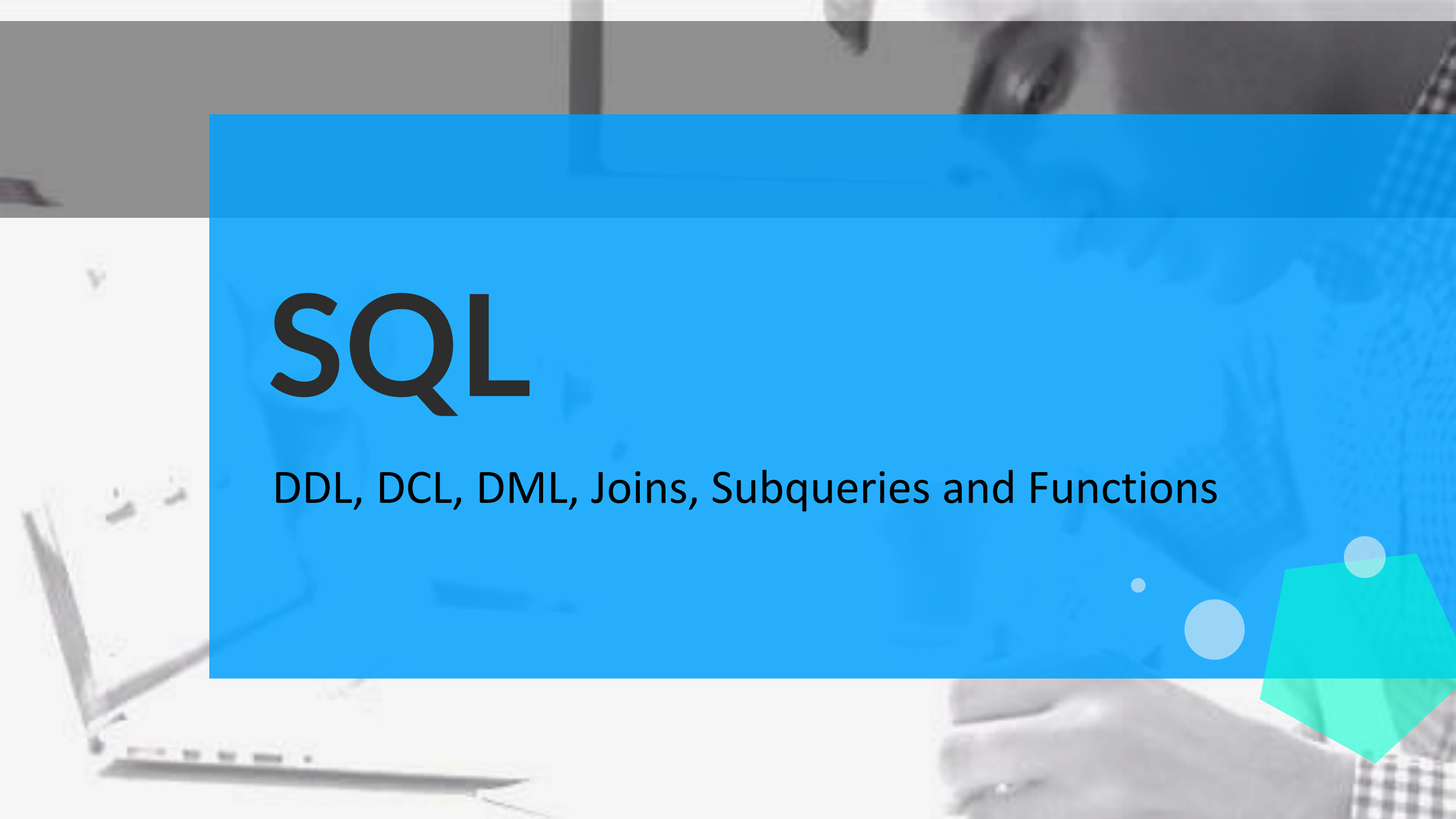


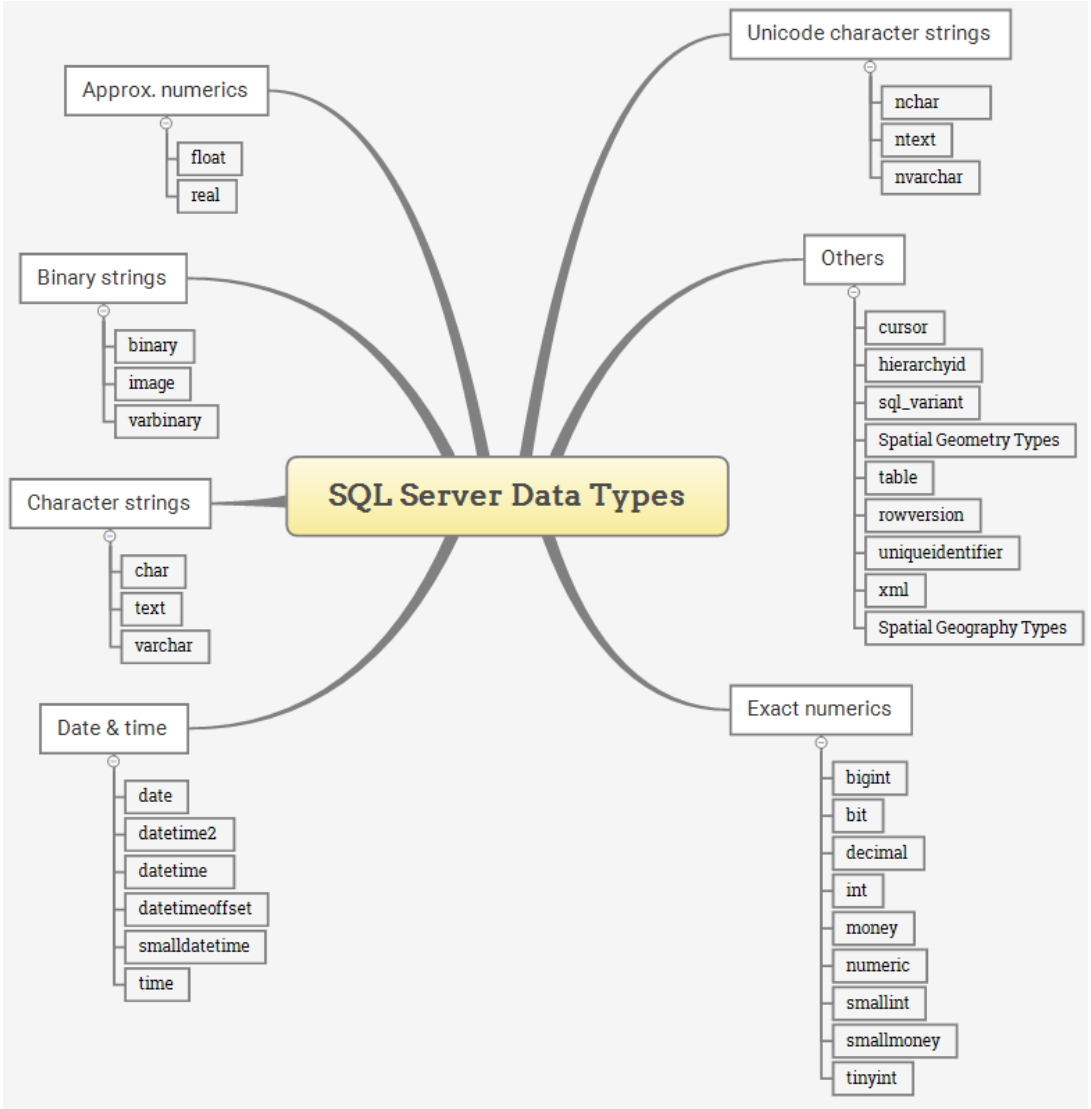
SQL

DDL, DCL, DML, Joins, Subqueries and Functions

Data types

In SQL Server, a column, variable, and parameter holds a value that associated with a type, or also known as a data type. A data type is an attribute that specifies the type of data that these objects can store. It can be an integer, character string, monetary, date and time, and so on.

