# Exercises: Database Programmability and Transactions

This document defines the **exercise assignments** for the [MySQL course @ Software University.](https://softuni.bg/opencourses/databases-basics-mysql)

# Part I – Queries for SoftUni Database

## Employees with Salary Above 35000

Create stored procedure usp\_get\_employees\_salary\_above\_35000 that returns all employees' first and last names for whose salary is above 35000. The result should be sorted by first\_name then by last\_name alphabetically, and id **ascending**. Submit your query statement as Run skeleton, run queries & check DB in Judge.

**CREATE PROCEDURE usp\_get\_employees\_salary\_above\_35000()**

**BEGIN**

**SELECT first\_name,last\_name**

**FROM employees**

**WHERE salary>35000**

**order by first\_name,last\_name,employee\_id;**

**END**

### Example

|  |  |
| --- | --- |
| **first\_name** | **last\_name** |
| Amy | Alberts |
| Brian | Welcker |
| Dan | Wilson |
| … | … |

## Employees with Salary Above Number

Create stored procedure sa that **accept a decimal number** (with precision, **4 digits** after the decimal point) as parameter and return **all employee's first and last names** whose salary is **above or equal** to the given number. The result should be sorted by first\_name then by last\_name **alphabetically** and id **ascending**. Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_get\_employees\_salary\_above (desired\_salary DECIMAL(19,4))**

**BEGIN**

**SELECT first\_name,last\_name**

**FROM employees**

**WHERE salary>=desired\_salary**

**order by first\_name,last\_name,employee\_id;**

**END**

### Example

Supplied number for that example is 45000.

|  |  |
| --- | --- |
| **first\_name** | **last\_name** |
| Amy | Alberts |
| Brian | Welcker |
| Dylan | Miller |
| … | … |

## Town Names Starting With

Write a stored procedure usp\_get\_towns\_starting\_with that **accept string as parameter** and returns **all town names starting with that string.** The result should be sorted by town\_name alphabetically. Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_get\_towns\_starting\_with (town\_substring VARCHAR(50))**

**BEGIN**

**SELECT `name`**

**FROM towns**

**WHERE `name` like CONCAT(town\_substring,'%')**

**order by `name`;**

**END**

### Example

Here is the list of all towns **starting with "b".**

|  |
| --- |
| **town\_name** |
| Bellevue |
| Berlin |
| Bordeaux |
| Bothell |

## Employees from Town

Write a stored procedure usp\_get\_employees\_from\_town that accepts town\_name as parameter and return the **employees' first and last name that live in the given town.** The result should be sorted by first\_name then by last\_name alphabetically and id **ascending**. Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_get\_employees\_from\_town (searched\_town VARCHAR(50))**

**BEGIN**

**SELECT first\_name,last\_name**

**FROM employees as e**

**JOIN addresses as a on e.address\_id=a.address\_id**

**JOIN towns as t on a.town\_id=t.town\_id**

**WHERE t.name like searched\_town**

**order by first\_name,last\_name;**

**END**

### Example

Here it is a list of employees **living in Sofia.**

|  |  |
| --- | --- |
| **first\_name** | **last\_name** |
| George | Denchev |
| Martin | Kulov |
| Svetlin | Nakov |

## Salary Level Function

Write a function ufn\_get\_salary\_level that receives **salary of an employee** and returns the **level of the salary**.

* If salary is **< 30000** return **"Low"**
* If salary is **between 30000 and 50000 (inclusive)** return **"Average"**
* If salary is **> 50000** return **"High"**

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE FUNCTION ufn\_get\_salary\_level(salary DECIMAL(18, 4))**

**RETURNS VARCHAR(10)**

**DETERMINISTIC**

**BEGIN**

**DECLARE salary\_level VARCHAR(10);**

**IF salary<30000 then SET salary\_level = "Low";**

**elseIF salary between 30000 and 50000 THEN SET salary\_level = "Average";**

**else SET salary\_level = "High";**

**end if;**

**return salary\_level;**

**END**

### Example

|  |  |
| --- | --- |
| **salary** | **salary\_Level** |
| 13500.00 | Low |
| 43300.00 | Average |
| 125500.00 | High |

## Employees by Salary Level

Write a stored procedure usp\_get\_employees\_by\_salary\_levelthat receive as **parameter** **level of salary** (low, average or high) and print the **names of all employees** that have given level of salary. The result should be sorted by first\_name then by last\_name both in **descending order**.

