Birla Institute of Technology and Science, Pilani
Database Systems
Lab No #8

Today's Topics

- ❖ Triggers
- ❖ Indexing

Triggers

It is a TSQL block or TSQL procedure associated with tables, views, schemas, or databases. It implicitly gets executed whenever a particular event takes place and the event might be in the following forms:

- System event
- DML statement being issued against the table

Features of Triggers

- Trigger includes execution of SQL and TSQL statements as a unit.
- Triggers are stored separately from their associated tables in the Database.
- Triggers are similar to subprograms in the following ways:
  - Implementing TSQL blocks with declarative, executable, and exception handling sections
  - Can be stored in Database and can invoke other stored procedures

Benefits of Triggers

- Audit data modifications
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicated tables
- Enable building complex updatable views
- Useful in tracking system events

Difference between Triggers and Subprograms

Triggers	Subprograms
Defined with CREATE TRIGGER	Defined with CREATE PROCEDURE / CREATE FUNCTION
Are executed implicitly whenever the triggering event occurs	Are executed explicitly by a user, application, or a trigger
Do not accept arguments	Can accept arguments
Data dictionary contains source code in USER_TRIGGERS	Data dictionary contains source code in USER_SOURCE
Implicitly invoked	Explicitly invoked
COMMIT, SAVEPOINT, and ROLLBACK are not allowed	COMMIT, SAVEPOINT, and ROLLBACK are allowed

Types of Triggers

Application Triggers

- Are executed implicitly when a DML event occurs with respect to an application (usually a frontend application)
- Operates in the Client layer in client/server architecture

Database Triggers

- Executes whenever a data event or system event occurs on a schema or database
  Data Event implies DML or DDL. System Event implies SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.
- Are TSQL program units that are associated with a specific table or view
- Operate in Client/Server architecture in the server layer
- Are compiled and stored permanently in the database
  - Hence the name "database" triggers!
- Are integrated with middle tier applications

Parts of Database Triggers

- Whenever the trigger event occurs the database trigger is implicitly fired and the TSQL block performs the action.
- The four parts of database trigger are:
  - Trigger Timing
    - ✔ Determines when the trigger executes in relation to the triggering event - Can be FOR, AFTER, or INSTEAD OF

- Triggering Event
    - ✔ The event may be a SQL statement that causes a trigger to be executed - Can be an INSERT, UPDATE, or DELETE statement for a specific table or view – Trigger Action
    - ✔ A procedure that contains SQL statements
    - ✔ Is executed when a triggering statement is issued and the trigger restriction (if present) evaluates to TRUE

Defining Database Triggers

- Database triggers can be defined on:
    - DML statements
        - ✔ INSERT, UPDATE, and DELETE
    - DDL events
        - ✔ CREATE, DROP, and ALTER
    - Database operations
        - ✔ SERVERERROR, LOGON, LOGOFF, STARTUP, and SHUTDOWN

Creating Database Trigger

- CREATE TRIGGER is used to create a new trigger.
- ALTER TRIGGER is used to replace an existing trigger.
- To create a trigger the user must have:

```
CREATE [ OR ALTER ] TRIGGER trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]  ]
AS { sql_statement  [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }
GO
```

A trigger that fires before an operation will make sure the operation can execute correctly. A trigger that fires after the table modification usually will cause another action to take place. Triggers are added to the database using the CREATE TRIGGER command.

FOR | AFTER :

AFTER specifies that the DML trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. All referential cascade actions and constraint checks also must succeed before this trigger fires. AFTER is the default when FOR is the only keyword specified. AFTER triggers cannot be defined on views.

INSTEAD OF :

Specifies that the DML trigger is executed *instead of* the triggering SQL statement, therefore, overriding the actions of the triggering statements. At most, one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement can be defined on a table or view. However, you can define multiple views on a schema where each view has its own INSTEAD OF trigger. INSTEAD OF triggers are not allowed on updatable views that use WITH CHECK OPTION. SQL Server raises an error when an INSTEAD OF trigger is added to an updatable view WITH CHECK OPTION specified. The user must remove that option by using ALTER VIEW before defining the INSTEAD OF trigger.

