

## 2nd Normal Form Definition

A database is in second normal form if it satisfies the following conditions:

- It is in first normal form
- All non-key attributes are fully functional dependent on the primary key

In a table, if attribute B is functionally dependent on A, but is not functionally dependent on a proper subset of A, then B is considered fully functional dependent on A. Hence, in a 2NF table, all non-key attributes cannot be dependent on a subset of the primary key. Note that if the primary key is not a composite key, all non-key attributes are always fully functional dependent on the primary key. A table that is in 1st normal form and contains only a single key as the primary key is automatically in 2nd normal form.

**The example that was used in the book to explain this was orders, if two primary keys are used then the non-key attribute must be depended on both. If a non-key attribute depended on only one of the two primary keys then the Normalization form would be violated.**

**TABLE\_PURCHASE\_DETAIL**

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location]. In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key. Therefore, this table does not satisfy second normal form.

**TABLE\_PURCHASE**

Customer ID	Store ID
1	1
1	3
2	1
3	2
4	3

**TABLE\_STORE**

Store ID	Purchase Location
1	Los Angeles
2	New York
3	San Francisco

What we have done is to remove the partial functional dependency that we initially had. Now, in the table [TABLE\_STORE], the column [Purchase Location] is fully dependent on the primary key of that table, which is [Store ID].

### 3rd Normal Form Definition

A database is in third normal form if it satisfies the following conditions:

- It is in second normal form
- There is no transitive functional dependency

By transitive functional dependency, we mean we have the following relationships in the table: A is functionally dependent on B, and B is functionally dependent on C. In this case, C is transitively dependent on A via B.

**Informally, this rule means that all nonkey attributes must be mutually independent. In other words, one nonkey attribute cannot be dependent on another nonkey attribute. (Non-key attributes must not be dependent on one another)**

**TABLE\_BOOK\_DETAIL**

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

In the table above, [Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type]. Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

To bring this table to third normal form, we split the table into two as follows:

**TABLE\_BOOK**

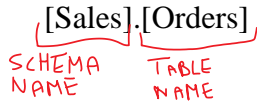
Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

**TABLE\_GENRE**

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

Now all non-key attributes are fully functional dependent only on the primary key. In [TABLE\_BOOK], both [Genre ID] and [Price] are only dependent on [Book ID]. In [TABLE\_GENRE], [Genre Type] is only dependent on [Genre ID].

**SCHEMA:** A container of objects, such as tables, views, stored procedures, and others. Schema is a namespace Sales.Orders

  
[Sales].[Orders]  
SCHEMA NAME      TABLE NAME


## SYNTAX

*USE(name of database)*

**DROP TABLE IF EXISTS:** command drops the table if it already exists

 **DROP TABLE IF EXISTS** dbo.Employees;

**CREATE TABLE:** Here you can specify the name of the table and, in parenthesis, the definition of its attributes

 **CREATE TABLE** dbo.Employees  
(  
    empid **int** NOT NULL  
    firstname **varchar**(20) NOT NULL  
)


## CONSTRAINTS

**Primary Key Constraint :** Enforces the uniqueness of rows and also disallows NULL's in the attribute

**Foreign Key Constraint:**


**Unique Constraint:** Enforces the uniqueness of rows, allowing you to implement the concept of alternate keys from the relational model in your database. Unlike with primary keys you can define multiple unique constraints within the same table. NULL's are ALLOWED

**Check Constraint:** A predicate (True or False) that a row must satisfy to be entered into the table or to be modified

 **ALTER TABLE** dbo.Employees  
**ADD CONSTRAINT** CHK\_Employees\_Salary  
**CHECK**(SALARY > 0)

1000 is rejected but 5000 or NULL is accepted

**Default Constraint:** A default constraint is associated with a particular attribute. It's an expression that is used as the default value when an explicit value is not specified for the attribute when you insert a row.

 **ALTER TABLE** dbo.Employees  
**ADD CONSTRAINT** DFT\_Orders\_Orderdt  
**DEFAULT**(SYSDATETIME()) **FOR** Orderdt

## LOGICAL QUERY PROCESSING

In most programming languages, the lines of code are processed in the order that they are written. In SQL, things are different. Even though the **SELECT** clause appears first in the query, it is logically processed almost last.

