# Relational modelling

## Database structure

* Data is organized in database in the form of tables as rows and columns
* Constraints define how data is saved
* Entities are essential building blocks in database. It is a basic object with independent existence.
* An entity instance is a single occurrence of an entity
* Attributes provide information for each entity. i.e, it describes the characteristics or additional details about the entity.

## Normalization

* Formal technique for analyzing a relation based on its primary key and functional dependencies between its attributes.

**Goal**: reduce redundancy, reduce anomalies

* As normalization proceeds, relations become progressively more restricted (stronger) in format and also less vulnerable to insert/update/delete anomalies

## Relationships

Meaningful business associations.

### Multiplicity

* Cardinality - maximum number of possible relationship occurrences for an entity
* Participation - optional or mandatory

### Types of relations

There are 3 different types of relations in the database:

* one-to-one
* one-to-many, and
* many-to-many

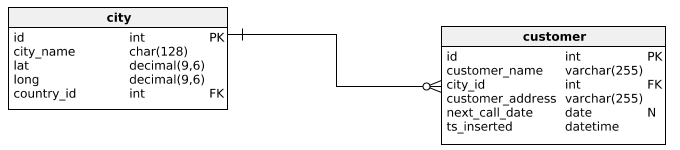
### One-to-many relation

It’s the most commonly used and the remaining two are “subtypes” of this one. Let’s start with a real-life problem.

**Example**

Imagine that we want to store a list of all our customers in the database. For each customer, we also want to store the city where this customer is located, and we know that the customer will be in exactly one city.

This the typical example of one-to-many relation and this is how we solved it in our model:



We simply established a relation from the **city.id** to **customer.city\_id**. And this works, because the customer can be only in one city and the city could have many different customers located in it.

When you want to determine the nature of the relation you need to establish between two tables just do this. In our example – For **one** city, we could have **many** different customers located in it. And the other way around – For **one** customer, we can have only **one** city it’s located in.

So, how to choose between these 3 different types of relations? If you said the word “many” only once, then this is one-to-many relation. If you would use the word “many” two times, the relation would be many-to-many. And if you wouldn’t use it at all, then it would be one-to-one.

**Tip**: put the primary key of one side as a FK in the many side table.

### **Many-to-many relation**

The second out of three types of relations is a many-to-many type. This type is used when both tables could have multiple rows on the other side. Let’s see an example.

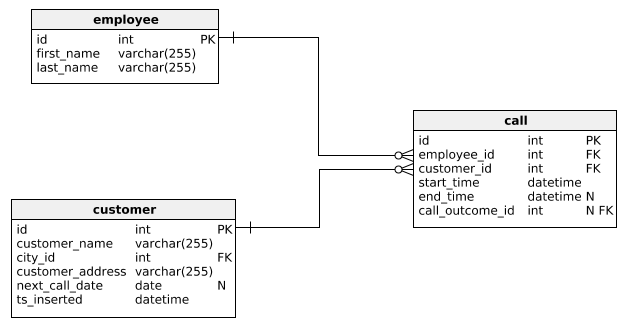
**Example**

We need to store calls between employees and customers.

**One** employee, during the time, could call **many** customers. Also, **one** customer, during the time, could receive calls from **many** employees.

Notice that we’ve mentioned the word “many” two times. This is the signal we need to resolve this using many-to-many relation (out of 3 types of relations we have on disposal). To solve it we’ll create an associative entity as follows:

* Add a table between tables **employee** and **customer**
* Add foreign keys (**employee\_id** & **customer\_id**) to that new table (**call**)



Now, when we look from the employee perspective, **one** employee could make **many** (multiple) calls. On the other hand, **one** customer could be related to **many** (multiple) calls. Therefore, many-to-many relation is implemented with adding a new table and one-to-many relations from both sides.

### **One-to-one relation**

Compared to previously mentioned types of relations, this one is really rarely used. Let’s go with an example.

**Example**

In the database, we want to store employees, but also their valid identity cards. We’re not interested in storing any other types of documents or identity cards that were previously valid, so we need exactly 1 (or none) identity card for 1 employee.

Let’s check this truly is a one-to-one relation. We’ve been given these rules: **One** employee could have only **one** valid identity card in our system. **One** identity card could belong to only **one** employee. We haven’t used the word “many”, so this can’t be any type of relation including the word “many”.

## Denormalization

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database.

**When and Why to Use Denormalization**

1. **Maintaining history**: Data can change during time, and we need to store values that were valid when a record was created. What kind of changes do we mean? Well, a person’s first and last name can change; a client also can change their business name or any other data. Task details should contain values that were actual at the moment a task was generated. We wouldn’t be able to recreate past data correctly if this didn’t happen. We could solve this problem by adding a table containing the history of these changes. In that case, a select query returning the task and a valid client name would become more complicated. Maybe an extra table isn’t the best solution.
2. **Improving query performance**: Some of the queries may use multiple tables to access data that we frequently need. Think of a situation where we’d need to join 10 tables to return the client’s name and the products that were sold to them. Some tables along the path could also contain large amounts of data. In that case, maybe it would be wise to add a client\_id attribute directly to the products\_sold table.
3. **Speeding up reporting**: We need certain statistics very frequently. Creating them from live data is quite time-consuming and can affect overall system performance. Let’s say that we want to track client sales over certain years for some or all clients. Generating such reports out of live data would “dig” almost throughout the whole database and slow it down a lot. And what happens if we use that statistic often?
4. **Computing commonly-needed values up front:** We want to have some values ready-computed so we don’t have to generate them in real time.

It’s important to point out that you don’t need to use denormalization if there are no performance issues in the application. But if you notice the system is slowing down – or if you’re aware that this could happen – then you should think about applying this technique. Before going with it, though, consider other options, like query optimization and proper indexing.

# SQL Server Essentials

## Components

* Database Engine
* Replication
* Full Text Search
* Analysis Services (SSAS)
* Reporting Services (SSRS)
* Integration Services (SSIS – ETL)
* Management Studio (SSMS)
* Configuration Manager (SSCM)
* Development Tools (SSD

# Transact SQL (T-SQL)

It is a query language, based on ANSI SQL Standard.

T-SQL has three general types of languages

1. **Data Definition Language - DDL**

CREATE, ALTER, DROP

1. **Data Manipulation Language – DML**

SELECT, INSERT, UPDATE, DELETE

1. **Data Control Language - DCL**

GRANT, REVOKE

## Data Types

### Main Categories

* Numeric
  + Exact numeric
  + Approximate numeric
* Temporal
* String
  + Unicode characters
  + Binary strings
* Other

### Exact numerics

* [bigint](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql?view=sql-server-ver15)
* [numeric](https://docs.microsoft.com/en-us/sql/t-sql/data-types/decimal-and-numeric-transact-sql?view=sql-server-ver15)
* [bit](https://docs.microsoft.com/en-us/sql/t-sql/data-types/bit-transact-sql?view=sql-server-ver15)
* [smallint](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql?view=sql-server-ver15)
* [decimal](https://docs.microsoft.com/en-us/sql/t-sql/data-types/decimal-and-numeric-transact-sql?view=sql-server-ver15)
* [smallmoney](https://docs.microsoft.com/en-us/sql/t-sql/data-types/money-and-smallmoney-transact-sql?view=sql-server-ver15)
* [int](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql?view=sql-server-ver15)
* [tinyint](https://docs.microsoft.com/en-us/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql?view=sql-server-ver15)
* [money](https://docs.microsoft.com/en-us/sql/t-sql/data-types/money-and-smallmoney-transact-sql?view=sql-server-ver15)

### Approximate numerics

* [float](https://docs.microsoft.com/en-us/sql/t-sql/data-types/float-and-real-transact-sql?view=sql-server-ver15)
* [real](https://docs.microsoft.com/en-us/sql/t-sql/data-types/float-and-real-transact-sql?view=sql-server-ver15)

### Date and time

* [date](https://docs.microsoft.com/en-us/sql/t-sql/data-types/date-transact-sql?view=sql-server-ver15)
* [datetimeoffset](https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetimeoffset-transact-sql?view=sql-server-ver15)
* [datetime2](https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetime2-transact-sql?view=sql-server-ver15)
* [smalldatetime](https://docs.microsoft.com/en-us/sql/t-sql/data-types/smalldatetime-transact-sql?view=sql-server-ver15)
* [datetime](https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetime-transact-sql?view=sql-server-ver15)
* [time](https://docs.microsoft.com/en-us/sql/t-sql/data-types/time-transact-sql?view=sql-server-ver15)

