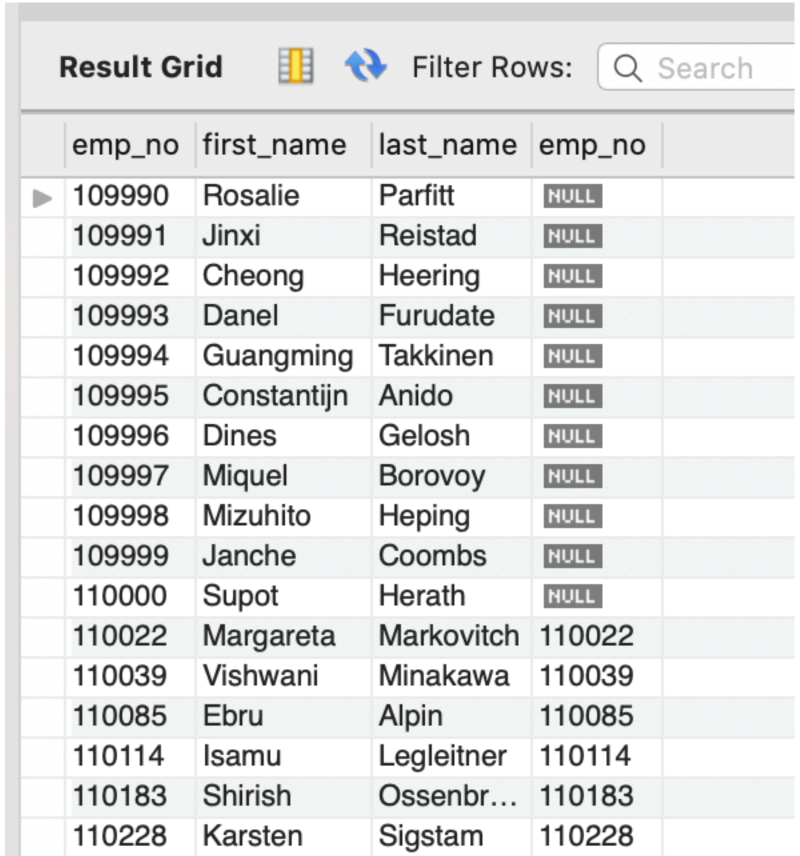
```
SELECT e.emp_no,
       e.first_name,
       e.last_name,
       dm.emp_no
FROM employees e LEFT                       ger dm
           ON e.emp_
WHERE e.emp_no BETWE                       .000;
```

Result Grid   Filter Rows: Search

emp_no	first_name	last_name	emp_no
109990	Rosalie	Parfitt	NULL
109991	Jinxi	Reistad	NULL
109992	Cheong	Heering	NULL
109993	Danel	Furudate	NULL
109994	Guangming	Takkinen	NULL
109995	Constantijn	Anido	NULL
109996	Dines	Gelosh	NULL
109997	Miquel	Borovoy	NULL
109998	Mizuhito	Heping	NULL
109999	Janche	Coombs	NULL
110000	Supot	Herath	NULL
110022	Margareta	Markovitch	110022
110039	Vishwani	Minakawa	110039
110085	Ebru	Alpin	110085
110114	Isamu	Legleitner	110114
110183	Shirish	Ossenbr…	110183
110228	Karsten	Sigstam	110228

CASE Statement

- Because this is a left join, we match the emp_no, but keep both of them
  - the second one is for dept_manager
- This means that we can use the case statement


