

SQL-Optimization and Performance



PIVOT

PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output. And PIVOT runs aggregations where they're required on any remaining column values that are wanted in the final output.

Syntax:

```
SELECT <non-pivoted column>, [first pivoted column] AS <column name>,
[second pivoted column] AS <column name>, ...
[last pivoted column] AS <column name>
FROM
(<SELECT query that produces the data>) AS <alias for the source
query> PIVOT ( <aggregation function>(<column being aggregated>)
FOR
<column that contains the values that will become column headers>
IN ( [first pivoted column], [second pivoted column],
... [last pivoted column])
) AS <alias for the pivot table>
<optional ORDER BY clause>;
```

PIVOT

Create a pivot table that depicts the average fuel prices sorted by first five stores

```
SELECT 'AverageFuelPrice' AS Price_sorted_by_store,  
[1], [2], [3], [4],[5] into Pivot_table  
FROM ( SELECT Store, Fuel_Price  
FROM Feature) AS SourceTable PIVOT  
( AVG(Fuel_Price)  
FOR store IN ([1],[2],[3],[4],[5] ) AS PivotTable;
```

Price_sorted_by_store	1	2	3	4	5
AverageFuelPrice	3.24347928995211	3.24347928995211	3.24347928995211	3.2409940852216	3.24347928995211

UNPIVOT

UNPIVOT carries out the opposite operation to PIVOT by rotating columns of a table-valued expression into column values.

Reverse the pivoting mentioned in the previous question using the unpivot operator and display the records.

```
SELECT Store, AverageFuelPrice  
into UnPivot_Table  
FROM  
(SELECT Price_sorted_by_store,  
[1],[2],[3],[4],[5]  
FROM Pivot_table) u  
UNPIVOT  
(AverageFuelPrice FOR Store IN  
([1],[2],[3],[4],[5])  
AS unpvt;
```

Store	AverageFuelPrice
1	3.24347928995211
2	3.24347928995211
3	3.24347928995211
4	3.2409940852216
5	3.24347928995211

STUFF

The STUFF function inserts a string into another string. It deletes a specified length of characters in the first string at the start position and then inserts the second string into the first string at the start position.

By using STUFF effectively, you can handle complex string manipulations directly within your SQL queries, streamlining data processing tasks.

Syntax:

*STUFF (character_expression ,
start , length , replacewith_expression)*

```
SELECT Store_Name, STUFF('Walmart',  
7, 8, 'store') as Stuffed_column from  
Store_details
```

Store_Name	Stuffed_column
Walmart	Walmarstore

Indexes

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

SQL indexes are a critical aspect of database performance optimization. They improve the speed of data retrieval operations on a database table at the cost of additional storage and slower write operations.

Let's dive into the details: An index is a database object that enhances the speed of data retrieval by providing quick access to rows in a table based on the values in one or more columns.

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)

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

Search	ProductID	ProductName	UnitPrice	
	48	Chocolade	12.75	Row 48
	58	Escargots de Bourgogne	13.25	Row 58
	77	Original Frankfurter	13.00	Row 77

Copyright IntelliPaat. All rights reserved

With No Indexes Defined..

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).

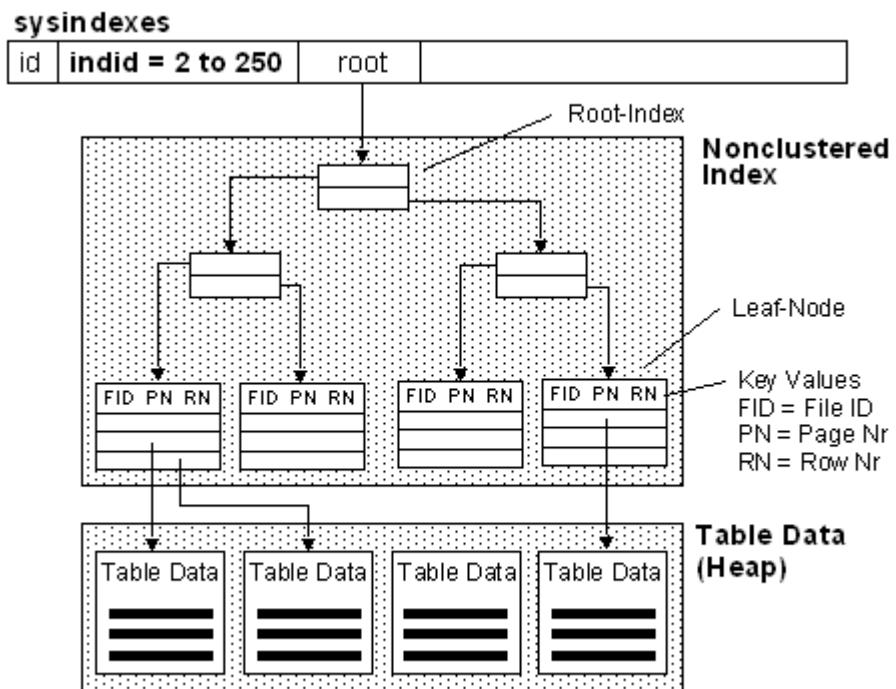
```
CREATE INDEX IDX_UnitPrice] ON Products (UnitPrice)
```

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.



Indexes can aid queries looking for values inside of a range (as we demonstrated earlier), as well as queries looking for a specific value.

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 for 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.



The following query counts the number of products at each price by grouping together records with the same UnitPrice value.

```
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

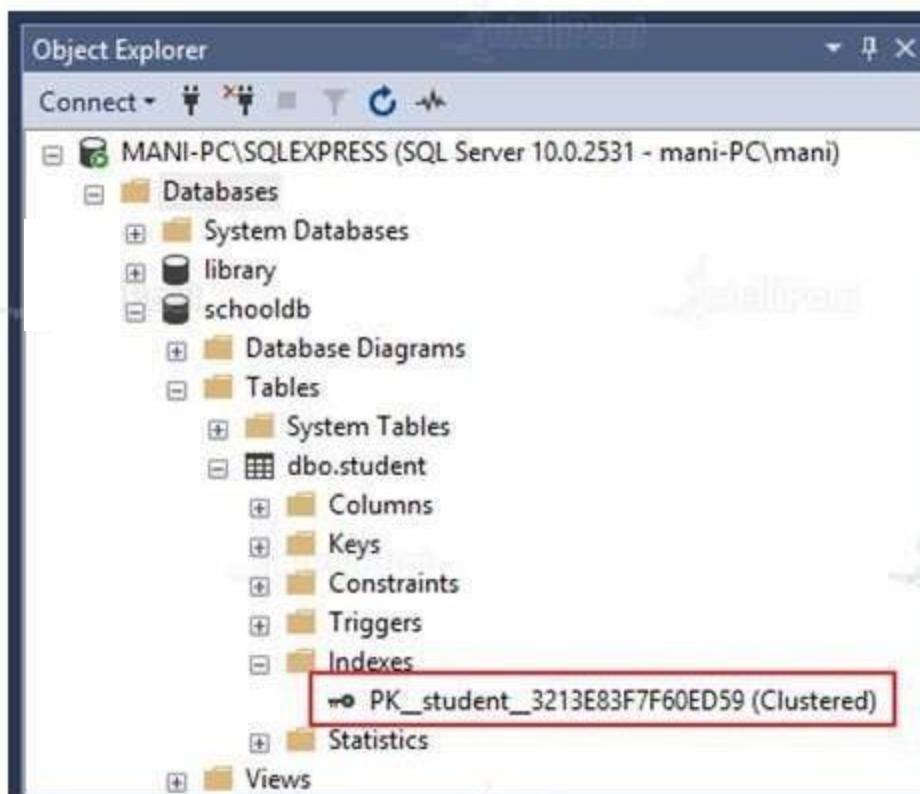
CREATE TABLE student
(
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    gender VARCHAR(50) NOT NULL,
    DOB datetime NOT NULL,
    total_score INT NOT NULL,
    city VARCHAR(50) NOT NULL
)
```

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

Clustered Index

It defines the order in which the table data will be sorted and stored. Defining a clustered index on a column will sort the data based on the column values and will store it. Thus, it helps in faster retrieval of the data.

