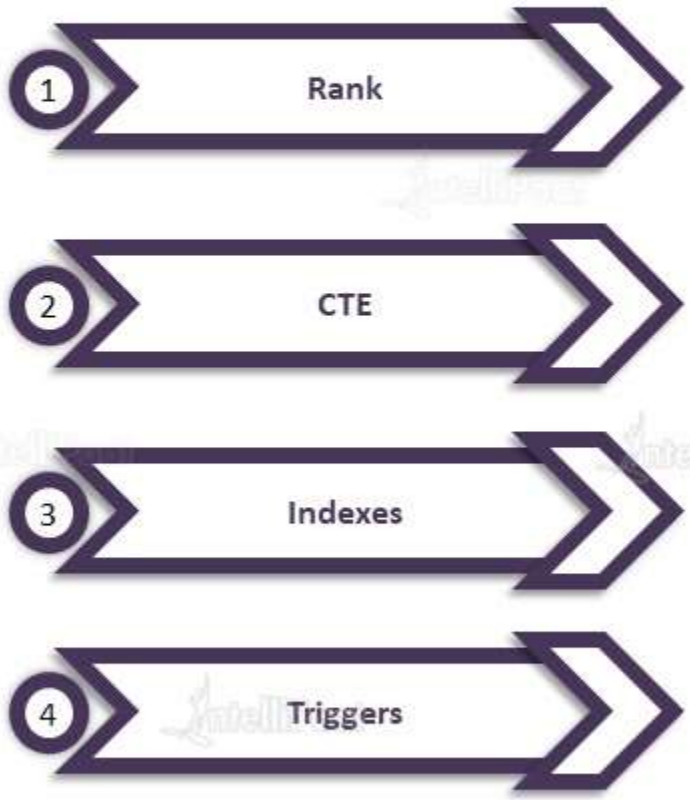
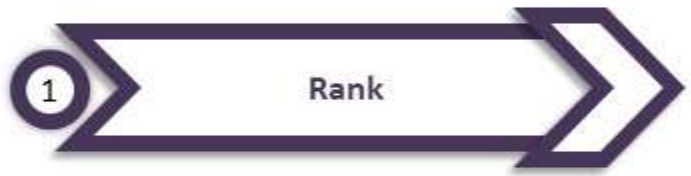
SQL Tutorial

SQL

Agenda



Agenda



Rank Functions

Ranking functions return a ranking value for each row in a partition. Depending on the function that is used, some rows might receive the same value as other rows.

Ranking functions are nondeterministic.

- **RANK**
- **DENSE RANK**
- **NTILE**
- **ROW NUMBER**

RANK (Transact-SQL)

For example:

If the two top salespeople have the same SalesYTD value,
they are both ranked one.

The salesperson with the next highest SalesYTD is ranked
number three, because there are two rows that are ranked higher.
Therefore, the RANK function does not always return consecutive integers.

DENSE RANK (Transact-SQL)

The rank of a row is one plus the number of distinct ranks that come before the row in question.

NTILE()

Distributes the rows in an ordered partition
into a specified number of groups.
It divides the partitioned result set into specified
number of groups in an order.

Example for NTILE(2)

```
select Name,Subject,Marks,  NTILE(2)
over(partition  by name order by Marks desc)
Quartile  From ExamResult order by name,subject
```

ROW NUMBER()

Example for ROW NUMBER()

```
select Name,Subject,Marks,
ROW_NUMBER
over(order by Name)
RowNumber From
ExamResult
order by name,subject
```

Agenda

CTE (Common Table Expressions)

The Common Table Expressions (CTE) were introduced into standard SQL in order to simplify various classes of SQL Queries for which a derived table was just unsuitable.


```
SELECT * FROM ( SELECT A.Address, E.Name, E.Age
From Address A Inner join Employee E on E.EID =
A.EID) T
WHERE T.Age > 50 ORDER BY T.NAME
```

Continued.....