### Character strings

* [char](https://docs.microsoft.com/en-us/sql/t-sql/data-types/char-and-varchar-transact-sql?view=sql-server-ver15)
* [varchar](https://docs.microsoft.com/en-us/sql/t-sql/data-types/char-and-varchar-transact-sql?view=sql-server-ver15)
* [text](https://docs.microsoft.com/en-us/sql/t-sql/data-types/ntext-text-and-image-transact-sql?view=sql-server-ver15)

### Binary strings

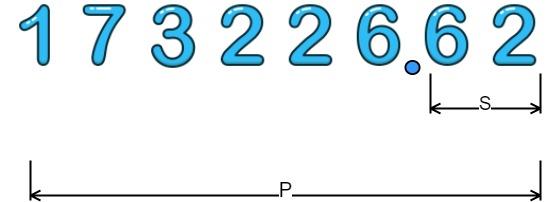
* [binary](https://docs.microsoft.com/en-us/sql/t-sql/data-types/binary-and-varbinary-transact-sql?view=sql-server-ver15)
* [varbinary](https://docs.microsoft.com/en-us/sql/t-sql/data-types/binary-and-varbinary-transact-sql?view=sql-server-ver15)
* [image](https://docs.microsoft.com/en-us/sql/t-sql/data-types/ntext-text-and-image-transact-sql?view=sql-server-ver15)

### Other data types

* [cursor](https://docs.microsoft.com/en-us/sql/t-sql/data-types/cursor-transact-sql?view=sql-server-ver15)
* [rowversion](https://docs.microsoft.com/en-us/sql/t-sql/data-types/rowversion-transact-sql?view=sql-server-ver15)
* [hierarchyid](https://docs.microsoft.com/en-us/sql/t-sql/data-types/hierarchyid-data-type-method-reference?view=sql-server-ver15)
* [uniqueidentifier](https://docs.microsoft.com/en-us/sql/t-sql/data-types/uniqueidentifier-transact-sql?view=sql-server-ver15)
* [sql\_variant](https://docs.microsoft.com/en-us/sql/t-sql/data-types/sql-variant-transact-sql?view=sql-server-ver15)
* [xml](https://docs.microsoft.com/en-us/sql/t-sql/xml/xml-transact-sql?view=sql-server-ver15)
* [Spatial Geometry Types](https://docs.microsoft.com/en-us/sql/t-sql/spatial-geometry/spatial-types-geometry-transact-sql?view=sql-server-ver15)
* [Spatial Geography Types](https://docs.microsoft.com/en-us/sql/t-sql/spatial-geography/spatial-types-geography?view=sql-server-ver15)
* [table](https://docs.microsoft.com/en-us/sql/t-sql/data-types/table-transact-sql?view=sql-server-ver15)

### decimal and numeric (Transact-SQL)

**decimal**[ **(**p[ **,**s] **)**] and **numeric**[ **(**p[ **,**s] **)**]



p (precision)  
The maximum total number of decimal digits to be stored. This number includes both the left and the right sides of the decimal point. The precision must be a value from 1 through the maximum precision of 38. The default precision is 18.

s (scale)  
The number of decimal digits that are stored to the right of the decimal point. This number is subtracted from p to determine the maximum number of digits to the left of the decimal point. Scale must be a value from 0 through p, and can only be specified if precision is specified. The default scale is 0 and so 0 <= s <= p.

There is a small difference between NUMERIC(p,s) and DECIMAL(p,s). NUMERIC determines the **exact precision and scale**. DECIMAL specifies **only the exact scale**; the precision is equal or greater than what is specified by the coder. DECIMAL columns can have a larger-than-specified precision if this is more convenient or efficient for the database system.

## Manage data with Transact- SQL

To write correct and robust T-SQL code, it’s important to first understand the roots of the language, as well as a concept called logical query processing. You also need to understand the structure of the SELECT statement and how to use its clauses to perform data manipulation tasks like filtering and sorting. You often need to combine data from different sources, and one of the ways to achieve this in T-SQL is using set operators.

### Foundations of T-SQL

T-SQL is based on strong mathematical foundations. Understanding some of the key principles from those foundations can help you better understand the language you are dealing with. Then you will think in T-SQL terms when coding in T-SQL, as opposed to coding with T-SQL while thinking in procedural terms.

SQL Server also supports other languages, like Microsoft Visual C# and Microsoft Visual Basic, but T-SQL is usually the preferred language for data management and manipulation.

Standard SQL is based on the relational model, which is a mathematical model for data management and manipulation. A relation in the relational model is what SQL represents with a table. A relation has a heading and a body. The heading is a set of attributes (what SQL attempts to represent with columns), each of a given type. An attribute is identified by name and type name. The body is a set of tuples (what SQL attempts to represent with rows). Each tuple’s heading is the heading of the relation. Each value of each tuple’s attribute is of its respective type.

Even when the table doesn’t allow duplicate rows, a query against the table can still return duplicate rows in its result. Consider the following query:

USE TSQLV4;

SELECT country

FROM HR.Employees;

T-SQL does give you enough tools so that if you want to follow relational theory, you can do so. For example, the language provides you with a DISTINCT clause to remove duplicates. Here’s the revised query:

SELECT DISTINCT country

FROM HR.Employees;

Another fundamental aspect of a set is that there’s no relevance to the order of the elements. For this reason, rows in a table have no particular order, conceptually.

SQL Server can choose between different physical access methods to process the query, knowing that it doesn’t need to guarantee the order in the result. For example, SQL Server could decide to parallelize the query or scan the data in file order (as opposed to index order).

If you do need to guarantee a specific presentation order to the rows in the result, you need to add an ORDER BY clause to the query, as follows:

SELECT empid, lastname

FROM HR.Employees

ORDER BY empid;

This time, the result isn’t relational, it’s what standard SQL calls a cursor. The order of the rows in the output is guaranteed based on the empid attribute.

T-SQL does keep track of ordinal positions of columns based on their order of appearance in the table definition. When you issue a query with SELECT \*, you are guaranteed to get the columns in the result based on definition order. Also, T-SQL allows referring to ordinal positions of columns from the result in the ORDER BY clause, as follows:

SELECT empid, lastname

FROM HR.Employees

ORDER BY 1;

However, this practice is not recommended.

T-SQL has another deviation from the relational model in that it allows defining result columns based on an expression without assigning a name to

the target column. For example, the following query is valid in T-SQL:

SELECT empid, firstname + ' ' + lastname

FROM HR.Employees;

But according to the relational model, all attributes must have names. In order for the query to be relational, you need to assign an alias to the target

attribute. You can do so by using the AS clause, as follows:

SELECT empid, firstname + ' ' + lastname AS fullname

FROM HR.Employees;

Also, with T-SQL a query can return multiple result columns with the same name. For example, consider a join between two tables, T1 and T2, both with a column called keycol. With T-SQL, a SELECT list can look like the following:

SELECT T1.keycol, T2.keycol ...

For the result to be relational, all attributes must have unique names, so you would need to use different aliases for the result attributes as follows:

SELECT T1.keycol AS key1, T2.keycol AS key2 ...

#### Two value vs three value logic:

As for predicates, following the law of excluded middle in mathematical

logic, a predicate can evaluate to true or false. In other words, predicates are

supposed to use two-valued logic. However, Codd wanted to reflect the

possibility for values to be missing in his model. He referred to two kinds of

missing values: missing but applicable (A-Values marker) and missing but

inapplicable (I-Values marker). As an example for a missing but applicable

case, consider a mobilephone attribute of an employee. Suppose that an

employee has a mobile phone, but did not want to provide this information,

for example, for privacy reasons. As an example for a missing but

inapplicable case, consider a salescommission attribute of an employee. This

attribute is applicable only to sales people, but not to other kinds of

employees. According to Codd, a language based on his model should

provide two different markers for the two cases. T-SQL—again, based on

standard SQL—implements only one general-purpose marker called NULL

for any kind of missing value. This leads to three-valued predicate logic.

