SQL Coding Fundamentals

**Structured Query Language (SQL)** is a language for processing data in a **relational database**. Think of a relational database as a collection of tables. A table is a set of rows and columns, like a spreadsheet. Each row, or **record**, of a table contains info about a single entity (e.g. one member). Each column, or **field**, of a table contains a single attribute for all rows in the table (e.g. birthdate). A **query** is a request to pull data from a database table or combination of tables.

## Use best practices

* + Comment your code. Make your code clear for readers, including your future self!
  + Format your code.
    - Use new lines and indentation.
    - Use\_snake\_case\_for\_legibility.
  + Use clear aliases. Be succinct but still transparent.
  + Eliminate extra subqueries whenever possible.
  + Apply filters appropriately at every level.
  + Use IN rather than OR whenever possible.
  + Use BETWEEN whenever possible.

## Select columns and filter rows

Select a column from a table:

select HCG\_SETTING

from MillimanData.USR.VW\_CLAIM;

Select multiple columns from a table, limiting to top 100 rows:

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM;

Select all columns from a table, limiting to top 100 rows:

select top(100) \*

from MillimanData.USR.VW\_CLAIM;

Select unique (distinct) values from a column:

select distinct RELATION

from MillimanData.USR.VW\_CLAIM;

Count the number of rows in one or more columns:

select count(\*)

from MillimanData.USR.VW\_CLAIM;

-- claims data has > 57 million rows

Count the number of *non-missing* values in a specific column:

select count(MEMBER\_ID)

from MillimanData.USR.VW\_CLAIM;

-- claims data member ID has > 57 million non-missing values

Count the number of *distinct* values in a specific column:

select count(distinct MEMBER\_ID)

from MillimanData.USR.VW\_CLAIM;

-- claims data member ID has ~650,000 distinct values

Filter based on text and numeric values with WHERE and *comparison operators*:

= equal

<> not equal

< less than

> greater than

<= less than or equal to

>= greater than or equal to

*Note the order of operations (SELECT, FROM, WHERE):*

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

where AGE = 5;

Select data based on multiple conditions (AND and OR):

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

where AGE > 5

and AGE < 10;

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

where AGE = 5

or AGE = 10;

When combining conditions, enclose clauses in parentheses:

select top(100) MEMBER\_ID, AGE, MEMBER\_STATE

from MillimanData.USR.VW\_CLAIM

where (AGE = 5 or AGE = 10)

and (MEMBER\_STATE = 'WI' or MEMBER\_STATE = 'IL');

Filter values within a range, *including the beginning and end* values with BETWEEN:

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

where AGE between 5 and 10;

Specify multiple OR conditions with IN:

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

where AGE in (2, 4, 6);

Count the number of missing (NULL) values in a column:

select count(\*)

from MillimanData.USR.VW\_CLAIM

where AGE IS NULL;

Filter out missing (NULL) values:

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

where AGE IS NOT NULL;

## 

## Apply aggregate functions and arithmetic

Use *aggregate functions* to do some basic calculations:

select

min(AMT\_BILLED),

max(AMT\_BILLED),

avg(AMT\_BILLED),

sum(AMT\_BILLED)

from MillimanData.USR.VW\_CLAIM;

Use *aliases* to make results more legible:

select

max(AMT\_BILLED) as max\_billed,

max(AMT\_ALLOWED) as max\_allowed

from MillimanData.USR.VW\_CLAIM;

Use decimals for arithmetic. SQL assumes integer/integer = integer.

select 45 / 10 \* 100.0;

-- this will give 400 (integer)

select 45 \* 100.0 / 10;

-- this will give 450 because the numerator is a decimal

Apply functions for rounding:

· CEILING(): rounds up to the nearest integer value

· FLOOR(): rounds down to the nearest integer value

· ROUND(expression, length): rounds the expression to the specified length

## Sort and group results

*Note the order of operations (SELECT, FROM, GROUP BY, ORDER BY):*

Sort results by a column, ascending (default) or descending:

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

order by MEMBER\_ID; -- sort ascending

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

order by MEMBER\_ID desc; -- sort descending

Sort results by multiple columns. Note column order is important:

select top(100) MEMBER\_ID, AGE

from MillimanData.USR.VW\_CLAIM

order by MEMBER\_ID, AGE; -- sort by member ID, then by age

To aggregate results, group a result by one or more columns:

select AGE, count(\*) as n\_rows

from MillimanData.USR.VW\_CLAIM

group by AGE;