- 1) **FROM**
- 2) **WHERE**
- 3) **GROUP BY**
- 4) **HAVING**
- 5) **SELECT**
- 6) **ORDER BY**


**FROM:** Specify from where to get the table from

**WHERE:** Returns rows for which the **predicate** evaluates to TRUE (returns for true only not for FALSE or UNKNOWN)


 **WHERE** custid = 71

**GROUP BY:** It is used to group a selected set of rows into a set of summary rows by the table value of one or more columns or expression. It is always used in conjunction with one or more aggregate function

**HAVING:** Filters the rows

 **HAVING** COUNT(\*) > 1

**SELECT:** Selects what columns to display, or give an alias name

 **SELECT** empid, YEAR(orderdate) **AS** orderyear, COUNT(\*) **AS** numorders

**ORDER BY:** Orders the output based on the previous 5 steps

## GROUP BY EXAMPLE

USE TSQLV4

```
SELECT TOP(5) country, COUNT(DISTINCT supplierid) AS totalsuppliers
FROM Production.Suppliers
GROUP BY country
ORDER BY totalsuppliers DESC
```

country	totalsuppliers
USA	4
France	3
Germany	3
Australia	2
Canada	2

First its **GROUP BY** country

Then **SELECT TOP(5)** is used to show the top 5 based on the **ORDER BY** clause

Next **COUNT(DISTINCT supplierid) AS totalsuppliers** is used to Count the distinct number of supplierid's based on the **GROUP BY** (country) clause and is given an alias name totalsuppliers

Last it is ordered by the totalsuppliers count

```
USE TSQLV4
```

```
SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    COUNT(custid) AS total
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
```

Figure 1

empid	orderyear	total
1	2015	2
3	2015	2
5	2015	3
6	2015	3
8	2015	4
1	2016	3
2	2016	2
4	2016	3
7	2016	2

```
USE TSQLV4
```

```
SELECT
    empid,
    YEAR(orderdate) AS Orderyear
FROM Sales.Orders
WHERE custid = 71
ORDER BY empid
```

Figure 2

empid	Orderyear
1	2014
1	2015
1	2015
1	2016
1	2016
1	2016
2	2016
2	2016
2	2015
2	2014
3	2015
3	2015
4	2016
4	2016
4	2016
4	2015
5	2015
5	2015
5	2015

Does not qualify because the HAVING clause states

**HAVING** COUNT(\*) > 1

COUNT is counting the number of customer id 71's there are based on the GROUP BY clause which is

**GROUP BY** empid, YEAR(orderdate)

In figure 1 values like (empid 1, orderyear 2014) or (empid 2, orderyear 2015) or (empid 2, orderyear 2014)..etc are not displayed because they do not satisfy the condition which is **HAVING** COUNT(\*) > 1

## Problem 1: DISTINCT, GROUP BY

```
SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    COUNT(DISTINCT custid) AS total
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate)
ORDER BY empid
```

```
SELECT
    empid,
    COUNT(DISTINCT custid) AS total
FROM Sales.Orders
GROUP BY empid
ORDER BY empid
```

empid	orderyear	total
1	2014	22
1	2015	40
1	2016	32
2	2014	15
2	2015	35
2	2016	34
3	2014	16
3	2015	46
3	2016	30
4	2014	26
4	2015	57
4	2016	33
5	2014	10
5	2015	13
5	2016	11
6	2014	15
6	2015	24
6	2016	17
7	2014	11

does not add up  
why?

empid	total
1	65
2	59
3	63
4	75
5	29
6	43
7	45
8	56
9	29

This doesn't add up because even though distinct gets rid of duplicates, GROUP BY has a higher logical order, so it is executed first. Meaning that DISTINCT is only applies inside the group's. Example Group 1 can only have a unique customerID lets say custid =10, This does not mean that another group cannot have custid = 10. So in Figure 1 all the custid inside(empid 1, 2014) is unique and all the custid inside (empid1,2015) is unique, but between (empid 1, 2014) and (empid 1, 2015) there can be values that are repeated

**DISTINCT:** Selects only unique values, and if there are duplicate, it only selects one of the duplicates

```
COUNT(DISTINCT supplierid) AS totalsuppliers
```

**DESCENDING/ASCENDING:** If you want to sort in descending order, you need to specify *DESC* after the expression, as in *orderyear DESC*. When you want to sort by an expression in ascending order, you either specify *ASC* right after the expression, as in *orderyear ASC*, or don't specify anything after the expression, because *ASC* is the default.

```
ORDER BY empid DESC
```