Result Grid Filter Rows: Search

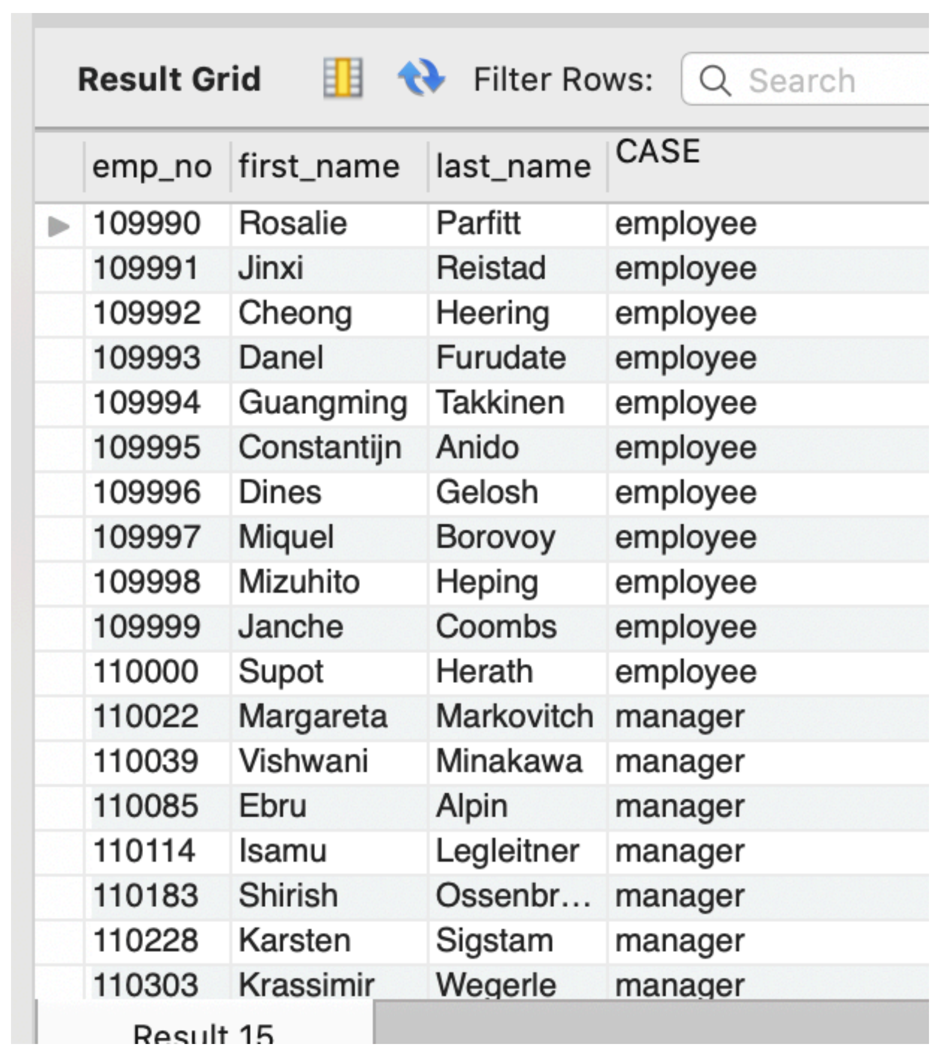
emp_no	first_name	last_name	emp_no
109990	Rosalie	Parfitt	NULL
109991	Jinxi	Reistad	NULL
109992	Cheong	Heering	NULL
109993	Danel	Furudate	NULL
109994	Guangming	Takkinen	NULL
109995	Constantijn	Anido	NULL
109996	Dines	Gelosh	NULL
109997	Miquel	Borovoy	NULL
109998	Mizuhito	Heping	NULL
109999	Janche	Coombs	NULL
110000	Supot	Herath	NULL
110022	Margareta	Markovitch	110022
110039	Vishwani	Minakawa	110039
110085	Ebru	Alpin	110085
110114	Isamu	Legleitner	110114
110183	Shirish	Ossenbr...	110183
110228	Karsten	Sigstam	110228

CASE Statement

```
SELECT e.emp_no,
       e.first_name,
       e.last_name,
       CASE
          WHEN dm.emp_no IS NOT NULL THEN 'manager'
          ELSE 'employee'
       END
FROM employees e LEFT JOIN dept_manager dm
                 ON e.emp_no = dm.emp_no
WHERE e.emp_no BETWEEN 109990 AND 111000;
```

CASE Statement

- The last column of the result is now the result of the CASE expression
- The column name is bad, so we change it with an AS clause



Result Grid | Filter Rows: Search

emp_no	first_name	last_name	CASE
109990	Rosalie	Parfitt	employee
109991	Jinxi	Reistad	employee
109992	Cheong	Heering	employee
109993	Danel	Furudate	employee
109994	Guangming	Takkinen	employee
109995	Constantijn	Anido	employee
109996	Dines	Gelosh	employee
109997	Miquel	Borovoy	employee
109998	Mizuhito	Heping	employee
109999	Janche	Coombs	employee
110000	Supot	Herath	employee
110022	Margareta	Markovitch	manager
110039	Vishwani	Minakawa	manager
110085	Ebru	Alpin	manager
110114	Isamu	Legleitner	manager
110183	Shirish	Ossenbr…	manager
110228	Karsten	Sigstam	manager
110303	Krassimir	Wegerle	manager

Result 15

CASE Statement

```
SELECT e.emp_no,
       e.first_name,
       e.last_name,
       CASE
          WHEN dm.emp_no IS NOT NULL THEN 'manager'
          ELSE 'employee'
     END AS 'role'
FROM employees e LEFT JOIN dept_manager dm
           ON e.emp_no = dm.emp_no
WHERE e.emp_no BETWEEN 109990 AND 111000;
```

CASE Statement

```
SELECT e.emp_no,
       e.first_name,
       e.last_name,
       CASE
          WHEN dm.emp_no IS N
          ELSE 'employee'
     END AS 'role'
FROM employees e LEFT JOIN de
          ON e.emp_no = dm.e
WHERE e.emp_no BETWEEN 10999
```