Organize the data of the store table using cluster index

Syntax:

```
CREATE CLUSTERED INDEX <index_name>
ON
[schema.]<table_name>(column_name
[asc|desc]);
```

```
CREATE CLUSTERED INDEX
CIX_store_storeid
ON dbo.store(store_id)
select * from store
```

Store_id	Accessories	ExpiryDate
8	Laptop	NULL
5	Laptop	NULL
1	Laptop	NULL
3	Laptop	NULL
6	Laptop	NULL
2	Laptop	NULL
4	Laptop	NULL
7	Laptop	NULL

Unsorted data

Store_id	Accessories	ExpiryDate
1	Laptop	NULL
2	Laptop	NULL
3	Laptop	NULL
4	Laptop	NULL
5	Laptop	NULL
6	Laptop	NULL
7	Laptop	NULL
8	Laptop	NULL

Sorted data

Non-Clustered Index

Data is stored in one place and index is stored in another place same as an index of a book

Apply non cluster index on store_name of the given below store table

Store_id	store_name	Sales
1	walmart	25000
2	titan	18000
3	h and m	11000
4	Amazon	75000
5	Wipro	95000
6	TCS	67000
7	Infosys	55000
8	abc	15000

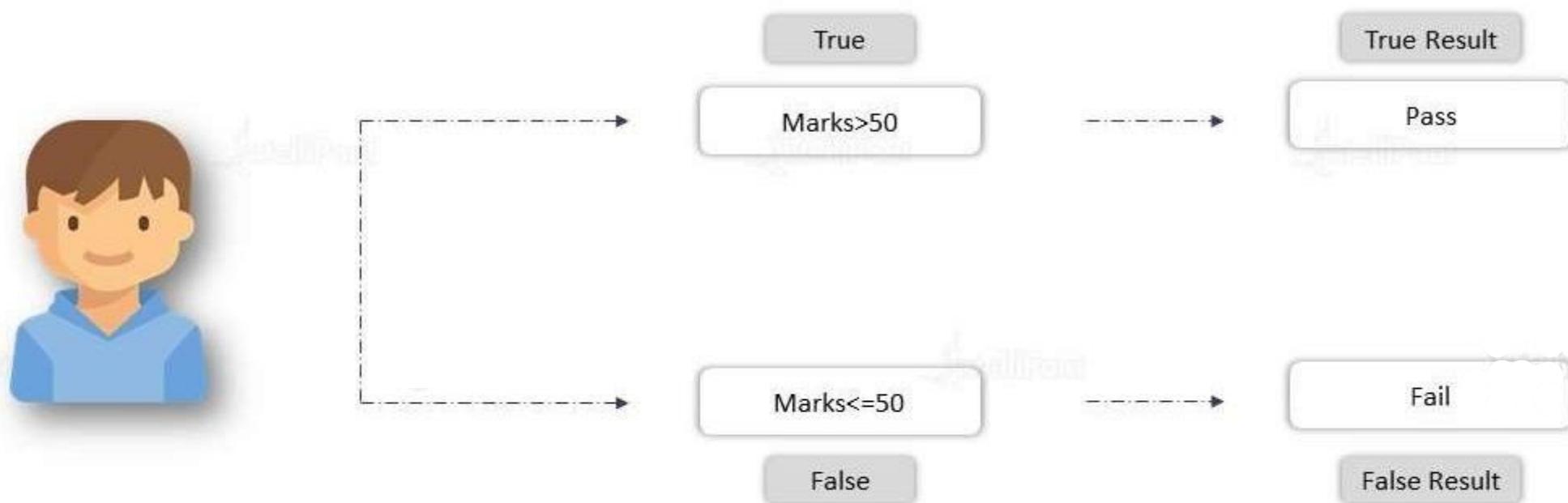
Syntax:

***CREATE NONCLUSTERED INDEX
<index_name> ON <table_name>(column)***

```
create nonclusteredindex nci_storefinal  
on store10(Store_name Asc);  
sp_helpindex store10
```

index_name	index_description	index_keys
nci_storefinal	nonclustered located on PRIMARY	store_name

IIF Condition



IIF Condition

IIF() function is an alternative for the case statement.

```
IIF (boolean_expression, true_value,  
false_value )
```

IIF Condition

Create a Store_Status column in the Sales to check whether the Store is open or closed.

IIF()

The IIF condition accepts three parameters.

- 1) CONDITION
- 2) TRUE
- 3) FALSE

Syntax:

IIF (Condition, TRUE statement, FALSE statement)

```
Select *, IIF(IsHoliday = 'TRUE',  
'Open','Closed') as Store_Status  
from Sales
```

	Store	Dept	Date	Weekly_Sales	IsHoliday	Store_Status
1	8	5	05/08/2011	16183.33	FALSE	Closed
2	8	5	12/08/2011	16067.32	FALSE	Closed
3	8	5	19/08/2011	15966.62	FALSE	Closed
4	8	5	26/08/2011	11533.31	FALSE	Closed
5	8	5	02/09/2011	13594.54	FALSE	Closed
6	8	5	09/09/2011	12781.84	TRUE	Open
7	8	5	16/09/2011	15666.95	FALSE	Closed
8	8	5	23/09/2011	15107.2	FALSE	Closed
9	8	5	30/09/2011	16594.95	FALSE	Closed
10	8	5	07/10/2011	22372.88	FALSE	Closed
11	8	5	14/10/2011	18907.09	FALSE	Closed

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.



- ✓ First, you can use a composite index to cover a query.
- ✓ Secondly, you can use a composite index to help match the search criteria of specific queries. We will go onto more detail and give examples of these two areas in the following sections.

Purpose of correlated subqueries



Sometimes you have to answer more than one question in one sentence

Multiple Questions

One Statement

Your friend might ask you if you have enough money for a cinema ticket, popcorn, and a drink



Before you can answer your friend, you need to know the prices of the ticket, the popcorn, and the drink.

Prerequisites



So actually, what seemed like an easy question, turns into four questions that you need answers to before you can say "Yes" or "No."

Safety



You also need to see how much money you have in your pocket.

Purpose of correlated subqueries

In business, you might get asked to produce a report of all employees earning more than the average salary for their departments.

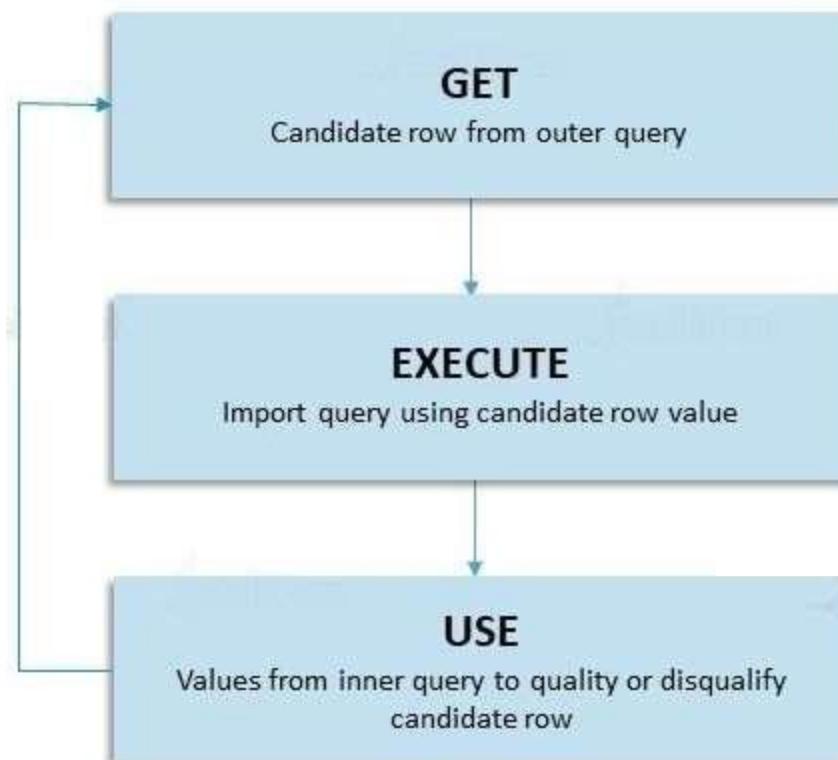
So here you first have to calculate the average salary per department, and then compare the salary for each employee to the average salary of that employee's department.