CTE (Common Table Expressions)

CTE allows you to define the subquery at once, name it using an alias and later call the same data using the alias just like what you do with a normal table.

```
With T(Address, Name, Age) --Column names for Temporary table
AS ( SELECT A.Address, E.Name, E.Age from Address A INNER JOIN
EMP E ON E.EID = A.EID )
SELECT * FROM T --SELECT or USE CTE temporary Table
WHERE T.Age > 50 ORDER BY T.NAME
```

Multiple CTE

To use multiple CTE's in a single query you just need to finish the first CTE, add a comma, declare the name and optional columns for the next CTE, open the CTE query with a comma, write the query, and access it from a CTE query later in the same query or from the final query outside the CTEs.

```
With TI (Address, Name, Age) --Column names for Temporary table
AS ( SELECT A.Address, E.Name, E.Age from Address A INNER JOIN EMP E ON E.EID = A.EID )
T2(Name, Desig)
AS
( SELECT NAME, DESIG FROM Designation)
SELECT T1.*, T2.Desig FROM T1 --SELECT or USE CTE temporary Table WHERE T1.Age > 50 AND TI .Name = T2.Name
ORDER BY T1.NAME
WITH ShowMessage(STATEMENT, LENGTH)
AS
( SELECT STATEMENT = CAST('I Like ' AS VARCHAR(300)), LEN('I Like ')
UNION ALL
SELECT CAST(STATEMENT + 'CodeProject! ' AS VARCHAR(300))
, LEN(STATEMENT) FROM ShowMessage WHERE LENGTH < 300 )
SELECT STATEMENT, LENGTH FROM ShowMessage
```

Agenda

Indexes

An INDEX can be defined as a mechanism for providing fast access to table rows and for enforcing constraints.

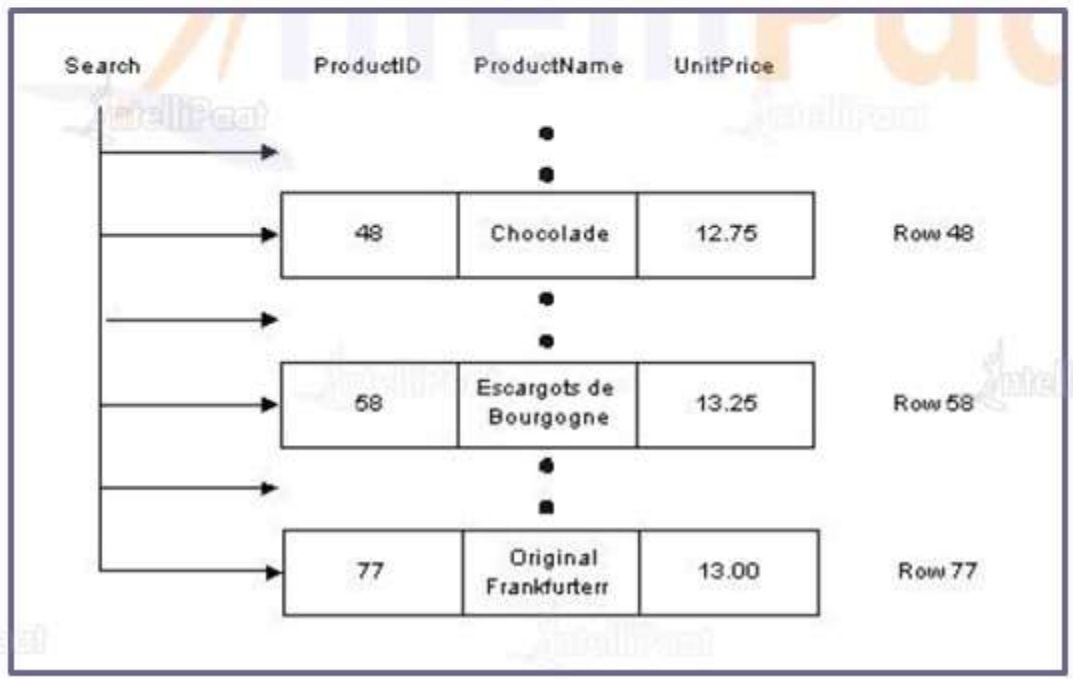
- When data volumes increase, organizations face the problems related to data retrieval and posting. They feel the need for a mechanism that will increase the speed of data access.
- An index, like the index of a book, enables the database retrieve and present data to the end user with ease.
- When a SQL Server has NO index to use for searching, the result is similar to the reader who looks at every page in a book to find a word: the SQL engine needs to visit every row in a table.
- In database terminology we call this behaviour a **TABLE SCAN**, or just **SCAN**.