{ [DELETE] [,] [INSERT] [,] [UPDATE] } :

Specifies the data modification statements that activate the DML trigger when it is tried against this table or view. At least one option must be specified. Any combination of these options in any order is allowed in the trigger definition.

For INSTEAD OF triggers, the DELETE option is not allowed on tables that have a referential relationship specifying a cascade action ON DELETE. Similarly, the UPDATE option is not allowed on tables that have a referential relationship specifying a cascade action ON UPDATE.

WITH APPEND

Specifies that an additional trigger of an existing type should be added. WITH APPEND cannot be used with INSTEAD OF triggers or if AFTER trigger is explicitly stated. WITH APPEND can be used only when FOR is specified, without INSTEAD OF or AFTER, for backward compatibility reasons.

Execution of Trigger

- It is executed implicitly when DML, DDL, or system event occurs.
- The act of executing a trigger is also known as **firing the trigger**.
- When multiple triggers are to be executed, they have to be executed in a sequence.
- Database triggers execute with the privileges of the owner, not the current user.
- The following list shows the sample execution sequence:

Example:

Assume that we always have a 100% markup on the price of items. In other words, the price of an item is always twice the cost of the sum of the ingredients. We can create a trigger to update the price of a meal when the ingredient costs are modified.

```
CREATE TRIGGER markup ON ingredients
AFTER UPDATE as
UPDATE items SET price = (SELECT 2 * SUM(quantity * unitprice)
FROM madewith m , ingredients i
WHERE m.ingredientid = i.ingredientid AND items.itemid = m.itemid)
```

Two special tables inserted and deleted can be used in a trigger.
Inserted: Table containing the new values of row / rows being updated / inserted.
Deleted: Table containing the previous values of the row / rows being updated / deleted.

If one wants to create a custom *insert* trigger that will execute when the user tries to insert data normally, then inside the trigger construct it would be essential to get access to the data that the user tried to insert. This data is retrieved from the insert table. Once the data is retrieved, our custom *insert* trigger can now insert into the underlying tables with the appropriate logic that is needed. Note the *inserted* table and the *deleted* table can only be referenced inside the trigger construct. Any attempt at referencing these tables from somewhere else will result in an error.

** Think of how this could help if we wanted to insert into a view that normally wouldn't allow us to do so ? **

Example:

Create a view new_emp_dept

```
Create view new_emp_dept as
Select e.employeeid, firstname, e.deptcode, e.salary, d.name
From employees e, departments d
Where e.deptcode = d.code;
```

Now suppose when a delete statement is executed for the view specifying a department code, following actions are to be performed simultaneously:

1. Delete all rows from the *employees* table for a particular department code.
2. Delete the corresponding department information from the *departments* table

```
CREATE TRIGGER new_emp_dept_remove
ON new_emp_dept
INSTEAD OF DELETE
AS
DELETE FROM employees where employeeid in(select employeeid from
     deleted);
DELETE FROM departments where code in (select deptcode from deleted)
GO
```

<u>Managing Triggers</u>

- Triggers can be enabled, disabled, and recompiled.
  - To disable or enable one database trigger:
    ```
    ALTER TABLE <TABLE_NAME> DISABLE TRIGGER { [ schema_name . ]
    trigger_name [ ,...n ] | ALL};
    ```
  - To disable or enable or alter all triggers for a table:
    ```
    ALTER TABLE <TABLE_NAME> DISABLE TRIGGER ALL;
    ```
  - To recompile a trigger for a table (here *object* is the name of any object in the database) :
    ```
    sp_recompile [ @objname = ] 'object'
    ```

The queries used by stored procedures, or triggers, and user-defined functions are optimized only when they are compiled. As indexes or other changes that affect statistics are made to the database, compiled stored procedures, triggers, and user-defined functions may lose efficiency. By recompiling stored procedures and triggers that act on a table, you can re-optimize the queries.

