# CET1031 - Databases and Data Modelling - Practicum 2

**Topics Covered**: CRUD statements, Stored Functions, Stored Procedures and Triggers.

**Learning Objectives**:

- Familiarizing with porting databases between systems.
- Applying stored functions, procedures and triggers to a database.

**Deliverables**

- A zip file of all completed `.sql` files using the given filenames as defined in this document, the naming format of the zip file is `CET1031_P02_<Your_Name>.zip`. e.g. `CET1031_P02_John_Doe.zip`.
- Note that a non-loadable submission will result in a zero

## Background

You have been tasked with adding **stored functions, stored procedures and triggers** to a database for a fictional company that sells knock-off antique items. You **DO NOT** need to create these tables as they have been provided to you along with a small test dataset to work with. The **incomplete** text schema is as follows (there are 7 tables):

1. `employee(employee_id, first_name, last_name, contact)` where
   - `employee_id` is the primary key
2. `customer(cust_id, company_name, address, contact)` where
   - `cust_id` is the primary key
3. `product(product_id, name, unit_price, remaining_stock, max_stock)` where
   - `product_id` is the primary key
4. `supplier_order(supp_order_id, product_id, order_date, stock, delivered)` where:
   - `supp_order_id` is the primary key
   - `product_id` is the foreign key of the `product` table.
   - `order_date` is filled via the default timestamp value with current timestamp
   - `delivered` is by default is `false` (aka `0`)
5. `order_detail(order_detail_id, employee_id, product_id, cust_id, order_date, quantity, unit_price, discount)` where:
   - `order_detail_id` is the primary key
   - `employee_id` is the foreign key referencing the `employee` table
   - `product_id` is the foreign key referencing the `product` table
   - `cust_id` is the foreign key referencing the `customer` table
   - `order_date` is filled via the default timestamp value with current timestamp
   - `unit_price` values are taken from the `product` table, attribute `unit_price`

6. `payment(payment_id, order_detail_id, pay_method, pay_amt, pay_date, cc_no, cc_exp_date, cc_owner_name)` where:
    - `payment_id` & `order_detail_id` are the primary keys. **Note** that payment and order numbers are treated as a candidate key and **no 2 orders have the same** `order_detail_id`.
    - `order_detail_id` is the foreign key referencing the `order_detail` table.
    - `pay_date` is filled/updated via the default timestamp value with current timestamp
7. `shipping_order (shipping_order_id, order_detail_id, shipping_method, shipping_addr)` where:
    - `shipping_order_id` is the primary key
    - `order_detail_id` is the foreign key referencing the `order_detail` table

# Tasks

**Important notes**

1. All `.sql` files **must be loadable repeatedly** via the command prompt.
2. All tasks are to be answered using **multiple** `sql` **files**. Use the format `task_<x>.sql` to name your `.sql` files, where `<x>` refers to the task number.
3. We are using MariaDB therefore your SQL constructs **must conform** to the SQL constructs supported by MariaDB.
4. Be mindful of the foreign key checks when designing your triggers.
5. You may need to change the delimiter to something else when creating your triggers, stored functions and stored procedures.
6. All tasks are **linked** in some way.
7. There can be many orders for a particular product but the product stock can **only** be updated **after** payment has be processed. Make sure you are able to test all your stored functions and triggers before attempting task 5.
8. Assume that the input values for all tasks are valid.
9. Make sure that all your Tasks are **well tested** before submitting.

check whether we need to turn off autocommit for the triggers.

we have to figure out the link.

Load the given database ( `create_tables.sql` ) and data ( `populate_tables.sql` ) into MariaDB before starting in the tasks.

## Task 1

Create a **stored function** that returns the estimated amount of stock left given the number of units to sell and the product id. The return value can be either positive or negative. **Name the stored function** `est_remaining_stock`.

careful, don't name wrongly.

## Task 2

Create a **stored function** that calculates the total payable amount (inclusive of discount, if any). This function takes the `order_detail_id` and returns the total payable amount rounded up to the nearest cent. **Name the stored function** `total_payable`.

careful, don't name wrongly.

you need to show a message box, don't use try catch. there is no statement to actually show a message box, you want the error to happen so that the message pops up.

## Task 3

Create a **trigger** that makes sure that there is sufficient product stock for all new orders coming in (use stored function created in Task 1), if there is not enough stock, an error message is thrown. In addition, a 5% discount is given for all orders with a purchase quantity of more than 40 (ie: 41 and above) items (regardless of product). Minimum stock for all products cannot drop below 10, exactly 10 is okay. **So long as payment has not been processed, orders can still be made.** Examine Test Case for Task 3 & Point 7 (from above) carefully. **Name the trigger** `before_order`.

need to find out how to throw error message.

careful, don't name wrongly.