With No Indexes Defined ...

Consider the following query on the Products table (with NO index of the Northwind database. This query retrieves products in a specific price range (12.5 — 14).

```
SELECT ProductID, ProductName, UnitPrice FROM Products WHERE (UnitPrice > 12.5) AND (UnitPrice < 14)
```

In the diagram below, the database search touches a total of 77 records to find just three matches



With No Indexes Defined ...

- Now imagine if we CREATED an index, just like a book index, on the data in the UnitPrice column.
- Each index entry would contain a copy of the UnitPrice value for a row, and a reference (just like a page number) to the row where the value originated.
- SQL will sort these index entries into ascending order.
- The index will allow the database to quickly narrow in on the three rows to satisfy the query, and avoid scanning every row in the table.

Creating Indexes

The command specifies the name of the index (IDX_UnitPrice), the table name (Products), and the column to index (UnitPrice).

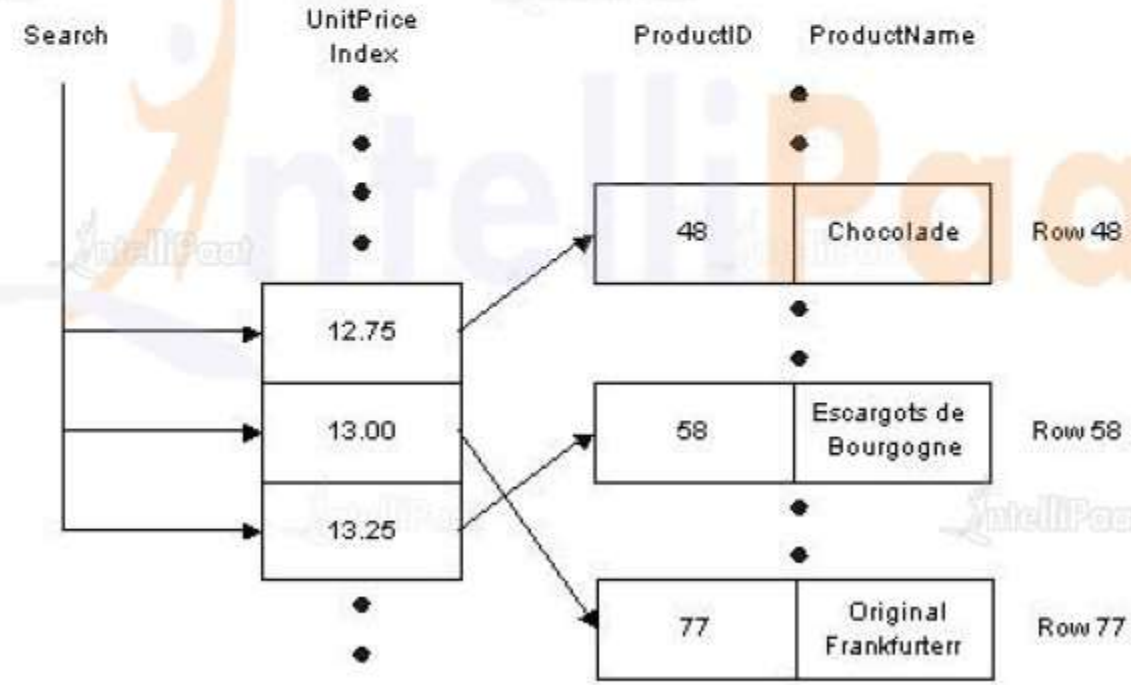
```
CREATE INDEX IIDX_UnitPrice] ON Products (UnitPrice)
```

To verify that the index is created, use the following stored procedure to see a list of all indexes on the Products table:

```
EXEC sp_helpindex Products
```

How It Works

The database takes the columns specified in a **CREATE INDEX** command and sorts the values into a special data structure known as a B-tree.