Result Grid | Filter Rows: Search | Export:

emp_no	first_name	last_name	role
109990	Rosalie	Parfitt	employee
109991	Jinxi	Reistad	employee
109992	Cheong	Heering	employee
109993	Danel	Furudate	employee
109994	Guangming	Takkinen	employee
109995	Constantijn	Anido	employee
109996	Dines	Gelosh	employee
109997	Miquel	Borovoy	employee
109998	Mizuhito	Heping	employee
109999	Janche	Coombs	employee
110000	Supot	Herath	employee
110022	Margareta	Markovitch	manager
110039	Vishwani	Minakawa	manager
110085	Ebru	Alpin	manager
110114	Isamu	Legleitner	manager
110183	Shirish	Ossenbrug…	manager
110228	Karsten	Sigstam	manager
110303	Krassimir	Wegerle	manager
110344	Rosine	Cools	manager
110386	Shem	Kieras	manager
110420	Oscar	Ghazalie	manager
110511	DeForest	Hagimont	manager
110567	Leon	DasSarma	manager
110725	Peternela	Onuegbe	manager
110765	Rutger	Hofmeyr	manager
110800	Sanjoy	Quadeer	manager
110854	Dung	Pesch	manager

CASE Statement

- **GOTCHA**
- NULL and NOT NULL cannot be compared
- So: this does NOT work

```
SELECT e.emp_no,
       e.first_name,
       e.last_name,
       CASE dm.emp_no
        WHEN  NOT NULL THEN 'manager'
            ELSE 'employee'
     END AS 'role'
FROM employees e LEFT JOIN dept_manager dm
       ON e.emp_no = dm.emp_no
WHERE e.emp_no BETWEEN 109990 AND 111000;
```

CASE Statement

- Result:
  - Everyone is an employee because NOT NULL comparison is never-ever true

emp_no	first_name	last_name	role
109990	Rosalie	Parfitt	employee
109991	Jinxi	Reistad	employee
109992	Cheong	Heering	employee
109993	Danel	Furudate	employee
109994	Guangming	Takkinen	employee
109995	Constantijn	Anido	employee
109996	Dines	Gelosh	employee
109997	Miquel	Borovoy	employee
109998	Mizuhito	Heping	employee
109999	Janche	Coombs	employee
110000	Supot	Herath	employee
110022	Margareta	Markovitch	employee
110039	Vishwani	Minakawa	employee
110085	Ebru	Alpin	employee
110114	Isamu	Legleitner	employee
110183	Shirish	Ossenbrug…	employee
110228	Karsten	Sigstam	employee
110303	Krassimir	Wegerle	employee
110344	Rosine	Cools	employee
110386	Shem	Kieras	employee
110420	Oscar	Ghazalie	employee
110511	DeForest	Hagimont	employee
110567	Leon	DasSarma	employee
110725	Peternela	Onuegbe	employee
110765	Rutger	Hofmeyr	employee

CASE Statement

- MySQL has an IF expression
  - Can have only one test
  - Syntax is IF(condition, valiftrue, valiffalse)

CASE Statement

- Inside a stored procedure, we can use the case statement to set a variable

```
CREATE PROCEDURE GetDeliveryStatus(
    IN pOrderNumber INT,
    OUT pDeliveryStatus VARCHAR(100)
)
BEGIN
    DECLARE waitingDay INT DEFAULT 0;
    SELECT
        DATEDIFF(requiredDate, shippedDate)
    INTO waitingDay
    FROM orders
    WHERE orderNumber = pOrderNumber;

    CASE
        WHEN waitingDay = 0 THEN
            SET pDeliveryStatus = 'On Time';
        WHEN waitingDay >= 1 AND waitingDay < 5 THEN
            SET pDeliveryStatus = 'Late';
        WHEN waitingDay >= 5 THEN
            SET pDeliveryStatus = 'Very Late';
        ELSE
            SET pDeliveryStatus = 'No Information';
    END CASE;
```

Transactions in Stored Procedures

- Go back to the banks example
- Create a table checking
  - ```
    DROP TABLE IF EXISTS checking;

    CREATE TABLE checking (
       id INT PRIMARY KEY,
        amount DECIMAL(28,2),
        CONSTRAINT account_holder_exists
        FOREIGN KEY (id) REFERENCES users(id)
        ON DELETE CASCADE
        );
    ```
    ```

Transactions in Stored Procedures

- Create checking accounts:

```
INSERT INTO checking(id, amount)
SELECT id, 1000 FROM users WHERE name = 'Thomas Schwarz';

SELECT * from checking;

INSERT INTO checking(id, amount)
SELECT id, 1000 FROM users WHERE name = 'Dennis Brylow';