SQL Server provides a list of data types that define all types of data that you can use e.g., defining a column or declaring a variable.



Exact numeric data types

Exact numeric data types store exact numbers such as integer, decimal, or monetary amount.

- The bit store one of three values 0, 1, and NULL
- The int, bigint, smallint, and tinyint data types store integer data.
- The decimal and numeric data types store numbers that have fixed precision and scale. Note that decimal and numeric are synonyms.
- The money and smallmoney data type store currency values.

Data Type	Lower limit	Upper limit	Memory
bigint	-2^63 (-9,223,372, 036,854,775,808)	2^63-1 (-9,223,372, 036,854,775,807)	8 bytes
int	-2^31 (-2,147, 483,648)	2^31-1 (-2,147, 483,647)	4 bytes
smallint	-2^15 (-32,767)	2^15 (-32,768)	2 bytes
tinyint	0	255	1 byte
bit	0	1	1 byte/8bit column
decimal	-10^38+1	10^381-1	5 to 17 bytes
numeric	-10^38+1	10^381-1	5 to 17 bytes
money	-922,337, 203, 685,477.5808	+922,337, 203, 685,477.5807	8 bytes
smallmoney	-214,478.3648	+214,478.3647	4 bytes

Approximate numeric data types

Data Type	Lower limit	Upper limit	Memory	Precision
float(n)	–1.79E+308	1.79E+308	Depends on the value of n	7 Digit
real	–3.40E+38	3.40E+38	4 bytes	15 Digit

- The approximate numeric data type stores floating point numeric data. They are often used in scientific calculations.

Date & Time data types

Data Type	Storage size	Accuracy	Lower Range	Upper Range
datetime	8 bytes	Rounded to increments of .000, .003, .007	1753-01-01	9999-12-31
smalldatetime	4 bytes, fixed	1 minute	1900-01-01	2079-06-06
date	3 bytes, fixed	1 day	0001-01-01	9999-12-31
time	5 bytes	100 nanoseconds	00:00:00.0000000	23:59:59.9999999
datetimeoffset	10 bytes	100 nanoseconds	0001-01-01	9999-12-31
datetime2	6 bytes	100 nanoseconds	0001-01-01	9999-12-31

The date and time data types store data and time data, and the date time offset.

Character strings data types

Data Type	Lower limit	Upper limit	Memory
char	0 chars	8000 chars	n bytes
varchar	0 chars	8000 chars	n bytes + 2 bytes
varchar (max)	0 chars	2^31 chars	n bytes + 2 bytes
text	0 chars	2,147,483,647 chars	n bytes + 4 bytes

- Character strings data types allow you to store either fixed-length (char) or variable-length data (varchar). The text data type can store non-Unicode data in the code page of the server.

Unicode character string data types

Data Type	Lower limit	Upper limit	Memory
nchar	0 chars	4000 chars	2 times n bytes
nvarchar	0 chars	4000 chars	2 times n bytes + 2 bytes
ntext	0 chars	1,073,741,823 char	2 times the string length

- Unicode character string data types store either fixed-length (nchar) or variable-length (nvarchar) Unicode character data.

Binary string data types

The binary data types stores fixed and variable length binary data.

Data Type	**Lower limit**	**Upper limit**	**Memory**
binary	0 bytes	8000 bytes	n bytes
varbinary	0 bytes	8000 bytes	The actual length of data entered + 2 bytes
image	0 bytes	2,147,483,647 bytes	

Data Type	Description
cursor	for variables or stored procedure OUTPUT parameter that contains a reference to a cursor
rowversion	expose automatically generated, unique binary numbers within a database.
hierarchyid	represent a tree position in a tree hierarchy
uniqueidentifier	16-byte GUID
sql_variant	store values of other data types
XML	store XML data in a column, or a variable of XML type
Spatial Geometry type	represent data in a flat coordinate system.
Spatial Geography type	store ellipsoidal (round-earth) data, such as GPS latitude and longitude coordinates.
table	store a result set temporarily for processing at a later time

OTHER DATA TYPES

Select - Syntax

```
SELECT
   select_list
FROM
   schema_name.table_name;
```

SQL Server SELECT – retrieve some columns of a table example

```
SELECT
   first_name,
   last_name
FROM
   sales.customers;
```

SQL Server SELECT – sort the result set

```
SELECT
   *
FROM
   sales.customers
WHERE
   state = 'CA';
```

SQL Server SELECT – group rows into groups example

```
SELECT   city,   COUNT (*)
FROM   sales.customers
WHERE   state = 'CA'
GROUP BY   city
ORDER BY   city;
```

Order By - Syntax

```
SELECT    select_list
FROM    table_name
ORDER BY
   column_name | expression [ASC | DESC ];
```

```
SELECT
   first_name,
   last_name
FROM
   sales.customers
ORDER BY
   LEN(first_name) DESC;
```

```
SELECT
   city,
   first_name,
   last_name
FROM
   sales.customers
ORDER BY
   city DESC,
   first_name ASC;
```

```
SELECT
   city,
   first_name,
   last_name
FROM
   sales.customers
ORDER BY
   state;
```

```
SELECT
   first_name,
   last_name
FROM
   sales.customers
ORDER BY
   1,
   2;
```

Offset- Syntax

```
ORDER BY column_list [ASC |DESC]
OFFSET offset_row_count {ROW | ROWS}
FETCH {FIRST | NEXT} fetch_row_count {ROW | ROWS} ONLY
```

```
SELECT    product_name,
   list_price
FROM    production.products
ORDER BY    list_price,
   product_name
OFFSET 10 ROWS;
```

```
SELECT
   product_name,
   list_price
FROM
   production.products
ORDER BY
   list_price,
   product_name
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY;
```

```
SELECT
   product_name,
   list_price
FROM
   production.products
ORDER BY
   list_price DESC,
   product_name
OFFSET 0 ROWS
FETCH FIRST 10 ROWS ONLY;
```

TOP - Syntax

```
SELECT TOP (expression) [PERCENT]
  [WITH TIES]
FROM
  table_name
ORDER BY
  column_name;
```

```
SELECT TOP 10
  product_name,
  list_price
FROM
  production.products
ORDER BY
  list_price DESC;
```

```
SELECT TOP 1 PERCENT
  product_name,
  list_price
FROM
  production.products
ORDER BY
  list_price DESC;
```

```
SELECT TOP 3 WITH TIES
  product_name,
  list_price
FROM
  production.products
ORDER BY
  list_price DESC;
```

Distinct - Syntax

```
SELECT DISTINCT
   column_name
FROM
   table_name;
```

```
SELECT DISTINCT
      column_name1,
      column_name2 ,
      ...
FROM
      table_name;
```

```
SELECT DISTINCT
   city
FROM
   sales.customers
ORDER BY
   city;
```

```
SELECT
      city,
      state,
      zip_code
FROM
      sales.customers
GROUP BY
      city, state, zip_code
ORDER BY
      city, state, zip_code
```

```
SELECT
      DISTINCT
   city,
   state,
   zip_code
FROM
      sales.customers;
```

Where - Syntax

```
SELECT
  select_list
FROM
  table_name
WHERE
  search_condition;
```

```
SELECT product_id, product_name, category_id,
  model_year, list_price
FROM production.products
WHERE category_id = 1
ORDER BY
  list_price DESC;
```

```
SELECT product_id, product_name,
  category_id, model_year, list_price
FROM
  production.products
WHERE
  category_id = 1 AND model_year = 2018
ORDER BY
  list_price DESC;
```

```
SELECT product_id, product_name,
  category_id, model_year, list_price
FROM
  production.products
WHERE
  list_price > 300 AND model_year = 2018
ORDER BY
  list_price DESC;
```

```
SELECT product_id, product_name,
category_id, model_year, list_price
FROM production.products
WHERE
  list_price BETWEEN 1899.00 AND 1999.99
ORDER BY
  list_price DESC;
```

```
SELECT product_id, product_name,
  category_id, model_year, list_price
FROM
  production.products
WHERE
  list_price IN (299.99, 369.99, 489.99)
ORDER BY
  list_price DESC;
```

```
SELECT product_id, product_name,
  category_id, model_year, list_price
FROM
  production.products
WHERE
  product_name LIKE '%Cruiser%'
ORDER BY
  list_price;
```

