CS 240A : Databases and Knowledge Base
Homework #3

Ronak Sumbaly
UID : 604591897

October 21, 2015

Question 1

Given the relational database schema Employee(Name, Salary, Department) Department(DeptNo, Manager) define the following active rules in Starburst, Oracle, and DB2.

1.a

A rule that, whenever a department is deleted from the database, sets to null the value of the Department attribute for those tuples in relation Employee having the number of the deleted department.

STARBURST

```
CREATE RULE SetValueNull ON Department
WHEN DELETED
THEN UPDATE Employee
        SET Department = Null
        WHERE Department IN (SELECT DeptNo FROM DELETED);
```

ORACLE

```
CREATE TRIGGER SetValueNull
AFTER DELETE ON Department
REFERENCING OLD AS DeleteDept
FOR EACH ROW
        UPDATE Employee
        SET Department = Null
       WHERE Department = DeleteDept.DeptNo;
```

DB2

```
CREATE TRIGGER SetValueNull
AFTER DELETE ON Department
REFERENCING OLD AS DeleteDept
FOR EACH ROW
        UPDATE Employee
        SET Department = Null
        WHERE Department IN (SELECT DeptNo FROM DeleteDept);
```

1.b

A rule that, whenever a department is deleted from the database, deletes all employees in the deleted department.

STARBURST

```
CREATE RULE DeleteEmp ON Department
WHEN DELETED
THEN DELETE FROM Employee
        WHERE Department IN (SELECT DeptNo FROM DELETED)
```

ORACLE

```
CREATE TRIGGER DeleteEmp
AFTER DELETE ON Department
REFERENCING OLD AS DeleteDept
FOR EACH ROW
        DELETE FROM Employee
        WHERE Department = DeleteDept.DeptNo
```

DB2

```
CREATE TRIGGER DeleteEmp
AFTER DELETE ON Department
REFERENCING OLD AS DeleteDept
FOR EACH ROW
        DELETE FROM Employee
        WHEN Department IN (SELECT Dept-No FROM DeleteDept)
```

1.c

A rule that, whenever the salary of an employee exceeds the salary of its manager, sets the salary of the employee to the salary of the manager.

STARBURST

```
CREATE RULE ExceedSal ON Employee
WHEN INSERTED, UPDATED Salary
THEN UPDATE Emp E
        SET salary =
                (SELECT salary FROM Emp, Department
                WHERE E.Department = DeptNo AND Name = Manager)
        WHERE salary > (Select salary From Emp , Department
                WHERE E.Department = DeptNo AND Name = Manager);
```

ORACLE

```
CREATE TRIGGER ExceedSal
AFTER INSERT OR UPDATE OF Salary ON Emp
REFERENCING NEW AS NEmp
FOR EACH ROW
BEGIN
        WHEN NEmp.Salary > (SELECT Salary FROM Emp, Department
                WHERE NEmp.Department = DeptNo AND Name = Manager)
        UPDATE NEmp
        SET Salary =
                (SELECT Salary FROM Emp, Department
                WHERE NEmp.Department = DeptNo AND Name = Manager)
        WHERE Name = NEmp.Name;
END;
```

DB2

```
CREATE TRIGGER ExceedSal
AFTER INSERT OR UPDATE OF Salary ON Emp
REFERENCING NEW AS NEmp
FOR EACH ROW
BEGIN
        WHEN NEmp.Salary > (SELECT Salary FROM Emp, Department
                WHERE NEmp.Department = DeptNo AND Name = Manager)
        UPDATE NEmp
        SET Salary =
                (SELECT Salary FROM Emp, Department
                WHERE NEmp.Department = DeptNo AND Name = Manager);
        WHERE Name = NEmp.Name;
END;
```

Question 2

Consider the (Oracle) Reorder Rule of Example 2.3 of the ADS book. To stop the reordering when there is already a pending order, we can use an active rule on PendingOrders, instead of the PL/SQL code currently used in Example 2.3. Please write such a DB2 rule and also revise the Reorder rule accordingly (and to conform to the syntax of DB2 triggers). You are not required to test your triggers.

