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 of the updated or inserted rows 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 inserted;
      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;

      -- close and deallocate cursors
      close employee_cursor;
      deallocate employee_cursor;

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
instead of 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
                        -- 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

            -- close and deallocate the cursors
            close delete_cursor;
            deallocate delete_cursor;
      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,@inser
      t_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_sal
      ary,@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,@inser
                t_salary,@insert_code;
                fetch next from delete_cursor into
                @delete_id,@delete_first_name,@delete_last_name,@delet
                e_salary,@delete_code;

            end

            -- close and deallocate cursor
            close delete_cursor;
            deallocate delete_cursor;

        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);`

2. 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);`

3. 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);
```

4. 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);
```

5. 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);
```