NULL and three-valued logic

```
NULL = 0
NULL <> 0
NULL > 0
NULL = NULL
```

```
SELECT customer_id, first_name,
  last_name, phone
FROM sales.customers
WHERE
  phone = NULL
ORDER BY
  first_name,
  last_name;
```

```
SELECT customer_id, first_name,
  last_name, phone
FROM
  sales.customers
WHERE
  phone IS NULL
ORDER BY
  first_name,
  last_name;
```

```
SELECT customer_id, first_name,
  last_name, phone
FROM
  sales.customers
WHERE
  phone IS NOT NULL
ORDER BY
  first_name,
  last_name;
```

AND operator

boolean_expression AND boolean_expression

boolean_expression AND
..... boolean_expression AND
boolean_expression

```
SELECT
  *
FROM   production.products
WHERE
  category_id = 1
AND list_price > 400
AND brand_id = 1
ORDER BY
  list_price DESC;
```

```
SELECT
  *
FROM   production.products
WHERE
  category_id = 1AND list_price > 400
ORDER BY
  list_price DESC;
```

```
SELECT
  *
FROM
  production.products
WHERE
  brand_id = 1
OR brand_id = 2
AND list_price > 1000
ORDER BY
  brand_id DESC;
```

OR operator

boolean_expression OR boolean_expression

boolean_expression OR
..... boolean_expression OR
boolean_expression

```
SELECT product_name, brand_id
FROM
  production.products
WHERE
  brand_id = 1
OR brand_id = 2
OR brand_id = 4
ORDER BY
  brand_id DESC;
```

```
SELECT product_name, list_price
FROM
  production.products
WHERE
  list_price < 200
OR list_price > 6000
ORDER BY
  list_price;
```

```
SELECT
  product_name,
  brand_id
FROM
  production.products
WHERE
  brand_id IN (1, 2, 3)
ORDER BY
  brand_id DESC;
```

BETWEEN operator

column | expression BETWEEN start_expression AND end_expression

column | expression <= end_expression AND column | expression >= start_expression

column | expression NOT BETWEEN start_expression AND end_expresion

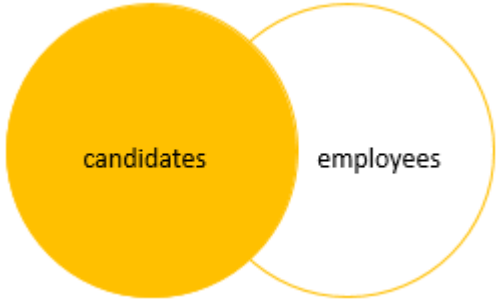
column | expression < start_expression AND column | expression > end_expression

```
SELECT
   product_id,
   product_name,
   list_price
FROM
   production.products
WHERE
   list_price BETWEEN 149.99 AND 199.99
ORDER BY
   list_price;
```

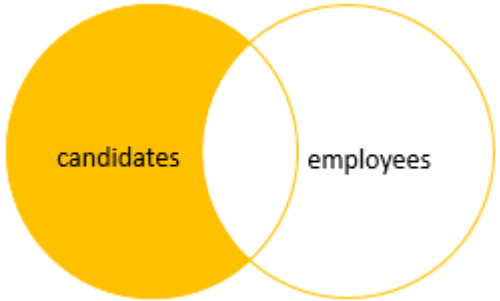
```
SELECT
   product_id,
   product_name,
   list_price
FROM
   production.products
WHERE
   list_price NOT BETWEEN 149.99 AND 199.99
ORDER BY
   list_price;
```

Left Join

```
SELECT   c.id candidate_id,   c.fullname candidate_name,
       e.id employee_id,   e.fullname employee_name
FROM
       hr.candidates c
       LEFT JOIN hr.employees e
              ON e.fullname = c.fullname;
```

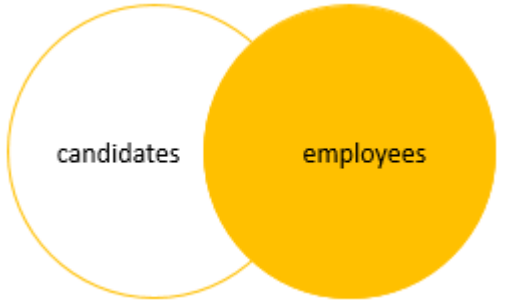



```
SELECT     c.id candidate_id,    c.fullname candidate_name,
   e.id employee_id,    e.fullname employee_name
FROM     hr.candidates c
   LEFT JOIN hr.employees e       ON e.fullname = c.fullname
WHERE
   e.id IS NULL;
```

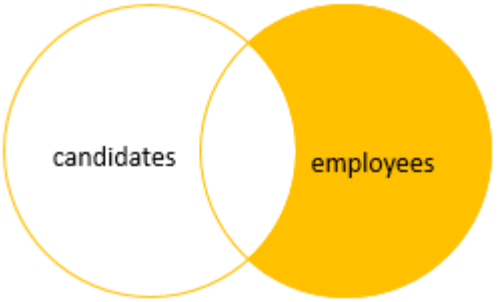


Right Join

```
SELECT     c.id candidate_id,    c.fullname candidate_name,
   e.id employee_id,    e.fullname employee_name
FROM
   hr.candidates c
   RIGHT JOIN hr.employees e
      ON e.fullname = c.fullname;
```

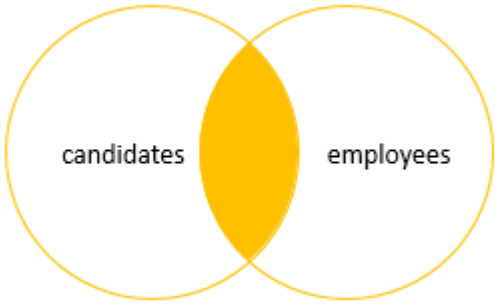



```
SELECT     c.id candidate_id,    c.fullname candidate_name,
   e.id employee_id,    e.fullname employee_name
FROM     hr.candidates c
   RIGHT JOIN hr.employees e        ON e.fullname = c.fullname
WHERE
   c.id IS NULL;
```



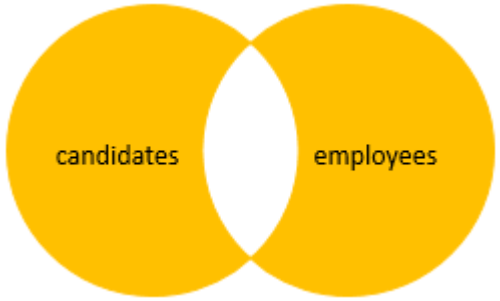
Inner Join

```
SELECT     c.id candidate_id,    c.fullname candidate_name,
   e.id employee_id,    e.fullname employee_name
FROM     hr.candidates c
   INNER JOIN hr.employees e        ON e.fullname = c.fullname;
```



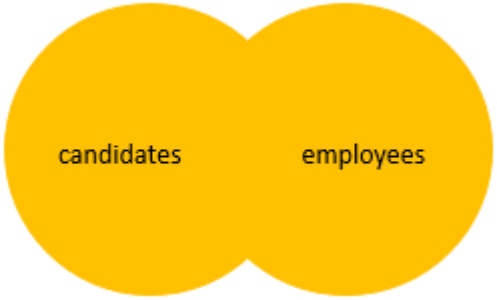
Full Join

```
SELECT     c.id candidate_id,    c.fullname candidate_name,
   e.id employee_id,    e.fullname employee_name
FROM     hr.candidates c
   FULL JOIN hr.employees e        ON e.fullname = c.fullname
WHERE    c.id IS NULL OR    e.id IS NULL;
```



Full Join

```
SELECT     c.id candidate_id,    c.fullname candidate_name,
   e.id employee_id,    e.fullname employee_name
FROM     hr.candidates c
   FULL JOIN hr.employees e
      ON e.fullname = c.fullname;
```



Self Join

A self join allows you to join a table to itself. It helps query hierarchical data or compare rows within the same table.