Correlated Subqueries

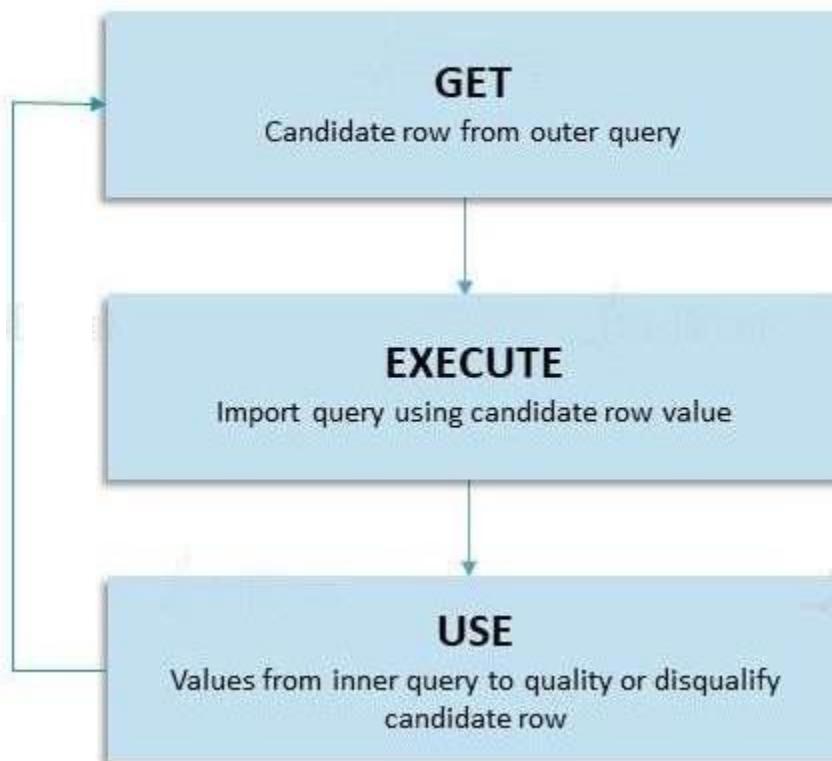
Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



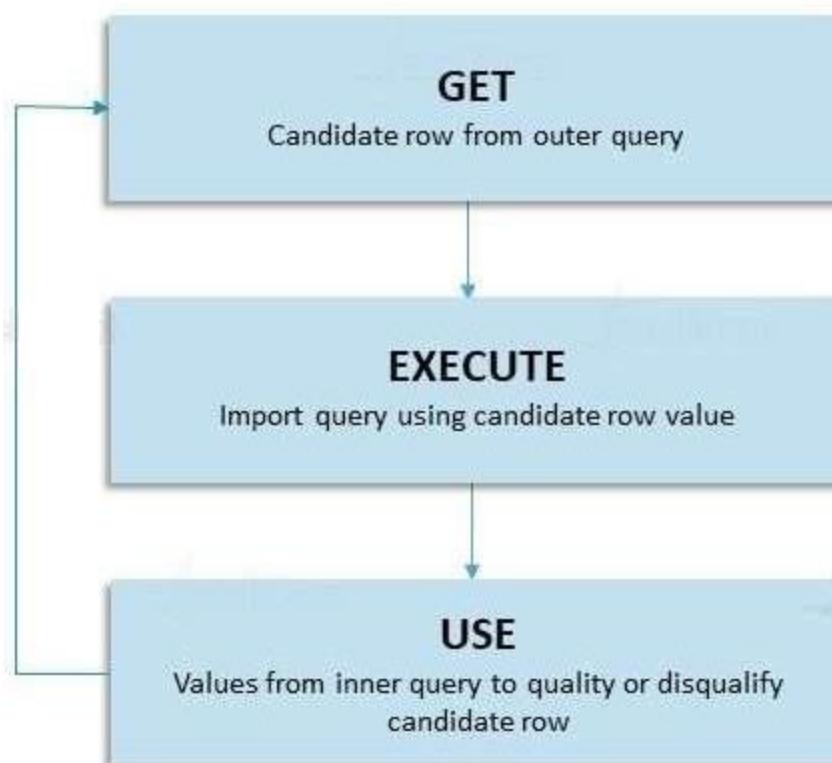
Correlated Subqueries

A correlated subquery is evaluated once for each row processed by the parent statement.

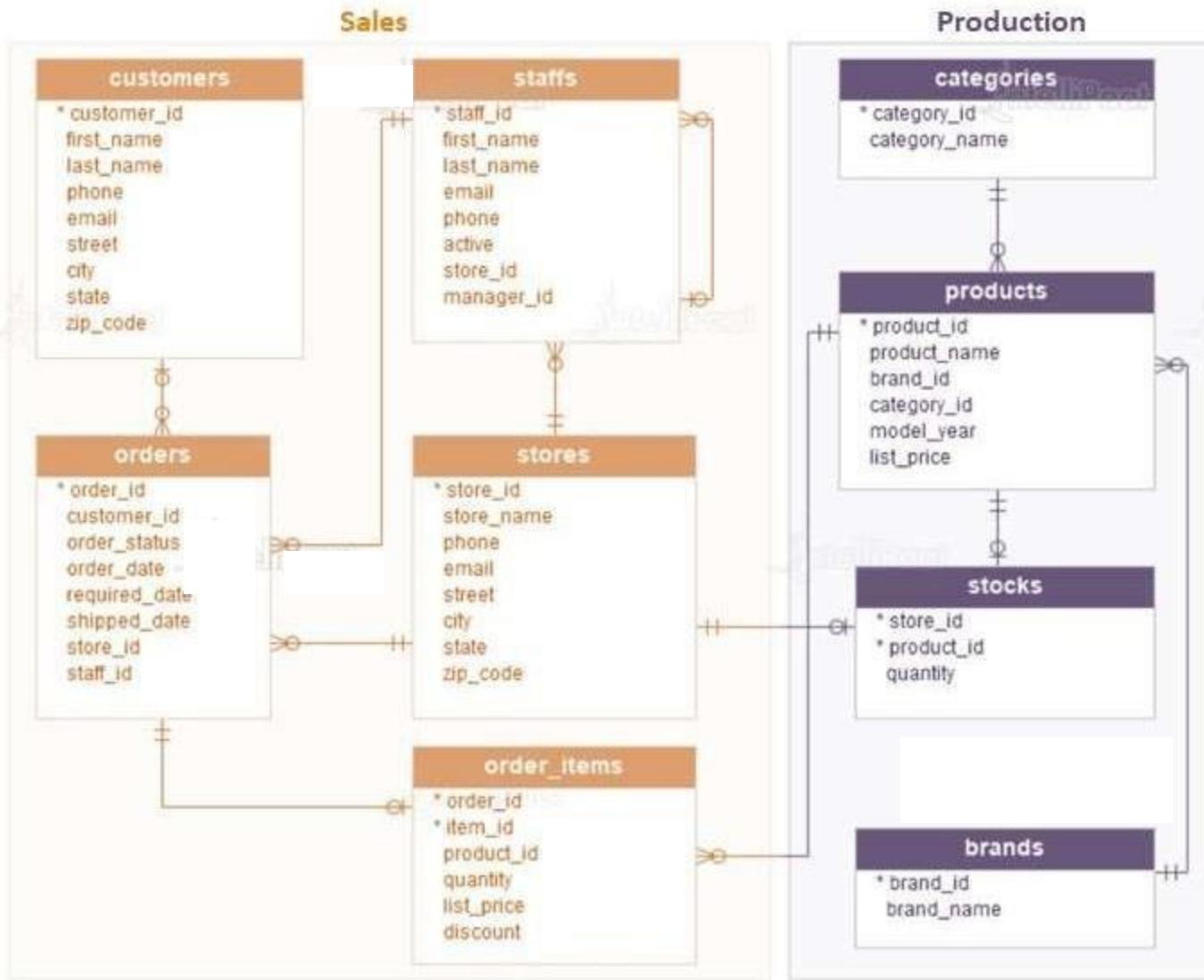


Correlated Subqueries

The parent statement can be a SELECT, UPDATE, or DELETE statement.