Namely, when a predicate compares two values, for example, mobilephone =

‘(425) 555-0136’, if both are present, the result evaluates to either true or

false. But if at least one of them is NULL, the result evaluates to a third

logical value—unknown. That’s the case both when you use the equality

operator = and when you use an inequality operator such as: <>, >, >=, <, <=.

**Using correct terminology**

As an example of incorrect terms in T-SQL, people often use the terms

“field” and “record” to refer to what T-SQL calls “column” and “row,”

respectively. Fields and records are physical. Fields are what you have in user

interfaces in client applications, and records are what you have in files and

cursors. Tables are logical, and they have logical rows and columns.

Another example of an incorrect term is referring to “NULL values.” A

NULL is a marker for a missing value—not a value itself. Hence, the correct

usage of the term is either just “NULL” or “NULL marker.” Personally, I

prefer the former.

### Understanding logical query processing

T-SQL has both logical and physical sides to it. The logical side is the

conceptual interpretation of the query that explains what the correct result of

the query is. The physical side is the processing of the query by the database

engine. Physical processing must produce the result defined by logical query

processing. To achieve this goal, the database engine can apply optimization.

Optimization can rearrange steps from logical query processing or remove

steps altogether, but only as long as the result remains the one defined by

logical query processing. The focus of this section is logical query processing

—the conceptual interpretation of the query that guarantees returning what I

defined as the correct result.

T-SQL, being based on standard SQL, is a declarative English-like language.

In this language, declarative means you define what you want, as opposed to

imperative languages that define also how to achieve what you want.

You provide your instructions in an Englishlike manner. For example, consider the instruction, “Bring me a soda from the refrigerator.” Observe that in the English instruction, the object comes before the location. Consider the following request in T-SQL:

SELECT shipperid, phone, companyname

FROM Sales.Shippers;

Observe the similarity of the query’s keyed-in order to English. The query

first indicates the SELECT list with the attributes you want to return, and

then the FROM clause with the table you want to query.

Now try to think of the order in which the request needs to be logically

interpreted. For example, how would you define the instructions to a robot

instead of a human? The original English instruction to get a soda from the

refrigerator would probably need to be revised to something like, “Go to the

refrigerator; open the door; get a soda; bring it to me.”

Similarly, the logical processing of a query must first know which table is

being queried before it can know which attributes can be returned from that

table. Therefore, contrary to the keyed-in order of the previous query, the

logical query processing has to be as follows:

FROM Sales.Shippers

SELECT shipperid, phone, companyname

Following are the main query clauses specified in the order that you are

supposed to type them (known as “keyed-in order”):

**1.** SELECT

**2.** FROM

**3.** WHERE

**4.** GROUP BY

**5.** HAVING

**6.** ORDER BY

But as mentioned, the logical query processing order, which is the

conceptual interpretation order, is different. It starts with the FROM clause.

Here is the logical query processing order of the six main query clauses:

**1.** FROM

**2.** WHERE

**3.** GROUP BY

**4.** HAVING

**5.** SELECT

**6.** ORDER BY

Consider the following query as an example:

SELECT country, YEAR(hiredate) AS yearhired, COUNT(\*) AS

numemployees

FROM HR.Employees

WHERE hiredate >= '20140101'

GROUP BY country, YEAR(hiredate)

HAVING COUNT(\*) > 1

ORDER BY country, yearhired DESC;

This query is issued against the HR.Employees table. It filters only

employees that were hired in or after the year 2014. It groups the remaining

employees by country and the hire year. It keeps only groups with more than

one employee. For each qualifying group, the query returns the hire year and

count of employees, sorted by country and hire year, in descending order.

Note that an alias created by the SELECT phase isn’t even visible to other

expressions that appear in the same SELECT list. For example, the following

query isn’t valid:

SELECT empid, country, YEAR(hiredate) AS yearhired,

yearhired - 1 AS prevyear

FROM HR.Employees;

This query generates error:

The reason that this isn’t allowed is that all expressions that appear in the

same logical query-processing step are treated as a set, and a set has no order.

In other words, conceptually, T-SQL evaluates all expressions that appear in

the same phase in an all-at-once manner

#### Getting started with the SELECT statement

**The FROM clause**

According to logical query processing, the FROM clause is the first clause to

be evaluated logically in a SELECT query. The FROM clause has two main

roles:

* It’s the clause where you indicate the tables that you want to query.
* It’s the clause where you can apply table operators like joins to input

tables.

SELECT empid, firstname, lastname, country

FROM HR.Employees;

Observe the use of the two-part name to refer to the table. The first part

(HR) is the schema name and the second part (Employees) is the table name.

In some cases, T-SQL supports omitting the schema name, as in FROM

Employees, in which case it uses an implicit schema name resolution process.

It is considered a best practice to always explicitly indicate the schema name.

This practice can prevent you from ending up with a schema name that you

did not intend to use, and can also remove the cost involved in the implicit

resolution process, although this cost is minor.

In the FROM clause, you can alias the queried tables with your chosen

names. You can use the form <table> <alias>, as in HR.Employees E, or

<table> AS <alias>, as in HR.Employees AS E. The latter form is more

readable. When using aliases, the convention is to use short names, typically

one letter that is somehow indicative of the queried table, like E for

Employees. Then, when referring to an ambiguous column name in a multitable query (same column name appears in multiple queried tables), to avoid ambiguity, you add the table alias as a column prefix.

Note that if you assign an alias to a table, you basically rename the table

for the duration of the query. The original table name isn’t visible anymore;

only the alias is. Normally, you can prefix a column name you refer to in a

query with the table name, as in Employees.empid. However, if you aliased

the Employees table as E, the reference Employees.empid is invalid; you

have to use E.empid, as the following example demonstrates:

SELECT E.empid, firstname, lastname, country

FROM HR.Employees AS E;

If you try running this code by using the full table name as the column

prefix, the code will fail.

**The SELECT clause**

The SELECT clause of a query has two main roles:

* It evaluates expressions that define the attributes in the query’s result, assigning them with aliases if needed.
* Using a DISTINCT clause, you can eliminate duplicate rows in the

result if needed.

T-SQL supports using an asterisk (\*) as an alternative to listing all

attributes from the input tables, but this is considered a bad practice for a

number of reasons. Often, you need to return only a subset of the input

attributes, and using an \* is just a matter of laziness. By returning more

attributes than you really need, you can prevent SQL Server from using what

would normally be considered covering indexes with respect to the

interesting set of attributes. You also send more data than is needed over the

network, and this can have a negative impact on the system’s performance. In

addition, the underlying table definition could change over time; even if,

when the query was initially authored, \* really represented all attributes you

needed; it might not be the case anymore at a later point in time. For these

reasons and others, it is considered a best practice to always explicitly list the

attributes that you need.

In the SELECT clause, you can assign your own aliases to the expressions

that define the result attributes. There are a number of supported forms of

aliasing:

* <expression> AS <alias> as in empid AS employeeid,
* <expression> <alias> as in empid employeeid,
* <alias> = <expression> as in employeeid = empid.

There are two main uses for intentional attribute aliasing. One is renaming

—when you need the result attribute to be named differently than the source

attribute—for example, if you need to name the result attribute employeeid

instead of empid, as follows:

SELECT empid AS employeeid, firstname, lastname

FROM HR.Employees;

Another use is to assign a name to an attribute that results from an

expression that would otherwise be unnamed. For example, suppose you need

to generate a result attribute from an expression that concatenates the

firstname attribute, a space, and the lastname attribute. You use the following

query:

SELECT empid, firstname + N' ' + lastname

FROM HR.Employees;

By aliasing the expression, you assign a name to the result attribute,

making the result of the query relational, as follows.

SELECT empid, firstname + N' ' + lastname AS fullname

FROM HR.Employees;

There’s an interesting difference between standard SQL and T-SQL in

terms of minimal SELECT query requirements. According to standard SQL,

a SELECT query must have at minimum FROM and SELECT clauses.