How It Works

- On the left, each index entry contains the index key **(UnitPrice)**.
- Each entry also includes a reference (which points) to the table rows which share that particular value and from which we can retrieve the required information.
- Much like the index in the back of a book helps us to find keywords quickly, so the database is able to quickly narrow the number of records it must examine to a minimum by using the sorted list of **UnitPrice** values stored in the index.
- We have avoided a table scan to fetch the query results

Index Advantages

- The database engine can use indexes to boost performance in a number of different queries.
- An important feature of versions SQL Server 2005 and above is a component known as the query optimizer.
- The query optimizer's job is to find the fastest and least resource intensive means of executing incoming queries.
- An important part of this job is selecting the best index or indexes to perform the task.

Searching for Records

The most obvious use for an index is in finding a record or set of records matching a WHERE clause.

By way of example, the following queries can all benefit from an index on **UnitPrice**:

```
DELETE FROM Products WHERE UnitPrice = 1
UPDATE Products SET Discontinued = 1 WHERE UnitPrice > 15
SELECT * FROM PRODUCTS WHERE UnitPrice BETWEEN 14 AND 16
```

Indexes work just as well when searching for a record in DELETE and UPDATE commands as they do for SELECT statements.

Sorting Records

When we ask for a sorted dataset, the database will try to find an index and avoid sorting the results during execution of the query.

We control sorting of a dataset by specifying a field, or fields, in an ORDER BY clause, with the sort order as ASC (ascending) or DESC (descending)


```
SELECT * FROM Products ORDER BY UnitPrice ASC
```

The database can simply scan the index from the first entry to the last entry and retrieve the rows in sorted order.
The same index works equally well with the following query, simply by scanning the index in reverse.

```
SELECT * FROM Products ORDER BY UnitPrice DESC
```

Grouping Records

We can use a **GROUP BY** clause to group records and aggregate values, for example, counting the number of orders placed by a customer.

To process a query with a GROUP BY clause, the database will often sort the results on the columns included in the GROUP BY.

SELECT Count(*), UnitPrice FROM Products GROUP BY UnitPrice

The database can use the IDX UnitPrice index to retrieve the prices in order.

- ✓ **Since matching prices appear in consecutive index entries, the database is able count the number of products at each price quickly.**

Clustered Indexes

Notice here in the "student" table we have set primary key constraint on the "id" column. This automatically creates a clustered index on the "id" column. To see all the indexes on a particular table execute "sp_helpindex" stored procedure. This stored procedure accepts the name of the table as a parameter and retrieves all the indexes of the table. The following query retrieves the indexes created on student table.

```
USE schooldb

EXECUTE sp_helpindex student
```

The above query will return this result:

index_name	index_description	index_keys
PK__student__3213E83F7F60ED59	clustered, unique, primary key located on PRIMARY	id

Clustered Indexes

A clustered index defines the order in which data is physically stored in a table. Table data can be sorted in only way, therefore, there can be only one clustered index per table. In SQL Server, the primary key constraint automatically creates a clustered index on that particular column.

Let's take a look. First, create a "student" table inside "schooldb" by executing the following script:

