**Exercise 1**

Q.1 ) Create a trigger on the table employees, which after an update or insert, converts all the values of first and last names to upper case. (Hint: Use cursors to retrieve the values of each row and modify them.)

**Solution:**

>> create trigger converttoupper on employees

after update , insert

as

declare employee\_cursor CURSOR FOR SELECT firstname, lastname from employees; declare @fname varchar(100), @lname varchar(100);

OPEN employee\_cursor;

fetch next from employee\_cursor into @fname,@lname;

WHILE @@FETCH\_STATUS = 0 begin

update employees set firstname = upper(@fname), lastname = upper(@lname) where CURRENT OF employee\_cursor;

FETCH NEXT FROM employee\_cursor INTO @fname,@lname

end;

go

DROP INDEX employees.i\_func\_employees\_lastname;

2) Create a trigger that restores the values before an update operation on the employees table if the salary exceeds 100000. (Hint: Use the inserted and deleted tables to look at the old and new values in the table respectively.)

**Solution :**

>> create trigger salary\_checker on employees

after UPDATE

as

begin transaction trans;

save transaction check\_1;

if exists(select \* from inserted where salary > 100000)

begin RAISERROR ('salary > 100000 not allowed', 16, 1);

rollback transaction check\_1;

end;

commit transaction trans;

go;

3) Create a trigger to insert into the view *alex* such that the underlying base tables are *populated* correctly. Handle cases where the input is incorrect and rollback if a wrong input is asked to be *inserted*. Assume that you cannot create a new department by an insert query.

>> create trigger tr\_alex\_insert

on alex

instead of insert

as

begin transaction temp;

-- Just in case we get an incorrect tuple to insert

SAVE TRANSACTION p1;

declare @employeeid numeric(9);

declare @firstname varchar(10), @lastname varchar(20);

declare @code char(5);

declare @salary numeric(9,2);

select @employeeid = employeeid from inserted;

select @firstname = firstname from inserted;

select @lastname = lastname from inserted;

select @code = code from inserted;

select @salary = salary from inserted;

-- Ensure that primary key constraint is followed and --Ensure that department code is valid

if ( @employeeid in (select employeeid from employees) ) or (@code not in (select code from departments) ) begin

rollback transaction p1;

end

else begin

insert into employees values(@employeeid,@firstname,@lastname,@code,@salary;

end

commit transaction temp;

go

4) Create a trigger to delete from the view *alex* such that the corresponding rows in the underlying base tables are *removed* correctly. Handle cases where the rows requested to be *deleted* are not possible and *rollback* accordingly. Assume that a department cannot be left without a manager.

>> create trigger tr\_alex\_delete

on alex

instead of delete

as

-- make sure to mention the name of the transaction

begin transaction temp

save transaction t1;

declare @id numeric(9);

declare @code char(5);

declare @first\_name varchar(10), @last\_name varchar(20);

declare @salary numeric(9,2);

declare @int\_1 numeric(9);

declare @int\_2 numeric(9);

select @int\_1 = count(\*) from (select distinct managerid from departments except select employeeid from deleted) d2;

select @int\_2 = count(\*) from (select distinct managerid from departments) d1;

-- if any of the employees are managers do nothing

if (@int\_2 > @int\_1) begin

-- make sure to mention the name of the transaction

rollback transaction t1;

end

-- else continue with the deletion

else begin

declare delete\_cursor CURSOR for select \* from deleted;

open delete\_cursor;

fetch next from delete\_cursor into @id,@first\_name,@last\_name,@salary,@code;

while @@FETCH\_STATUS = 0

begin

select @id = employeeid from deleted;

select @code = code from deleted;

-- delete all workson entries

delete from workson where employeeid = @id;

-- delete all employee tables

delete from employees where employeeid = @id;

-- fetch next

fetch next from delete\_cursor into @id,@first\_name,@last\_name,@salary,@code;

end

end

-- make sure to mention the name of the transaction

commit transaction temp;

go

5) Create a trigger to modify from the view *alex* such that the corresponding rows in the underlying base tables are *changed* correctly. Handle cases where the *modifications* asked for are *not possible* and *rollback* accordingly. Assume that a department cannot be left without a manager.

>> create trigger tr\_alex\_update

on alex

instead of update

as

begin transaction trans;

save transaction check\_1;

declare @int\_1 numeric(9);

declare @int\_2 numeric(9);

declare @int\_3 numeric(9);

select @int\_1 = count( distinct employeeid ) from inserted;

select @int\_2 = count(\*) from inserted;

select @int\_3 = count(\*) from (select employeeid from inserted where employeeid in (select employeeid from employees)) d1;

-- check if the primary keys are same for any two rows

-- check if primary key does not already exist

if( (@int\_1 != @int\_2) or (@int\_3 > 0) ) begin

rollback transaction check\_1;

end

else begin

declare @insert\_id numeric(9);

declare @insert\_code char(5);

declare @insert\_first\_name varchar(10), @insert\_last\_name varchar(20);

declare @insert\_salary numeric(9,2);

declare @delete\_id numeric(9);

declare @delete\_code char(5);

declare @delete\_first\_name varchar(10), @delete\_last\_name varchar(20);

declare @delete\_salary numeric(9,2);

select \* from inserted;

select \* from deleted;

declare insert\_cursor cursor for select \* from inserted;

open insert\_cursor;

fetch next from insert\_cursor into @insert\_id,@insert\_first\_name,@insert\_last\_name,@insert\_salary,@insert\_code;

declare delete\_cursor CURSOR for select \* from deleted;

open delete\_cursor;

fetch next from delete\_cursor into @delete\_id,@delete\_first\_name,@delete\_last\_name,@delete\_salary,@delete\_code;

while @@FETCH\_STATUS = 0 begin

-- insert the new row

insert into employees

values(@insert\_id,@insert\_first\_name,@insert\_last\_name,@insert\_code,@insert\_salary);

-- modify the foreign key relation values

update workson set employeeid=@insert\_id where employeeid=@delete\_id;

update departments set managerid=@insert\_id where managerid=@delete\_id;

-- delete the old row

delete from employees where employeeid=@delete\_id;

-- fetching the new rows

fetch next from insert\_cursor into @insert\_id,@insert\_first\_name,@insert\_last\_name,@insert\_salary,@insert\_code;

fetch next from delete\_cursor into @delete\_id,@delete\_first\_name,@delete\_last\_name,@delete\_salary,@delete\_code;

end

end

commit transaction trans;

go

**Exercise 2**

1. Create a unique composite non-clustered index on the first\_name and order in the descending order.

>> create UNIQUE Nonclustered index IX\_student\_info on

student\_info(first\_name asc);

1. Create a composite clustered index on the first\_name and the last\_name and order both in the increasing order.

>> create clustered index IX\_student\_info2 on

student\_info(first\_name asc, last\_name asc);

1. Create a unique Clustered index on the age and order in the increasing order

>> drop index student\_info.IX\_student\_info2 create UNIQUE clustered index IX\_student\_info3 on student\_info(age asc);

1. Create composite non-clustered index on the id,age both in the increasing order

>> create Nonclustered index IX\_student\_info4 on student\_info(id asc ,age asc);

1. Create a non-clustered unique function based index on the upper(first\_name + ' ' + lastname)

>> alter table student\_info add complete\_name as upper(first\_name + ' ' + last\_name);

create unique Nonclustered index IX\_student\_info5 on student\_info(complete\_name);