<u>Modifying Triggers</u>

- It is possible to straight away modify the existing triggers using the ALTER triggers command with the same syntax as in CREATE.
- ALTER TRIGGER supports manually updatable views through INSTEAD OF triggers on tables and views. SQL Server applies ALTER TRIGGER the same way for all kinds of triggers (AFTER, INSTEAD-OF).

<u>Removing Triggers</u>

- Triggers can be deleted from the system memory.
- They are removed using the DROP TRIGGER command.
- To drop a trigger one must either :
  - Own the trigger or
  - Have the DROP ANY TRIGGER system privilege

Syntax:

```
DROP TRIGGER [ IF EXISTS ] [schema_name.]trigger_name [ ,...n ] [ ; ]
```

Example:

```
DROP TRIGGER new_emp_dept_remove;
```

<u>**Schema for Temporary view**</u>

```
        create view alex as
        select employeeid,firstname,lastname,salary,code
        from employees inner join departments
        on employees.deptcode = departments.code;
```

The above view is used for questions 3 to 5.

<u>**Exercise:**</u>

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)
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.)
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.
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.
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.

Indexes

An index for a database table is similar in concept to a book index.

- The downside of indexes is that when a row is added to the table, additional time is required to update the index for the new row.
- Generally, you should create an index on a column when you are retrieving a small number of rows from a table containing many rows. A good rule of thumb is to create an index when a query retrieves <= 10 percent of the total rows in a table.
- This means the column for the index should contain a wide range of values. These types of indexes are called "B-tree" indexes

Creating a B-Tree Index

Syntax:

```
CREATE [UNIQUE][CLUSTERED|NONCLUSTERED] INDEX index_name ON
table_name(column_name[, column_name ...]);
```

where

- UNIQUE means that the values in the indexed columns must be unique.
- index_name is the name of the index.
- table_name is a database table.
- column_name is the indexed column. You can create an index on multiple columns (such an index is known as a composite index).

Query to be executed

```
SELECT employeeid, firstname, lastname FROM employees WHERE lastname =
'Advice';
```

The following CREATE INDEX statement creates an index named i_employees_lastname on the lastname column of the employees table (I always put i_ at the start of index names):

```
CREATE INDEX i_employees_lastname ON employees(lastname);
```

Once the index has been created, the latter query will take less time to complete. You can enforce uniqueness of column values using a unique index. For example, the following statement creates a unique index named i_employees_salary on the employees.salary column:

```
CREATE UNIQUE INDEX i_employees_salary ON  employees(salary);
```

You can also create a composite index on multiple columns. For example, the following statement creates a composite index named i_employees_first_lastname on the firstname and lastname columns of the employees table:

```
CREATE INDEX i_employees_first_lastname ON  employees(firstname, lastname);
```

Note, there are multiple ways to define an index. The index can be a *clustered* or a *non-clustered* index. Similarly it can also be *unique* or without a unique constraint.

Clustered Index and Non-Clustered Index

Consider a dictionary. If tomorrow we decide to insert a new word "alex" into the dictionary, then we would insert the word in the *'a' section* and possibly push all the words in *'b'* one place lower. Hence it is easy to notice that such an ordering directly affects the way the data (in this case words) is stored.

Now consider the glossary at the end of the dictionary, any change to the glossary does not change the way the data is stored in the dictionary. This means that the glossary is a smart way to navigate through the book but does NOT affect how the data is stored.

