Data Bases II
Trigger

Michele Beretta

michele.beretta@unibg.it

Exercise T.1

```
ClientOrder(OrderId, ProductId, Qty, ClientId, TotalSubItems)
ProductionProcess(ProdProcId, ObtainedProdId, StartingProdId, Qty, ProcessDuration, ProductionCost)
ProductionPlan(BatchId, ProdProcId, Qty, OrderId)
PurchaseOrder(PurchaseId, ProdId, Qty, OrderId)
```

Table `ProductionProcess` describes how a product can be obtained by (possibly several) other products, which can be themselves obtained from other products from outside.

Build a trigger system that reacts to the insertion of orders from clients and creates new items in `ProductionPlan` or in `PurchaseOrder`, depending on the ordered product, so as to manage the client's order (for the generation of the identifiers, use a function `GenerateId()`).

The triggers should also update the value of `TotalSubItems` (initially always set to 0) to describe the number of sub-products (internally produced or outsourced) that are used overall in the production plan deriving from the order.

Also briefly discuss the termination of the trigger system.

Exercise T.1 – Solution

```
CREATE TRIGGER NewClientOrder
AFTER INSERT ON ClientOrder
FOR EACH ROW
DECLARE
  _ProcId := NULL;
  _Qty    := 0;
BEGIN
  SELECT ProdProcId INTO _ProcId, Qty INTO _Qty
  FROM ProductionProcess
  WHERE ObtainedProdId = NEW.ProductId;

  IF _ProcId IS NOT NULL THEN
    INSERT INTO ProductionPlan
    VALUES (GenerateId(), _ProcId, NEW.Qty, NEW.OrderId)
  ELSE
    INSERT INTO PurchaseOrder
    VALUES (GenerateId(), NEW.ProductId, NEW.Qty, NEW.OrderId)
  END IF;
END;
```

Exercise T.1 – Solution

```
CREATE TRIGGER UpdateSubItemsAfterPurchase
AFTER INSERT ON PurchaseOrder
FOR EACH ROW
BEGIN
  UPDATE ClientOrder
  SET TotalSubItems = TotalSubItems + NEW.Qty
  WHERE OrderId = NEW.OrderId;
END;
```

```
CREATE TRIGGER UpdateSubItemsAfterProduction
AFTER INSERT ON PurchaseOrder
FOR EACH ROW
BEGIN
  UPDATE ClientOrder
  SET TotalSubItems = TotalSubItems + NEW.Qty
  WHERE OrderId = NEW.OrderId;
END;
```

Exercise T.1 – Solution

```
CREATE TRIGGER NewProductionPlan
AFTER INSERT ON ProductionPlan
FOR EACH ROW
DECLARE
  _WantedProd := NULL;
  _Qty        := 0;
BEGIN
  SELECT StartingProdId INTO _WantedProd, Qty INTO _Qty
  FROM ProductionProcess
  WHERE ProdProcId = NEW.ProdProcId;

  IF _WantedProd IS NOT NULL THEN
    INSERT INTO ProductionPlan
    VALUES (GenerateId(), _WantedProd, _Qty * NEW.Qty, NEW.OrderId)
  ELSE
    INSERT INTO PurchaseOrder
    VALUES (GenerateId(), _WantedProd, _Qty, NEW.OrderId)
  END IF;
END;
```

Exercise T.2

Consider the following relational scheme, describing a rental system for music rooms by groups.

```
Client(Id, Name, Surname, Type)
Reservation(ClientId, RoomCode, Day, TimeBegin, TimeEnd)
Usage(ClientId, RoomCode, Day, TimeBegin, TimeEnd, Cost)
Room(Code, HourlyCost)
```

The effective usage can only be present with a corresponding reservation, and possibly start after the reservation's `TimeBegin` or end before its `TimeEnd`.

1. Write a trigger that disallows reservations for already booked rooms
2. Suppose that data about effective usage are compiled at the end of the room's use. Implement a set of rules that assigns the value `"Unreliable"` to the client's `Type` field if they have wasted at least 50 hours, i.e., hours that have been booked but haven't been used.