**TOP:** Limits the number of rows your query returns. It relies on Two elements

- 1) **SELECT TOP (5):** will return the top 5 rows. But we don't know of what top 5 rows we want, so this is why we need the second element
- 2) **ORDER BY:** This is to define which of the top rows you want

TOP can be used without a ORDER BY but SQL servers returns which ever rows it accesses first

You can also do **TOP (5) PERCENT** to show the top 5 percent of the rows

**TIE BREAKER:** If you want the query to be deterministic, you need to make the **ORDER BY** list unique; in other words, add a tiebreaker. For example, you can add **orderid DESC** to the **ORDER BY** list as shown, so that, in case of ties, the row with the greater order ID value will be preferred.

```
SELECT TOP (5) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC;
```

**TIES:** Since top relies on both a value and ORDER BY, you can use **WITH TIES** to display the values that are the same based on the **ORDER BY**

```
SELECT TOP (5) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

orderid	orderdate	custid	empid
11074	2016-05-06	73	7
11075	2016-05-06	68	8
11076	2016-05-06	9	4
11077	2016-05-06	65	1
11070	2016-05-05	44	2
11071	2016-05-05	46	1
11072	2016-05-05	20	4
11073	2016-05-05	58	2

Notice that the output has eight rows, even though you specified *TOP (5)*. SQL Server first returned the *TOP (5)* rows based on *orderdate DESC* ordering, and it also returned all other rows from the table that had the same *orderdate* value as in the last of the five rows that were accessed. Using the **WITH TIES** option, the selection of rows is deterministic, but the presentation order among rows with the same order date isn't.

**OFFSET-FETCH:** There are two parts to this OFFSET and FETCH, first the OFFSET this says how many rows should be skipped. The FETCH part says how many rows should be fetched after those skipped rows. So this similar to but a more efficient method than TOP

```
OFFSET 10 ROWS FETCH NEXT 7 ROWS ONLY
```

Relies on

- 1) Note that a query that uses *OFFSET-FETCH* must have an *ORDER BY* clause.

## PARTITION:

```
SELECT orderid, custid, val, ROW_NUMBER() OVER(PARTITION BY custid ORDER BY val) AS rownum
FROM Sales.OrderValues
ORDER BY custid, val;
```

**ROW\_NUMBER()**: The *ROW\_NUMBER* function assigns unique, sequential, incrementing integers to the rows in the result **within the respective partition**, based on the indicated ordering

Note that the *ROW\_NUMBER* function must produce unique values within each partition. This means that even when the ordering value doesn't increase, the row number still must increase. Therefore, if the *ROW\_NUMBER* function's *ORDER BY* list is non-unique, as in the preceding example, the query is nondeterministic. That is, more than one correct result is possible. If you want to make a row number calculation deterministic, you must add elements to the *ORDER BY* list to make it unique. For example, in our sample query you can achieve this by adding the *orderid* attribute as a tiebreaker.

orderid	custid	val	rownum
10702	1	330.00	1
10952	1	471.20	2
10643	1	814.50	3
10835	1	845.80	4
10692	1	878.00	5
11011	1	933.50	6
10308	2	88.80	1
10759	2	320.00	2
10625	2	479.75	3
10926	2	514.40	4
10682	3	375.50	1

**OVER**: The *OVER* clause in this example function partitions the window by the *custid* attribute; hence, the row numbers are unique to each customer. The *OVER* clause also defines ordering in the window by the *val* attribute, so the sequential row numbers are incremented within the partition based on the values in this attribute.

## PREDICATE AND OPERATORS:

**IN**: checks whether a value, or scalar expression, is equal to at least one of the elements in a set. the IN method is used very similar to a equals sign, it basically means where orderid is equal to x,y and z

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderid IN(10248, 10249, 10250);
```

orderid	empid	orderdate
10248	5	2014-07-04
10249	6	2014-07-05
10250	4	2014-07-08

**BETWEEN**: The between keyword is used to select everything in between (x,y) This is inclusive meaning that x and y are included in the selection