INSERT INTO checking(id, amount)
SELECT id, 100 FROM users WHERE name = 'Donald Trump';
```

Transactions in Stored Procedures

```
DELIMITER //
CREATE PROCEDURE transfer(
    IN sender_id INT,
    IN receiver_id INT,
    IN amount DECIMAL(10,2)
)
BEGIN
   DECLARE rollback_message VARCHAR(255) DEFAULT
       'Transaction rolled back: Insufficient funds';
    DECLARE commit_message VARCHAR(255) DEFAULT
        'Transaction committed successfully';
```

Transactions in Stored Procedures

```
 START TRANSACTION;

    UPDATE checking
    SET checking.amount = checking.amount - my_amount
    WHERE id = sender_id;

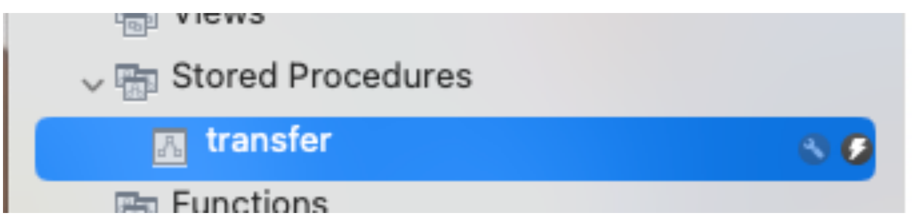
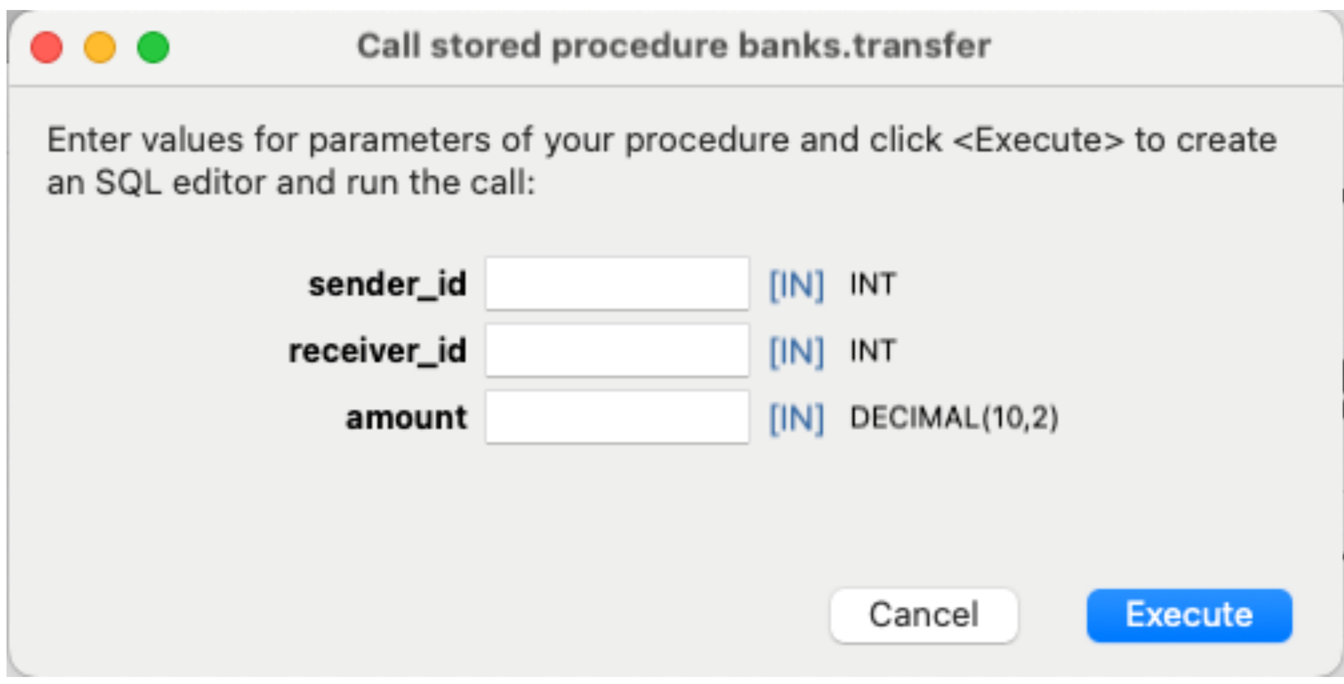
    UPDATE checking
    SET checking.amount = checking.amount + my_amount
    WHERE id = receiver_id;

IF (SELECT amount FROM checking WHERE id = sender_id) < 0 THEN
      ROLLBACK;
        SIGNAL SQLSTATE '45000'
             SET MESSAGE_TEXT = rollback_message;
   ELSE
      COMMIT;
        SELECT commit_message AS 'Result';
   END IF;
END //