Exercise T.2 – Solution

Part 1.

```
CREATE TRIGGER CantBook
BEFORE INSERT ON Reservation
FOR EACH ROW
WHEN EXISTS (
  SELECT *
  FROM Reservation
  WHERE RoomCode = NEW.RoomCode
    AND Day = NEW.Day
    AND (
         NEW.TimeBegin BETWEEN TimeBegin AND TimeEnd
      OR NEW.TimeEnd   BETWEEN TimeBegin AND TimeEnd
      OR NEW.TimeBegin < TimeBEgin AND NEW.TimeEnd > TimeEnd
    )
)
ROLLBACK;
```

Exercise T.2 – Solution

Part 2. Let's add `WastedHours` to `Client`.

```
CREATE TRIGGER UpdateWastedHours
AFTER INSERT INTO Usage
FOR EACH ROW
BEGIN
  UPDATE Client
  SET WastedHours = WastedHours + (
    SELECT SUM(TimeEnd - TimeBegin) - SUM(NEW.TimeEnd - NEW.TimeBegin)
    FROM Reservation
    WHERE ClientId = NEW.ClientId
      AND RoomCode = NEW.RoomCode
      AND Day      = NEW.Day
  )
  WHERE Id = NEW.ClientId
END;
```

Exercise T.2 – Solution

We have to handle clients that never showed up, since they do not appear in the database and the previous trigger does not cover them. We suppose there is an event at the end of the day which checks for all clients that never showed up.

```
CREATE TRIGGER UpdateWastedHours2
AFTER End_Day
BEGIN
  UPDATE Client C
  SET WastedHours = WastedHours + (
    SELECT SUM(TimeEnd - TimeBegin)
    FROM Reservation R
    WHERE R.ClientId = C.Id
      AND R.Day = Today()
      AND (R.ClientId, RoomCode, Day) NOT IN (
        SELECT ClientId, RoomCode, Day
        FROM Usage U
        WHERE U.TimeBegin BETWEEN R.TimeBegin AND R.TimeEnd
          AND U.ClientId = C.Id
          AND U.RoomCode = R.RoomCode
          AND U.Day = R.Day
```

```
        ))
END;
```

Exercise T.2 – Solution

Update `Type` in `Client`

```
CREATE TRIGGER UpdateClientType
AFTER UPDATE OF WastedHours ON Client
FOR EACH ROW
WHEN OLD.WastedHours < 50 AND NEW.WastedHours >= 50
BEGIN
  UPDATE Client
  SET Type = "Unreliable"
  WHERE Id = NEW.Id
END;
```

Exercise T.3

```
Owns(Company1, Company2, Percentage)
```

For the sake of simplicity, assume `Owns` starts empty, and that data is gradually added and never edited.

Using triggers, build the `Controls` relation, where company A controls company B if A owns (directly or indirectly) more than 50% of B.

Assume that `Percentage` is a decimal value between 0 and 1. Remember that:

- If A owns more than 50% of B, then A controls B
- If A owns some companies, and these own B, then A controls B if the sum of the percentages of companies owned by A is greather than 50%

Exercise T.3 – Solution

What we want to obtain:

```
Controls(ControllingCompany, ControlledCompany)
```

A can control B in two ways:

1. Directly: A owns more than 50% of B
2. Indirectly: the sum of the directly controlled percentage and the indirectly controlled percentage (through other controlling companies) is over 50%, even if none of the single percentages is greater than 50%.

A simple example: A controls 40% of B, A controls 51% of C, and C controls 20% of B. Since A controls C, A also controls 20% of B through C. Then, A controls B too (40% + 20% > 50%).