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderid BETWEEN 10300 AND 10305;
```



**LIKE:** With the *LIKE* predicate, you can check whether a character string value meets a specified pattern. For example, the following query returns employees whose last names start with the letter *D*:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname LIKE N'd%';
```

empid	firstname	lastname
1	Sara	Davis
9	Patricia	Doyle

Notice the use of the letter *N* to prefix the string 'D%'; it stands for *National* and is used to denote that a character string is of a Unicode data type (*NCHAR* or *NVARCHAR*),

## OPERATORS:

T-SQL supports the following comparison operators: =, >, <, >=, <=, <>, !=, !>, !<, of which the last three are not standard. Because the nonstandard operators have standard alternatives (such as <> instead of !=)

**OR/AND:** If you need to combine logical expressions, you can use the logical operators *OR* and *AND*. If you want to negate an expression, you can use the *NOT* operator

```
SELECTorderid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20160101'
AND empid IN(1, 3, 5);
```

the following query returns orders placed on or after January 1, 2016, that were handled by one of the employees whose ID is 1, 3, or 5:

**T-SQL supports the four obvious arithmetic operators: +, -, \*, and /. It also supports the % operator**

```
SELECTorderid, productid, qty, unitprice, discount,
qty * unitprice * (1 - discount) AS val
FROM Sales.OrderDetails;
```

## CAST:

Note that the data type of a scalar expression involving two operands is determined in T-SQL by the higher of the two in terms of data-type precedence. If both operands are of the same data type, the result of the expression is of the same data type as well. For example, a division between two integers (*INT*) yields an integer. The expression 5/2 returns the integer 2 and not the numeric 2.5. This is not a problem when you are dealing with constants, because you can always specify the values as numeric ones with a decimal point. But when you are dealing with, say, two integer columns, as in *col1/col2*, you need to cast the operands to the appropriate type if you want the calculation to be a numeric one: *CAST(col1 AS NUMERIC(12, 2))/CAST(col2 AS NUMERIC(12, 2))*. The data type *NUMERIC(12, 2)* has a precision of 12 and a scale of 2, meaning that it has 12 digits in total, 2 of which are after the decimal point.

If the two operands are of different types, the one with the lower precedence is promoted to the one that is higher. For example, in the expression 5/2.0, the first operand is *INT* and the second is *NUMERIC*. Because *NUMERIC* is considered higher than *INT*, the *INT* operand 5 is implicitly converted to the *NUMERIC* 5.0 before the arithmetic operation, and you get the result 2.5.

**CASE:** Every much the same as in java used as an if statement, END AS is to set the name of the alias

```
SELECT productid, productname, categoryid,
CASE categoryid
WHEN 1 THEN 'Beverages'
WHEN 2 THEN 'Condiments'
WHEN 3 THEN 'Confections'
WHEN 4 THEN 'Dairy Products'
WHEN 5 THEN 'Grains/Cereals'
WHEN 6 THEN 'Meat/Poultry'
WHEN 7 THEN 'Produce'
WHEN 8 THEN 'Seafood'
ELSE 'Unknown Category'
END AS categoryname
FROM Production.Products;

USE TSQLV4
SELECTorderid, custid, val,
CASE
WHEN val < 1000.00 THEN 'Less than 1000'
WHEN val BETWEEN 1000.00 AND 3000.00 THEN 'Between 1000 and 3000'
WHEN val > 3000.00 THEN 'More than 3000'
ELSE 'Unknown'
END AS valuecategory
FROM Sales.OrderValues;
```

## ORDER OF PRECEDENCE

1. ( ) (Parentheses)
2. \* (Multiplication), / (Division), % (Modulo)
3. + (Positive), - (Negative), + (Addition), + (Concatenation), - (Subtraction)
4. =, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
5. NOT
6. AND
7. BETWEEN, IN, LIKE, OR
8. = (Assignment)

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE
  custid = 1
AND empid IN(1, 3, 5)
OR custid = 85
AND empid IN(2, 4, 6);
```