Conversely, T-SQL supports a SELECT query with only a SELECT clause

and without a FROM clause. Such a query is as if issued against an imaginary

table that has only one row. For example, the following query is invalid

according to standard SQL, but is valid according to T-SQL:

SELECT 10 AS col1, 'ABC' AS col2;

The output of this query is a single row with attributes resulting from the

expressions with names assigned using the aliases:

col1 col2

----------- ----

10 ABC

#### Filtering data with predicates

Filtering data is one of the most fundamental aspects of T-SQL querying.

Almost every query that you write involves some form of filtering.

**Predicates and three-valued-logic**

Consider the following query, which filters only employees from the US:

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE country = N'USA';

In case you’re wondering why the literal ‘USA’ is preceded with the letter

N as a prefix, that’s to denote a Unicode character string literal, because the

country column is of the data type NVARCHAR. Had the country column

been of a regular character string data type, such as VARCHAR, the literal

should have been just ‘USA’.

When NULLs are not possible in the data that you’re filtering, such as in

the above example, T-SQL uses two-valued logic; namely, for any given row

the predicate can evaluate to either true or false. The filter returns only the

rows for which the predicate evaluates to true and discards the ones for which

the predicate evaluates to false.

However, when NULLs are possible in the data, things get trickier. For

instance, consider the following query:

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region = N'WA';

Here you’re looking for only those employees who are from Washington

(have WA in the region attribute). It’s clear that the predicate evaluates to

true for rows that have WA in the region attribute and that those rows are

returned. It’s also clear that the predicate would have evaluated to false had

there been any rows with a present region other than WA, for example CA,

and that those rows would have been discarded. However, remember that the

predicate evaluates to unknown for rows that have a NULL in the region

attribute, and that the WHERE clause discards such rows. This happens to be

the desired behavior in our case because you know that when the region is

NULL, it can’t be Washington. However, remember that even when you use

the inequality operator <> a comparison with a NULL yields unknown. For

instance, suppose that you wanted to return only employees with a region

other than Washington, and that you used the following query in attempt to

achieve this:

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region <> N'WA';

The predicate evaluates to false for rows with WA in the region attribute

and those rows are discarded. The predicate would have evaluated to true had

there been rows with a present region other than WA, and those rows would

have been returned. However, the predicate evaluates to unknown for rows

with NULL in the region attribute, and those rows get discarded, even though

you know that if region is NULL, it cannot be Washington. This query

returns an empty set because our sample data contains only rows with either

WA or NULL in the region attribute:

This is an example where you need to intervene and add some logic to

your query to also return the rows where the region attribute is NULL. Be

careful though not to use an equality operator when looking for a NULL

because remember that nothing is considered equal to a NULL—not even

another NULL. The following query still returns an empty set:

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region <> N'WA'

OR region = NULL;

T-SQL supports the IS NULL and IS NOT NULL operators to check if a

NULL is or isn’t present, respectively. Here’s the solution query that

correctly handles NULLs:

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region <> N'WA'

OR region IS NULL;

**Combining predicates**

You can combine predicates in the WHERE clause by using the logical

operators AND and OR. You can also negate predicates by using the NOT

logical operator.

What can be surprising to some is what happens when you negate

unknown—NOT unknown is still unknown. Recall from the previous section

the query that returned all employees from Washington; the query used the

predicate region = N’WA’ in the WHERE clause. Suppose that you want to

return the employees that are not from WA, and for this you use the predicate

NOT region = N’WA’. It’s clear that cases that return false from the positive

predicate (say the region is NY) return true from the negated predicate. It’s

also clear that cases that return true from the positive predicate (say the

region is WA) return false from the negated predicate. However, when the

region is NULL, both the positive predicate and the negated one return

unknown and the row is discarded. So the right way for you to include NULL

cases in the result—if that’s what you know that you need to do—is to use

the IS NULL operator, as in NOT region = N’WA’ OR region IS NULL.

As for combining predicates, there are several interesting things to note.

Some precedence rules determine the logical evaluation order of the different

predicates. The NOT operator precedes AND and OR, and AND precedes

OR. For example, suppose that the WHERE filter in your query had the

following combination of predicates:

**Click here to view code image**

WHERE col1 = 'w' AND col2 = 'x' OR col3 = 'y' AND col4 =

'z'

Because AND precedes OR, you get the equivalent of the following:

**Click here to view code image**

WHERE (col1 = 'w' AND col2 = 'x') OR (col3 = 'y' AND col4

= 'z')

Trying to express the operators as pseudo functions, this combination of

operators is equivalent to OR( AND( col1 = ‘w’, col2 = ‘x’ ), AND( col3 =

‘y’, col4 = ‘z’ ) ).

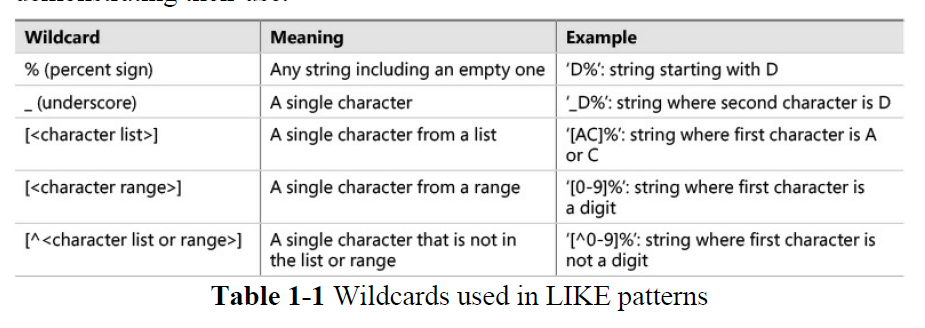
**Filtering character data**

T-SQL provides the LIKE predicate, which you can use to filter character

string data (regular and Unicode) based on pattern matching. The form of a

predicate using LIKE is as follows:

<column> LIKE <pattern>



As an example, suppose you want to return all employees whose last name

starts with the letter D. You would use the following query:

SELECT empid, firstname, lastname

FROM HR.Employees

WHERE lastname LIKE N'D%';

If you want to look for a character that is considered a wildcard, you can

indicate it after a character that you designate as an escape character by using

the ESCAPE keyword. For example, the expression col1 LIKE ‘!\_%’

ESCAPE ‘!’ looks for strings that start with an underscore (\_) by using an

exclamation point (!) as the escape character. Alternatively, you can place the

wildcard in square brackets, as in col1 LIKE ‘[\_]%’.

**Filtering date and time data**

You use the following query:

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate = '02/12/16';

If you’re an American, this form probably means February 12, 2016, to

you. However, if you’re British, this form probably means December 2,

2016. If you’re Japanese, it probably means December 16, 2002.

The question is, when SQL Server converts this character string to a date and time

type to align it with the filtered column’s type, how does it interpret the

value? As it turns out, it depends on the language of the login that runs the

code. Each login has a default language associated with it, and the default

language sets various session options on the login’s behalf, including one

called DATEFORMAT. A login with us\_english will have the

DATEFORMAT setting set to mdy, British to dmy, and Japanese to ymd.

The problem is, how do you as a developer express a date if you want it to be

interpreted the way you intended, regardless of who runs your code?

There are two main approaches. One is to use a form that is considered

language-neutral. For example, the form ‘20160212’ is always interpreted as

ymd, regardless of your language. Note that the form ‘2016-02-12’ is

considered language-neutral only for the data types DATE, DATETIME2,

and DATETIMEOFFSET. Unfortunately, due to historic reasons, this form is

considered language-dependent for the types DATETIME and

SMALLDATETIME. The advantage of the form without the separators is

that it is language-neutral for all date and time types. So the recommendation

is to write the query as follows:

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate = '20160212';

Another approach is to explicitly convert the string to the target type using

the CONVERT function, and indicating the style number that represents the

style that you used. You can find the documentation of the CONVERT

function with the different style numbers that it supports at

*https://msdn.microsoft.com/en-GB/library/ms187928.aspx*. For instance, to

use the U.S. style, specify style number 101, as CONVERT(DATE,

‘02/12/2016’, 101).

When filtering data stored in a DATETIME data type, you need to be very

careful with ranges. The recommended way to express a date and time range is with a closed-open interval as follows:

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders2

WHERE orderdate >= '20160401' AND orderdate < '20160501';

This time the output contains only the orders placed in April 2016.

**Sorting data**

A query that doesn’t have an

explicit instruction to return the rows in a particular order doesn’t guarantee

the order of rows in the result. When you do need such a guarantee, the only

way to provide it is by adding an ORDER BY clause to the query.