Exercise T.3 – Solution

The simple one: direct control.

```
CREATE TRIGGER InsOwns
AFTER INSERT INTO Owns
FOR EACH ROW
BEGIN
  IF NEW.Percentage > 0.5 THEN
    INSERT INTO Controls
    VALUES (NEW.Company1, NEW.Company2)
  END IF;
END;
```

Exercise T.3 – Solution

```
CREATE VIEW DirectControl(Company1, Company2, Percentage) AS
SELECT C.ControllingCompany, O.Company2, SUM(O.Percentage)
FROM Owns O
    JOIN Controls C ON O.Company1 = C.ControlledCompany
WHERE (C.ControllingCompany, O.Company2) NOT IN (SELECT * FROM Controls)
GROUP BY C.ControllingCompany, O.Company2
```

```
CREATE TRIGGER UpdControls
AFTER INSERT INTO Controls
FOR EACH ROW
BEGIN
  INSERT INTO Controls
  SELECT C.Company1, C.Company2
  FROM DirectControl C
    LEFT JOIN Owns O ON C.Company1 = O.Company2 AND C.Company2 = O.Company2
  WHERE C.Company1 = NEW.ControllingCompany
    AND ((O.Percentage IS NULL AND C.Percentage > 0.5) OR O.Percentage + C.Percentage > 0.5)
END;
```

Exercise T.4 – Transactions

Given the following:

```
Item(Id, Name, Type)
Transaction(TransId, SellerCode, BuyerCode, ItemId, Qty, Value, Date, Time)
Operator(Code, Name, Address, Balance)
```

Build a trigger system that keeps updated the `Balance` field of `Operator` after the insertion of tuples in `Transaction`. Specifically, for every transaction where the operators sells, the corresponding value must increase the balance, and where the operator buys, the value must decrease the balance.

Moreover, when an operator's `Balance` becomes negative, they must be inserted into a specific table.

```
Overdraft(Code, Name, Address)
```

Exercise T.4 – Solution

```
CREATE TRIGGER InsTransaction
AFTER INSERT ON Transaction
FOR EACH ROW
BEGIN
  UPDATE Operator
  SET Balance = Balance + NEW.Qty * NEW.Value
  WHERE Code = NEW.SellerCode;

  UPDATE Operator
  SET Balance = Balance - NEW.Qty * NEW.Value
  WHERE Code = NEW.BuyerCode;
END;
```

Exercise T.4 – Solution

```
CREATE TRIGGER InsOverdraft
AFTER UPDATE OF Balance ON Operator
FOR EACH ROW
WHEN NEW.Balance < 0 AND OLD.Balance >= 0
INSERT INTO Overdraft
VALUES (NEW.Code, NEW.Name, NEW.Address);
```

```
CREATE TRIGGER DelOverdraft
AFTER UPDATE OF Balance ON Operator
FOR EACH ROW
WHEN NEW.Balance >= 0 and OLD.Balance < 0
DELETE FROM Overdraft
WHERE Code = NEW.Code;
```

Exercise T.5 – Trips

Consider the following tables:

```
Employee(Code, Name, Qualification, Sector)
Trip(Code, EmployeeCode, Destination, Date, Km, CarPlate)
Car(Plate, Model, CostByKm)
Destination(Name, State)
```

And the following view:

```
Trips(Employee, TotalKm, TotalCost)
```

Write two triggers that fills this view after insertions of new trips, the first in an incremental way, and the second by re-computing the entire view every time.

Exercise T.5 – Trips

```
CREATE TRIGGER IncrementalView
AFTER INSERT INTO Trip
FOR EACH ROW
BEGIN
  UPDATE Trips
  SET TotalKm = TotalKm + NEW.Km,
      TotalCost = TotalCost + (SELECT NEW.Km * CostByKm FROM Car WHERE Plate = NEW.CarPlate)
  WHERE Employee = NEW.EmployeeCode
END;
```

```
CREATE TRIGGER OneShotView
AFTER INSERT INTO Trip
FOR EACH ROW
BEGIN
  DELETE FROM Trips;
  INSERT INTO Trips
  SELECT EmployeeCode, SUM(Km), SUM(Km * CostByKm)
  FROM Trip JOIN Car ON CarPlate = Plate
  GROUP BY EmployeeCode
END;
```