The former is an example of a Clustered Index and the latter is an example of the Non-Clustered Index. If we create a clustered index on a column of a table. Then as and when we insert rows into the table, the entries are stored in the manner prescribed by the clustering index. Hence it is intuitive to state that there is only one clustering index for each table. Note by default if you mention a column as the primary key for a table, then it becomes the clustering index for the table.

```
ex. Create clustered index IX_temp on employees(lastname asc)
Should this give you an error ?
```

In the same way, just like a glossary many other secondary indices can be created to efficiently index into the database without changing the format of the stored data. Hence one table can have multiple non-clustered indexes.

```
ex. Create Nonclustered index IX_temp on employees(lastname asc)
```

Creating a Function-Based Index

```
SELECT firstname, lastname FROM employees WHERE lastname = UPPER('ADVICE');
```

Because this query uses a function UPPER(), in this case - the i_employees_lastname index isn't used. If you want an index to be based on the results of a function, you must create a function-based index, such as:

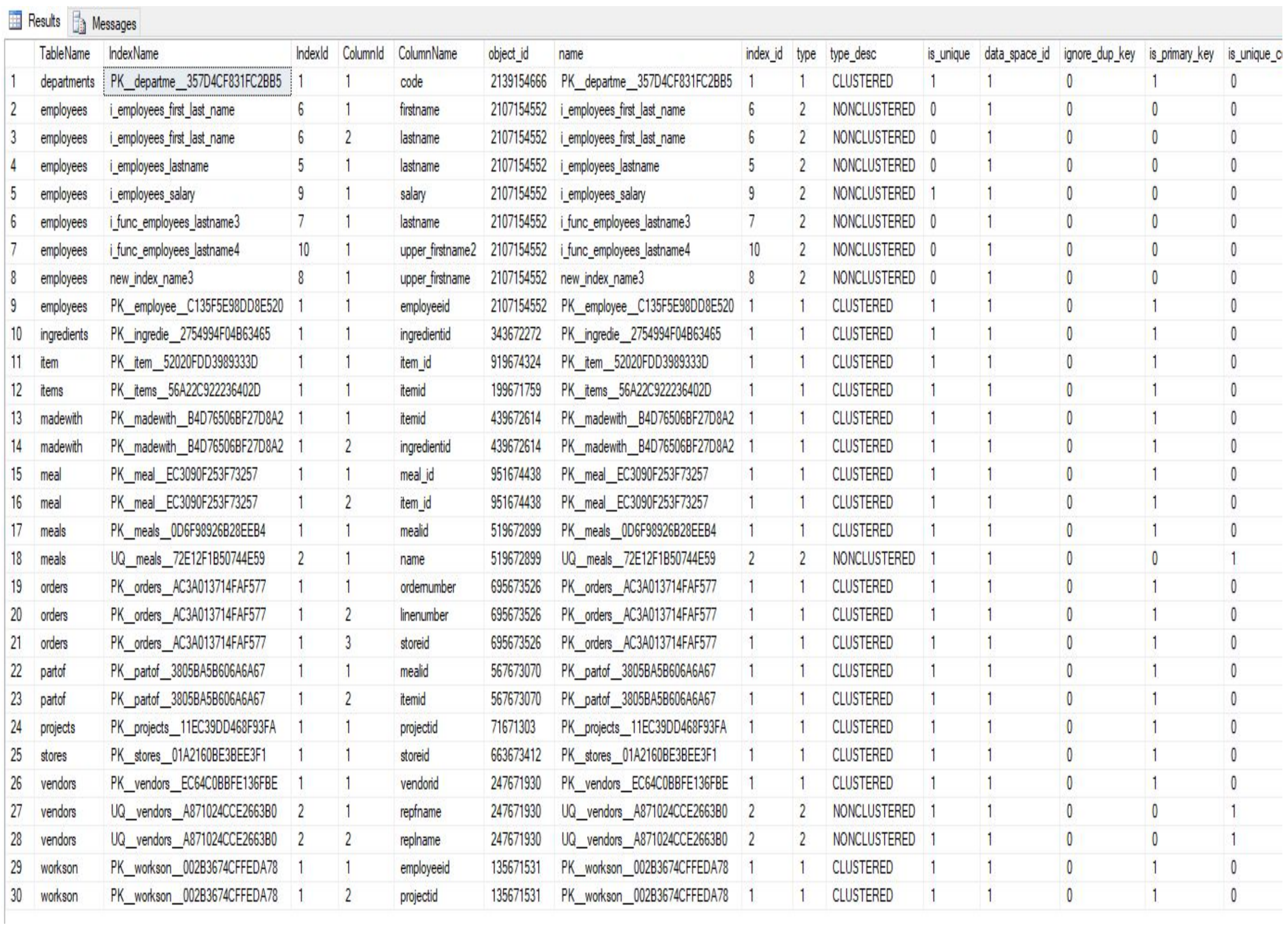
First execute query 1 to create a computed column and then index on the computed column

```
1) ALTER TABLE dbo.employees ADD upper_firstname AS UPPER(firstname)

2) CREATE INDEX i_func_employees_lastname ON employees(upper_firstname);
```