To group and order results:

select AGE, count(\*) as n\_rows

from MillimanData.USR.VW\_CLAIM

group by AGE

order by n\_rows desc;

Aggregate functions can’t be used in WHERE clauses. Use HAVING instead.

select AGE, count(\*) as n\_rows

from MillimanData.USR.VW\_CLAIM

group by AGE

having count(\*) > 100000;

Search Databases

**FINDING the data you need is even more important than knowing how to manipulate it! 😊**

**Evaluate tables in a SQL server database (via SSMS)**

Say you know that there's some MARA risk score data stored somewhere in the database, but you're not sure about the details. Is there a table for 2021 MARA scores and another for 2022 MARA scores? How can you be sure that you're using the correct/current table? Start with a high-level search and narrow it down from there:

-- evaluate all tables and views in a database (e.g. MillimanData):

select

T.name as Table\_Name,

C.name as Column\_Name,

P.name as Data\_Type

from sys.objects as T

join sys.columns as C on T.object\_id = C.object\_id

join sys.types as P on C.system\_type\_id = P.system\_type\_id

where T.type\_desc = 'USER\_TABLE' or T.type\_desc = 'VIEW'

order by T.name;

**Evaluate fields in a SQL server table (via SSMS)**

Say you have a list of tables, and you need to figure out which you should use. What column names are in your table of interest? What data types do these columns contain? Query the column names and data types for a specific table to learn more:

-- evaluate a table or view in a database (e.g. VW\_CLAIM in MillimanData):

select

T.name as Table\_Name,

C.name as Column\_Name,

P.name as Data\_Type

from sys.objects as T

join sys.columns as C on T.object\_id = C.object\_id

join sys.types as P on C.system\_type\_id = P.system\_type\_id

where (T.type\_desc = 'USER\_TABLE' or T.type\_desc = 'VIEW')

and T.name = 'VW\_CLAIM'

order by T.name;

**Evaluate fields in a CLUNITY / UNITY table (via SQL Developer):**

Say you're looking for BMI data, but you have no idea where to find it. Is there a field with a name like "BMI" somewhere in the CLUNITY or UNITY database? If so, what database and table is the field in? Note: Use % as a wildcard character before and after your search keyword. Doing so will return results if the field is called "Member\_BMI" or "BMI\_Date" etc.

SELECT

owner,

table\_name,

column\_name

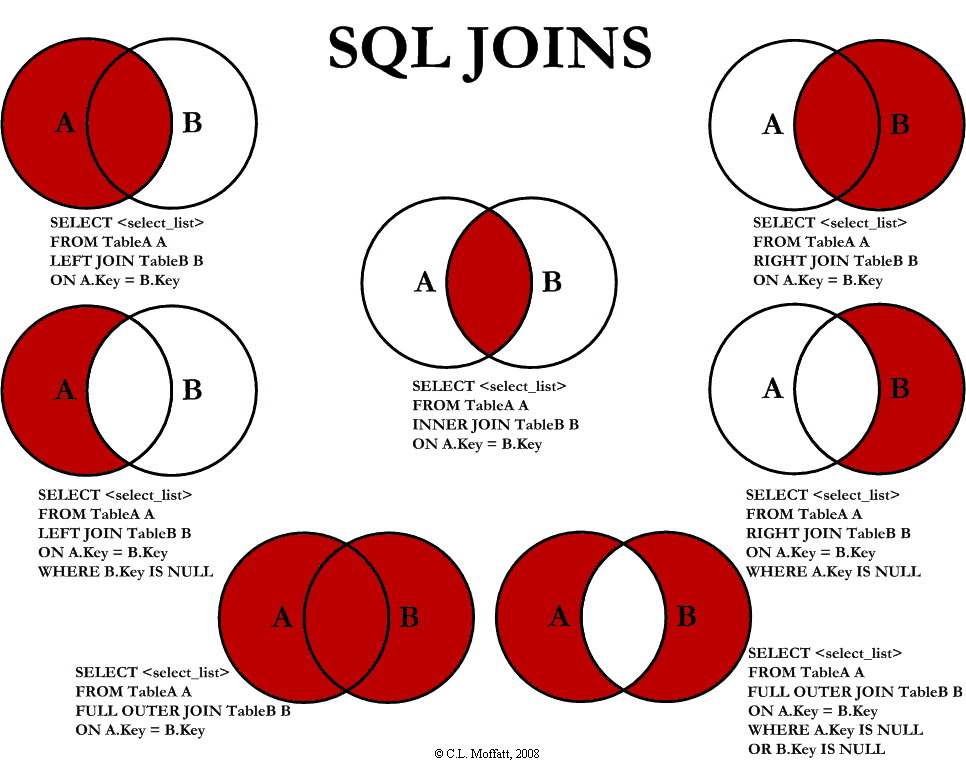
FROM all\_tab\_columns

WHERE column\_name LIKE '%BMI%'