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE FUNCTION ufn\_get\_salary\_level(salary DECIMAL(18, 4))**

**RETURNS VARCHAR(10)**

**return (case when salary<30000 then "Low"**

**when salary>50000 then "High"**

**else "Average" end);**

**CREATE PROCEDURE usp\_get\_employees\_by\_salary\_level(salary\_level VARCHAR(10))**

**BEGIN**

**SELECT e.first\_name,e.last\_name**

**FROM employees as e**

**WHERE ufn\_get\_salary\_level(e.salary)=salary\_level**

**order by e.first\_name DESC,e.last\_name DESC;**

**END**

### Example

Here is the list of all employees with **high salary**.

|  |  |
| --- | --- |
| **first\_name** | **last\_name** |
| Terri | Duffy |
| Laura | Norman |
| Ken | Sanchez |
| … | … |

## Define Function

Define a function ufn\_is\_word\_comprised(set\_of\_letters varchar(50), word varchar(50)) that returns 1 or 0 depending on that if the word is a comprised of the given set of letters.

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE FUNCTION ufn\_is\_word\_comprised(set\_of\_letters VARCHAR(50), word VARCHAR(50))**

**RETURNS INT(1)**

**BEGIN**

**DECLARE count\_let INT;**

**DECLARE length INT;**

**DECLARE current\_char VARCHAR(5);**

**SET length = CHAR\_LENGTH(word);**

**SET count\_let =1;**

**iter\_word: LOOP**

**SET current\_char = SUBSTR(word,count\_let,1);**

**IF LOCATE(current\_char,set\_of\_letters)=0 THEN RETURN 0;**

**ELSEIF count\_let=length THEN RETURN 1;**

**END IF;**

**SET count\_let = count\_let + 1;**

**END LOOP iter\_word;**

**END**

### Example

|  |  |  |
| --- | --- | --- |
| **set\_of\_letters** | **word** | **result** |
| Oistmiahf | Sofia | 1 |
| Oistmiahf | halves | 0 |
| Bobr | Rob | 1 |
| Pppp | Guy | 0 |

# PART II – Queries for Bank Database

## Find Full Name

You are given a database schema with tables:

* account\_holders(id (PK), first\_name, last\_name, ssn)

and

* accounts(id (PK), account\_holder\_id (FK), balance).

Write a stored procedure **usp\_get\_holders\_full\_name** that selects the full names of all people**.** The result should be sorted by **full\_name** alphabetically and **id ascending**. Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_get\_holders\_full\_name()**

**BEGIN**

**SELECT concat(first\_name," ", last\_name) as full\_name**

**FROM account\_holders**

**order by full\_name,id;**

**END**

### Example

|  |
| --- |
| **full\_name** |
| Bjorn Sweden |
| Jimmy Henderson |
| Kim Novac |
| … |

## People with Balance Higher Than

Your task is to create a stored procedure usp\_get\_holders\_with\_balance\_higher\_than that accepts a **number as a parameter** and returns all **people who have more money in total of all their accounts than the supplied number**. The result should be sorted by **account\_holders.id** ascending.

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_get\_holders\_with\_balance\_higher\_than(money decimal(12,4))**

**BEGIN**

**SELECT first\_name,last\_name**

**FROM account\_holders as h**

**Left JOIN accounts as a**

**ON h.id=a.account\_holder\_id**

**GROUP BY first\_name,last\_name**

**HAVING SUM(a.balance) > money;**

**END**

### Example

Supplied number for that example is 7000.

|  |  |
| --- | --- |
| **first\_name** | **last\_name** |
| Susan | Cane |
| Petar | Kirilov |
| Zlatina | Pateva |
| … | … |

## Future Value Function

Your task is to create a function ufn\_calculate\_future\_value that accepts as parameters – **sum** (with precision, **4 digits** after the decimal point), **yearly interest rate (double)** and **number of years(int)**. It should calculate and return the **future value of the initial sum**. The result from the function **must** be **decimal, with percision 4**.