Exercise T.6 – Campus

Given the following

```
Students(Id, Name, Address, Phone, Degree, CourseYear, Campus, TotalCredits)
Enrollments(StudentId, Course, Year, Date)
CourseYear(CourseId, Year, Teacher, Semester, NumStudents, NumOutsideCampus)
CourseDegree(CourseId, Degree)
Courses(CourseId, Title, Credits, Type)
```

1. Write a trigger that removes every insertion on `Enrollments` if there are no corresponding values in `Students` or `CourseYear` (i.e., the enrollment is not valid, due to either the student being unknown or the course not being active).
2. Suppose that `NumOutsideCampus` in `CourseYear` represents the number of students enrolled in that course with a campus *different* from the course's one. Write a trigger that updates `NumOutsideCampus` accordingly after edits on the `Campus` attribute of `Students`.

Exercise T.6 – Part 1

```
CREATE TRIGGER InsEnrollment
BEFORE INSERT ON Enrollments
WHEN NOT Exists(
  SELECT *
  FROM Students
  WHERE Id = NEW.StudentId
) OR NOT EXISTS (
  SELECT *
  FROM CourseYear
  WHERE CourseId = NEW.Course AND Year = NEW.Year
)
ROLLBACK;
```

Exercise T.6 – Part 2

```
CREATE TRIGGER UpdStudentCampus
AFTER UPDATE OF Campus ON Students
FOR EACH ROW
BEGIN
  UPDATE CourseYear
  SET NumOutsideCampus = NumOutsideCampus - 1
  WHERE CourseId IN (
    SELECT CourseId
    FROM Courses
    WHERE Campus = NEW.Campus
  ) AND (CourseId, Year) IN (
    SELECT CourseId, Year
    FROM Enrollments
    WHERE StudentId = NEW.Id
  )

  UPDATE CourseYear
  SET NumOutsideCampus = NumOutsideCampus + 1
  WHERE CourseId IN (
    SELECT CourseId
    FROM Courses
    WHERE Campus = OLD.Campus
  ) AND (CourseId, Year) IN (
    SELECT CourseId, Year
    FROM Enrollments
    WHERE StudentId = OLD.Id
  )
END;
```

We have two things to do: the student becomes 'from outside' for all their courses in the old campus, and 'on campus' for all their courses on the new campus.

Watch out for the use of `OLD` and `NEW` for `Campus`.

Exercise T.7 – Products

The following schema describes a hierarchical set of products, i.e., where a product may contain others. Here, `Level` describes how deep the product is in the hierarchy, with zero being the root(s). If a product is not contained in any other product, we assume `Level` to be 0 and `SuperProduct` to be `NULL`.

```
Product(Code, Name, Description, SuperProduct, Level)
```

1. Write a trigger that, after the deletion of a product, deletes all its sub-products.
2. Write a trigger that, after the creation of a product (possibly a sub-product), computes the value of `Level`.
3. By using a set of triggers, implement a pseudo-integrity check on `SuperProduct`, so that the only admissible values are either `NULL` or the code of another existing product (not itself).

Exercise T.7 – Part 1

```
CREATE TRIGGER DelSubProd
BEFORE DELETE ON Product
FOR EACH ROW
BEGIN
  DELETE FROM Product
  WHERE SuperProduct = OLD.Code
END;
```

Exercise T.7 – Part 2

```
CREATE TRIGGER InsProd
AFTER INSERT ON Product
FOR EACH ROW
BEGIN
  UPDATE Product
  SET Level = 0
  WHERE Code = NEW.Code AND SuperProduct IS NULL;

  UPDATE Product
  SET Level = 1 + (
    SELECT Level
    FROM Product
    WHERE Code = NEW.SuperProduct
  )
  WHERE Code = NEW.Code AND SuperProduct IS NOT NULL;
END;
```

Exercise T.7 – Part 3

```
CREATE TRIGGER IntegrProd
AFTER INSERT ON Product
FOR EACH ROW
WHEN NEW.SuperProduct IS NOT NULL
  AND (NEW.SuperProduct = NEW.Code OR NEW.SuperProduct NOT IN (SELECT Code FROM Product))
ROLLBACK;
```