AND OWNER IN ('CLUNITY', 'UNITY');

Join Data

Most queries require joining data across multiple sources. For example, you might have data about members in one table and data about providers in another table. This page describes various types of joins and how to write them.



select distinct

-- note "m" and "c" prefixes correspond to aliases below

m.MEMBER\_ID,

m.MEM\_ZIP,

c.AGE

from MillimanData.USR.MEMBER as m

**inner join** MillimanData.USR.VW\_CLAIM as c

-- join based on the tables' shared field (member ID)

on m.MEMBER\_ID = c.MEMBER\_ID;

-- compare results for inner join vs. left join

select distinct

m.MEMBER\_ID,

m.MEM\_ZIP,

c.AGE

from MillimanData.USR.MEMBER as m

**left join** MillimanData.USR.VW\_CLAIM as c

on m.MEMBER\_ID = c.MEMBER\_ID;

-- see how there are sometimes NULL values for AGE

-- compare results for left join vs. full join

select distinct

m.MEMBER\_ID,

m.MEM\_ZIP,

c.AGE

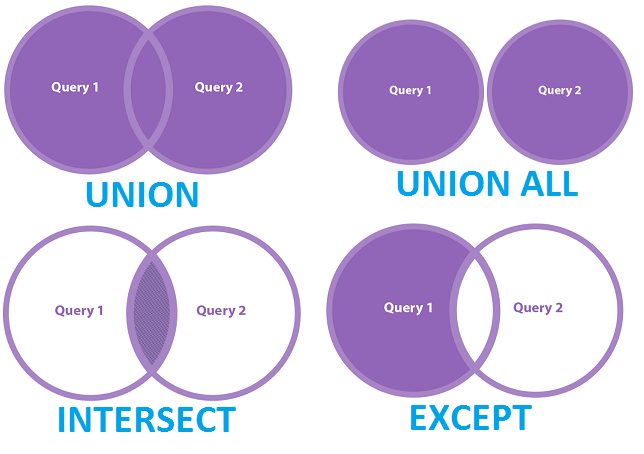
from MillimanData.USR.MEMBER as m

**full join** MillimanData.USR.VW\_CLAIM as c

on m.MEMBER\_ID = c.MEMBER\_ID;

-- see how there are sometimes NULL values for ZIP and AGE

UNION and UNION ALL are like joins, but “stack” records on top of each other from one table to the next. Records must be of the same data type (e.g. both characters). For R users, JOINS are like cbind and UNION ALL is like rbind. UNION ALL will extract all records from two tables, while INTERSECT will only return records that both tables have in common.



select distinct MEMBER\_ID

from MillimanData.USR.VW\_CLAIM

**union**

select distinct MEMBER\_ID

from MillimanData.USR.MEMBER

order by MEMBER\_ID;

-- union stacks member ID values from both tables

-- note the large number of rows returned with union

-- compare results for union vs. intersect

select distinct MEMBER\_ID

from MillimanData.USR.VW\_CLAIM

**intersect**

select distinct MEMBER\_ID

from MillimanData.USR.MEMBER

order by MEMBER\_ID;

-- see how there are fewer rows returned with intersect

Categorize and Filter Data

Often we need to create a categorical variable based on data from an existing field (e.g. obese vs. non-obese, based on BMI) to filter query results, to group data for summary tables, or to evaluate a variable’s distribution. This page shows how to group data in various ways.