A self join uses the inner join or left join clause. Because the query that uses the self join references the same table, the table alias is used to assign different names to the same table within the query.

```
SELECT
    select_list
FROM
    T t1
[INNER | LEFT]  JOIN T t2 ON
    join_predicate;
```

```
SELECT
    e.first_name + ' ' + e.last_name employee,
    m.first_name + ' ' + m.last_name manager
FROM
    sales.staffs e
INNER JOIN sales.staffs m ON m.staff_id = e.manager_id
ORDER BY
    manager;
```

```
SELECT
    e.first_name + ' ' + e.last_name employee,
    m.first_name + ' ' + m.last_name manager
FROM
    sales.staffs e
LEFT JOIN sales.staffs m ON m.staff_id = e.manager_id
ORDER BY
    manager;
```

Using self join to compare rows within a table

```
SELECT
    c1.city,
    c1.first_name + ' ' + c1.last_name
customer_1,
    c2.first_name + ' ' + c2.last_name
customer_2
FROM
    sales.customers c1
INNER JOIN sales.customers c2 ON
c1.customer_id > c2.customer_id
AND c1.city = c2.city
ORDER BY
    city,
    customer_1,
    customer_2;
```

```
SELECT
    c1.city,
        c1.first_name + ' ' + c1.last_name customer_1,
    c2.first_name + ' ' + c2.last_name customer_2
FROM
    sales.customers c1
INNER JOIN sales.customers c2 ON c1.customer_id <> c2.customer_id
AND c1.city = c2.city
WHERE c1.city = 'Albany'
ORDER BY
        c1.city,
    customer_1,
    customer_2;
```

Cross Join

The CROSS JOIN joined every row from the first table (T1) with every row from the second table (T2). In other words, the cross join returns a Cartesian product of rows from both tables.

Unlike the INNER JOIN or LEFT JOIN, the cross join does not establish a relationship between the joined tables.

Suppose the T1 table contains three rows 1, 2, and 3 and the T2 table contains three rows A, B, and C.

The CROSS JOIN gets a row from the first table (T1) and then creates a new row for every row in the second table (T2). It then does the same for the next row for in the first table (T1) and so on.

```
SELECT
        select_list
FROM
        T1
CROSS JOIN T2;
```

```
SELECT
   product_id,
   product_name,
   store_id,
   0 AS quantity
FROM
   production.products
CROSS JOIN sales.stores
ORDER BY
   product_name,
   store_id;
```

Group By

```
SELECT
   select_list
FROM
   table_name
GROUP BY
   column_name1,
   column_name2 ,...;
```

```
SELECT  customer_id,  YEAR (order_date) order_year
FROM  sales.orders
WHERE
   customer_id IN (1, 2)
GROUP BY
   customer_id,
   YEAR (order_date)
ORDER BY
   customer_id;
```

```
SELECT  customer_id,  YEAR (order_date) order_year,
COUNT (order_id) order_placed
FROM  sales.orders
WHERE  customer_id IN (1, 2)
GROUP BY  customer_id,  YEAR (order_date)
ORDER BY  customer_id;
```

```
SELECT  city,  state,  COUNT (customer_id) customer_count
FROM  sales.customers
GROUP BY  state,  city
ORDER BY  city,  state;
```

```
SELECT  order_id,  SUM (  quantity * list_price * (1 - discount)  ) net_value
FROM  sales.order_items
GROUP BY  order_id;
```

Having

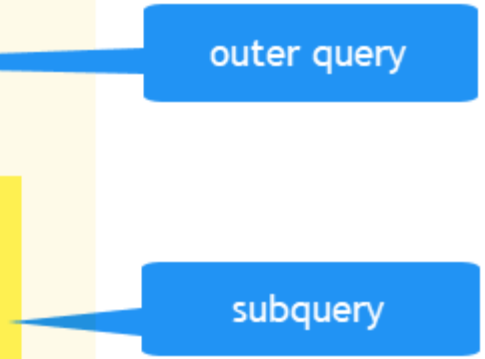
```
SELECT    select_list
FROM    table_name
GROUP BY    group_list
HAVING    conditions;
```

```
SELECT    column_name1,    column_name2,
   aggregate_function (column_name3)
column_alias
FROM    table_name
GROUP BY    column_name1,    column_name2
HAVING    column_alias > value;
```

```
SELECT
   customer_id,
   YEAR (order_date),
   COUNT (order_id) order_count
FROM
   sales.orders
GROUP BY
   customer_id,
   YEAR (order_date)
HAVING
   COUNT (order_id) >= 2
ORDER BY
   customer_id;
```

Sub Query

```sql
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```




```sql
SELECT
    product_name,
    list_price
FROM
    production.products
WHERE
    list_price > (
        SELECT
            AVG (list_price)
        FROM
            production.products
        WHERE
            brand_id IN (
                SELECT
                    brand_id
                FROM
                    production.brands
                WHERE
                    brand_name = 'Strider'
                OR brand_name = 'Trek'
            )
    )
ORDER BY
    list_price;
```

```sql
SELECT
   order_id,
   order_date,
   (
      SELECT
         MAX (list_price)
      FROM
         sales.order_items i
      WHERE
         i.order_id = o.order_id
   ) AS max_list_price
FROM
   sales.orders o
order by order_date desc;
```

```sql
SELECT
   product_id,
   product_name
FROM
   production.products
WHERE
   category_id IN (
      SELECT
         category_id
      FROM
         production.categories
      WHERE
         category_name = 'Mountain Bikes'
      OR category_name = 'Road Bikes'
   );
```

CTE in SQL Server

CTE stands for common table expression. A CTE allows you to define a temporary named result set that available temporarily in the execution scope of a statement such as SELECT, INSERT, UPDATE, DELETE, or MERGE.

The following shows the common syntax of a CTE in SQL Server:

```
WITH expression_name[(column_name [,...])]
AS
   (CTE_definition)
SQL_statement;
```

In this syntax:

First, specify the expression name (expression_name) to which you can refer later in a query.

Next, specify a list of comma-separated columns after the expression_name. The number of columns must be the same as the number of columns defined in the CTE_definition.

Then, use the AS keyword after the expression name or column list if the column list is specified.

After, define a SELECT statement whose result set populates the common table expression.

Finally, refer to the common table expression in a query (SQL_statement) such as SELECT, INSERT, UPDATE, DELETE, or MERGE.

```
WITH cte_sales_amounts (staff, sales, year) AS (
    SELECT
        first_name + ' ' + last_name,
        SUM(quantity * list_price * (1 - discount)),
        YEAR(order_date)
    FROM
        sales.orders o
    INNER JOIN sales.order_items i ON i.order_id = o.order_id
    INNER JOIN sales.staffs s ON s.staff_id = o.staff_id
    GROUP BY
        first_name + ' ' + last_name,
        year(order_date)
)

SELECT
    staff,
    sales
FROM
    cte_sales_amounts
WHERE
    year = 2018;
```

```
WITH cte_sales AS (
    SELECT
        staff_id,
        COUNT(*) order_count
    FROM
        sales.orders
    WHERE
        YEAR(order_date) = 2018
    GROUP BY
        staff_id