Using the following formula:



* **I** – Initial sum
* **R** – Yearly interest rate
* **T** – Number of years

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE FUNCTION ufn\_calculate\_future\_value(sum double(15,4),rate double,years int)**

**RETURNS decimal (15,4)**

**RETURN sum\*Power(1+rate,years);**

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Initial sum:** 1000  **Yearly Interest rate:** 50%  **years:** 5  ufn\_calculate\_future\_value(1000, 0.5, 5) | 7593.7500 |

## Calculating Interest

Your task is to create a stored procedure usp\_calculate\_future\_value\_for\_account that accepts as parameters – **id** of account and **interest** rate. The procedure uses the function from the previous problem to give an interest to a person's account **for 5 years**, along with information about his/her **account id, first name, last name and current balance** as it is shown in the example below. It should take the **account\_id** and the **interest\_rate** as parameters. Interest rate should have precision up to 0.0001, same as the calculated balance after 5 years. **Be extremely careful to achieve the desired precision!**

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE FUNCTION ufn\_calculate\_future\_value(sum decimal(19,4),rate decimal(19,4),years int)**

**RETURNS decimal(19,4)**

**deterministic**

**RETURN sum\*pow(1+rate,years);**

**CREATE PROCEDURE usp\_calculate\_future\_value\_for\_account(id int, rate double(10,4))**

**BEGIN**

**SELECT a.id, h.first\_name,h.last\_name, a.balance as current\_balance,**

**ufn\_calculate\_future\_value(a.balance,rate,5) as balance\_in\_5\_years from account\_holders as h**

**Left JOIN accounts as a**

**ON h.id=a.account\_holder\_id**

**where a.id=id;**

**END**

### Example

Here is the result for **account\_id = 1** and **interest\_rate = 0.1.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **account\_id** | **fist\_name** | **last\_name** | **current\_balance** | **balance\_in\_5\_years** |
| 1 | Susan | Cane | 123.1200 | 198.2860 |

## Deposit Money

Add stored procedure usp\_deposit\_money(account\_id, money\_amount) that operate in transactions.

Make sure to guarantee valid positive money\_amount with precision up to **fourth sign after decimal point**. The procedure should produce exact results working with the specified precision.

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_deposit\_money(id int,money\_amount decimal(19,4))**

**BEGIN**

**START TRANSACTION;**

**IF(money\_amount<=0 ) THEN**

**ROLLBACK;**

**ELSE**

**UPDATE accounts as ac SET ac.balance = ac.balance + money\_amount**

**WHERE ac.id = id;**

**END IF;**

**END**

### Example

Here is the result for **account\_id = 1** and **money\_amount = 10.**

|  |  |  |
| --- | --- | --- |
| **account\_id** | **account\_holder\_id** | **balance** |
| 1 | 1 | 133.1200 |

## Withdraw Money

Add stored procedures usp\_withdraw\_money(account\_id, money\_amount) that operate in transactions.

Make sure to guarantee withdraw is done only when balance is enough and money\_amount is valid positive number. **Work with precision up to fourth sign after decimal point**. The procedure should produce exact results working with the specified precision.

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_withdraw\_money(id int,money\_amount decimal(19,4))**

**BEGIN**

**START TRANSACTION;**

**IF(money\_amount<=0 or (select balance from accounts as a where a.id=id)<money\_amount) THEN**

**ROLLBACK;**

**ELSE**

**UPDATE accounts as ac SET ac.balance = ac.balance - money\_amount**

**WHERE ac.id = id;**

**END IF;**

**END**

### Example

Here is the result for **account\_id = 1** and **money\_amount = 10.**

|  |  |  |
| --- | --- | --- |
| **account\_id** | **account\_holder\_id** | **balance** |
| 1 | 1 | 123.1200 |

## Money Transfer

Write stored procedure usp\_transfer\_money(from\_account\_id, to\_account\_id, amount)that **transfers money from one account to another**. Consider cases when one of the account\_ids is not valid, the amount of **money is negative number, outgoing balance** is enough or transferring **from/to one and the same account.** Make sure that the whole procedure **passes without errors** and **if error occurs make** **no change in the database.**

Make sure to guarantee exact results working with precision up to fourth sign after decimal point.