## Filter query results with CASE WHEN

CASE statements allow you to categorize data that you want to pull vs. data to exclude. You can use a CASE statement as a filter in the WHERE clause:

select top(100)

MEMBER\_ID,

AGE

from MillimanData.USR.VW\_CLAIM

where

case

when AGE <= 5 then 'Include'

when AGE > 5 then 'Exclude'

end = 'Include';

## Filter query results with DECLARE

-- declare and set variables

declare @start date

declare @stop date;

set @start = '2022-01-01'

set @stop = '2022-01-31';

select top(100)

MEMBER\_ID,

PAID\_DATE

from MillimanData.USR.VW\_CLAIM

-- use declared variables to filter

where PAID\_DATE between @start and @stop;

## Create a categorical variable with CASE WHEN

select top(100)

MEMBER\_ID,

AGE,

case

when AGE >= 65 then 'senior'

when AGE between 18 and 64 then 'adult'

else 'child'

end as AGE\_GROUP

from MillimanData.USR.VW\_CLAIM;

Subqueries and CTEs

**A subquery is a query nested inside another query.** A subquery can be placed in any part of your query: SELECT, FROM, WHERE, or GROUP BY clause. Where you place it depends on what you want your final data to look like. A subquery can return a variety of info: a list, a table, or a single value.

**Subqueries allow you to:**

* + compare summarized values to detailed data.
  + better structure or reshape your data.
  + combine data from tables where you can't perform a join directly.

**Subqueries in the WHERE clause**

These can only return a single column – very useful for generating a filtering list.

**Subqueries in the FROM clause**

These are useful for restructuring and transforming your data, or when calculating aggregates of aggregate information. You can create more than one subquery in the FROM statement of any main query. When you do so, make sure that you give each subquery an alias, and make sure that you are able to JOIN them to each other. You can also join a subquery to any existing table in your database. Again, however, you need to make sure you have a column in the subquery that you can use with the JOIN you'd like to perform.

**Subqueries in the SELECT clause**

These return a single, aggregate value. For example, you may want to see how much an individual score deviates from an average score. The subquery needs to return a single value (e.g. an average score). If your subquery result returns multiple rows, your entire query will generate an error. Note that filtering the main query does not filter the subquery, and vice versa. Since the subquery is processed before the main query, you'll need to include relevant filters in the subquery AND in the main query.

**Common Table Expressions (CTEs)**

CTEs are a special type of subquery that is declared ahead of your main query. Instead of wrapping subqueries inside a statement, you name it using the WITH statement, and then reference it by name later in the FROM statement, as if it were any other table in your database.

**CTEs have numerous benefits over a subquery:**

* The CTE is run only once, and then stored in memory, so it **often reduces query runtime**.
* CTEs are an excellent tool for **organizing long and complex queries**. You can declare as many CTEs as you need, one after another. You can also reference info in CTEs declared earlier. For example, if you have 3 CTEs in a query, your third CTE can retrieve info from the first and second CTE.

-- identify active members with a CTE

with act as (

select

mem\_number

from

clunity.coverage\_member\_list mem

join clunity.coverage cov

on mem.coverage\_id = cov.coverage\_id

join clunity.plan\_grp ppg

on cov.plan\_grp\_id = ppg.plan\_grp\_id

join clunity.plan\_grp\_ben\_plan ben

on ppg.plan\_grp\_id = ben.plan\_grp\_id

join clunity.clarity\_epm pay

on ben.ben\_plan\_payor\_id = pay.payor\_id

where

-- member currently in effect

-- NOTE: NVL assigns a default value to a field where it may return NULL. For example, Member Effective To Date can sometimes be NULL due to no termination date listed for the member. We often use 12/31/2999 to replace these NULL values.

mem.mem\_eff\_from\_date <= sysdate

and nvl(mem.mem\_eff\_to\_date,'31-DEC-2999') >=

to\_date(to\_char(sysdate,'dd-mm-yyyy'),'dd-mm-yyyy')

-- and benefit plan is active

and ben.ben\_plan\_eff\_date <= sysdate

and nvl(ben.ben\_plan\_term\_dt,'31-DEC-2999') >=

to\_date(to\_char(sysdate,'dd-mm-yyyy'),'dd-mm-yyyy')

-- and coverage is verified an in effect

and mem.mem\_covered\_yn = 'Y'

-- and under a license of QHBPC, QTZ, QHPB, QHPMC, or QHIC

and pay.rpt\_grp\_six IN (1,2,3,4,5)

)

-- get real member number and deidentified member number

select map.mem\_number, map.mem\_deident

from unity.map\_mem\_number map

-- join with active members CTE to get results only for active members

inner join act

on act.mem\_number = map.mem\_number;