For example, if you want to return information about employees from

Washington in the US, sorted by city, you specify the city column in the

ORDER BY clause as follows:

SELECT empid, firstname, lastname, city, MONTH(birthdate)

AS birthmonth

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY city;

If you don’t indicate a direction for sorting, ascending order is assumed by

default. You can be explicit and specify city ASC, but it means the same

thing as not indicating the direction. For descending ordering, you need to

explicitly specify DESC, as follows:

SELECT empid, firstname, lastname, city, MONTH(birthdate)

AS birthmonth

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY city DESC;

The city column isn’t unique within the filtered country and region, and

therefore, the ordering of rows with the same city (see Seattle, for example)

isn’t guaranteed. Fortunately,

you can specify multiple expressions in the ORDER BY list, separated by

commas. One use case of this capability is to apply a tiebreaker for ordering.

For example, you could define empid as the secondary sort column, as

follows:

SELECT empid, firstname, lastname, city, MONTH(birthdate)

AS birthmonth

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY city, empid;

You can indicate the

ordering direction on an expression-by-expression basis, as in ORDER BY

col1 DESC, col2, col3 DESC (col1 descending, then col2 ascending, then

col3 descending).

With T-SQL, you can sort by ordinal positions of columns in the SELECT

list, but it is considered a bad practice. Consider the following query as an

example:

SELECT empid, firstname, lastname, city, MONTH(birthdate)

AS birthmonth

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY 4, 1;

Note that you can order the result rows by elements that you’re not

returning. For example, the following query returns, for each qualifying

employee, the employee ID and city, ordering the result rows by the

employee birth date:

SELECT empid, city

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY birthdate;

Here’s the output of this query:

empid city

----------- ---------------

4 Redmond

1 Seattle

2 Tacoma

8 Seattle

3 Kirkland

Of course, the result would appear much more meaningful if you included

the birthdate attribute, but if it makes sense for you not to, it’s perfectly valid.

The rule is that you can order the result rows by elements that are not part of

the SELECT list, as long as those elements would have normally been

allowed there. This rule changes when the DISTINCT clause is also

specified, and for a good reason. When DISTINCT is used, duplicates are

removed; then the result rows don’t necessarily map to source rows in a one-to-one manner, rather than one-to-many. For example, try to reason why the

following query isn’t valid:

SELECT DISTINCT city

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY birthdate;

You can have multiple employees—each with a different birth date—from

the same city. But you’re returning only one row for each distinct city in the

result. So given one city (say, Seattle) with multiple employees, which of the

employee birth dates should apply as the ordering value? The query won’t

just pick one; rather, it simply fails.

So, in case the DISTINCT clause is used, you are limited in the ORDER

BY list to only elements that appear in the SELECT list, as in the following

query:

SELECT DISTINCT city

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY city;

What’s also interesting to note about the ORDER BY clause is that it gets

evaluated conceptually after the SELECT clause—unlike most other query

clauses. This means that column aliases assigned in the SELECT clause are

actually visible to the ORDER BY clause. As an example, the following

query uses the MONTH function to return the birth month, assigning the

expression with the column alias birthmonth. The query then refers to the

column alias birthmonth directly in the ORDER BY clause:

SELECT empid, firstname, lastname, city, MONTH(birthdate)

AS birthmonth

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY birthmonth;

Another tricky aspect of ordering is treatment of NULLs. Recall that a

NULL represents a missing value, so when comparing a NULL to anything,

you get the logical result unknown. That’s the case even when comparing two

NULLs. So it’s not that trivial to ask how NULLs should behave in terms of

sorting. Should they all sort together? If so, should they sort before or after

non-NULL values? Standard SQL says that NULLs should sort together, but

leaves it to the implementation to decide whether to sort them before or after

non-NULL values. In SQL Server the decision was to sort them before non-

NULLs (when using an ascending direction). As an example, the following

query returns for each order the order ID and shipped date, ordered by the

latter:

SELECT orderid, shippeddate

FROM Sales.Orders

WHERE custid = 20

ORDER BY shippeddate;

Remember that unshipped orders have a NULL in the shippeddate column;

hence, they sort before shipped orders, as the query output shows:

orderid shippeddate

----------- -----------

11008 NULL

11072 NULL

10258 2014-07-23

10263 2014-07-31

#### Filtering data with TOP and OFFSET-FETCH

Besides supporting filters that are based on predicates, like the WHERE

filter, T-SQL also supports filters that are based on a number, or percent of

rows and ordering. Those are the TOP and OFFSET-FETCH filters. The

former is used in a lot of common filtering tasks, and the latter is typically

used in more specialized paging-related tasks.

**Filtering data with TOP**

With the TOP option, you can filter a requested number or percent of rows

from the query result based on indicated ordering. You specify the TOP

option in the SELECT clause followed by the requested number of rows in

parentheses (as a BIGINT typed value). The ordering specification of the

TOP filter is based on the same ORDER BY clause that is normally used for

presentation ordering.

As an example, the following query returns the three most recent orders:

SELECT TOP (3) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

You can also specify a percent of rows to filter instead of a number. To do

so, specify a FLOAT value in the range 0 through 100 in the parentheses, and

the keyword PERCENT after the parentheses, as follows:

SELECT TOP (1) PERCENT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

The PERCENT option computes the ceiling of the resulting number of

rows if it’s not whole. In this example, without the TOP option, the number

of rows in the result is 830. Filtering 1 percent gives you 8.3, and then the

ceiling of this value gives you 9; hence, the query returns 9 rows:

The TOP option isn’t limited to a constant input; instead, it allows you to

specify a self-contained expression. From a practical perspective, this

capability is especially important when you need to pass a parameter or a

variable as input, as the following code demonstrates:

DECLARE @n AS BIGINT = 5;

SELECT TOP (@n) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

In most cases, you need your TOP option to rely on some ordering

specification, but as it turns out, an ORDER BY clause isn’t mandatory. For

example, the following query is technically valid:

SELECT TOP (3) orderid, orderdate, custid, empid

FROM Sales.Orders;

However, the query isn’t deterministic. The query filters three rows, but

you have no guarantee which three rows will be returned.

If you are really after three arbitrary rows, it might be a good

idea to add an ORDER BY clause with the expression (SELECT NULL) to

let people know that your choice is intentional and not an oversight. Here’s

how your query would look:

SELECT TOP (3) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY (SELECT NULL);

Note that even when you do have an ORDER BY clause, in order for the

query to be completely deterministic, the ordering must be unique. For

example, consider again the first query from this section:

SELECT TOP (3) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

The orderdate column isn’t unique, so the ordering in case of ties is

arbitrary.

consider again the first query from this section:

SELECT TOP (3) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

what if there are other rows in the result without TOP that have the

same order date as in the last row here? You don’t always care about

guaranteeing deterministic or repeatable results; but if you do, two options

are available to you. One option is to ask to include all ties with the last row

by adding the WITH TIES option, as follows:

SELECT TOP (3) WITH TIES orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

Of course, this could result in returning more rows than you asked for.

**Filtering data with OFFSET-FETCH**

The OFFSET-FETCH option is a filtering option that, like TOP, you can use

to filter data based on a specified number of rows and ordering. But unlike

TOP, it is standard, and also has a skipping capability, making it useful for

ad-hoc paging purposes.

The OFFSET and FETCH clauses appear right after the ORDER BY

clause, and in fact, in T-SQL, they require an ORDER BY clause to be

present. You first specify the OFFSET clause indicating how many rows you

want to skip (0 if you don’t want to skip any); you then optionally specify the

FETCH clause indicating how many rows you want to filter. For example,

the following query defines ordering based on order date descending,

followed by order ID descending; it then skips the first 50 rows and fetches

the next 25 rows:

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;

In T-SQL—contrary to standard SQL—a

FETCH clause requires an OFFSET clause to be present. So if you do want to

filter some rows but skip none, you still need to specify the OFFSET clause

with 0 ROWS.