Correlated Subqueries: Example



Correlated Subqueries: Example

Consider the following products table from the sample database:

products

- * product_id
- product_name
- brand_id
- category_id
- model_year
- list_price

Correlated Subqueries: Example

The following example finds the products whose list price is equal to the highest list price of the products within the same category:

```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Correlated Subqueries: Example

The following example finds the products whose list price is equal to the highest list price of the products within the same category:

```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Output

	product_name	list_price	category_id
1	Electra Straight 8 3i (20-inch) - Boy's - 2017	489.99	1
2	Electra Townie 3i EQ (20-inch) - Boys' - 2017	489.99	1
3	Trek Superfly 24 - 2017/2018	489.99	1
4	Electra Townie Go! 8i - 2017/2018	2599.99	2
5	Electra Townie Commute Go! - 2018	2999.99	3
6	Electra Townie Commute Go! Ladies' - 2018	2999.99	3
7	Trek Boone 7 Disc - 2018	3999.99	4
8	Trek Powerfly 7 FS - 2018	4999.99	5
9	Trek Powerfly 8 FS Plus - 2017	4999.99	5
10	Trek Super Commuter+ 8S - 2018	4999.99	5
11	Trek Fuel EX 9.8 27.5 Plus - 2017	5299.99	6
12	Trek Remedy 9.8 - 2017	5299.99	6
13	Trek Domane SLR 9 Disc - 2018	11999.99	7

Correlated Subqueries: Example

For each product evaluated by the outer query, the subquery finds the highest price of all products in its category.

```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Output

	product_name	list_price	category_id
1	Electra Straight 8 3i (20-inch) - Boys' - 2017	489.99	1
2	Electra Townie 3i EQ (20-inch) - Boys' - 2017	489.99	1
3	Trek Superfly 24 - 2017/2018	489.99	1
4	Electra Townie Go! 8i - 2017/2018	2599.99	2
5	Electra Townie Commute Go! - 2018	2999.99	3
6	Electra Townie Commute Go! Ladies' - 2018	2999.99	3
7	Trek Boone 7 Disc - 2018	3999.99	4
8	Trek Powerfly 7 FS - 2018	4999.99	5
9	Trek Powerfly 8 FS Plus - 2017	4999.99	5
10	Trek Super Commuter+ 8S - 2018	4999.99	5
11	Trek Fuel EX 9.8 27.5 Plus - 2017	5299.99	6
12	Trek Remedy 9.8 - 2017	5299.99	6
13	Trek Domane SLR 9 Disc - 2018	11999.99	7

Correlated Subqueries: Example

If the price of the current product is equal to the highest price of all products in its category, the product is included in the result set. This process continues for the next product and so on.

```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Output

	product_name	list_price	category_id
1	Electra Straight 8 3i (20-inch) - Boys' - 2017	489.99	1
2	Electra Townie 3i EQ (20-inch) - Boys' - 2017	489.99	1
3	Trek Superfly 24 - 2017/2018	489.99	1
4	Electra Townie Go! 8i - 2017/2018	2599.99	2
5	Electra Townie Commute Go! - 2018	2999.99	3
6	Electra Townie Commute Go! Ladies' - 2018	2999.99	3
7	Trek Boone 7 Disc - 2018	3999.99	4
8	Trek Powerfly 7 FS - 2018	4999.99	5
9	Trek Powerfly 8 FS Plus - 2017	4999.99	5
10	Trek Super Commuter+ 8S - 2018	4999.99	5
11	Trek Fuel EX 9.8 27.5 Plus - 2017	5299.99	6
12	Trek Remedy 9.8 - 2017	5299.99	6
13	Trek Domane SLR 9 Disc - 2018	11999.99	7

Exists

Exists

The EXISTS operator is a logical operator that allows you to check whether a subquery returns any row.

The EXISTS operator returns TRUE if the subquery returns one or more rows.

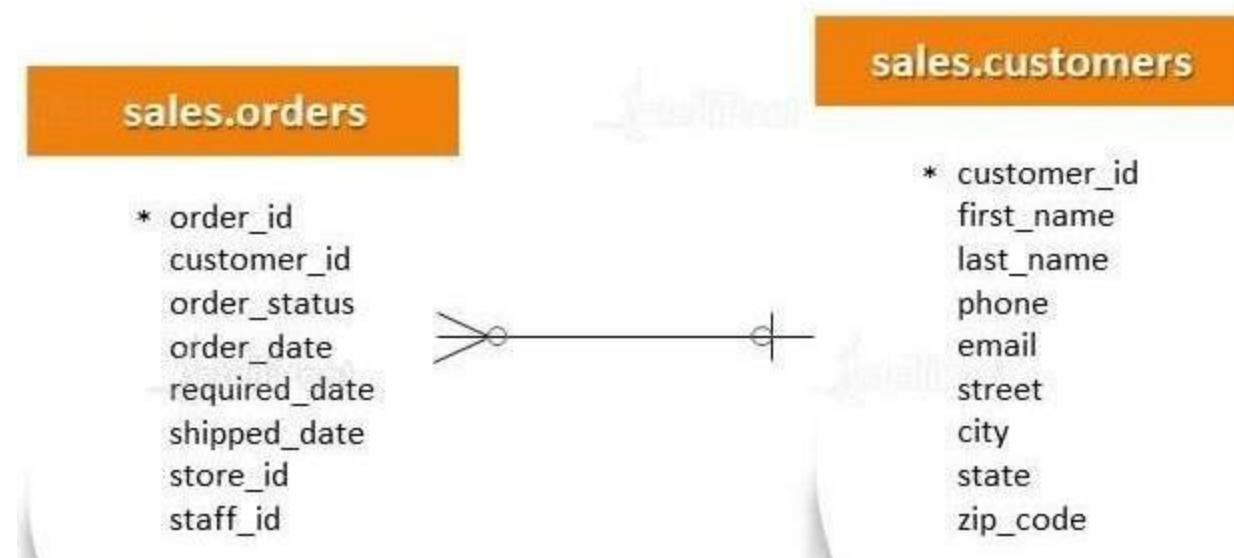
Syntax: EXISTS (subquery)

NOTE

In this syntax, the subquery is a SELECT statement only. As soon as the subquery returns rows, the EXISTS operator returns TRUE and stop processing immediately. Though the subquery returns a NULL value, the EXISTS operator is still evaluated to TRUE.

Using EXISTS with a correlated query

Consider the following customers and orders tables



Using EXISTS with a correlated query

This following example finds all customers who have placed more than two orders

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers c
WHERE
    EXISTS (
        SELECT
            COUNT (*)
        FROM
            sales.orders o
        WHERE
            customer_id = c.customer_id
        GROUP BY
            customer_id
        HAVING
            COUNT (*) > 2
    )
ORDER BY
    first_name,
    last_name;
```

Using EXISTS with a correlated query

This following example finds all customers who have placed more than two orders

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers c
WHERE
    EXISTS (
        SELECT
            COUNT (*)
        FROM
            sales.orders o
        WHERE
            customer_id = c.customer_id
        GROUP BY
            customer_id
        HAVING
            COUNT (*) > 2
    )
ORDER BY
    first_name,
    last_name;
```

Output

	customer_id	first_name	last_name
1	20	Aleta	Shepard
2	32	Araceli	Golden
3	64	Bobbie	Foster
4	47	Bridgette	Guerra
5	17	Caren	Stephens
6	5	Charolette	Rice
7	50	Cleotilde	Booth
8	24	Corene	Wall
9	4	Daryl	Spence
10	1	Debra	Burks
11	33	Deloris	Burke
12	11	Deshawn	Mendoza
13	61	Elinore	Aguilar
14	16	Emmitt	Sanchez
15	14	Garry	Espinosa
16	9	Genoveva	Baldwin
17	18	Georgetta	Hardin
18	8	Jacqueline	Duncan
19	30	Jamaal	Albert
...