DELIMITER ;
```

Transactions in Stored Procedures

```
DELIMITER ;

SELECT * FROM checking;

CALL transfer(1,2,500.00);

SELECT * FROM checking;
```

Transactions in Stored Procedures

- In the workbench, you can use the stored procedure flash to execute the procedure





Transactions in Stored Procedures

- Example:
  - `call banks.transfer(2, 1, 3000);`

69	13:41:26	call banks.transfer(2, 1, 3000)	Error Code: 1644. Transaction rolled back: Insufficient funds	0.00095 sec

Constraints and Triggers

Keys and Foreign Keys

- SQL Primary Key declaration
  - Equivalent to NOT NULL and UNIQUE
  - Creates an index, so lookup with key are faster
- SQL Foreign Key declaration
  - Insures that a value in a foreign table exists
  - That value must be declared UNIQUE

Keys and Foreign Keys

- Two declarations in SQL

```
CREATE TABLE studio(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC#  INT REFERENCES MovieExec(cert#)
);

CREATE TABLE studio(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC#  INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
);
```

Keys and Foreign Keys

- What happens if we try to insert into studio a president or change a presC# whose certificate number does <u>not</u> match a certificate number in movieExecs?
- What happens if we delete a row from movieExecs or update a cert# in movieExecs
  - (1) Reject modification.
  - (2) Cascade operation
  - (3) Set NULL

Keys and Foreign Keys

```
CREATE TABLE studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

- If we delete a movieExec tuple with a studio president, then the presC# value in studio is replaced by NULL
- If we change a movieExec tuple with a studio president, then the presC# value gets changed as well

Keys and Foreign Keys

- A tuple with foreign key is "dangling" if the foreign key does not exist
- Similarly, a tuple that does not participate in a join is called dangling.

Keys and Foreign Keys

- A table with a foreign key needs to be populated first
- But there are examples of circular references
  - To deal with them:
    - Make the two insertions part of a single transaction
    - Tells the DBMS to not check constraints until the transaction is finished
      - Can declare deferrable
        - INITIALLY DEFERRED — check just before a transaction commits
        - INITIALLY IMMEDIATE — check after each statement is executed

Keys and Foreign Keys

```
CREATE TABLE studio (
   name CHAR(30) PRIMARY KEY,
   address VARCHAR(255),
   presC# INT UNIQUE
      REFERENCES MovieExec(cert#)
      DEFERRABLE INITIALLY DEFERRED
);
```

Keys and Foreign Keys

- Can also give constraints names
- Then change is enforcement policy

```
SET CONSTRAINT myConstraint DEFERRED;
```

```
SET CONSTRAINT myConstraint IMMEDIATE;
```

Constraints on Attributes

- NOT NULL

Constraints on Attributes

- CHECK
    - Enforces conditions on an attribute

```
CREATE TABLE movieExec (
   name CHAR(30) PRIMARY KEY,
   address VARCHAR(255),
   presC# INT REFERENCES MovieExec(cert#)
              CHECK(presC# >= 100000
);
```

Constraints on Attributes

```
CREATE TABLE movieStar(
   name VARCHAR(255) PRIMARY KEY;
   address VARCHAR(255);
   gender CHAR(1)
        CHECK(gender IN ('F', 'M', 'X'))
);
```

Constraints on Attributes

```
CREATE TABLE parts (
    part_no VARCHAR(18) PRIMARY KEY,
    description VARCHAR(40),
    cost DECIMAL(10,2 ) NOT NULL CHECK (cost >= 0),
    price DECIMAL(10,2) NOT NULL CHECK (price >= 0)
);
Code language: SQL (Structured Query Language) (sql)
```

Constraints on Attributes

- Checks cannot be used to replace foreign keys

```
…
presC# INT CHECK
    (presC# IN (SELECT cert# FROM movieExec)
…
```

- The check is only executed by the time the tuple is inserted or changed
- If movieExec changes, our table is NOT updated
- Also, NULL values would be rejected

Constraints on Tuples

- Tuple based checks are executed on Insertion and on Update
  - Checks do not trigger checks for relations mentioned in checks

```
CREATE TABLE movieStar(
   name CHAR(30) PRIMARY KEY,
   address VARCHAR(255),
   gender CHAR(1),
   birthdate DATE,
   CHECK(gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

Constraints on Tuples

- Attribute based checks are executed when
  - Attribute is changed
  - Tuple inserted
- Tuple based checks are executed when
  - Tuple changes
  - Tuple inserted

Constraint Modifications

- You should give your constraints names
    - Helps with error messages
    - Used for changing constraints