<u>Retrieving Information on Indexes</u> Employees and employees tables:

```
SELECT
    TableName = t.name,
    IndexName = ind.name,
    IndexId = ind.index_id,
    ColumnId = ic.index_column_id,
    ColumnName = col.name,
    ind.*,
    ic.*,
    col.*
FROM
    sys.indexes ind
INNER JOIN
    sys.index_columns ic ON  ind.object_id = ic.object_id and ind.index_id = ic.index_id
INNER JOIN
    sys.columns col ON ic.object_id = col.object_id and ic.column_id = col.column_id
INNER JOIN
    sys.tables t ON ind.object_id = t.object_id
ORDER BY
    t.name, ind.name, ind.index_id, ic.index_column_id;
```



Results | Messages

	TableName	IndexName	IndexId	ColumnId	ColumnName	object_id	name	index_id	type	type_desc	is_unique	data_space_id	ignore_dup_key	is_primary_key	is_unique_c
1	departments	PK__departme__357D4CF831FC2BB5	1	1	code	2139154666	PK__departme__357D4CF831FC2BB5	1	1	CLUSTERED	1	1	0	1	0
2	employees	i_employees_first_last_name	6	1	firstname	2107154552	i_employees_first_last_name	6	2	NONCLUSTERED	0	1	0	0	0
3	employees	i_employees_first_last_name	6	2	lastname	2107154552	i_employees_first_last_name	6	2	NONCLUSTERED	0	1	0	0	0
4	employees	i_employees_lastname	5	1	lastname	2107154552	i_employees_lastname	5	2	NONCLUSTERED	0	1	0	0	0
5	employees	i_employees_salary	9	1	salary	2107154552	i_employees_salary	9	2	NONCLUSTERED	1	1	0	0	0
6	employees	i_func_employees_lastname3	7	1	lastname	2107154552	i_func_employees_lastname3	7	2	NONCLUSTERED	0	1	0	0	0
7	employees	i_func_employees_lastname4	10	1	upper_firstname2	2107154552	i_func_employees_lastname4	10	2	NONCLUSTERED	0	1	0	0	0
8	employees	new_index_name3	8	1	upper_firstname	2107154552	new_index_name3	8	2	NONCLUSTERED	0	1	0	0	0
9	employees	PK__employee__C135F5E98DD8E520	1	1	employeeid	2107154552	PK__employee__C135F5E98DD8E520	1	1	CLUSTERED	1	1	0	1	0
10	ingredients	PK__ingredie__2754994F04B63465	1	1	ingredientid	343672272	PK__ingredie__2754994F04B63465	1	1	CLUSTERED	1	1	0	1	0
11	item	PK__item__52020FDD3989333D	1	1	item_id	919674324	PK__item__52020FDD3989333D	1	1	CLUSTERED	1	1	0	1	0
12	items	PK__items__56A22C922236402D	1	1	itemid	199671759	PK__items__56A22C922236402D	1	1	CLUSTERED	1	1	0	1	0
13	madewith	PK__madewith__B4D76506BF27D8A2	1	1	itemid	439672614	PK__madewith__B4D76506BF27D8A2	1	1	CLUSTERED	1	1	0	1	0
14	madewith	PK__madewith__B4D76506BF27D8A2	1	2	ingredientid	439672614	PK__madewith__B4D76506BF27D8A2	1	1	CLUSTERED	1	1	0	1	0
15	meal	PK__meal__EC3090F253F73257	1	1	meal_id	951674438	PK__meal__EC3090F253F73257	1	1	CLUSTERED	1	1	0	1	0
16	meal	PK__meal__EC3090F253F73257	1	2	item_id	951674438	PK__meal__EC3090F253F73257	1	1	CLUSTERED	1	1	0	1	0
17	meals	PK__meals__0D6F98926B28EEB4	1	1	mealid	519672899	PK__meals__0D6F98926B28EEB4	1	1	CLUSTERED	1	1	0	1	0