VIEW



View is a virtual table based on the result of an SQL statement.

e_id	e_name	e_salary	e_age	e_gender	e_dept
1	Sam	95000	45	Male	Operations
2	Bob	80000	21	Male	Support
3	Anne	125000	25	Female	Analytics
4	Julia	73000	30	Female	Analytics
5	Matt	159000	33	Male	Sales
6	Jeff	112000	27	Male	Operations

Employee Table



e_id	e_name	e_salary	e_age	e_gender	e_dept
1	3	Anne	125000	25	Female
2	4	Julia	112000	30	Male

“Female_Employee” View

VIEW



```
CREATE VIEW view_name  
AS  
SELECT column1, column2,..  
FROM table_name  
WHERE condition;
```

VIEW

The views are generally referred to as a virtual table. It is just like a real table.

Create a view to include the Store_Status column.

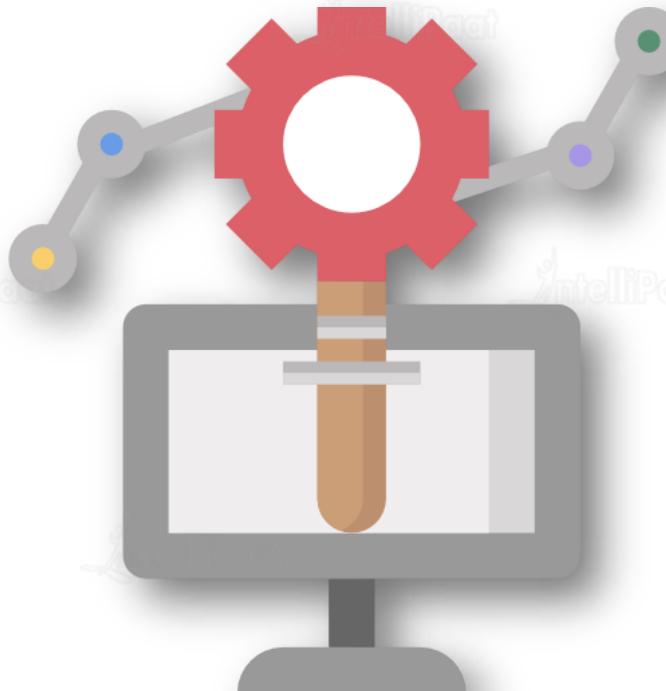
```
create view Store_Status_View as  
Select *, IIF(IsHoliday ='TRUE',  
'Open','Closed') as Store_Status  
from Sales
```

	Store	Dept	Date	Weekly_Sales	IsHoliday	Store_Status
1	8	5	05/08/2011	16183.33	FALSE	Closed
2	8	5	12/08/2011	16067.32	FALSE	Closed
3	8	5	19/08/2011	15966.62	FALSE	Closed
4	8	5	26/08/2011	11533.31	FALSE	Closed
5	8	5	02/09/2011	13594.54	FALSE	Closed
6	8	5	09/09/2011	12781.84	TRUE	Open
7	8	5	16/09/2011	15666.95	FALSE	Closed
8	8	5	23/09/2011	15107.2	FALSE	Closed
9	8	5	30/09/2011	16594.95	FALSE	Closed
10	8	5	07/10/2011	22372.88	FALSE	Closed
11	8	5	14/10/2011	18907.09	FALSE	Closed

STORED PROCEDURE

IntelliPact

STORED PROCEDURE is a prepared SQL code which can be saved and reused.



Stored Procedure without Parameter: Syntax



```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

```
EXEC procedure_name
```

STORED PROCEDURE

The Stored Procedure can be created once and re-used many times. The Stored Procedure will also accept the parameters.

Create a Stored Procedure to estimate the 10% increase in fuel price.

```
// Stored Procedure Creation without  
Parameter
```

```
CREATE PROCEDURE Proc_Fuel_Price as  
Select Fuel_Price,  
Fuel_Price+(Fuel_Price + 10)/100 as  
Estimate_Increase  
from Features
```

```
// Calling Stored Procedure  
Exec Proc_Fuel_Price
```

Fuel_Price	Estimate_Increase
2.57200002670288	2.69772002696991
2.54800009727478	2.67348009824753
2.51399993896484	2.63913993835449
2.56100010871887	2.68661010980606
2.625	2.75125
2.66700005531311	2.79367005586624
2.72000002861023	2.84720002889633
2.73200011253357	2.8593201136589

Stored Procedure with Parameter: Syntax



```
CREATE PROCEDURE procedure_name  
@param1 data-type, @param2 data-type  
AS  
sql_statement  
GO;
```

STORED PROCEDURE

Create a Stored Procedure to check whether the Weekly_Sales is greater than the user's Sales value.

```
// Stored Procedure Creation with  
Parameter (with float data type)  
CREATE PROCEDURE Proc_Weekly_Sales  
@Sales float as  
Select Date,Weekly_Sales from Sales  
Where Weekly_Sales > @Sales  
// Calling Stored Procedure Exec  
Proc_Weekly_Sales @Sales = 1500
```

	Date	Weekly_Sales
1	05/08/2011	16183.33
2	12/08/2011	16067.32
3	19/08/2011	15966.62
4	26/08/2011	11533.31
5	02/09/2011	13594.54
6	09/09/2011	12781.84
7	16/09/2011	15666.95
8	23/09/2011	15107.2
9	30/09/2011	16594.95
10	07/10/2011	22372.88
11	14/10/2011	18907.09

STORED PROCEDURE

Create a Stored Procedure to fetch the Store details based on the Storetype.

```
// Stored Procedure Creation with  
Parameter (with varchar data type)  
CREATE PROCEDURE Proc_Type @type  
varchar(10) as  
Select * from Stores  
where Type=@type  
// Calling Stored Procedure Exec Exec  
Proc_Type @type ='C'
```

	Store	Type	Size
1	30	C	42988
2	37	C	39910
3	38	C	39690
4	42	C	39690
5	43	C	41062
6	44	C	39910

Transactions in SQL

Transaction is a group of commands that change the data stored in a database.

```
begin try
    begin transaction
        update employee set
            e_salary=50 where
            e_gender='Male'
        update employee set
            e_salary=195/0 where
            e_name='Female'
        commit transaction
        Print 'transaction committed'
    end try
    begin
    catch
        rollback transaction
        print 'transaction
        rolledback' end catch
```

→ Single Unit

Transactions in SQL

Transaction is a group of commands that change the data stored in a database.

```
begin try
    begin transaction
        update employee set
            e_salary=50 where
            e_gender='Male'
        update employee set
            e_salary=195/0 where
            e_name='Female'
        commit
        transaction Print
        'transaction
        committed'
    end try
    begin
    catch
        rollback transaction
        print 'transaction
        rolledback' end catch
```



Transactions in SQL

Transaction is a group of commands that change the data stored in a database.

```
begin try
    begin transaction
        update employee
        set e_salary=50 where
        e_gender='Male'
        update employee set
        e_salary=195 where
        e_name='Female'
        commit transaction
    Print 'transaction
    committed'
end try
begin
catch
    rollback transaction
    print 'transaction
    rolledback' end catch
```

Commands
Success

Committed ..

TRANSACTION

A Transaction is an action which can be performed to make one or more changes to the database.
The transaction can take DML commands (Insert, Update and Delete)

Create a transaction to update your Feature table and save it.