Data Types

**Working with TEXT data types (CHAR, VARCHAR, TEXT):**

* + LENGTH() or CHAR\_LENGTH() returns the number of characters from the provided string, excluding the blanks at the end. It has only one parameter: the string whose length you want to calculate.
  + CHARINDEX() looks for a character expression in a given string and returns the starting position of the first occurrence it finds. It has 2 mandatory parameters and 1 optional parameter: the expression we are looking for, the string in which we do the search and a value expressing the starting position of the search. If this is not specified, the search will start at the beginning.
  + PATINDEX() is similar to CHARINDEX() but more powerful. It returns the starting position of the first occurrence of a pattern in a specified expression. You can use wildcard characters (% and \_) in the expression you are looking for. If you want to match any of the characters in a list, you should include them in square brackets.
  + LIKE() and ILIKE() are akin to grep and grepl in R.
  + POSITION() returns an integer which represents the number of characters from left to right before the search string is located, e.g. the position of the @ in the email column.
  + LOWER() returns the same string, with all characters converted to lowercase.
  + UPPER() returns the same string, with all characters converted to uppercase.
  + INITCAP() returns the same string, converted to title case.
  + LEFT() and RIGHT() receive two parameters: the string you're working with and the number of characters you need from it. LEFT() will return that number of characters from the beginning of the string and RIGHT() will do the same, starting with the end of the string.
  + TRIM() removes blank characters from both the beginning and end of the string.
  + LTRIM() and RTRIM() remove blank characters from the beginning/end of a string.
  + LPAD() and RPAD() append a character or string to another string by a specified number of characters. This is useful when you need a field to be the same length and want to pad the string with a certain character like a space or a tab.
  + REPLACE() needs three parameters: the expression in which we're performing the search, the expression we're searching for and the replacement.
  + SUBSTRING() needs three parameters: the expression from which you want to extract the substring, the starting position of your substring and the total number of characters you want to return.

select distinct

RATE\_CELL\_IND,

substring(RATE\_CELL\_IND, 1, 1) as RC\_1,

substring(RATE\_CELL\_IND, 2, 2) as RC\_2\_3,

substring(RATE\_CELL\_IND, 4, 2) as RC\_4\_5

from PROD\_LND\_UTY.DHS\_820\_LAND

group by RATE\_CELL\_IND;

* + LEFT() extracts the first n characters of a string.
  + RIGHT() extracts the last n characters of a string.
  + CONCAT() joins values together. CONCAT\_WS(), meaning CONCAT with separator, receives a character value as the first parameter, which is called "the separator". This value is introduced between the strings that are being concatenated.
  + STRING\_AGG() is also used to concatenate a series of strings. The separator is placed between each string but not at the end. It has an optional order\_clause.
  + STRING\_SPLIT() is the opposite of STRING\_AGG(): it splits a bigger string into pieces based on a given separator. Because the result of the function is a table, it cannot be used as a column in the SELECT clause; you can only use it in the FROM clause, just like a normal table.

**Working with DATE and TIME data types:**

* TIMESTAMPs contain both a date value and a time value with microsecond precision. They are commonly used to record an exact point in time like when a payment was made or a record was last updated. TIMESTAMPs in PostgreSQL use the ISO 8601 format (YYYY-MM-DD).
* NOW() returns a timestamp for the current date and time at the microsecond precision with time zone.
* CURRENT\_TIMESTAMP() returns the same result of the NOW() function, but you can specify a precision parameter to have the seconds rounded to the number of fractional digits specified.
* CURRENT\_DATE() and CURRENT\_TIME() return either a date value or a time value, without the precision of a timestamp.
* DATE and TIME types are essentially the date and time values of the TIMESTAMP. DATE types contain a date value with no time of day while TIME types contain the time of day but without a date.
* INTERVAL types store date and time data as a period of time in years, months, days, hours, seconds, etc. They are useful when you want to do arithmetic on date and time columns (e.g. add 5 days):

SELECT

CURRENT\_TIMESTAMP::timestamp AS right\_now,

INTERVAL '5 days' + CURRENT\_TIMESTAMP AS five\_days\_from\_now;