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE PROCEDURE usp\_transfer\_money(fromID int, toID int,money\_amount decimal(19,4))**

**BEGIN**

**START TRANSACTION;**

**IF(money\_amount<=0 or (select balance from accounts where id=fromID)<money\_amount**

**or fromID=toID or (SELECT count(id) FROM accounts WHERE id like fromID)<>1**

**or (SELECT count(id) FROM accounts WHERE id like toID)<>1) THEN**

**ROLLBACK;**

**ELSE**

**UPDATE accounts SET balance = balance - money\_amount**

**WHERE id = fromID;**

**UPDATE accounts SET balance = balance + money\_amount**

**WHERE id = toID;**

**END IF;**

**END**

### Example

Here is the result for **from\_account\_id = 1, to\_account\_id = 2** and **money\_amount = 10.**

|  |  |  |
| --- | --- | --- |
| **account\_id** | **account\_holder\_id** | **balance** |
| 1 | 1 | 113.1200 |
| 2 | 3 | 4364.2300 |

## Log Accounts Trigger

Create another table – logs(log\_id, account\_id, old\_sum, new\_sum). Add a **trigger** to the **accounts** table that enters a new entry into the **logs** table every time the sum on an **account** changes.

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE TABLE `logs`(**

**log\_id INT PRIMARY KEY AUTO\_INCREMENT,**

**account\_id INT,**

**old\_sum DECIMAL(19, 4),**

**new\_sum DECIMAL(19, 4);**

**CREATE TRIGGER tr\_update\_account**

**AFTER UPDATE**

**ON accounts**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO `logs`(account\_id, old\_sum, new\_sum)**

**VALUES (OLD.id, OLD.balance, NEW.balance);**

**END;**

### Example

The following data in logs table is inserted after updating balance of account with **account\_id = 1** with 10.

|  |  |  |  |
| --- | --- | --- | --- |
| **log\_id** | **account\_id** | **old\_sum** | **new\_sum** |
| 1 | 1 | 123.12 | 113.12 |
| 2 | 1 | 145.43 | 155.43 |

## Emails Trigger

Create another table – notification\_emails(id, recipient, subject, body). Add a trigger to logs table to **create new email whenever new record is inserted in logs table.** The following data is required to be filled for each email:

* recipient – account\_id
* subject – "Balance change for account: **{**account\_id**}**"
* body - "On **{date (current date)}** your balance was changed from **{**old**}** to **{**new**}.**"

Submit your query statement as **Run skeleton, run queries & check DB in Judge.**

**CREATE TABLE `logs`(**

**log\_id INT PRIMARY KEY AUTO\_INCREMENT,**

**account\_id INT NOT NULL,**

**old\_sum DECIMAL(19, 4) NOT NULL,**

**new\_sum DECIMAL(19, 4) NOT NULL);**

**CREATE TRIGGER tr\_change\_balance**

**AFTER UPDATE**

**ON accounts**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO `logs`(account\_id, old\_sum, new\_sum)**

**VALUES (OLD.id, OLD.balance, NEW.balance);**

**END;**

**CREATE TABLE notification\_emails(**

**id INT PRIMARY KEY AUTO\_INCREMENT,**

**recipient INT NOT NULL,**

**`subject` TEXT,**

**body TEXT);**

**CREATE TRIGGER tr\_email\_on\_change\_balance**

**AFTER INSERT**

**ON `logs`**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO notification\_emails(recipient, `subject`, body)**

**VALUES (NEW.account\_id, concat\_ws(' ', 'Balance change for account:', NEW.account\_id), concat\_ws(' ', 'On', NOW(), 'your balance was changed from', NEW.old\_sum, 'to', NEW.new\_sum, '.'));**

**END;**

### Example

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **recipient** | **subject** | **body** |
| 1 | 1 | Balance change for account: 1 | On Sep 15 2016 at 11:44:06 AM your balance was changed from 133 to 143. |
| … | … | … | … |