```
// Transaction to update a record
begin transaction Update_Features
update Features
set Markdown1='No Data',Markdown2='No Data',Markdown3='No Data',
    Markdown4='No Data', Markdown5='No Data'
// Commit Transaction (Saving the update)
commit transaction Update_Features
Select * from Features
```

TRANSACTION

OUTPUT

	Store	Date	Temperature	Fuel_Price	MarkDown1	MarkDown2	MarkDown3	MarkDown4	MarkDown5	CPI	Unemployment	IsHoliday
1	1	05/02/2010	42.31	2.572	No Data	211.0963582	8.106	FALSE				
2	1	12/02/2010	38.51	2.548	No Data	211.2421698	8.106	TRUE				
3	1	19/02/2010	39.93	2.514	No Data	211.2891429	8.106	FALSE				
4	1	26/02/2010	46.63	2.561	No Data	211.3196429	8.106	FALSE				
5	1	05/03/2010	46.5	2.625	No Data	211.3501429	8.106	FALSE				
6	1	12/03/2010	57.79	2.667	No Data	211.3806429	8.106	FALSE				
7	1	19/03/2010	54.58	2.72	No Data	211.215635	8.106	FALSE				
8	1	26/03/2010	51.45	2.732	No Data	211.0180424	8.106	FALSE				
9	1	02/04/2010	62.27	2.719	No Data	210.8204499	7.808	FALSE				
10	1	09/04/2010	65.86	2.77	No Data	210.6228574	7.808	FALSE				
11	1	16/04/2010	66.32	2.808	No Data	210.4887	7.808	FALSE				
12	1	23/04/2010	64.84	2.795	No Data	210.4391228	7.808	FALSE				

TRANSACTION

Create a transaction to roll back the changes.

```
// Transaction to Delete a record  
begin transaction Delete_Stores  
Delete Stores where Type='A'  
// Rollback Transaction (Undoing the deletion)  
rollback transaction Delete_Stores  
Select * from Features
```

Messages

Commands completed successfully.

Completion time: 2022-11-21T20:03:58.5214813+05:30

SAVEPOINT

A Savepoint is like a marker. It saves a particular portion of a transaction without rolling the entire transaction.

Create a transaction to update your Feature table and save it.

Before Rollback

```
begin transaction Insert_Stores
Insert into Stores values (4,'C',35677)
Save transaction First_Record
Insert into Stores values (5,'D',55751)
Select * from stores
```

	Store	Type	Size
1	4	C	35677
2	5	D	55751
3	3	B	37392
4	7	B	70713
5	9	B	125833
6	10	B	126512
7	12	B	112238
8	15	B	123737
9	16	B	57197
10	17	B	93188
11	18	B	120653
12	21	B	140167
13	22	B	119557

SAVEPOINT

After Rollback - (The (5,'D',55751) record was removed and the first record was saved)

```
rollback transaction First_Record  
select * from stores
```

	Store	Type	Size
1	4	C	35677
2	3	B	37392
3	7	B	70713
4	9	B	125833
5	10	B	126512
6	12	B	112238
7	15	B	123737
8	16	B	57197
9	17	B	93188
10	18	B	120653
11	21	B	140167
12	22	B	119557
13	23	B	114533

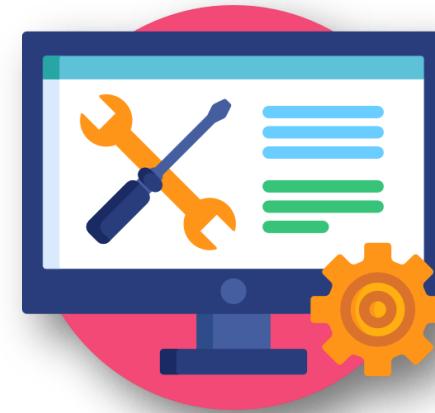
Exception Handling

An error condition during a program execution is called an exception.



Exception

The mechanism for resolving such an exception is exception handling.



Exception Handling

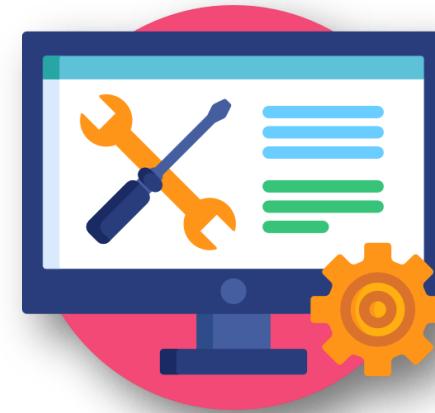
Exception Handling

An error condition during a program execution is called an exception.



Exception

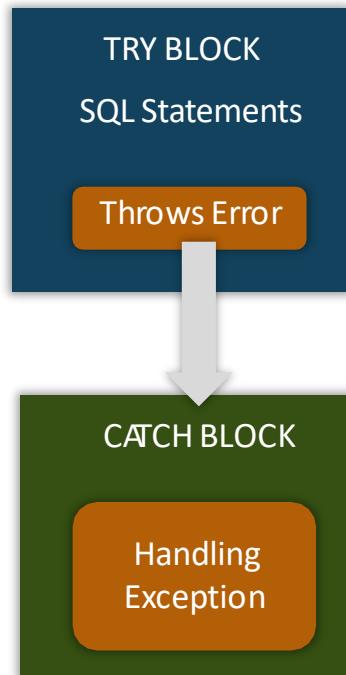
The mechanism for resolving such an exception is exception handling.



Exception Handling

Try/Catch

SQL provides try/catch blocks for exception handling.



Try/Catch: Syntax



BEGIN TRY

SQL Statements

END TRY

BEGIN CATCH

- Print Error OR
- Rollback Transaction

END CATCH

EXCEPTION HANDLING IN TRANSACTION

In the transaction, if the try blocks execute, the record will be automatically committed. If the catch blocks execute the record will be rolled back.

Create a transaction to divide the Weekly_Sales column by zero. Use the exception-handling method

```
begin transaction Sales_Division
begin try
Select (Weekly_Sales)/0 from Sales
end try
begin catch
Select ERROR_MESSAGE()as Error
end catch
// The Transaction will be rolled back
since we cannot divide Weekly_Sales by 0.
```

(No column name)	
	Error
1	Divide by zero error encountered.

EXCEPTION HANDLING IN TRANSACTION

Create a transaction to divide the Weekly_Sales column by five. Use the exception-handling method.

```
begin transaction Sales_Division
begin try
Select *,(Weekly_Sales)/5 as
New_Weekly_Sales from Sales
end try
begin catch
Select ERROR_MESSAGE()as Error
end catch
// The Transaction will be committed since
// we can divide a Weekly_Sales by 5.
```

	Store	Dept	Date	Weekly_Sales	IsHoliday	New_Weekly_Sales
1	8	5	05/08/2011	16183.33	FALSE	3236.666
2	8	5	12/08/2011	16067.32	FALSE	3213.464
3	8	5	19/08/2011	15966.62	FALSE	3193.324
4	8	5	26/08/2011	11533.31	FALSE	2306.662
5	8	5	02/09/2011	13594.54	FALSE	2718.908
6	8	5	09/09/2011	12781.84	TRUE	2556.368
7	8	5	16/09/2011	15666.95	FALSE	3133.39
8	8	5	23/09/2011	15107.2	FALSE	3021.44
9	8	5	30/09/2011	16594.95	FALSE	3318.99
10	8	5	07/10/2011	22372.88	FALSE	4474.576
11	8	5	14/10/2011	18907.09	FALSE	3781.418

ROLLUP

ROLLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

Let's consider an example. The following CUBE (d1, d2, d3) defines eight possible grouping sets:

- (d1, d2, d3)
- (d1, d2)
- (d2, d3)
- (d1, d3)
- (d1)
- (d2)
- (d3)
- ()

ROLLUP

ROLLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

And the ROLLUP (d1 , d2 , d3) creates only four grouping sets, assuming the hierarchy d1 > d2 > d3, as follows:

(d1, d2, d3)
(d1, d2)
(d1)
()

ROLLUP

ROLLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns.

The ROLLUP is commonly used to calculate the aggregates of hierarchical data such as sales by year > quarter > month.

ROLLUP

ROLLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

Syntax

```
SELECT  
    d1,  
    d2,  
    d3,  
    aggregate_function(c4)  
FROM  
    table_name  
GROUP BY  
    ROLLUP (d1, d2, d3);
```

In this syntax, d1, d2, and d3 are the dimension columns. The statement will calculate the aggregation of values in the column c4 based on the hierarchy d1 > d2 > d3.

ROLLUP

ROLLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

Syntax

```
SELECT  
    d1,  
    d2,  
    d3,  
    aggregate_function(c4)  
FROM  
    table_name  
GROUP BY  
    d1,  
    ROLLUP (d2, d3);
```

You can also do a partial roll up to reduce the subtotals generated by using the following syntax

ROLLUP

SQL Roll up will generate multiple grouping sets. Roll-up can be used along with the group by function.

Give the total weekly sales value with the Date and Dept details. Use roll-up to pull the data in hierarchical order.