In order to make the syntax intuitive, you can use the keywords NEXT or

FIRST interchangeably. When skipping some rows, it might be more

intuitive to you to use the keywords FETCH NEXT to indicate how many

rows to filter; but when not skipping any rows, it might be more intuitive to

you to use the keywords FETCH FIRST, as follows:

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET 0 ROWS FETCH FIRST 25 ROWS ONLY;

For similar reasons, you can use the singular form ROW or the plural form

ROWS interchangeably, both for the number of rows to skip and for the

number of rows to filter. But it’s not like you will get an error if you say

FETCH NEXT 1 ROWS or FETCH NEXT 25 ROW.

In T-SQL, a FETCH clause requires an OFFSET clause, but the OFFSET

clause doesn’t require a FETCH clause. In other words, by indicating an

OFFSET clause, you’re requesting to skip some rows; then by not indicating

a FETCH clause, you’re requesting to return all remaining rows. For

example, the following query requests to skip 50 rows, returning all the rest.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET 50 ROWS;

the OFFSET-FETCH option requires an ORDER

BY clause. But what if you need to filter a certain number of rows based on

arbitrary order? To do so, you can specify the expression (SELECT NULL)

in the ORDER BY clause, as follows:

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY (SELECT NULL)

OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY;

With both the OFFSET and the FETCH clauses, you can use expressions

as inputs. This is very handy when you need to compute the input values

dynamically. For example, suppose that you’re implementing a paging

solution where you return to the user one page of rows at a time. The user

passes as input parameters to your procedure or function the page number

they are after (@pagenum parameter) and page size (@pagesize parameter).

This means that you need to skip as many rows as @pagenum minus one

times @pagesize, and fetch the next @pagesize rows. This can be

implemented using the following code (using local variables for simplicity):

DECLARE @pagesize AS BIGINT = 25, @pagenum AS BIGINT = 3;

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET (@pagenum - 1) \* @pagesize ROWS FETCH NEXT

@pagesize ROWS ONLY;

## Combining sets with set operators

Set operators operate on two result sets of queries, comparing complete rows

between the results. Depending on the result of the comparison and the

operator used, the operator determines whether to return the row or not. TSQL supports the following operators:

* UNION,
* UNION ALL,
* INTERSECT,
* EXCEPT.

The general form of code using these operators is as follows:

<query 1>

<operator>

<query 2>

[ORDER BY <order\_by\_list>];

**Guidelines:**

Because complete rows are matched between the result sets, the

number of columns in the queries has to be the same and the column

types of corresponding columns need to be compatible (implicitly

convertible).

These operators use distinctness-based comparison and not equality

based. Consequently, a comparison between two NULLs yields true,

and a comparison between a NULL and a non-NULL value yields a

false. This is in contrast to filtering clauses like WHERE, ON, and

HAVING, which yield unknown when comparing a NULL with

anything using both equality and inequality operators.

Because these operators are set operators and not cursor operators, the

individual queries are not allowed to have ORDER BY clauses.

You can optionally add an ORDER BY clause that determines

presentation ordering of the result of the set operator.

The column names of result columns are determined by the first query.

### UNION and UNION ALL

The UNION operator unifies the results of the two input queries. As a set

operator, UNION has an implied DISTINCT property, meaning that it does

not return duplicate rows. Figure 1-2 shows an illustration of the UNION

operator.

As an example for using the UNION operator, the following query returns

locations that are employee locations, or customer locations, or both:

SELECT country, region, city

FROM HR.Employees

UNION

SELECT country, region, city

FROM Sales.Customers;

If you want to keep the duplicates—for example, to later group the rows

and count occurrences—you need to use the UNION ALL operator instead of

UNION. The UNION ALL operator unifies the results of the two input

queries, but doesn’t try to eliminate duplicates.

As an example, the following query unifies employee locations and

customer locations using the UNION ALL operator:

SELECT country, region, city

FROM HR.Employees

UNION ALL

SELECT country, region, city

FROM Sales.Customers;

It’s important to use UNION ALL in such a case from a performance standpoint because with UNION, SQL Server can try to eliminate duplicates, incurring unnecessary cost.

### INTERSECT

The INTERSECT operator returns only distinct rows that are common to

both sets. In other words, if a row appears at least once in the first set and at

least once in the second set, it appears once in the result of the INTERSECT

operator.

As an example, the following code uses the INTERSECT operator to

return distinct locations that are both employee and customer locations

(locations where there’s at least one employee and at least one customer):

SELECT country, region, city

FROM HR.Employees

INTERSECT

SELECT country, region, city

FROM Sales.Customers;

### EXCEPT

The EXCEPT operator performs set difference. It returns distinct rows that

appear in the result of the first query but not the second. In other words, if a

row appears at least once in the first query result and zero times in the

second, it’s returned once in the output.

As an example for using EXCEPT, the following query returns locations

that are employee locations but not customer locations:

SELECT country, region, city

FROM HR.Employees

EXCEPT

SELECT country, region, city

FROM Sales.Customers;

With UNION and INTERSECT, the order of the input queries doesn’t

matter. However, with EXCEPT, there’s different meaning to:

<query 1> EXCEPT <query 2>

Versus:

<query 2> EXCEPT <query 1>

Finally, set operators have precedence: INTERSECT precedes UNION and

EXCEPT, and UNION and EXCEPT are evaluated from left to right based on

their position in the expression. Consider the following set operators:

<query 1> UNION <query 2> INTERSECT <query 3>;

First, the intersection between query 2 and query 3 takes place, and then a

union between the result of the intersection and query 1. You can always

force precedence by using parentheses. So, if you want the union to take

place first, you use the following form:

(<query 1> UNION <query 2>) INTERSECT <query 3>;

## Query multiple tables by using joins

Often, data that you need to query is spread across multiple tables. The tables

are usually related through keys, such as a foreign key in one side and a

primary key in the other. Then you can use joins to query the data from the

different tables and match the rows that need to be related.

### Cross joins

A cross join is the simplest type of join, though not the most commonly used

one. This join performs what’s known as a Cartesian product of the two input

tables. In other words, it performs a multiplication between the tables,

yielding a row for each combination of rows from both sides. If you have m

rows in table T1 and n rows in table T2, the result of a cross join between T1

and T2 is a virtual table with m × n rows.

Consider an example from the TSQLV4 sample database. This database

contains a table called dbo.Nums that has a column called n with a sequence

of integers from 1 and on. Your task is to use the Nums table to generate a

result with a row for each weekday (1 through 7) and shift number (1 through

3), assuming there are three shifts a day. The result can later be used as the

basis for building information about activities in the different shifts in the

different days. With seven days in the week, and three shifts every day, the

result should have 21 rows.

Here’s a query that achieves the task by performing a cross join between

two instances of the Nums table—one representing the days (aliased as D),

and the other representing the shifts (aliased as S):

USE TSQLV4;

SELECT D.n AS theday, S.n AS shiftno

FROM dbo.Nums AS D

CROSS JOIN dbo.Nums AS S

WHERE D.n <= 7

AND S.N <= 3

ORDER BY theday, shiftno;

Here’s the output of this query:

theday shiftno

----------- -----------

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

4 1

...

The Nums table has 100,000 rows. According to logical query processing,

the first step in the processing of the query is evaluating the FROM clause.

The cross join operates in the FROM clause, performing a Cartesian product

between the two instances of Nums, yielding a table with 10,000,000,000

rows (not to worry, that’s only conceptually). Then the WHERE clause filters

only the rows where the column D.n is less than or equal to 7, and the column

S.n is less than or equal to 3. After applying the filter, the result has 21

qualifying rows. The SELECT clause then returns D.n aliasing it theday, and

S.n aliasing it shiftno.

Fortunately, SQL Server doesn’t have to follow logical query processing

literally as long as it can return the correct result. That’s what optimization is

all about—returning the result as fast as possible. SQL Server knows that

with a cross join followed by a filter it can evaluate the filters first (which is

especially efficient when there are indexes to support the filters), and then

match the remaining rows. This optimization technique is called *predicate*

*pushdown*.

Both standard SQL and T-SQL support an older

syntax where you specify a comma between the table names, as in FROM T1,

T2. However, for a number of reasons, it is recommended to stick to the

newer syntax; it is less prone to errors and allows for more consistent code.

### Inner joins

With an inner join, you can match rows from two tables based on a predicate

—usually one that compares a primary key value in one side to a foreign key

value in another side.