)
SELECT
    AVG(order_count) average_orders_by_staff
FROM
    cte_sales;
```

```
WITH cte_category_counts (    category_id,    category_name,    product_count
AS (    SELECT        c.category_id,        c.category_name,        COUNT(p.product_id)
   FROM        production.products p
      INNER JOIN production.categories c          ON c.category_id = p.category_id
   GROUP BY        c.category_id,        c.category_name
),
cte_category_sales(category_id, sales) AS (    SELECT          p.category_id,
      SUM(i.quantity * i.list_price * (1 - i.discount))
   FROM          sales.order_items i
      INNER JOIN production.products p          ON p.product_id = i.product_id
      INNER JOIN sales.orders o          ON o.order_id = i.order_id
   WHERE order_status = 4 -- completed
   GROUP BY        p.category_id
)
SELECT    c.category_id,    c.category_name,    c.product_count,    s.sales
FROM    cte_category_counts c
   INNER JOIN cte_category_sales s        ON s.category_id = c.category_id
ORDER BY     c.category_name;
```

Recursive CTE

A recursive common table expression (CTE) is a CTE that references itself. By doing so, the CTE repeatedly executes, returns subsets of data, until it returns the complete result set.

A recursive CTE is useful in querying hierarchical data such as organization charts where one employee reports to a manager or multi-level bill of materials when a product consists of many components, and each component itself also consists of many other components.

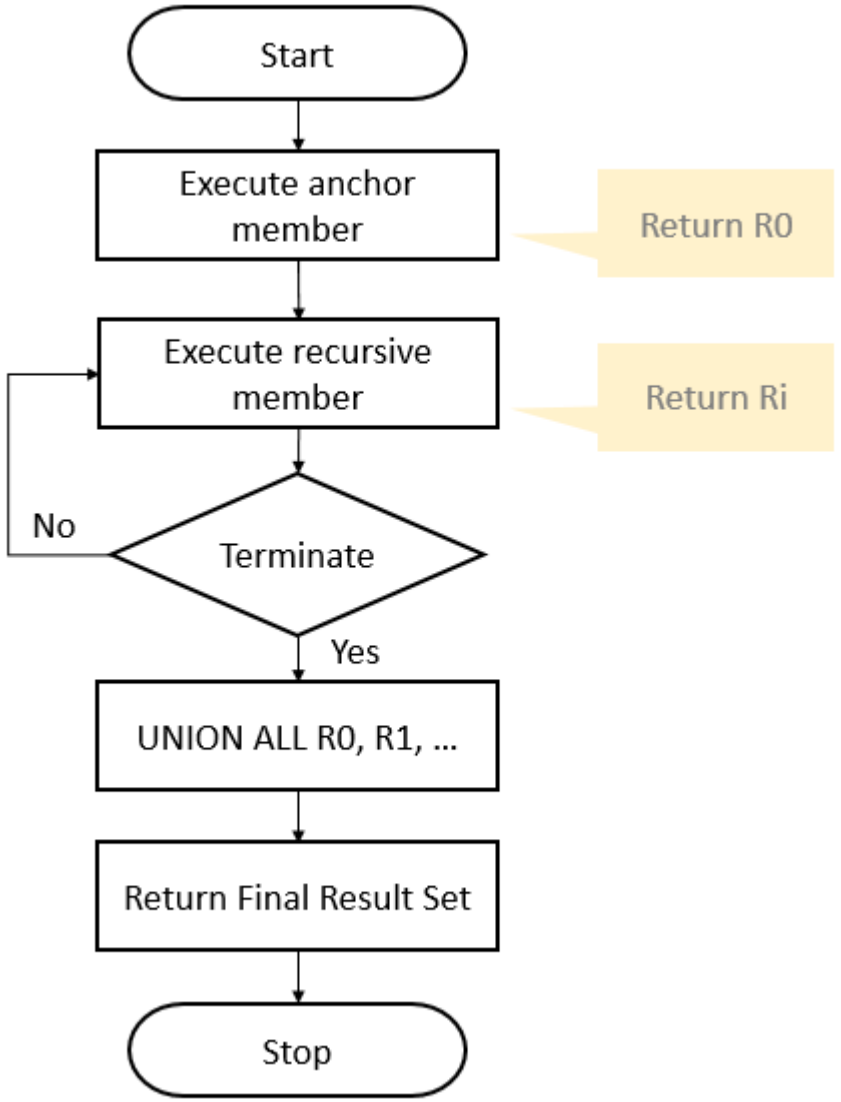
The following shows the syntax of a recursive CTE:

```
WITH expression_name (column_list)
AS
(
   -- Anchor member
   initial_query
   UNION ALL
   -- Recursive member that references expression_name.
   recursive_query
)
-- references expression name
SELECT *
FROM   expression_name
```

The execution order of a recursive CTE is as follows:

First, execute the anchor member to form the base result set (R0), use this result for the next iteration.
Second, execute the recursive member with the input result set from the previous iteration (Ri-1) and return a sub-result set (Ri) until the termination condition is met.
Third, combine all result sets R0, R1, … Rn using UNION ALL operator to produce the final result set.

The following flowchart illustrates the execution of a recursive CTE:



```
WITH cte_numbers(n, weekday)
AS (
   SELECT
      0,
      DATENAME(DW, 0)
   UNION ALL
   SELECT
      n + 1,
      DATENAME(DW, n + 1)
   FROM
      cte_numbers
   WHERE n < 6
)
SELECT
   weekday
FROM
   cte_numbers;
```

```
WITH cte_org AS (
   SELECT
      staff_id,
      first_name,
      manager_id

   FROM
      sales.staffs
   WHERE manager_id IS NULL
   UNION ALL
   SELECT
      e.staff_id,
      e.first_name,
      e.manager_id
   FROM
      sales.staffs e
      INNER JOIN cte_org o
         ON o.staff_id = e.manager_id
)
SELECT * FROM cte_org;
```

PIVOT operator

SQL Server PIVOT operator rotates a table-valued expression. It turns the unique values in one column into multiple columns in the output and performs aggregations on any remaining column values.

You follow these steps to make a query a pivot table:

First, select a base dataset for pivoting.
Second, create a temporary result by using a derived table or common table expression (CTE)
Third, apply the PIVOT operator.
Let's apply these steps in the following example.

First, select category name and product id from the production.products and production.categories tables as the base data for pivoting:

```
SELECT
   category_name,
   product_id
FROM
   production.products p
   INNER JOIN production.categories c
      ON c.category_id = p.category_id
```

```
   SELECT * FROM (
      SELECT
         category_name,
         product_id
      FROM
         production.products p
         INNER JOIN production.categories c
            ON c.category_id = p.category_id
   ) t
```

```
SELECT * FROM
(
   SELECT
      category_name,
      product_id
   FROM
      production.products p
      INNER JOIN production.categories c
         ON c.category_id = p.category_id
) t
PIVOT(
   COUNT(product_id)
   FOR category_name IN (
      [Children Bicycles],
      [Comfort Bicycles],
      [Cruisers Bicycles],
      [Cyclocross Bicycles],
      [Electric Bikes],
      [Mountain Bikes],
      [Road Bikes])
) AS pivot_table;
```

SQL Server Views

When you use the SELECT statement to query data from one or more tables, you get a result set.

For example, the following statement returns the product name, brand, and list price of all products from the products and brands tables:

```
SELECT
   product_name,
   brand_name,
   list_price
FROM
   production.products p
INNER JOIN production.brands b
      ON b.brand_id = p.brand_id;
```

Next time, if you want to get the same result set, you can save this query into a text file, open it, and execute it again.

SQL Server provides a better way to save this query in the database catalog through a view.

A view is a named query stored in the database catalog that allows you to refer to it later.

So the query above can be stored as a view using the CREATE VIEW statement as follows:

```
SELECT
  *
FROM (
  SELECT
    product_name,
    brand_name,
    list_price
  FROM
    production.products p
  INNER JOIN production.brands b
    ON b.brand_id = p.brand_id;
);
```

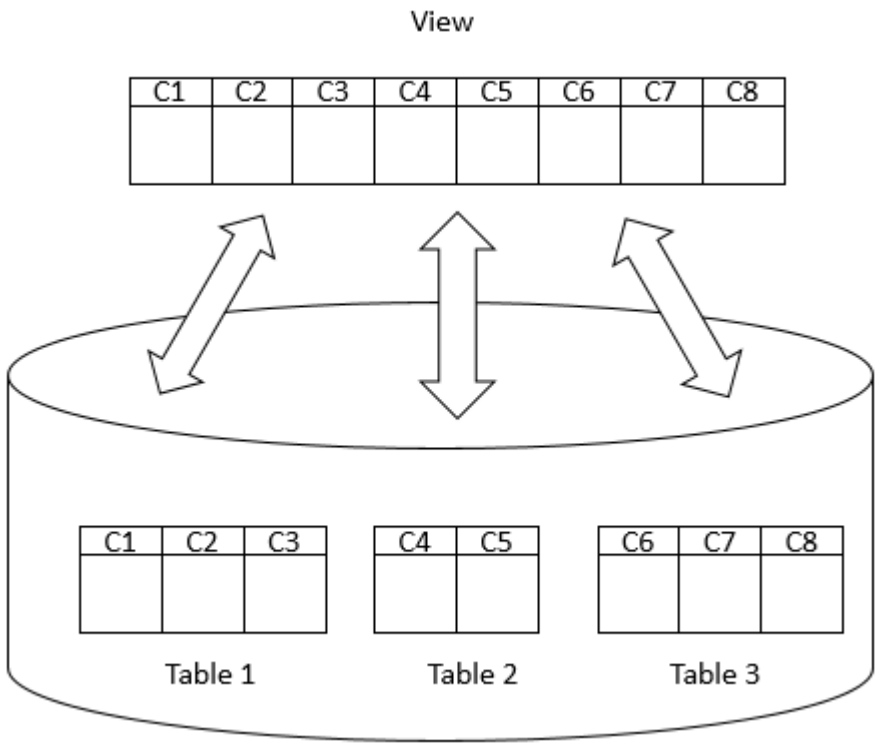
```
SELECT * FROM sales.product_info;
```

When receiving this query, SQL Server executes the following query:

```
SELECT
  *
FROM (
  SELECT
    product_name,
    brand_name,
    list_price
  FROM
    production.products p
  INNER JOIN production.brands b
    ON b.brand_id = p.brand_id;
);
```

By definition, views do not store data except for indexed views. A view may consist of columns from multiple tables using joins or just a subset of columns of a single table. This makes views useful for abstracting or hiding complex queries.

The following picture illustrates a view that includes columns from multiple tables:



```
CREATE VIEW [OR ALTER] schema_name.view_name
[(column_list)]
AS
   select_statement;
```

```
CREATE VIEW sales.daily_sales
AS
SELECT
   year(order_date) AS y,
   month(order_date) AS m,
   day(order_date) AS d,
   p.product_id,
   product_name,
   quantity * i.list_price AS sales
FROM
   sales.orders AS o
INNER JOIN sales.order_items AS i
   ON o.order_id = i.order_id
INNER JOIN production.products AS p
   ON p.product_id = i.product_id;
```

```
DROP VIEW [IF EXISTS] schema_name.view_name;
```

```
DROP VIEW [IF EXISTS]
   schema_name.view_name1,
   schema_name.view_name2,
   ...;
```

```
DROP VIEW IF EXISTS sales.daily_sales;
```

```
SELECT
      OBJECT_SCHEMA_NAME(v.object_id) schema_name,
      v.name
FROM
      sys.views as v;
```

Creating an SQL Server indexed view example

```
CREATE VIEW product_master
WITH SCHEMABINDING
AS
SELECT
   product_id,
   product_name,
   model_year,
   list_price,
   brand_name,
   category_name
FROM
   production.products p
INNER JOIN production.brands b
   ON b.brand_id = p.brand_id
INNER JOIN production.categories c
   ON c.category_id = p.category_id;
```

```
SET STATISTICS IO ON
GO

SELECT
  *
FROM
  production.product_master
ORDER BY
  product_name;
GO
```

```
Table 'Worktable'. Scan count 0, logical reads 0, physical reads
0, read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0,
read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.
Table 'products'. Scan count 1, logical reads 5, physical reads 1,
read-ahead reads 3, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.
Table 'categories'. Scan count 1, logical reads 2, physical reads
1, read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.
Table 'brands'. Scan count 1, logical reads 2, physical reads 1,
read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.
```

```
CREATE UNIQUE CLUSTERED INDEX
   ucidx_product_id
ON production.product_master(product_id);
```

```
CREATE NONCLUSTERED INDEX
   ucidx_product_name
ON production.product_master(product_name);
```

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'product_master'. Scan count 1, logical reads 6, physical reads 1, read-ahead reads 11, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Note that this feature is only available on SQL Server Enterprise Edition. If you use the SQL Server Standard or Developer Edition, you must use the WITH (NOEXPAND) table hint directly in the FROM clause of the query which you wanNote that this feature is only available on SQL Server Enterprise Edition. If you use the SQL Server Standard or Developer Edition, you must use the WITH (NOEXPAND) table hint directly in the FROM clause of the query which you want to use the view like the following query:t to use the view like the following query:

```
SELECT *
FROM production.product_master
  WITH (NOEXPAND)
ORDER BY product_name;
```

Indexes are special data structures associated with tables or views that help speed up the query. SQL Server provides two types of indexes: clustered index and non-clustered index.

Clustered indexes
Non clustered indexes

```
CREATE CLUSTERED INDEX index_name
ON schema_name.table_name (column_list);
```

```
CREATE [NONCLUSTERED] INDEX index_name
ON table_name(column_list);
```

Stored Procedures

SQL Server stored procedures are used to group one or more Transact-SQL statements into logical units. The stored procedure is stored as a named object in the SQL Server Database Server.

When you call a stored procedure for the first time, SQL Server creates an execution plan and stores it in the cache. In the subsequent executions of the stored procedure, SQL Server reuses the plan to execute the stored procedure very fast with reliable performance.

```
CREATE PROCEDURE uspProductList
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   ORDER BY
      product_name;
END;
```

```
EXECUTE sp_name;
```

```
ALTER PROCEDURE uspProductList
  AS
  BEGIN
     SELECT
        product_name,
        list_price
     FROM
        production.products
     ORDER BY
        list_price
  END;
```

```
DROP PROCEDURE sp_name;
```

Creating a stored procedure with one parameter

```
CREATE PROCEDURE uspFindProducts
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   ORDER BY
      list_price;
END;
```

```
ALTER PROCEDURE uspFindProducts(@min_list_price AS
DECIMAL)
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   WHERE
      list_price >= @min_list_price
   ORDER BY
      list_price;
END;
```

```
EXEC uspFindProducts 100;
```

Creating a stored procedure with multiple parameters

```
ALTER PROCEDURE uspFindProducts(
   @min_list_price AS DECIMAL
   ,@max_list_price AS DECIMAL
)
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   WHERE
      list_price >= @min_list_price AND
      list_price <= @max_list_price
   ORDER BY
      list_price;
END;
```

```
EXECUTE uspFindProducts 900, 1000;
```

```
EXECUTE uspFindProducts
   @min_list_price = 900,
   @max_list_price = 1000;
```

Creating text parameters

```
ALTER PROCEDURE uspFindProducts(
   @min_list_price AS DECIMAL
   ,@max_list_price AS DECIMAL
   ,@name AS VARCHAR(max)
)
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   WHERE
      list_price >= @min_list_price AND
      list_price <= @max_list_price AND
      product_name LIKE '%' + @name + '%'
   ORDER BY
      list_price;
END;
```

```
EXECUTE uspFindProducts
   @min_list_price = 900,
   @max_list_price = 1000,
   @name = 'Trek';
```

Creating optional parameters

```
ALTER PROCEDURE uspFindProducts(
   @min_list_price AS DECIMAL = 0
   ,@max_list_price AS DECIMAL = 999999
   ,@name AS VARCHAR(max)
)
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   WHERE
      list_price >= @min_list_price AND
      list_price <= @max_list_price AND
      product_name LIKE '%' + @name + '%'
   ORDER BY
      list_price;
END;
```

```
EXECUTE uspFindProducts
   @name = 'Trek';
```

```
EXECUTE uspFindProducts
   @min_list_price = 6000,
   @name = 'Trek';
```

Using NULL as the default value

```
ALTER PROCEDURE uspFindProducts(
    @min_list_price AS DECIMAL = 0
    ,@max_list_price AS DECIMAL = NULL
    ,@name AS VARCHAR(max)
)
AS
BEGIN
    SELECT
        product_name,
        list_price
    FROM
        production.products
    WHERE
        list_price >= @min_list_price AND
        (@max_list_price IS NULL OR list_price <=
@max_list_price) AND
        product_name LIKE '%' + @name + '%'
    ORDER BY
        list_price;
END;
```