```
name CHAR(30) CONSTRAINT nameIsKey PRIMARY KEY

CONSTRAINT rightTitle
    CHECK(gender = 'F' OR name not like 'Ms.%'
```

Constraint Modifications

- Dropping constraints
  - Use ALTER table

```
ALTER TABLE movieStar DROP CONSTRAINT nameIsKey;

ALTER TABLE movieStar ADD CONSTRAINT
              nameIsKey PRIMARY KEY(name)
```

Assertions

- Assertion:
    - A boolean valued SQL expression that must be true at all times
- Trigger:
    - Series of actions associated with certain events and triggered by them

Assertion

- Creating assertions

```
CREATE ASSERTION <name> CHECK (<condition>)
```

- Should be true when you call it, unless the assertion is deferred

Assertion

- Formulating assertions
  - Unlike checks, assertions need to specify the relation

```
movieExec(name, address, cert#, netWorth)
studio(name, address, presC#
```

```
CREATE ASSERTION richPres CHECK(
   (NOT EXISTS
       (SELECT studio.name
        FROM studio, movieExec
        WHERE presC# = cert# AND netWorth<1000000
       )
   );
```

Assertion

- Formulating assertions
  - All studios can only produce <10000 minutes of movies

Assertion

```
CREATE ASSERTION sumLength CHECK
    (10000 >= ALL
           (SELECT SUM(length)
            FROM movies
            GROUP BY studioName
           )
    );
```

Assertion

- Assertions are always checked when there is a change in the database
- Constraints for a tuple are only checked when a tuple is updated or inserted
  - Therefore, making the previous assertion a check has a different meaning:

```
ALTER TABLE movies ADD CONSTRAINT
    maxLength CHECK (10000 >= ALL
         (SELECT SUM(length) FROM movies
            GROUP BY studioName)
    );
```

Assertion

- Dropping assertions

```
DROP ASSERTION sumLength
```

Triggers

- A Trigger is awakened at certain events
  - insert, delete, updates to a particular relation
- A Trigger then tests a condition.
  - If condition is false, nothing more happens
  - Otherwise: The action associated with trigger is executed

Triggers

- Trigger's condition and action executed either :
  - state of DB before the triggering event
  - state of DB after the triggering event
- Condition and action can refer to both the new and the old values
- Update events can be limited to certain attribute(s)
- Trigger can execute
  - once for each modified tuple — *row-level trigger*
  - once for all tuples changed — *statement level trigger*

Triggers

- Example:

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE movieExec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#
```

Triggers

Creating and naming the trigger

- Example:

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE movieExec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#
```

Triggers

- Example:

This is the triggering event:
Update
Attribute
Table

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE movieExec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#
```

Triggers

- Example:

The referencing gives names OLD ROW and NEW ROW are defined with the obvious meanings

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE movieExec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#
```

Triggers

- Example:

OLD ROW is the old value of the tuple, NEW ROW is the new value of the tuple

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE movieExec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#
```

Triggers

- Example:

Here we decide whether we want the action to be performed once or for each row separate

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE movieExec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#
```

Triggers

- Example:

WHEN is the condition (like an if clause)

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
    OLD ROW AS OldTuple,
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
    UPDATE movieExec
    SET netWorth = OldTuple.netWorth
    WHERE cert# = NewTuple.cert#
```

Triggers

This is the conditional action:

An SQL statements that refers to the previous value of the tuple

- Example:

```
CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE movieExec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#
```

Triggers

- Trigger condition:
  - Can use "After" or "Before"
  - There is another option, "Instead", that is used with views

Triggers

- Trigger condition:
  - Triggering events can be:
    - UPDATE
    - INSERT
    - DELETE
  - OF clause is optional for updates and not possible for INSERT and DELETE
    - Because it would not make any sense

Triggers

- WHEN clause
  - Is optional, but supplements the AFTER clause

Triggers

- SQL statements:
  - There can be any number of SQL statements, bracketed by BEGIN … END
  - Separated by semi-colons

Triggers

- If the triggering event is an update, there is going to be an old and a new value of the tuple
  - We give them names in
    - OLD ROW AS
    - NEW ROW AS
- If the update is a delete, we only have an old tuple:
  - OLD ROW AS
- If the update is an insert, then we only have a new tuple:
  - NEW ROW AS

Triggers

- Statement level versus row level trigger
  - Statement level: default or FOR EACH STATEMENT
    - Executed once independent of the number of rows affected (zero, one, many, all)
    - Cannot have OLD ROW or NEW ROW but
      - OLD TABLE AS oldstuff
      - NEW TABLE AS newstuff
  - Row level: FOR EACH ROW
    - allows using OLD ROW and NEW ROW