18	meals	UQ__meals__72E12F1B50744E59	2	1	name	519672899	UQ__meals__72E12F1B50744E59	2	2	NONCLUSTERED	1	1	0	0	1
19	orders	PK__orders__AC3A013714FAF577	1	1	ordernumber	695673526	PK__orders__AC3A013714FAF577	1	1	CLUSTERED	1	1	0	1	0
20	orders	PK__orders__AC3A013714FAF577	1	2	linenumber	695673526	PK__orders__AC3A013714FAF577	1	1	CLUSTERED	1	1	0	1	0
21	orders	PK__orders__AC3A013714FAF577	1	3	storeid	695673526	PK__orders__AC3A013714FAF577	1	1	CLUSTERED	1	1	0	1	0
22	partof	PK__partof__3805BA5B606A6A67	1	1	mealid	567673070	PK__partof__3805BA5B606A6A67	1	1	CLUSTERED	1	1	0	1	0
23	partof	PK__partof__3805BA5B606A6A67	1	2	itemid	567673070	PK__partof__3805BA5B606A6A67	1	1	CLUSTERED	1	1	0	1	0
24	projects	PK__projects__11EC39DD468F93FA	1	1	projectid	71671303	PK__projects__11EC39DD468F93FA	1	1	CLUSTERED	1	1	0	1	0
25	stores	PK__stores__01A2160BE3BEE3F1	1	1	storeid	663673412	PK__stores__01A2160BE3BEE3F1	1	1	CLUSTERED	1	1	0	1	0
26	vendors	PK__vendors__EC64C0BBFE136FBE	1	1	vendorid	247671930	PK__vendors__EC64C0BBFE136FBE	1	1	CLUSTERED	1	1	0	1	0
27	vendors	UQ__vendors__A871024CCE2663B0	2	1	repfname	247671930	UQ__vendors__A871024CCE2663B0	2	2	NONCLUSTERED	1	1	0	0	1
28	vendors	UQ__vendors__A871024CCE2663B0	2	2	replname	247671930	UQ__vendors__A871024CCE2663B0	2	2	NONCLUSTERED	1	1	0	0	1
29	workson	PK__workson__002B3674CFFEDA78	1	1	employeeid	135671531	PK__workson__002B3674CFFEDA78	1	1	CLUSTERED	1	1	0	1	0
30	workson	PK__workson__002B3674CFFEDA78	1	2	projectid	135671531	PK__workson__002B3674CFFEDA78	1	1	CLUSTERED	1	1	0	1	0

Modifying an Index

You modify an index using ALTER INDEX. The following example renames the i_employees_phone index to i_employees_phone_number:

```
EXEC sp_rename N'employees.i_func_employees_lastname', N'new_index_name' , N'INDEX';
```

Dropping an Index

You drop an index using the DROP INDEX statement. The following example drops the I_employees_phone_number index:

```
        DROP INDEX employees.i_func_employees_lastname;
```

Consider the following table for the following exercise.

```
          create table student_info(
             first_name varchar(20),
             last_name varchar(20),
             id varchar(20),
             age numeric(2)
          );
```

Exercise:

1. Create a unique composite non-clustered index on the first_name and order in the descending order.
2. Create a composite clustered index on (first_name,last_name) and order both in the increasing order.
3. Create a unique Clustered index on the (age,order) in the increasing order.
4. Create composite non-clustered index on the (id,age) both in the increasing order.
5. Create a non-clustered unique function based index on the upper(first_name + ' ' + lastname).

****