```
EXECUTE uspFindProducts
    @min_list_price = 500,
    @name = 'Haro';
```

What is a variable

A variable is an object that holds a single value of a specific type e.g., integer, date, or varying character string.

We typically use variables in the following cases:

As a loop counter to count the number of times a loop is performed.
To hold a value to be tested by a control-of-flow statement such as WHILE.
To store the value returned by a stored procedure or a function
Declaring a variable
To declare a variable, you use the DECLARE statement. For example, the following statement declares a variable named @model_year:

```
DECLARE @model_year SMALLINT;

DECLARE @model_year AS SMALLINT;

DECLARE @model_year SMALLINT,
    @product_name VARCHAR(MAX);
```

```
SET @model_year = 2018;
```

```
SELECT
  product_name,
  model_year,
  list_price
FROM
  production.products
WHERE
  model_year = @model_year
ORDER BY
  product_name;
```

Storing query result in a variable

```
DECLARE @product_count INT;

SET @product_count = (
    SELECT
        COUNT(*)
    FROM
        production.products
);

SELECT @product_count;

PRINT @product_count;

PRINT 'The number of products is ' + CAST(@product_count
AS VARCHAR(MAX));
```

```
DECLARE
    @product_name VARCHAR(MAX),
    @list_price DECIMAL(10,2);

SELECT
    @product_name = product_name,
    @list_price = list_price
FROM
    production.products
WHERE
    product_id = 100;

SELECT
    @product_name AS product_name,
    @list_price AS list_price;
```

Accumulating values into a variable

```
CREATE  PROC uspGetProductList(
   @model_year SMALLINT
) AS
BEGIN
   DECLARE @product_list VARCHAR(MAX);

   SET @product_list = '';

   SELECT
      @product_list = @product_list + product_name
                  + CHAR(10)
   FROM
      production.products
   WHERE
      model_year = @model_year
   ORDER BY
      product_name;

   PRINT @product_list;
END;
```

The IF statement

```
IF boolean_expression
BEGIN
   { statement_block }
END
```

```
IF Boolean_expression
BEGIN
   -- Statement block executes when the
Boolean expression is TRUE
END
ELSE
BEGIN
   -- Statement block executes when the
Boolean expression is FALSE
END
```

```
BEGIN
   DECLARE @sales INT;

   SELECT
      @sales = SUM(list_price * quantity)
   FROM
      sales.order_items i
      INNER JOIN sales.orders o ON o.order_id = i.order_id
   WHERE
      YEAR(order_date) = 2018;

   SELECT @sales;

   IF @sales > 1000000
   BEGIN
      PRINT 'Great! The sales amount in 2018 is greater than
1,000,000';
   END
END
```

```
BEGIN
    DECLARE @sales INT;

    SELECT
        @sales = SUM(list_price * quantity)
    FROM
        sales.order_items i
        INNER JOIN sales.orders o ON o.order_id = i.order_id
    WHERE
        YEAR(order_date) = 2017;

    SELECT @sales;

    IF @sales > 10000000
    BEGIN
        PRINT 'Great! The sales amount in 2018 is greater than
10,000,000';
    END
    ELSE
    BEGIN
        PRINT 'Sales amount in 2017 did not reach 10,000,000';
    END
END
```

```
BEGIN
    DECLARE @x INT = 10,
            @y INT = 20;

    IF (@x > 0)
    BEGIN
        IF (@x < @y)
            PRINT 'x > 0 and x < y';
        ELSE
            PRINT 'x > 0 and x >= y';
    END
END
```

Overview of WHILE statement

```
WHILE Boolean_expression
    { sql_statement | statement_block}
```

```
DECLARE @counter INT = 1;

WHILE @counter <= 5
BEGIN
   PRINT @counter;
   SET @counter = @counter + 1;
END
```

SQL Server BREAK statement overview

```
WHILE Boolean_expression
BEGIN
   -- statements
   IF condition
      BREAK;
   -- other statements
END
```

```
WHILE Boolean_expression1
BEGIN
   -- statement
   WHILE Boolean_expression2
   BEGIN
      IF condition
         BREAK;
   END
END
```

```
DECLARE @counter INT = 0;

WHILE @counter <= 5
BEGIN
   SET @counter = @counter + 1;
   IF @counter = 4
      BREAK;
   PRINT @counter;
END
```

Introduction to the SQL Server CONTINUE statement

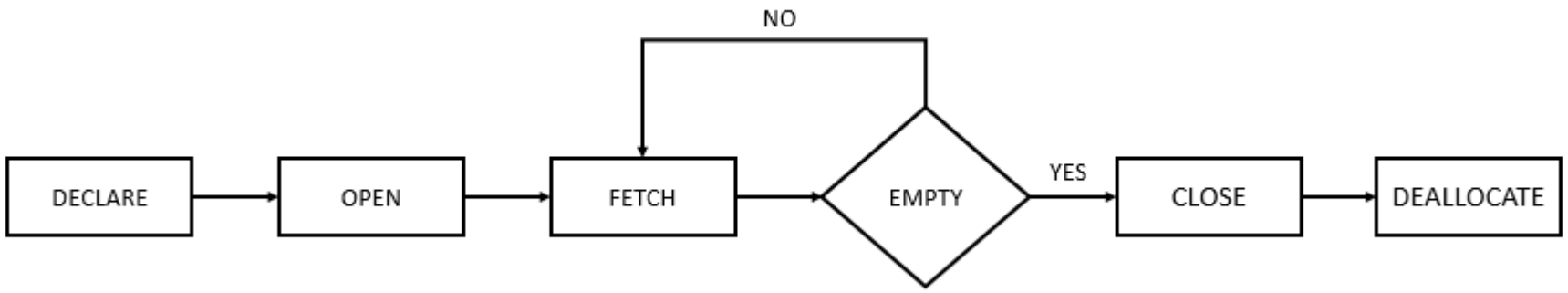
```
WHILE Boolean_expression
BEGIN
   -- code to be executed
   IF condition
      CONTINUE;
   -- code will be skipped if the condition is met
END
```

```
DECLARE @counter INT = 0;

WHILE @counter < 5
BEGIN
   SET @counter = @counter + 1;
   IF @counter = 3
      CONTINUE;
   PRINT @counter;
END
```

What is a database cursor

A database cursor is an object that enables traversal over the rows of a result set. It allows you to process individual row returned by a query.
SQL Server cursor life cycle
These are steps for using a cursor:



```
DECLARE cursor_name CURSOR
   FOR select_statement;


 OPEN cursor_name;


FETCH NEXT FROM cursor INTO variable_list;

 WHILE @@FETCH_STATUS = 0
    BEGIN
       FETCH NEXT FROM cursor_name;
    END;


CLOSE cursor_name;


DEALLOCATE cursor_name;
```

SQL Server provides the @@FETCHSTATUS function that returns the status of the last cursor FETCH statement executed against the cursor; If @@FETCHSTATUS returns 0, meaning the FETCH statement was successful. You can use the WHILE statement to fetch all rows from the cursor as shown in the following code:

```
DECLARE
   @product_name VARCHAR(MAX),
   @list_price   DECIMAL;

DECLARE cursor_product CURSOR
FOR SELECT
      product_name,
      list_price
   FROM
      production.products;


  OPEN cursor_product;
```

```
  FETCH NEXT FROM cursor_product INTO
     @product_name,
     @list_price;

  WHILE @@FETCH_STATUS = 0
     BEGIN
        PRINT @product_name + CAST(@list_price AS varchar);
        FETCH NEXT FROM cursor_product INTO
           @product_name,
           @list_price;
     END;
```

```
CLOSE cursor_product;


 DEALLOCATE cursor_product;
```

SQL Server TRY CATCH overview

```
BEGIN TRY
  -- statements that may cause exceptions
END TRY
```

```
BEGIN TRY
  -- statements that may cause exceptions
END TRY
BEGIN CATCH
  -- statements that handle exception
END CATCH
```