Triggers

- Example
  - Assume we want the average net worth of movie executives to drop below $500,000.—
  - Could use an assertion, but that would just result in our inability to insert low net worth executives
  - But could have several statements that allow us to insert, delete, and change net worth
    - We then want to refuse such a **block statement** only if afterwards the condition is not true

Triggers

- Example (continued):
  - We need to write one trigger for all modifications
  - But we only want to check after we made our modifications (and can still roll back to the previous state)

Triggers

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
    OLD TABLE AS oldStuff
    NEW TABLE AS newStuff
FOR EACH STATEMENT
    WHEN (500000 > SELECT(AVG(networth) FROM MovieExec;)
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert%, networth) IN newStuff;
    INSERT INTO MovieExec
          (SELECT * FROM oldStuff);
END
```

Triggers

After means check after statement has executed

This is the version for update

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
    OLD TABLE AS oldStuff
    NEW TABLE AS newStuff
FOR EACH STATEMENT
    WHEN (500000 > SELECT(AVG(networth) FROM MovieExec;)
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert%, networth) IN newStuff;
    INSERT INTO MovieExec
          (SELECT * FROM oldStuff);
END
```

Triggers

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
    OLD TABLE AS oldStuff
    NEW TABLE AS newStuff
FOR EACH STATEMENT
    WHEN (500000 > SELECT(AVG(networth) FROM MovieExec;)
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert%, networth) IN newStuff;
    INSERT INTO MovieExec
          (SELECT * FROM oldStuff);
END
```

This is the version for update.

We are creating a statement level trigger, so we give names to the new and the old table contents

Triggers

When checks whether the condition is violated

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
    OLD TABLE AS oldStuff
    NEW TABLE AS newStuff
FOR EACH STATEMENT
    WHEN (500000 > SELECT(AVG(networth) FROM MovieExec;)
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert%, networth) IN newStuff;
    INSERT INTO MovieExec
          (SELECT * FROM oldStuff);
END
```

Triggers

In which case we execute two sql statements, bracketed by BEGIN and END

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
    OLD TABLE AS oldStuff
    NEW TABLE AS newStuff
FOR EACH STATEMENT
    WHEN (500000 > SELECT(AVG(networth) FROM MovieExec;)
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert%, networth) IN newStuff;
    INSERT INTO MovieExec
          (SELECT * FROM oldStuff);
END
```

Triggers

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
    OLD TABLE AS oldStuff
    NEW TABLE AS newStuff
FOR EACH STATEMENT
    WHEN (500000 > SELECT(AVG(networth) FROM MovieExec;)
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert%, networth) IN newStuff;
    INSERT INTO MovieExec
          (SELECT * FROM oldStuff);
END
```

The first statement deletes all contents from MovieExec as it stands now, i.e. after the update

Triggers

The second statement then creates

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
    OLD TABLE AS oldStuff
    NEW TABLE AS newStuff
FOR EACH STATEMENT
    WHEN (500000 > SELECT(AVG(networth) FROM MovieExec;)
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert%, networth) IN newStuff;
    INSERT INTO MovieExec
          (SELECT * FROM oldStuff);
END
```

Triggers

- A common use of triggers is to fix up inserted values
  - Example:
    - `Movies(title, year, length, genre, studioName, producerC#)`
    - Write a trigger that changes a year value of NULL to the current year
      - YEAR(CURDATE)

Triggers

```
CREATE TRIGGER fixYearToCurrentYear
BEFORE INSERT ON Movies
REFERENCING
     NEW ROW AS newRow
     NEW TABLE AS newStuff
FOR EACH ROW
WHEN newRow.year IS NULL
UPDATE newStuff SET year = YEAR(CURDATE);
```

Triggers

- We need to be able to refer to
  - an attribute of the inserted row
  - the table in which we want to change a value
- ```
  CREATE TRIGGER fixYearToCurrentYear
  BEFORE INSERT ON Movies
  REFERENCING
       NEW ROW AS newRow
       NEW TABLE AS newStuff
  FOR EACH ROW
  WHEN newRow.year IS NULL
  UPDATE newStuff SET year = YEAR(CURDATE);
  ```
  ```

Triggers

We check before we actually do the update

```
CREATE TRIGGER fixYearToCurrentYear
BEFORE INSERT ON Movies
REFERENCING
     NEW ROW AS newRow
     NEW TABLE AS newStuff
FOR EACH ROW
WHEN newRow.year IS NULL
UPDATE newStuff SET year = YEAR(CURDATE);
```

Triggers

We have to refer to the row AND to the table

```
CREATE TRIGGER fixYearToCurrentYear
BEFORE INSERT ON Movies
REFERENCING
     NEW ROW AS newRow
     NEW TABLE AS newStuff
FOR EACH ROW
WHEN newRow.year IS NULL
UPDATE newStuff SET year = YEAR(CURDATE);
```