## Task 4

the update supplier order when product has arrived is linked to task 5. you need to do something manually to simulate this arrival of product.

Create a **trigger** that does the following:

- inserts a supplier order when the product's remaining stock drops to less than or equal to 10, when there is **no other (currently in delivery) orders** for the same product
- update the supplier order when the product has "arrive".
- **Name the trigger** `after_product_update`.

the currently in delivery orders are talking about the the supplier's delivery to you, not from you to the customer.

careful, don't name wrongly.

## Task 5

Create a **stored procedure** to process the payment of the order. This stored procedure accepts the `order_detail_id`, `payment_method`, `cc_no`, `cc_exp_date`, `cc_owner_name`, `shipping_mtd`, `shipping_addr` as inputs. There are **no outputs**. The procedure is to use the stored functions (from Task 1 & 2) to help with the logic. **Name the stored procedure** `process_payment`.

careful, don't name wrongly.

So long as the **estimated remaining stock is not 0, the payment will go through**, otherwise and error is thrown. A shipping order will also be created once the payment is successful.

**Note**: When testing task 5, if there is not enough product stock for your test cases, manually update the product's remaining stock to simulate the "arrival of new stock" (linked to Task 4).

For the maximum allocation of marks, refer to the table below.

| Description | Marks (%) |
|---|---|
| Task 1 - Successfully implemented and well tested | 10 |
| Task 2 - Successfully implemented and well tested | 10 |
| Task 3 - Successfully implemented and well tested | 15 |
| Task 4 - Successfully implemented and well tested | 15 |
| Task 5 - Successfully implementation and well tested | 40 |
| Proper naming of stored functions, procedures and triggers and using consistent SQL conventions. | 10 |

# Test Cases

The following are **non exhaustive** test cases to help test your stored functions and triggers.

## Test Cases for Task 1

Current stock = 44
SQL statement: `SELECT est_remaining_stock(4, '00000905');`
Output = 40

Current stock = 4
SQL statement: `SELECT est_remaining_stock(4, '00000905');`
Output = 0

Current stock = 4
SQL statement: `SELECT est_remaining_stock(8, '00000905');`
Output = -4

## Test Cases for Task 2

`order_detail_id` = `x0000001` (record available from the `populate_tables.sql`)
SQL statement: `SELECT total_payable('x0000001');`
Output = 523.60

take note that this does not test for discount and nearest cent so write your own test cases.

## Test Cases for Task 3

Current stock = 50
SQL statement:

```sql
INSERT INTO order_detail (order_detail_id, employee_id, product_id, cust_id,
    quantity, unit_price)
    SELECT 'x0000002', '00000001', '00000905', '00004002', 41, p.unit_price
    FROM product AS p
    WHERE p.product_id = '00000905';
```

Output: Error message shown

this one shows error message because it drops the stock to below 10.

Current stock = 50
SQL statement:

```sql
INSERT INTO order_detail (order_detail_id, employee_id, product_id, cust_id,
    quantity, unit_price)
    SELECT 'x0000002', '00000001', '00000905', '00004002', 40, p.unit_price
    FROM product AS p
    WHERE p.product_id = '00000905';
```

Output: Order created.

Current stock = 100
SQL statement:

make sure you can run multiple orders consecutively.

```
INSERT INTO order_detail (order_detail_id, employee_id, product_id, cust_id,
    quantity, unit_price)
    SELECT 'x0000002', '00000001', '00000904', '00004002', 41, p.unit_price
    FROM product AS p
    WHERE p.product_id = '00000904';

INSERT INTO order_detail (order_detail_id, employee_id, product_id, cust_id,
    quantity, unit_price)
    SELECT 'x0000003', '00000001', '00000904', '00004002', 4, p.unit_price
    FROM product AS p
    WHERE p.product_id = '00000904';
```

Output: 2 orders created. The order with 41 items purchased has a 5% discount added.

## Test Cases for Task 4

SQL statement: `UPDATE product SET remaining_stock = 4 WHERE product_id = '00000905';`
Output: Supplier order created for 96 items

SQL statement: `UPDATE product SET remaining_stock = 80 WHERE product_id = '00000905';`
Output: Previous supplier order `delivered` set to `1` (aka `true`)