* You can also add integer values to date data types. When adding integers to date values, the implied precision is days.
* AGE() takes two timestamp arguments and subtracts the first argument from the second and returns an INTERVAL as a result.
* EXTRACT() and DATE\_PART() both need two parameters – the field identifier and the source. The field parameter is an identifier (or string if you are using DATE\_PART) that indicates what sub-field that you want to extract from the source. The various field identifiers include year, month, quarter, day of week, etc. The source parameter needs to be a valid timestamp, time, or interval data type. Both EXTRACT and DATE\_PART will produce identical results and can be used interchangeably with only slight variations in how you pass in the field and source parameters.
* See <https://docs.microsoft.com/en-us/sql/t-sql/functions/datepart-transact-sql>
* DATE\_TRUNC() will truncate timestamp or interval data types to return a timestamp or interval at a specified precision. The precision values are a subset of the field identifiers that can be used with the EXTRACT() and DATE\_PART() functions. Unlike EXTRACT() and DATE\_PART() functions, DATE\_TRUNC() will return an interval or timestamp rather than a numeric value.

-- Show each date part for the fall of the Berlin Wall

DECLARE

@BerlinWallFalls DATETIME2(7) = '1989-11-09 23:49:36.2294852';

SELECT

DATEPART(YEAR, @BerlinWallFalls) AS TheYear,

DATEPART(MONTH, @BerlinWallFalls) AS TheMonth,

DATEPART(DAY, @BerlinWallFalls) AS TheDay,

DATEPART(DAYOFYEAR, @BerlinWallFalls) AS TheDayOfYear,

-- Day of week is WEEKDAY

DATEPART(WEEKDAY, @BerlinWallFalls) AS TheDayOfWeek,

DATEPART(WEEK, @BerlinWallFalls) AS TheWeek,

DATEPART(SECOND, @BerlinWallFalls) AS TheSecond,

DATEPART(NANOSECOND, @BerlinWallFalls) AS TheNanosecond;

-- Show the NAME of each date part

DECLARE

@BerlinWallFalls DATETIME2(7) = '1989-11-09 23:49:36.2294852';

SELECT

DATENAME(YEAR, @BerlinWallFalls) AS TheYear,

DATENAME(MONTH, @BerlinWallFalls) AS TheMonth,

DATENAME(DAY, @BerlinWallFalls) AS TheDay,

DATENAME(DAYOFYEAR, @BerlinWallFalls) AS TheDayOfYear,

-- Day of week is WEEKDAY

DATENAME(WEEKDAY, @BerlinWallFalls) AS TheDayOfWeek,

DATENAME(WEEK, @BerlinWallFalls) AS TheWeek,

DATENAME(SECOND, @BerlinWallFalls) AS TheSecond,

DATENAME(NANOSECOND, @BerlinWallFalls) AS TheNanosecond;

-- Round down @SomeTime to the day, hour, and minute

DECLARE

@SomeTime DATETIME2(7) = '2018-06-14 16:29:36.2248991';

SELECT

DATEADD(DAY, DATEDIFF(DAY, 0, @SomeTime), 0) AS RoundedToDay,

DATEADD(HOUR, DATEDIFF(HOUR, 0, @SomeTime), 0) AS RoundedToHour,

DATEADD(MINUTE, DATEDIFF(MINUTE, 0, @SomeTime), 0) AS RoundedToMin;

**Changing data types**

* + CAST() lets you convert one data type to another data type, such as an integer to a decimal. When you cast a column as a different type, the data is converted to the new type only for the current query.
    - Casting from numeric to integer rounds the value to the nearest integer.
  + CONVERT() is used for converting between data types, just like CAST(). It takes three parameters: a data type, an input, and an optional style.
  + FORMAT() takes three parameters: an input value, a format code, and an optional culture. The FORMAT() function can be slower than CAST() or CONVERT(), so keep this in mind as you scale up your data.

DECLARE

@Python3ReleaseDate DATETIME2(3) = '2008-12-03 19:45:00.033';

SELECT

-- 20081203

FORMAT(@Python3ReleaseDate, 'yyyyMMdd') AS F1,

-- 2008-12-03

FORMAT(@Python3ReleaseDate, 'yyyy-MM-dd') AS F2,

-- Dec 03+2008 (the + is just a "+" character)

FORMAT(@Python3ReleaseDate, 'MMM dd+yyyy') AS F3,

-- 12 08 03 (month, two-digit year, day)

FORMAT(@Python3ReleaseDate, 'MM yy dd') AS F4,

-- 03 07:45 2008.00

-- (day hour:minute year.second)

FORMAT(@Python3ReleaseDate, 'dd hh:mm yyyy.ss') AS F5;