Triggers

This is a row level trigger, not a statement level trigger

```
CREATE TRIGGER fixYearToCurrentYear
BEFORE INSERT ON Movies
REFERENCING
     NEW ROW AS newRow
     NEW TABLE AS newStuff
FOR EACH ROW
WHEN newRow.year IS NULL
UPDATE newStuff SET year = YEAR(CURDATE);
```

Triggers

The when clause refers to the inserted row BEFORE it is inserted

```
CREATE TRIGGER fixYearToCurrentYear
BEFORE INSERT ON Movies
REFERENCING
     NEW ROW AS newRow
     NEW TABLE AS newStuff
FOR EACH ROW
WHEN newRow.year IS NULL
UPDATE newStuff SET year = YEAR(CURDATE);
```

Triggers

We check before we actually do the update

```
CREATE TRIGGER fixYearToCurrentYear
BEFORE INSERT ON Movies
REFERENCING
     NEW ROW AS newRow
     NEW TABLE AS newStuff
FOR EACH ROW
WHEN newRow.year IS NULL
UPDATE newStuff SET year = YEAR(CURDATE);
```

Triggers

We then change with 'SET' the value of the year.
This is SQL to change a variable

```
CREATE TRIGGER fixYearToCurrentYear
BEFORE INSERT ON Movies
REFERENCING
     NEW ROW AS newRow
     NEW TABLE AS newStuff
FOR EACH ROW
WHEN newRow.year IS NULL
UPDATE newStuff SET year = YEAR(CURDATE);
```

Triggers

MySQL peculiarities:

MySQL does NOT have referencing.
Instead use NEW and OLD

Triggers

- Problems
  - In the employee database, create a table employees_audit
  - `employees_audit(rid, emp_no, last_name, change_date, action)`
    - rid: autoincrement
    - emp_no non-null integer
    - last_name non-null VARCHAR(50)
    - changedat a datetime value with default NULL
    - action a varchar value with default NULL

Triggers

- Problem (continued)
  - Basically, we create a record whenever we change something in the employees table
  - HINT: If you want to avoid loading the employees table afterwards, you should disable automatic commits.
  - Do a commit before you do anything
  - And a roll-back afterwards

Triggers

- Problem (continued)
  - After creating the table, create a trigger auditInsert that on each insert into the employees table creates an audit entry in employees_audit
    - You do this before a row has been inserted
    - You do this for each row
    - You insert into the employees_audit table
    - Use SET syntax
    - and use NOW( ) for the time stamp

Triggers

- Create a fictitious employee with high emp_no
- Change something about this employee with an update
- Check what happened to employee_audit

Triggers

- Solution

```
CREATE
     TRIGGER  before_employee_update
       BEFORE UPDATE ON employees
          FOR EACH ROW
       INSERT INTO employee_audit
                SET action = 'update' ,
                    employeeNumber = OLD.emp_no,
                    lastname = OLD.last_name ,
                    changedate = NOW();
```

Triggers

- Problem (continued)

```
INSERT INTO employees VALUES (500000, '1980-12-01',
'Friedrich', 'Chopin', 'M', '2020-03-01');
```

- You can check that this does not trigger anything, since we are not updating

Triggers

```
UPDATE employees
SET
    hire_date = '2020-02-01'
WHERE
    emp_no = 500000;
```

Triggers

- This should have triggered an action.
- Check your result panel (below) in the MySQL workbench to insure that you did not have a typo in your trigger
    - If you had one, just drop the trigger and redefine it.
        - DROP TRIGGER before_employee_update;
- Now you can check the audit table:

id	employeeNumb...	lastname	changedat	action
1	500000	Chopin	2020-03-23 20:58:48	update
NULL	NULL	NULL	NULL	NULL

MySQL Triggers

- If a trigger has more than a single statement:
  - you need to change the delimiter away from the standard semi-colon to something else and back

```
DELIMITER $$
 …

DELIMITER ;
```

MySQL Triggers

- If your trigger has multiple statements:
  - Use the BEGIN … END construct
  - Within the construct, you can then use the standard delimiter

MySQL Triggers

- Recall that instead of WHEN, MySQL has a different construct:
    - IF … THEN … ELSEIF … THEN … END IF;
    - IF … THEN … END IF;

Triggers

- In the employees database:
  - Create a trigger that checks if the hire date is NULL.
  - If this is true, set this date to the current date
  - The current date is given by NOW( )
- Also, instead of referencing you need to use OLD or NEW

MySQL Triggers

- Solution
  -
```
DELIMITER $$
CREATE TRIGGER hireDate
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
   IF NEW.hire_date IS NULL THEN
      SET NEW.hire_date = NOW();
   END IF;

END $$

DELIMITER ;
```

Triggers

- And now you do a rollback or you reload the employees database table