```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE
  (custid = 1
  AND empid IN(1, 3, 5))
OR
  (custid = 85
  AND empid IN(2, 4, 6));
```

The two top queries are the same but the parenthesis makes it much more readable

## NULL's

Let's start with three-valued predicate logic. A logical expression involving only non-*NULL* values evaluates to either *TRUE* or *FALSE*. When the logical expression involves a missing value, it evaluates to *UNKNOWN*. For example, consider the predicate *salary* > 0. When *salary* is equal to 1,000, the expression evaluates to *TRUE*. When *salary* is equal to -1,000, the expression evaluates to *FALSE*. When *salary* is *NULL*, the expression evaluates to *UNKNOWN*.

SQL treats *TRUE* and *FALSE* in an intuitive and probably expected manner. For example, if the predicate *salary* > 0 appears in a query filter (such as in a *WHERE* or *HAVING* clause), rows or groups for which the expression evaluates to *TRUE* are returned, whereas those for which the expression evaluates to *FALSE* are discarded. Similarly, if the predicate *salary* > 0 appears in a *CHECK* constraint in a table, *INSERT* or *UPDATE* statements for which the expression evaluates to *TRUE* for all rows are accepted, whereas those for which the expression evaluates to *FALSE* for any row are rejected.

SQL has different treatments for *UNKNOWN* in different language elements (and for some people, not necessarily the expected treatments). The treatment SQL has for query filters is "accept *TRUE*," meaning that both *FALSE* and *UNKNOWN* are discarded. Conversely, the definition of the treatment SQL has for *CHECK* constraints is "reject *FALSE*," meaning that both *TRUE* and *UNKNOWN* are accepted. Had SQL used two-valued predicate logic, there wouldn't have been a difference between the definitions "accept *TRUE*" and "reject *FALSE*." But with three-valued predicate logic, "accept *TRUE*" rejects *UNKNOWN*, whereas "reject *FALSE*" accepts it. With the predicate *salary* > 0 from the previous example, a *NULL* salary would cause the expression to evaluate to *UNKNOWN*. If this predicate appears in a query's *WHERE* clause, a row with a *NULL* salary will be discarded. If this predicate appears in a *CHECK* constraint in a table, a row with a *NULL* salary will be accepted.

Consider the following query, which attempts to return all customers where the region is equal to WA:

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = N'WA';
```

custid	country	region	city
43	USA	WA	Walla Walla
82	USA	WA	Kirkland
89	USA	WA	Seattle

Out of the 91 rows in the *Customers* table, the query returns the three rows where the *region* attribute is equal to WA. The query returns neither rows in which the value in the *region* attribute is present and different than WA (the predicate evaluates to *FALSE*) nor those where the *region* attribute is *NULL* (the predicate evaluates to *UNKNOWN*).

The following query attempts to return all customers for whom the region is different than WA:

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region <> N'WA';
```

custid	country	region	city
10	Canada	BC	Tsawassen
15	Brazil	SP	Sao Paulo
21	Brazil	SP	Sao Paulo
31	Brazil	SP	Campinas
32	USA	OR	Eugene
33	Venezuela	DF	Caracas
34	Brazil	RJ	Rio de Janeiro
35	Venezuela	Táchira	San Cristóbal
36	USA	OR	Elgin
37	Ireland	Co. Cork	Cork
38	UK	Isle of	Cornwall

If you expected to get 88 rows back (91 rows in the table minus 3 returned by the previous query), you might find this result (with just 28 rows) surprising. **But remember, a query filter “accepts *TRUE*,” meaning that it rejects both *FALSE* and *UNKNOWN*. So this query returned rows in which the *region* value was present and different than WA.** It returned neither rows in which the *region* value was equal to WA nor rows in which *region* was *NULL*.

**<> means NOT EQUAL TO**

## WRONG WAY

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = NULL;
```

If you want to return all rows for which *region* is *NULL*, do not use the predicate *region = NULL*, because the expression evaluates to *UNKNOWN* in all rows—both those in which the value is present and those in which the value is missing (is *NULL*). The following query returns an empty set:

## RIGHT WAY

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region IS NULL;
```

The following query returns all rows for which the *region* attribute is different than WA, including those in which the value is missing, you need to include an explicit test for *NULLs*, like this:

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region <> N'WA' →WHERE region IS NOT EQUAL TO(<>) 'WA'
OR region IS NULL; →OR region IS NULL
```

## DATA TYPE

SQL Server supports two kinds of character data types: regular and Unicode

### Regular

- Regular data types include *CHAR* and *VARCHAR*
- Regular characters use 1 byte of storage for each character
- If you choose a regular character type for a column, you are restricted to only one language in addition to English. The language support for the column is determined by the column's effective collation

### Unicode

- Unicode data types include *NCHAR* and *NVARCHAR*
- Unicode data requires 2 bytes per character
- With Unicode data types, multiple languages are supported.