As an example, the following query returns suppliers from Japan and the

products they supply:

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice

FROM Production.Suppliers AS S

INNER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

WHERE S.country = N'Japan';

A very common question is, “What’s the difference between the ON and

the WHERE clauses, and does it matter if you specify your predicate in one

or the other?” The answer is that for inner joins it doesn’t matter. Both

clauses serve the same filtering purpose. Both filter only rows for which the

predicate evaluates to true and discard rows for which the predicate evaluates

to false or unknown. In terms of logical query processing, the WHERE is

evaluated right after the FROM, so conceptually it is equivalent to

concatenating the predicates with an AND operator, forming a conjunction of

predicates. SQL Server knows this, and therefore can internally rearrange the

order in which it evaluates the predicates in practice, and it does so based on

cost estimates.

For these reasons, if you wanted, you could rearrange the placement of the

predicates from the previous query, specifying both in the ON clause, and

still retain the original meaning, as follows:

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice

FROM Production.Suppliers AS S

INNER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

AND S.country = N'Japan';

For many people, though, it’s intuitive to specify the predicate that

matches columns from both sides in the ON clause, and predicates that filter

columns from only one side in the WHERE clause.

As another example for an inner join, the following query joins two

instances of the HR.Employees table to match employees with their managers

(a manager is also an employee, hence the self-join):

SELECT E.empid,

E.firstname + N' ' + E.lastname AS emp,

M.firstname + N' ' + M.lastname AS mgr

FROM HR.Employees AS E

INNER JOIN HR.Employees AS M

ON E.mgrid = M.empid;

This query generates the following output:

empid emp mgr

----------- ------------------------------- --------------

-----------------

2 Don Funk Sara Davis

3 Judy Lew Don Funk

4 Yael Peled Judy Lew

5 Sven Mortensen Don Funk

6 Paul Suurs Sven Mortensen

Note that only eight rows were returned even though there are nine rows in

the table. The reason is that the CEO (Sara Davis, employee ID 1) has no

manager, and therefore, her mgrid column is NULL. Remember that an inner

join does not return rows that don’t find matches.

As with cross joins, both standard SQL and T-SQL support an older syntax

for inner joins where you specify a comma between the table names, and then

all predicates in the WHERE clause. But as mentioned, it is considered best

practice to stick to the newer syntax with the JOIN keyword. When using the

older syntax, if you forget to indicate the join predicate, you end up with an

unintentional cross join. When using the newer syntax, an inner join isn’t

valid syntactically without an ON clause, so if you forget to indicate the join

predicate, the parser will generate an error.

Because an inner join is the most commonly used type of join, the standard

decided to make it the default in case you specify just the JOIN keyword. So

T1 JOIN T2 is equivalent to T1 INNER JOIN T2.

### Outer joins

With outer joins, you can request to preserve all rows from one or both sides

of the join, never mind if there are matching rows in the other side based on

the ON predicate.

By using the keywords LEFT OUTER JOIN (or LEFT JOIN for short),

you ask to preserve the left table. The join returns what an inner join

normally would—that is, matches (call those inner rows). In addition, the join

also returns rows from the left table that have no matches in the right table

(call those outer rows), with NULLs used as placeholders in the right side.

As an example, the following query returns suppliers from Japan and the

products they supply, including suppliers from Japan that don’t have related

products.

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice

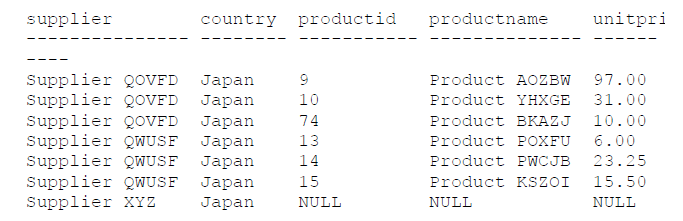
FROM Production.Suppliers AS S

LEFT OUTER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

WHERE S.country = N'Japan';

This query generates the following output:



Because the Production.Suppliers table is the preserved side of the join,

Supplier XYZ is returned even though it has no matching products. As you

recall, an inner join did not return this supplier.

It is very important to understand that, with outer joins, the ON and

WHERE clauses play very different roles, and therefore, they aren’t

interchangeable. The WHERE clause still plays a simple filtering role—

namely, it keeps true cases and discards false and unknown cases. In our

query, the WHERE clause filters only suppliers from Japan, so suppliers that

aren’t from Japan simply don’t show up in the output.

However, the ON clause doesn’t play a simple filtering role; rather, it’s a

more sophisticated matching role. In other words, a row in the preserved side

will be returned whether the ON predicate finds a match for it or not. So the

ON predicate only determines which rows from the nonpreserved side get

matched to rows from the preserved side—not whether to return the rows

from the preserved side. In our query, the ON clause matches rows from both

sides by comparing their supplier ID values. Because it’s a matching

predicate (as opposed to a filter), the join won’t discard suppliers; instead, it

only determines which products get matched to each supplier. But even if a

supplier has no matches based on the ON predicate, the supplier is still

returned. In other words, ON is not final with respect to the preserved side of

the join. WHERE is final. So when in doubt, whether to specify the predicate

in the ON or WHERE clauses, ask yourself: Is the predicate used to filter or

match? Is it supposed to be final or nonfinal?

With this in mind, guess what happens if you specify both the predicate

that compares the supplier IDs from both sides, and the one comparing the

supplier country to Japan in the ON clause? Try it.

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice

FROM Production.Suppliers AS S

LEFT OUTER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

AND S.country = N'Japan';

Now that both predicates appear in the ON clause, both serve a matching

purpose. What this means is that all suppliers are returned—even those that

aren’t from Japan.

Just like you can use a left outer join to preserve the left side, you can use a

right outer join to preserve the right side. Use the keywords RIGHT OUTER

JOIN (or RIGHT JOIN in short).

T-SQL also supports a full outer join (FULL OUTER JOIN, or FULL

JOIN in short) that preserves both sides.

A full outer join returns the matched rows, which are normally returned

from an inner join; plus rows from the left that don’t have matches in the

right, with NULLs used as placeholders in the right side; plus rows from the

right that don’t have matches in the left, with NULLs used as placeholders in

the left side. It’s not common to need a full outer join because most

relationships between tables allow only one of the sides to have rows that

don’t have matches in the other, in which case, a one-sided outer join is

needed.

### Queries with composite joins and NULLs in join Columns

Some joins can be a bit tricky to handle, for instance when the join columns

can have NULLs, or when you have multiple join columns—what’s known

as a composite join.

Earlier in the inner joins section is a query that matched employees and

their managers. Remember that the inner join eliminated the CEO’s row

because the mgrid is NULL in that row, and therefore the join found no

matching manager. If you want to include the CEO’s row, you need to use an

outer join to preserve the side representing the employees (E) as follows:

SELECT E.empid,

E.firstname + N' ' + E.lastname AS emp,

M.firstname + N' ' + M.lastname AS mgr

FROM HR.Employees AS E

LEFT OUTER JOIN HR.Employees AS M

ON E.mgrid = M.empid;

When you need to join tables that are related based on multiple columns,

the join is called a *composite join* and the ON clause typically consists of a

conjunction of predicates (predicates separated by AND operators) that match

the corresponding columns from the two sides. Sometimes you need more

complex predicates, especially when NULLs are involved. I’ll demonstrate

this by using a pair of tables. One table is called EmpLocations and it holds

employee locations and the number of employees in each location. Another

table is called CustLocations and it holds customer locations and the number

of customers in each location. Run the following code to create these tables

and populate them with sample data:

DROP TABLE IF EXISTS dbo.EmpLocations;

SELECT country, region, city, COUNT(\*) AS numemps

INTO dbo.EmpLocations

FROM HR.Employees

GROUP BY country, region, city;

ALTER TABLE dbo.EmpLocations ADD CONSTRAINT

UNQ\_EmpLocations

UNIQUE CLUSTERED(country, region, city);

DROP TABLE IF EXISTS dbo.CustLocations;

SELECT country, region, city, COUNT(\*) AS numcusts

INTO dbo.CustLocations

FROM Sales.Customers

GROUP BY country, region, city;

ALTER TABLE dbo.CustLocations ADD CONSTRAINT

UNQ\_CustLocations

UNIQUE CLUSTERED(country, region, city);

There’s a key defined in both tables on the location attributes: country,

region, and city. Instead of using a primary key constraint I used a unique

constraint to enforce the key because the region attribute allows NULLs, and

between the two types of constraints, only the latter allows NULLs. I also

specified the CLUSTERED keyword in the unique constraint definitions to

have SQL Server create a clustered index type to enforce the constraint’s

uniqueness property. This index will be beneficial in supporting joins

between the tables based on the location attributes as well filters based on

those attributes.