```
BEGIN TRY
   --- statements that may cause exceptions
END TRY
BEGIN CATCH
   -- statements to handle exception
   BEGIN TRY
      --- nested TRY block
   END TRY
   BEGIN CATCH
      --- nested CATCH block
   END CATCH
END CATCH
```

```
CREATE PROC usp_divide(
   @a decimal,
   @b decimal,
   @c decimal output
) AS
BEGIN
   BEGIN TRY
      SET @c = @a / @b;
   END TRY
   BEGIN CATCH
      SELECT
         ERROR_NUMBER() AS ErrorNumber
         ,ERROR_SEVERITY() AS ErrorSeverity
         ,ERROR_STATE() AS ErrorState
         ,ERROR_PROCEDURE() AS ErrorProcedure
         ,ERROR_LINE() AS ErrorLine
         ,ERROR_MESSAGE() AS ErrorMessage;
   END CATCH
END;
GO
```

```
DECLARE @r decimal;
EXEC usp_divide 10, 2, @r output;
PRINT @r;
```

```
DECLARE @r2 decimal;
EXEC usp_divide 10, 0, @r2 output;
PRINT @r2;
```

Introduction to Dynamic SQL

```
DECLARE
   @table NVARCHAR(128),
   @sql NVARCHAR(MAX);

SET @table = N'production.products';

SET @sql = N'SELECT * FROM ' + @table;

EXEC sp_executesql @sql;
```

```
CREATE PROC usp_query (
   @table NVARCHAR(128)
)
AS
BEGIN

   DECLARE @sql NVARCHAR(MAX);
   -- construct SQL
   SET @sql = N'SELECT * FROM ' + @table;
   -- execute the SQL
   EXEC sp_executesql @sql;

END;
```

```
CREATE OR ALTER PROC usp_query_topn(
   @table NVARCHAR(128),
   @topN INT,
   @byColumn NVARCHAR(128)
)
AS
BEGIN
   DECLARE
      @sql NVARCHAR(MAX),
      @topNStr NVARCHAR(MAX);

   SET @topNStr  = CAST(@topN as nvarchar(max));

   -- construct SQL
   SET @sql = N'SELECT TOP ' +  @topNStr  +
            ' * FROM ' + @table +
               ' ORDER BY ' + @byColumn + ' DESC';
   -- execute the SQL
   EXEC sp_executesql @sql;

END;
```

```
EXEC usp_query_topn
      'production.products',
      10,
      'list_price';
```

SQL Server User-defined Functions

What are table variables
Table variables are kinds of variables that allow you to hold rows of data, which are similar to temporary tables.

```
DECLARE @table_variable_name TABLE (
   column_list
);
```

```
DECLARE @product_table TABLE (
   product_name VARCHAR(MAX) NOT NULL,
   brand_id INT NOT NULL,
   list_price DEC(11,2) NOT NULL
);
```

```
INSERT INTO @product_table
SELECT
   product_name,
   brand_id,
   list_price
FROM
   production.products
WHERE
   category_id = 1;

   SELECT
      *
   FROM
      @product_table;
```

What are scalar functions

SQL Server scalar function takes one or more parameters and returns a single value.

The scalar functions help you simplify your code. For example, you may have a complex calculation that appears in many queries. Instead of including the formula in every query, you can create a scalar function that encapsulates the formula and uses it in each query.

Creating a scalar function
To create a scalar function, you use the CREATE FUNCTION statement as follows:

```
CREATE FUNCTION [schema_name.]function_name
(parameter_list)
RETURNS data_type AS
BEGIN
   statements
   RETURN value
END
```

```
CREATE FUNCTION sales.udfNetSale(
    @quantity INT,
    @list_price DEC(10,2),
    @discount DEC(4,2)
)
RETURNS DEC(10,2)
AS
BEGIN
    RETURN @quantity * @list_price * (1 - @discount);
END;



SELECT
    sales.udfNetSale(10,100,0.1) net_sale;
```

What is a table-valued function in SQL Server

A table-valued function is a user-defined function that returns data of a table type. The return type of a table-valued function is a table, therefore, you can use the table-valued function just like you would use a table.

Creating a table-valued function

The following statement example creates a table-valued function that returns a list of products including product name, model year and the list price for a specific model year:

```
CREATE FUNCTION udfProductInYear (
   @model_year INT
)
RETURNS TABLE
AS
RETURN
   SELECT
      product_name,
      model_year,
      list_price
   FROM
      production.products
   WHERE
      model_year = @model_year;
```

```
SELECT
   *
FROM
   udfProductInYear(2017);
```

Multi-statement table-valued functions (MSTVF)

```
CREATE FUNCTION udfContacts()
  RETURNS @contacts TABLE (
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(255),
    phone VARCHAR(25),
    contact_type VARCHAR(20)
  )
AS
BEGIN
  INSERT INTO @contacts
  SELECT
    first_name,
    last_name,
    email,
    phone,
    'Staff'
  FROM
    sales.staffs;
  INSERT INTO @contacts
  SELECT
    first_name,
    last_name,
    email,
    phone,
    'Customer'
  FROM
    sales.customers;
  RETURN;
END;
```

```
SELECT
  *
FROM
  udfContacts();
```

SQL Server Triggers

SQL Server triggers are special stored procedures that are executed automatically in response to the database object, database, and server events. SQL Server provides three type of triggers:

Data manipulation language (DML) triggers which are invoked automatically in response to INSERT, UPDATE, and DELETE events against tables.
Data definition language (DDL) triggers which fire in response to CREATE, ALTER, and DROP statements. DDL triggers also fire in response to some system stored procedures that perform DDL-like operations.
Logon triggers which fire in response to LOGON events

```
CREATE TRIGGER [schema_name.]trigger_name
ON table_name
AFTER  {[INSERT],[UPDATE],[DELETE]}
[NOT FOR REPLICATION]
AS
{sql_statements}
```

The schema_name is the name of the schema to which the new trigger belongs. The schema name is optional.

The trigger_name is the user-defined name for the new trigger.

The table_name is the table to which the trigger applies.

The event is listed in the AFTER clause. The event could be INSERT, UPDATE, or DELETE. A single trigger can fire in response to one or more actions against the table.

The NOT FOR REPLICATION option instructs SQL Server not to fire the trigger when data modification is made as part of a replication process.

The sql_statements is one or more Transact-SQL used to carry out actions once an event occurs.

DML event	INSERTED table holds	DELETED table holds
INSERT	rows to be inserted	empty
UPDATE	new rows modified by the update	existing rows modified by the update
DELETE	empty	rows to be deleted

```
CREATE TABLE production.product_audits(
    change_id INT IDENTITY PRIMARY KEY,
    product_id INT NOT NULL,
    product_name VARCHAR(255) NOT NULL,
    brand_id INT NOT NULL,
    category_id INT NOT NULL,
    model_year SMALLINT NOT NULL,
    list_price DEC(10,2) NOT NULL,
    updated_at DATETIME NOT NULL,
    operation CHAR(3) NOT NULL,
    CHECK(operation = 'INS' or operation='DEL')
);
```

```
CREATE TRIGGER production.trg_product_audit
ON production.products
AFTER INSERT, DELETE
AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO production.product_audits(
        product_id,
        product_name,
        brand_id,
        category_id,
        model_year,
        list_price,
        updated_at,
        operation
    )
SELECT
        i.product_id,
        product_name,
        brand_id,
        category_id,
        model_year,
        i.list_price,
        GETDATE(),
        'INS'
    FROM
        inserted i
    UNION ALL
    SELECT
        d.product_id,
        product_name,
        brand_id,
        category_id,
        model_year,
        d.list_price,
        GETDATE(),
        'DEL'
    FROM
        deleted d;
END
```

Testing the trigger

```
INSERT INTO production.products(
    product_name,
    brand_id,
    category_id,
    model_year,
    list_price
)
VALUES (
    'Test product',
    1,
    1,
    2018,
    599
);
```

```
SELECT
    *
FROM
    production.product_audits;
```