```
Select Date, Dept,  
SUM(Weekly_Sales)as Total_Sale from  
Sales  
group by Date, Dept  
with Rollup
```

Date	Dept	Total_Sale
01/04/2011	1	781166.1
01/04/2011	2	1875748.52
01/04/2011	3	348184.67
01/04/2011	4	1090598.68
01/04/2011	5	951711.6
01/04/2011	6	169476.42
01/04/2011	7	792626.93
01/04/2011	8	1320036.89
01/04/2011	9	750923.96
01/04/2011	10	744930.9
01/04/2011	11	510298.05

Date	Dept	Total_Sale
09/04/2010	1	1518946.82
09/04/2010	2	1903127.59
09/04/2010	3	363798.86
09/04/2010	4	1118048.3
09/04/2010	5	966470.35
09/04/2010	6	206312.51
09/04/2010	7	1095585.56
09/04/2010	8	1314310.42
09/04/2010	9	903029.69
09/04/2010	10	808857.42
09/04/2010	11	608841.17

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 **INSTEADOF TRIGGER** is defined
- Definer of the view on which the **INSTEADOF** 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.

Types of Triggers

```
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
```

TRIGGERS

DML Triggers:

The DML Trigger performs an event in Insert, Update or Delete commands in a **table or view**.

Create an insert trigger for the Sales table

```
//Insert Trigger Creation  
create trigger trigger_insert on Sales  
after insert  
as begin  
PRINT'YOU HAVE INSERTED A ROW'  
end
```

//To view the trigger activation

```
insert into Sales values (9,4,'04-11-2012',15670.50,'FALSE')
```

Messages

YOU HAVE INSERTED A ROW

(1 row affected)

Completion time: 2022-11-21T16:53:01.6538119+05:30

TRIGGERS

Create an update trigger in the Orders table.

```
//Update Trigger Creation
create trigger trigger_update on Stores
after update
as begin
PRINT'YOUR RECORD HAS BEEN UPDATED!'
end
SELECT * FROM [dbo].[Stores]
//To view the trigger activation
UPDATE Stores set Type='A' where Store =1
```

Messages
YOUR RECORD HAS BEEN UPDATED!
(1 row affected)
Completion time: 2022-11-21T17:06:48.9130465+05:30

TRIGGERS

Create a delete trigger in the Orders table.

```
//Delete Trigger Creation
create trigger trigger_delete on Features
after delete
as
begin
PRINT'YOUR RECORD HAS BEEN DELETED!'
end
select * from [dbo].[Features]
//To view the trigger activation
delete Features where Temperature='38.51'
```

Messages
YOUR RECORD HAS BEEN DELETED!
(3 rows affected)
Completion time: 2022-11-21T17:52:34.2282158+05:30

TRIGGERS

DDL Triggers:

The DDL Trigger performs an event in Create, Alter or Drop commands in a database.

Create a create trigger for your database

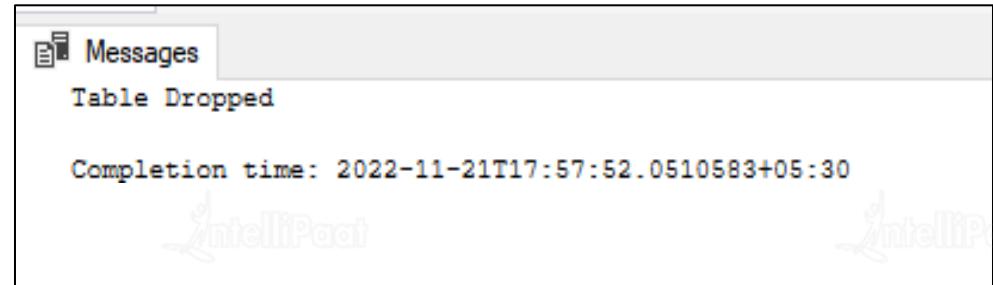
```
//Create Trigger Creation  
create trigger create_trigger  
on database  
after create_table  
as begin  
Print 'Table Created'  
End  
//To view the trigger activation  
create table sample_table (id int)
```

Messages
Table Created
Completion time: 2022-11-21T17:55:38.7361338+05:30

TRIGGERS

Alter a create trigger from your database to drop trigger.

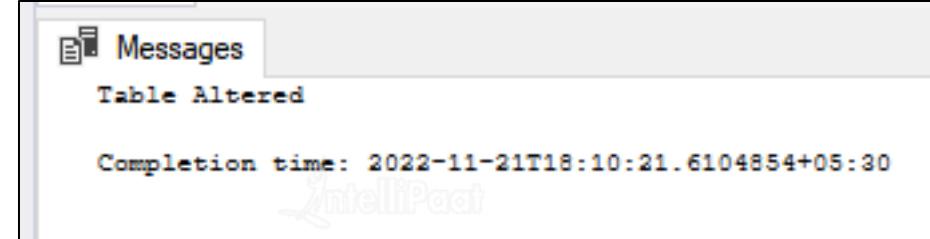
```
//Altering Create Trigger to Drop Trigger  
alter trigger create_trigger  
on database  
after drop_table  
as begin  
Print 'Table Dropped'  
End  
//To view the trigger activation  
drop table sample_table
```



TRIGGERS

Create a alter trigger for your database.

```
//Alter Trigger Creation
create trigger alter_trigger
on database
after alter_table
as begin
Print 'Table Altered'
end
//To view the trigger activation
alter table sample_table
alter column id varchar(5)
```



The screenshot shows a terminal window with the following output:

```
Messages
Table Altered
Completion time: 2022-11-21T18:10:21.6104854+05:30
```

COALESCE() FUNCTION

Use Coalesce function and print the first value without NULL records.

COALESCE()

return the first non-null value in a list

Syntax:

COALESCE(val1, val2,,val_n)

```
select coalesce(NULL,NULL,'Intellipaat',NULL,NULL,'Waseem');  
select coalesce('Shilpi',NULL,NULL,'Intellipaat',NULL);
```

(No column name)

Intellipaat

(No column name)

Shilpi

COALESCE() FUNCTION

Create a function using coalesce to concat the Store_Location and City to form a Address column.

// Coalesce Creation

```
Create FUNCTION try_CoalesceConcat  
(@city varchar(100))  
RETURNS NVARCHAR(MAX)  
AS  
BEGIN  
    DECLARE @str NVARCHAR(MAX);  
    SELECT @str = COALESCE(@str + ', ', '')  
    + Store_Location + City  
    FROM Store_Details  
        WHERE City = @city  
        ORDER BY Store_Location;  
    RETURN (@str);  
END
```

	Store	Store_Name	Sales	Order_No	Store_Location	City	Pincode
1	1	Walmart	374	246	Bentonville, Ark	Montgomery	36104
2	2	The Kroger Co	115	240	Cincinnati	Juneau	99801
3	3	Costco	93	567	Issaquah, Wash	Phoenix	85001
4	4	The Home Depot	91	639	Atlanta	Little Rock	72201
5	5	Walgreens Boots Alliance	82	484	Deerfield, Ill	Sacramento	95814
6	6	CVS Health Corporation	79	890	Woonsocket, R.I	Denver	80202
7	7	Target	71	251	Minneapolis	Hartford	6103
8	8	Lowe Companies	63	308	Mooresville, N.C	Dover	19901
9	9	Albertsons Companies	59	454	Boise, Idaho	Tallahassee	32301
10	10	Royal Ahold Delhaize USA	43	254	Carlisle, Pa	Atlanta	30303

Before Calling Coalesce function

COALESCE() FUNCTION

After Calling Coalesce function