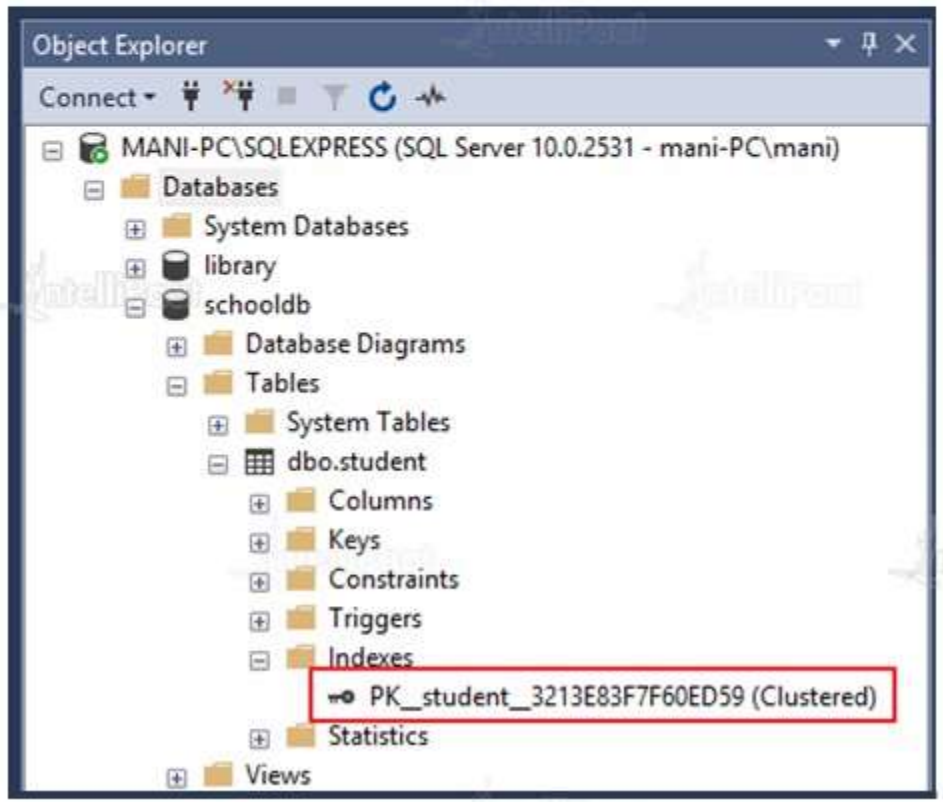
```
CREATE DATABASE schooldb

CREATE TABLE student
(
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    gender VARCHAR(50) NOT NULL,
    DOB datetime NOT NULL,
    total_score INT NOT NULL,
    city VARCHAR(50) NOT NULL
)
```

Notice here in the "student" table we have set primary key constraint on the "id" column. This automatically creates a clustered index on the "id" column. To see all the indexes on a particular table execute "sp_helpindex" stored procedure. This stored procedure accepts the name of the table as a parameter and retrieves all the indexes of the table. The following query retrieves the indexes created on student table.

Clustered Indexes

In the output you can see the only one index. This is the index that was automatically created because of the primary key constraint on the "**id**" column.

Another way to view table indexes is by going to "Object Explorer-> Databases-> Database_Name-> Tables-> Table_Name -> Indexes". Look at the following screenshot for reference.



Clustered Indexes

This clustered index stores the record in the student table in the ascending order of the “id”. Therefore, if the inserted record has the id of 5, the record will be inserted in the 5th row of the table instead of the first row. Similarly, if the fourth record has an id of 3, it will be inserted in the third row instead of the fourth row. This is because the clustered index has to maintain the physical order of the stored records according to the indexed column i.e. id. To see this ordering in action, execute the following script:

```
USE schooldb

INSERT INTO student

VALUES
(6, 'Kate', 'Female', '03-JAN-1985', 500, 'Liverpool'),
(2, 'Jon', 'Male', '02-FEB-1974', 545, 'Manchester'),
(9, 'Wise', 'Male', '11-NOV-1987', 499, 'Manchester'),
(3, 'Sara', 'Female', '07-MAR-1988', 600, 'Leeds'),
(1, 'Jolly', 'Female', '12-JUN-1989', 500, 'London'),
(4, 'Laura', 'Female', '22-DEC-1981', 400, 'Liverpool'),
(7, 'Joseph', 'Male', '09-APR-1982', 643, 'London'),
(5, 'Alan', 'Male', '29-JUL-1993', 500, 'London'),
(8, 'Mice', 'Male', '16-AUG-1974', 543, 'Liverpool'),
(10, 'Elis', 'Female', '28-OCT-1990', 400, 'Leeds');
```

Clustered Indexes

The above script inserts ten records in the student table. Notice here the records are inserted in random order of the values in the “id” column. But because of the default clustered index on the id column, the records are physically stored in the ascending order of the values in the “id” column. Execute the following SELECT statement to retrieve the records from the student table.

```
USE schooldb

SELECT * FROM student
```

The records will be retrieved in the following order:

Clustered Indexes

The records will be retrieved in the following order:

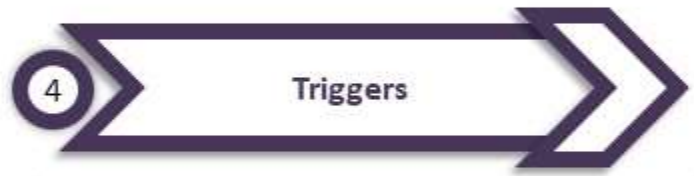
id	name	gender	DOB	total_score	city
1	Jolly	Female	1989-06-12 00:00:00.000	500	London
2	Jon	Male	1974-02-02 00:00:00.000	545	Manchester
3	Sara	Female	1988-03-07 00:00:00.000	600	Leeds
4	Laura	Female	1981-12-22 00:00:00.000	400	Liverpool
5	Alan	Male	1993-07-29 00:00:00.000	500	London
6	Kate	Female	1985-01-03 00:00:00.000	500	Liverpool
7	Joseph	Male	1982-04-09 00:00:00.000	643	London
8	Mice	Male	1974-08-16 00:00:00.000	543	Liverpool
9	Wise	Male	1987-11-11 00:00:00.000	499	Manchester
10	Elis	Female	1990-10-28 00:00:00.000	400	Leeds

Composite Indexes

A composite index is an index on two or more columns.

Both clustered and non-clustered indexes can be composite indexes. Composite indexes are especially useful in two different circumstances.

Agenda



Triggers

A trigger is a special kind of stored procedure that automatically executes when an event occurs in the database server. **DML** triggers execute when a user tries to modify data through a data manipulation language (DML) event. DML events are **INSERT**, **UPDATE**, or **DELETE** statements on a table or view

CREATE TRIGGER Statement

The **CREATE TRIGGER** statement defines a trigger in the database.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- **ALTER** privilege on the table on which the **BEFORE** or **AFTER** trigger is defined
- **CONTROL** privilege on the view on which the **INSTEAD OF TRIGGER** is defined
- Definer of the view on which the **INSTEAD OF** trigger is defined
- **ALTERIN** privilege on the schema of the table or view on which the trigger is defined
- **SYSADM** or **DBADM** authority and one of:
  - **IMPLICIT_SCHEMA** authority on the database, if the implicit or explicit schema name of the trigger does not exist
  - **CREATEIN** privilege on the schema, if the schema name of the trigger refers to an existing schema

Types of Triggers

There are three main types of triggers that fire on:

- INSERT
- DELETE
- UPDATE

actions. Like stored procedures, these can also be encrypted for extra security.

Syntax of a Trigger

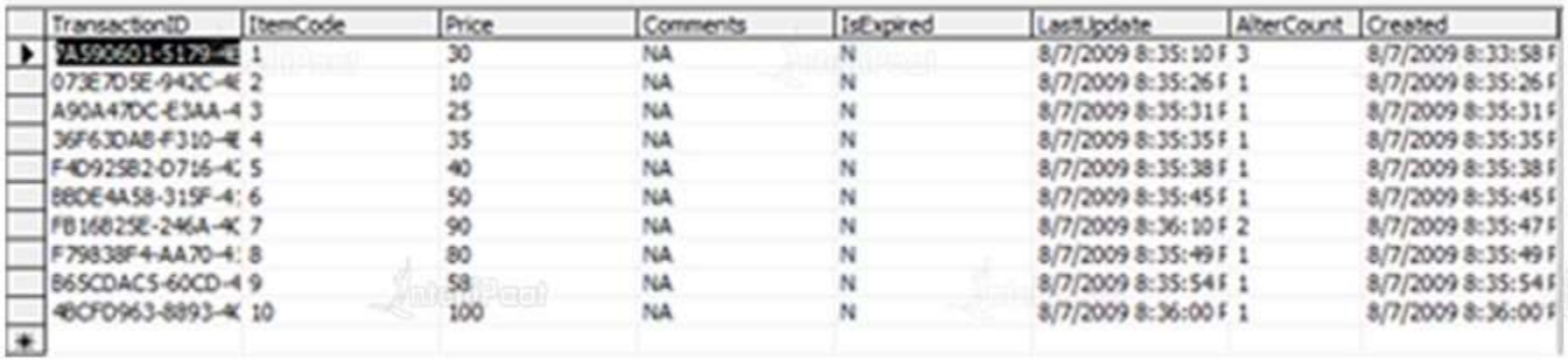
```
CREATE TRIGGER name ON table
  [WITH ENCRYPTION]
  [FOR/AFTER/INSTEAD OF]
  [INSERT, UPDATE, DELETE]
  [NOT FOR REPLICATION]
AS
BEGIN
--SQL statements
...
END
```

After observing its Syntax, now let's take an example.