A data type with the *VAR* element (*VARCHAR*, *NVARCHAR*) in its name has a variable length, which means that SQL Server uses as much storage space in the row as required to store the characters that appear in the character string, **plus two extra bytes for offset data**.

## COLLATION

Collation is a property of character data that encapsulates several aspects: language support, sort order, case sensitivity, accent sensitivity, and more.

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname = N'davis';
```

The query returns the row for Sara Davis, even though the casing doesn't match, because the effective casing is insensitive:

empid	firstname	lastname
1	Sara	Davis

If you want to make the filter case sensitive even though the column's collation is case insensitive, you can convert the collation of the expression:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname COLLATE Latin1_General_CS_AS = N'davis';
```

This time the query returns an empty set because no match is found when a case-sensitive comparison is used.

## OPERATORS AND FUNCTIONS

### String concatenation

the following query against the *Employees* table produces the *fullname* result column by concatenating *firstname*, a space, and *lastname*:

```
SELECT empid, firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

Standard SQL dictates that **a concatenation with a *NULL* should yield a *NULL***. This is the default behavior of T-SQL.

### COALESCE FUNCTION

The COALESCE function in SQL returns the first non-NULL expression among its arguments

For examples, say we have the following table,

Table *Contact\_Info*

Name	Business_Phone	Cell_Phone	Home_Phone
Jeff	531-2531	622-7813	565-9901
Laura	NULL	772-5588	312-4088
Peter	NULL	NULL	594-7477

and we want to find out the best way to contact each person according to the following rules:

1. If a person has a business phone, use the business phone number.
2. If a person does not have a business phone and has a cell phone, use the cell phone number.
3. If a person does not have a business phone, does not have a cell phone, and has a home phone, use the home phone number.

We can use the **COALESCE** function to achieve our goal:

```
SELECT Name, COALESCE (Business_Phone, Cell_Phone, Home_Phone) Contact_Phone
FROM Contact_Info;
```

<u>Name</u>	<u>Contact_Phone</u>
Jeff	531-2531
Laura	772-5588
Peter	594-7477

## SUBSTRING function

the following code returns the output 'abc':

```
SELECT SUBSTRING('abcde', 1, 3);
```

## LEFT and RIGHT functions

the following code returns the output 'cde':

```
SELECT right('abcde', 3);
```

the following code returns the output 'abc':

```
SELECT left('abcde', 3);
```

## LEN and DATALENGTH functions

The *LEN* function returns the number of characters in the input string.

the following code returns the output 5:

```
SELECT LEN('abcde')
```

Note that this function returns the number of characters in the input string and not necessarily the number of bytes. With regular characters, both numbers are the same because each character requires 1 byte of storage. With Unicode characters, each character requires at least 2 bytes of storage (in most cases, at least); therefore, the number of characters is half the number of bytes. To get the number of bytes, use the *DATALENGTH* function instead of *LEN*.

the following code returns the output 10:

```
SELECT DATALENGTH(N'abcde');
```

## CHARINDEX function

Output: 7

```
SELECT CHARINDEX('b', 'Itzik Ben-Gan');
```

The following code returns the first position of a space in 'Itzik Ben-Gan', so it returns the output 6:

```
SELECT CHARINDEX(' ', 'Itzik Ben-Gan');
```

## REPLACE function

`REPLACE(string, substring1, substring2)`

The function replaces all occurrences of *substring1* in *string* with *substring2*. For example, the following code substitutes all occurrences of a dash in the input string with a colon:

```
SELECT REPLACE('1-a 2-b', '-', ':');
```

This code returns the output: '1:a 2:b'.

## REPLICATE function

`REPLICATE(string, n)` For example

the following code replicates the string 'abc' three times, returning the string 'abcabcabc':

```
SELECT REPLICATE('abc', 3);
```

## STUFF function

`STUFF(string, pos, delete_length, insert_string)` This function operates on the input parameter *string*. It deletes as many characters as the number specified in the *delete\_length* parameter, starting at the character position specified in the *pos* input parameter. The function inserts the string specified in the *insert\_string* parameter in position *pos*. For example, the following code operates on the string 'xyz', removes one character from the second character, and inserts the substring 'abc' instead:

```
SELECT STUFF('xyz', 2, 1, 'abc');
```

The output of this code is 'xabcz'.

## UPPER and LOWER functions

The *UPPER* and *LOWER* functions return the input string with all uppercase or lowercase characters, respectively.

**Syntax:** `UPPER(string)`

`LOWER(string)`

## RTRIM and LTRIM functions

The *RTRIM* and *LTRIM* functions return the input string with leading or trailing spaces removed.

**Syntax:** `RTRIM(string)`

`LTRIM(string)`

If you want to remove both leading and trailing spaces, use the result of one function as the input to the other. For example, the following code removes both leading and trailing spaces from the input string, returning 'abc':

```
SELECT RTRIM(LTRIM(' abc '));
```

The output of this code is 'abc'.

## STRING\_SPLIT function

`SELECT value FROM STRING_SPLIT(string, separator);` Unlike the string functions described so far, which are all scalar functions, the *STRING\_SPLIT* function is a table function. It accepts as inputs a string with a separated list of values plus a separator, and it returns a table result with a string column called *val* with the individual elements. If you need the elements to be returned with a data type other than a character string, you will need to cast the *val* column to the target type. For example, the following code accepts the input string '10248,10249,10250' and separator ',', and it returns a table result with the individual elements:

```
SELECT CAST(value AS INT) AS myvalue
FROM STRING_SPLIT('10248,10249,10250', ',');

myvalue
-----
10248
10249
10250
```

## WILDCARD

### % (percent) wildcard

The percent sign represents a string of any size, including an empty string. For example, the following query returns employees where the last name starts with *D*:

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'D%';
```

### \_ (UNDERSCORE) wildcard

An underscore represents a single character.

You can use two UNDERSCORE to mean two single characters

For example, the following query returns employees where the second character in the last name is *e*:

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'_e%';
```

### [<list of characters>] wildcard

Square brackets with a list of characters (such as *[ABC]*) represent a single character that must be one of the characters specified in the list. For example, the following query returns employees where the first character in the last name is *A*, *B*, or *C*:

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'[abcd]%';
```

empid	lastname
8	Cameron
1	Davis
9	Doyle



## The [<character>-<character>] wildcard

Square brackets with a character range (such as [A-E]) represent a single character that must be within the specified range. For example, the following query returns employees where the first character in the last name is a letter in the range A through E, inclusive, taking the collation into account:

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'[A-E]%' ;
```

empid	lastname
8	Cameron
1	Davis
9	Doyle

## The [^<character list or range>] wildcard

Square brackets with a caret sign (^) followed by a character list or range (such as [^A-E]) represent a single character that is not in the specified character list or range. For example, the following query returns employees where the first character in the last name is not a letter in the range A through E:

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'[^A-E]%' ;
```

empid	lastname
2	Funk
7	King
3	Lew
5	Mortensen
4	Peled
6	Suurs

## Date and time data types

### T-SQL supports six date and time data types

1) DATETIME

2) SMALLDATETIME

3) DATE

4) TIME

Legacy types

The legacy types DATETIME and SMALLDATETIME include date and time components that are inseparable. The two types differ in their storage requirements, their supported date range, and their precision.

The DATE and TIME data types provide a separation between the date and time components if you need it

5) DATETIME2: The DATETIME2 data type has a bigger date range and better precision than the legacy types

6) DATETIMEOFFSET: The DATETIMEOFFSET data type is similar to DATETIME2, but it also includes the offset from UTC

<b>Data type</b>	<b>Storage (bytes)</b>	<b>Date range</b>	<b>Accuracy</b>	<b>Recommended entry format and example</b>
<i>DATETIME</i>	8	January 1, 1753, through December 31, 9999	3 1/3 milliseconds	'YYYYMMDD hh:mm:ss.nnn' '20160212 12:30:15.123'
<i>SMALLDATETIME</i>	4	January 1, 1900, through June 6, 2079	1 minute	''YYYYMMDD hh:mm' '20160212 12:30'
<i>DATE</i>	3	January 1, 0001, through December 31, 9999	1 day	'YYYY-MM-DD' '2016-02-12'
<i>TIME</i>	3 to 5	N/A	100 nanoseconds	'hh:mm:ss.nnnnnnn' '12:30:15.1234567'
<i>DATETIME2</i>	6 to 8	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnn' '2016-02-12 12:30:15.1234567'
<i>DATETIMEOFFSET</i>	8 to 10	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnn [+ -] hh:mm' '2016-02-12 12:30:15.1234567 +02:00'