```
SELECT *, Address =  
dbo.try_CoalesceConcat(City) FROM  
Store_Details
```

	Store	Store_Name	Sales	Order_No	Store_Location	City	Pincode	Address
1	1	Walmart	374	246	Bentonville, Ark	Montgomery	36104	Bentonville, Ark Montgomery
2	2	The Kroger Co	115	240	Cincinnati	Juneau	99801	Cincinnati Juneau
3	3	Costco	93	567	Issaquah, Wash	Phoenix	85001	Issaquah, Wash Phoenix
4	4	The Home Depot	91	639	Atlanta	Little Rock	72201	Atlanta Little Rock
5	5	Walgreens Boots Alliance	82	484	Deerfield, Ill	Sacramento	95814	Deerfield, Ill Sacramento
6	6	CVS Health Corporation	79	890	Woonsocket, R.I	Denver	80202	Woonsocket, R.I Denver
7	7	Target	71	251	Minneapolis	Hartford	6103	Minneapolis Hartford
8	8	Lowe Companies	63	308	Mooresville, N.C	Dover	19901	Mooresville, N.C Dover
9	9	Albertsons Companies	59	454	Boise, Idaho	Tallahassee	32301	Boise, Idaho Tallahassee
10	10	Royal Ahold Delhaize USA	43	254	Carlisle, Pa	Atlanta	30303	Carlisle, Pa Atlanta

ISNULL() FUNCTION

Fetch the below Customer's name without null values.

	Id	First_Name	Last_Name	Age
1	1	Harper	NULL	30
2	2	NULL	Charlie	31
3	3	Jane	Jones	25
4	4	NULL	Henry	27
5	5	Tina	Luis	24

	Customer_Name
1	Harper
2	Charlie
3	Jane
4	Henry
5	Tina

Select ISNULL(First_Name, Last_Name)as Customer_Name from Customer

CONVERT FUNCTION

It converts a value (of any type) into a specified datatype.

Convert the Weekly_Sales data type from float to int

Syntax -

CONVERT(data_type, expression)

```
select Weekly_Sales, convert  
(int,Weekly_Sales) as  
Converted_Weekly_Sales from Sales
```

	Results	Messages
1	Weekly_Sales	Converted_Weekly_Sales
1	16183.33	16183
2	16067.32	16067
3	15966.62	15966
4	11533.31	11533
5	13594.54	13594
6	12781.84	12781
7	15666.95	15666
8	15107.2	15107
9	16594.95	16594

Common Table Expression(CTE)

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.....

Common Table Expression(CTE)



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 T1(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 T1.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
```

Common Table Expression(CTE)

CTE allows us to define a temporary result set, that can be linked immediately with the select, insert, update or delete statement.

Using CTE display the records of stores data where Type is C

Syntax -

```
WITH expression_name(column1,column2,...)  
AS (CTE_definition)
```

```
with New_CTE  
as(select * from stores_data where  
Type='C'  
)select * from new_CTE
```

Store	Type	Size
30	C	42988
37	C	39910
38	C	39690
42	C	39690
43	C	41062
44	C	39910

Common Table Expression(CTE)

Create CTE with Store, Date and Fuel_Price column from features where holiday is there

```
// CTE with mutiple columns  
with New_CTE(Store,Date,Fuel_Price)  
as  
(  
select Store,Date,Fuel_Price from  
features where isHoliday=1  
)  
select Store,Date,Fuel_Price from New_CTE
```

Store	Date	Fuel_Price
1	2010-12-02	2.54800009727478
1	2010-10-09	2.56500005722046
1	2010-11-26	2.73499989509583
1	2010-12-31	2.94300007820129
1	2011-11-02	3.0220000743866
1	2011-09-09	3.5460000038147
1	2011-11-25	3.23600006103516
1	2011-12-30	3.12899994850159
1	2012-10-02	3.40899991989136
1	2012-07-09	3.73000001907349
1	2012-11-23	3.21099996566772

Common Table Expression(CTE)

Insert a record using CTE in Store Details table

```
// Inserting a record  
With New_CTE  
as  
(  
select * from Store_Details  
)  
insert new_CTE values(1,'H and  
M',100,1072,'HSR','Bangalore','560101')  
select * from Store_Details
```

Store	Store_Name	Sales	Order_No	Store_Location	City	pincode
3	Costco	93	567	Issaquah, Wash	Phoenix	85001
4	The Home Depot	91	639	Atlanta	Little Rock	72201
5	Walgreens Boots Alliance	82	484	Deerfield, Ill	Sacramento	95814
6	CVS Health Corporation	79	890	Woonsocket, R.I	Denver	80202
8	Lowe Companies	63	308	Mooresville, N.C	Dover	19901
9	Albertsons Companies	59	454	Boise, Idaho	Tallahassee	32301
10	Royal Ahold Delhaize USA	43	254	Carlisle, Pa	Atlanta	30303

Store	Store_Name	Sales	Order_No	Store_Location	City	pincode
1	H and M	100	1072	HSR	Bangalore	560101
3	Costco	93	567	Issaquah, Wash	Phoenix	85001
4	The Home Depot	91	639	Atlanta	Little Rock	72201
5	Walgreens Boots Alliance	82	484	Deerfield, Ill	Sacramento	95814
6	CVS Health Corporation	79	890	Woonsocket, R.I	Denver	80202
8	Lowe Companies	63	308	Mooresville, N.C	Dover	19901
9	Albertsons Companies	59	454	Boise, Idaho	Tallahassee	32301
10	Royal Ahold Delhaize USA	43	254	Carlisle, Pa	Atlanta	30303

Common Table Expression(CTE)

Update the store_name where store no is 1 using CTE in Store Details table

```
// Updating a record  
With New_CTE  
as  
(  
select * from Store_Details  
)  
update new_CTE set store_name='Jack &  
Jones' where Store=1  
select * from Store_Details
```

Store	Store_Name	Sales	Order_No	Store_Location	City	pincode
1	Jack & Jones	100	1072	HSR	Bangalore	560101
3	Costco	93	567	Issaquah, Wash	Phoenix	85001
4	The Home Depot	91	639	Atlanta	Little Rock	72201
5	Walgreens Boots Alliance	82	484	Deerfield, Ill	Sacramento	95814
6	CVS Health Corporation	79	890	Woonsocket, R.I	Denver	80202
8	Lowe Companies	63	308	Mooresville, N.C	Dover	19901
9	Albertsons Companies	59	454	Boise, Idaho	Tallahassee	32301
10	Royal Ahold Delhaize USA	43	254	Carlisle, Pa	Atlanta	30303

Common Table Expression(CTE)

Update the store_name where store no is 1 using CTE in Store Details table

// Deleting a record

```
With New_CTE  
as  
(  
select * from Store_Details  
)  
delete new_CTE where Store=1  
select * from Store_Details
```

Store	Store_Name	Sales	Order_No	Store_Location	City	pincode
3	Costco	93	567	Issaquah, Wash	Phoenix	85001
4	The Home Depot	91	639	Atlanta	Little Rock	72201
5	Walgreens Boots Alliance	82	484	Deerfield, Ill	Sacramento	95814
6	CVS Health Corporation	79	890	Woonsocket, R.I	Denver	80202
8	Lowe Companies	63	308	Mooresville, N.C	Dover	19901
9	Albertsons Companies	59	454	Boise, Idaho	Tallahassee	32301
10	Royal Ahold Delhaize USA	43	254	Carlisle, Pa	Atlanta	30303