We have a table with some columns. Our goal is to create a TRIGGER which will be fired on every modification of data in each column and track the number of modifications of that column. The sample example is given below:

```
-- ================================================
-- Author: Md. Marufuzzaman
-- Create date:
-- Description: Alter count for any modification
-- ================================================
CREATE TRIGGER [TRIGGER_ALTER_COUNT] ON [dbo].[tblTriggerExample]
FOR INSERT, UPDATE
AS
BEGIN
DECLARE @TransID VARCHAR(36)
 SELECT @TransID = TransactionID FROM INSERTED
 UPDATE [dbo].[tblTriggerExample] SET AlterCount = AlterCount + 1
       ,LastUpdate = GETDATE()
   WHERE TransactionID = @TransID
END
```

Syntax of a Trigger: Example Output



TransactionID	ItemCode	Price	Comments	IsExpired	LastUpdate	AlterCount	Created
7A590601-5179-4	1	30	NA	N	8/7/2009 8:35:10 F	3	8/7/2009 8:33:58 F
073E7D5E-942C-4	2	10	NA	N	8/7/2009 8:35:26 F	1	8/7/2009 8:35:26 F
A90A47DC-E3AA-4	3	25	NA	N	8/7/2009 8:35:31 F	1	8/7/2009 8:35:31 F
36F63DAB-F310-4	4	35	NA	N	8/7/2009 8:35:35 F	1	8/7/2009 8:35:35 F
F4D925B2-D716-4	5	40	NA	N	8/7/2009 8:35:38 F	1	8/7/2009 8:35:38 F
88DE4A58-315F-4	6	50	NA	N	8/7/2009 8:35:45 F	1	8/7/2009 8:35:45 F
FB16825E-246A-4	7	90	NA	N	8/7/2009 8:36:10 F	2	8/7/2009 8:35:47 F
F79838F4-AA70-4	8	80	NA	N	8/7/2009 8:35:49 F	1	8/7/2009 8:35:49 F
B65CDAC5-60CD-4	9	58	NA	N	8/7/2009 8:35:54 F	1	8/7/2009 8:35:54 F
4BCFD963-8893-4	10	100	NA	N	8/7/2009 8:36:00 F	1	8/7/2009 8:36:00 F

Quiz

Quiz

IntelliPaat

What is the full form of CTE?

A	Common Transaction Expression
B	Common Transaction Example
C	Collaborative Transaction Expression
D	All of the above

Answer

What is the full form of CTE?

A	Common Transaction Expression
B	Common Transaction Example
C	Collaborative Transaction Expression
D	All of the above

Quiz

Which statement is correct about CTE?

A	It allows you to create and delete the table
B	It allows you to adjust your table length
C	It allows you to define the subquery at once
D	All of the above

Answer

Which statement is correct about CTE?

A	It allows you to create and delete the table
B	It allows you to adjust your table length
C	It allows you to define the subquery at once
D	All of the above

Quiz

Which of the following is the simplest ranking function ?

A	RANK
B	NTILE
C	ROW_NUMBER
D	None of the mentioned

Answer

Which of the following is the simplest ranking function ?

A	RANK
B	NTILE
C	ROW_NUMBER
D	None of the mentioned

Quiz

Which of the clause is not mandatory ?

A	OVER Clause
B	ORDER BY Clause
C	PARTITION BY clause
D	All of the above

Answer

Which of the clause is not mandatory ?

A	OVER Clause
B	ORDER BY Clause
C	PARTITION BY clause
D	All of the above

Quiz

Which of the following functions are similar ?

A	RANK and NTILE
B	RANK and DENSE_RANK
C	DENSE_RANK and NTILE
D	None of the mentioned

Answer

Which of the following functions are similar ?

A	RANK and NTILE
B	RANK and DENSE_RANK
C	DENSE_RANK and NTILE
D	None of the mentioned

Thank You