DB2

```
CREATE TRIGGER Reorder
AFTER UPDATE OF PartOnHand ON Inventory
WHEN (New.PartOnHand < New.ReorderPoint)
REFERENCING NEW AS NOrder
FOR EACH ROW
        WHEN (NOrder.Part NOT IN (SELECT Part FROM PendingOrders))
        INSERT INTO PendingOrders VALUES (New.Part, New.OrderQuantity, SYSDATE)
```

Question 3

Revise your previous rule on PendingOrders so that when there is already a pending order for that part you only add in an order for 1/2 of the requested OrderQuantity instead of the requested quantity.ă

DB2

```
CREATE TRIGGER HalfQuantity
BEFORE INSERT ON PendingOrders
REFERENCING NEW AS NOrder
FOR EACH ROW
        WHEN (NOrder.Part IN (SELECT Part FROM PendingOrders))
        UPDATE NOrder
                SET NOrder.OrderQuantity = 0.5 * NOrder.OrderQuantity;
```

Question 4 - Deal Assignment.

You are given a set of functional dependencies such as

f1: A–>B
f2: B–>C
f3: CD–>E

Your assignment is to write a DeAL program to determine if some FD, BD–>E is implied by these FD. Basically, you can assume that the given fds are represented as follows:

ls(a, f1). rs(f1, b).
ls(b, f2). rs(f2, c).
ls(c, f3). ls(d, f3). rs(f3, e).

So assuming a fact base in this given form, what rules will you want to write to derive our goals FD, also represented in a suitable format. (Hint., you might to determine first how many attributes each FD has. Then you might want to write recursive rules that infer the right side of an FD as soon as the enough of its left sides have been inferred)

FunctionalDependency.fac

```
% Facts
% ls(Node:string, Dep:string)

ls('A','F1').
ls('B','F2').
ls('C','F3').
ls('D','F3').

% rs(Dep:string, Node:string)

rs('F1','B').
rs('F2','C').
rs('F3','E').

% tls(Node:string, Dep:string)

tls('B','F4').
tls('D','F4').

% trs(Dep:string, Node:string)

trs('F4','E').
```

FunctionalDependency.deal

```
% Schema
database({ls(Node:string, Dep:string), rs(Dep:string, Node:string),tls(Node:string, Dep:string),
trs(Dep:string, Node:string)}).

%Derivated predicates and rules

attributes(D," "," ") <- ls(_,D).

% add all rules to attributes table
attributes(D,V,V2) <- attributes(D,_,_), ls(V,D), rs(D,V2).
attributesClean(D,LS,RS) <- attributes(D,LS,RS), LS =" ", RS =" ".

% one step rules
oneStepRule(D,LS,LS2) <- attributesClean(D,LS,RS), attributesClean(D2,RS,LS2), D =D2.

%combine rules
allRules(LS,RS) <- attributesClean(_,LS,RS).
allRules(LS,RS) <- oneStepRule(_,LS,RS).
allRules(LS,LS2) <- allRules(LS,RS), oneStepRule(_,RS,LS2).

% get the test attribute table
testRule(D," "," ") <- tls(_,D).
testRule(D,V,V2) <- testRule(D,_,_), tls(V,D), trs(D,V2).
allTestRules(D,LS,RS) <- testRule(D,LS,RS), LS =" " , RS =" ".

% count the number of test cases for each rule
testCount(D, count<D>) <- allTestRules(D,_,_).

% check for valid rules
validRules(D,LS,RS) <- allTestRules(D,LS,RS), allRules(LS,RS).

% count the valid rules
validCount(D,count<D>) <- validRules(D,_,_).

% check if all rules are satisfied
outputTable(D,V) <- testCount(D,V1), validCount(D1,V2), if (V1=V2 then V=1 else V=0), D=D1.

% check for rule that is not present in output table and set output to 0
getTestQuery(D) <- testCount(D,_).
getOutputQuery(D) <- outputTable(D,_).

notPresent(D) <- getTestQuery(D),  getOutputQuery(D).
notValid(D,V) <- notPresent(D), V=0.

% add all invalid rules in the table even if they valid
preOutput(D,V) <- outputTable(D,V).
preOutput(D1,V2) <- preOutput(D,V), notValid(D1,V2), D =D1.

finalOutput(D,sum<V>) <- preOutput(D,V).
```

```
export allRules(L,R). % VALID RULES
export allTestRules(D,LS,RS). % TEST RULES
export testCount(D,C). % TEST RULES COUNT
export validCount(D,C). % COUNT OF VALID RULES
export finalOutput(D,V). % FINAL OUTPUT WHERE 1 denotes VALID FD and 0 INVALID
```

Output

Query: finalOutput(A,B)
finalOutput(f4,1)