Query the EmpLocations table to see its contents:

SELECT country, region, city, numemps

FROM dbo.EmpLocations;

Suppose that you needed to join the two tables returning only matched

locations, with both the employee and customer counts returned along with

the location attributes. Your first attempt might be to write a composite join

with an ON clause that has a conjunction of simple equality predicates as

follows:

SELECT EL.country, EL.region, EL.city, EL.numemps,

CL.numcusts

FROM dbo.EmpLocations AS EL

INNER JOIN dbo.CustLocations AS CL

ON EL.country = CL.country

AND EL.region = CL.region

AND EL.city = CL.city;

The problem is that the region column supports NULLs representing cases

where the region is irrelevant (missing but inapplicable) and when you

compare NULLs with an equality-based predicate the result is the logical

value unknown, in which case the row is discarded. For instance, the location

UK, NULL, London appears in both tables, and therefore you expect to see it

in the result of the join, but you don’t. A common way for people to resolve

this problem is to use the ISNULL or COALESCE functions to substitute a

NULL in both sides with a value that can’t normally appear in the data, and

this way when both sides are NULL you get a true back from the comparison.

Here’s an example for implementing this solution using the ISNULL

function:

SELECT EL.country, EL.region, EL.city, EL.numemps,

CL.numcusts

FROM dbo.EmpLocations AS EL

INNER JOIN dbo.CustLocations AS CL

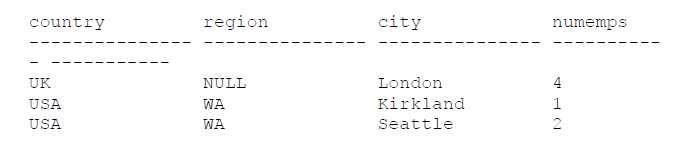
ON EL.country = CL.country

AND ISNULL(EL.region, N'<N/A>') = ISNULL(CL.region,

N'<N/A>')

AND EL.city = CL.city;

This time the query generates the correct result:



You can handle NULLs in a manner that gives you the desired logical

meaning and that at the same time is considered order preserving by the

optimizer using the predicate: (EL.region = CL.region OR (EL.region IS

NULL AND CL.region IS NULL)). Here’s the complete solution query:

SELECT EL.country, EL.region, EL.city, EL.numemps,

CL.numcusts

FROM dbo.EmpLocations AS EL

INNER JOIN dbo.CustLocations AS CL

ON EL.country = CL.country

AND (EL.region = CL.region OR (EL.region IS NULL AND

CL.region IS NULL))

AND EL.city = CL.city;

Recall that when set operators combine query results they compare

corresponding attributes using distinctness and not equality, producing true

when comparing two NULLs. However, one drawback that set operators

have is that they compare complete rows. Unlike joins, which allow

comparing a subset of the attributes and return additional ones in the result,

set operators must compare all attributes from the two input queries. But in

T-SQL, you can combine joins and set operators to benefit from the

advantages of both tools. Namely, rely on the distinctness-based comparison

of set operators and the ability of joins to return additional attributes beyond

what you compare. In our querying task, the solution looks like this:

SELECT EL.country, EL.region, EL.city, EL.numemps,

CL.numcusts

FROM dbo.EmpLocations AS EL

INNER JOIN dbo.CustLocations AS CL

ON EXISTS (SELECT EL.country, EL.region, EL.city

INTERSECT

SELECT CL.country, CL.region, CL.city);

### Multi-join queries

It’s important to remember that a join in T-SQL takes place conceptually

between two tables at a time. A multi-join query evaluates the joins

conceptually from left to right. So the result of one join is used as the left

input to the next join.

As an example, suppose that you wanted to return all suppliers from Japan,

and matching products where relevant. For this, you need an outer join

between Production.Suppliers and Production.Products, preserving Suppliers.

But you also want to include product category information, so you add an

inner join to Production.Categories, as follows:

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice,

C.categoryname

FROM Production.Suppliers AS S

LEFT OUTER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

INNER JOIN Production.Categories AS C

ON C.categoryid = P.categoryid

WHERE S.country = N'Japan';

Conceptually, the first join included outer rows (suppliers with no

products) but produced NULLs as placeholders in the product attributes in

those rows. Then the join to Production.Categories compared the NULLs in

the categoryid column in the outer rows to categoryid values in

Production.Categories, and discarded those rows. In short, the inner join that

followed the outer join nullified the outer part of the join. In fact, if you look

at the query plan for this query, you will find that the optimizer didn’t even

bother to process the join between Production.Suppliers and

Production.Products as an outer join. It detected the contradiction between

the outer join and the subsequent inner join, and converted the first join to an

inner join too.

There are a number of ways to address this problem. One is to use a LEFT

OUTER in both joins, like so:

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice,

C.categoryname

FROM Production.Suppliers AS S

LEFT OUTER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

LEFT OUTER JOIN Production.Categories AS C

ON C.categoryid = P.categoryid

WHERE S.country = N'Japan';

Another option is to use an interesting capability in the language—separate

some of the joins to their own independent logical phase. What you’re after is

a left outer join between Production.Suppliers and the result of the inner join

between Production.Products and Production.Categories. You can phrase

your query exactly like this:

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice,

C.categoryname

FROM Production.Suppliers AS S

LEFT OUTER JOIN

(Production.Products AS P

INNER JOIN Production.Categories AS C

ON C.categoryid = P.categoryid)

ON S.supplierid = P.supplierid

WHERE S.country = N'Japan';

## Implement functions and aggregate data

T-SQL supports many built-in functions that you can use to manipulate data.

Scalar-valued functions return a single value and table-valued functions

return a table result. Use of built-in functions can improve developer

productivity, but you also need to understand cases where their use in certain

context can end up negatively affecting query performance. It’s also

important to understand the concept of function determinism and its effects

on your queries.

### Type conversion functions

T-SQL supports a number of functions that can convert a source expression

to a target data type. In my examples I use constants as the source values to

demonstrate the use of the functions, but typically you apply such functions

to columns or expressions based on columns as part of a query.

The two fundamental functions that T-SQL supports for conversion

purposes are CAST and CONVERT. The former is standard whereas the

latter is proprietary in T-SQL. The CAST function’s syntax is

CAST(source\_expression AS target\_type. For example, CAST(‘100’ AS

INT) converts the source character string constant to the target integer value

100. The CONVERT function is handy when you need to specify a style for

the conversion. Its syntax is CONVERT(target\_type, source\_expression [,

style\_number])

For instance,

when converting a character string to a date and time type or the other way

around, you can specify the style number to avoid ambiguity in case the form

you use is considered language dependent. As an example, the expression

CONVERT(DATE, ‘01/02/2017’, 101) converts the input string to a date

using the U.S. style, meaning January 2, 2017. The expression

CONVERT(DATE, ‘01/02/2017’, 103) uses the British/French style,

meaning February 1, 2017.

The PARSE function is an alternative to CONVERT when you want to

parse a character string input to a target type, but instead of using cryptic

style numbers, it uses a more user-friendly .NET culture name. For instance,

the expression PARSE(‘01/02/2017’ AS DATE USING ‘en-US’) uses the

English US culture, parsing the input as a date meaning January 2, 2017. The

expression PARSE(‘01/02/2017’ AS DATE USING ‘en-GB’) uses the

English Great Britain culture, parsing the input as a date meaning February 1,

2017. Note though that this function is significantly slower than CONVERT,

so I personally stay away from using it.

One of the problems with CAST, CONVERT, and PARSE is that if the

function fails to convert a value within a query, the whole query fails and

stops processing. As an alternative to these functions, T-SQL supports the

TRY\_CAST, TRY\_CONVERT, and TRY\_PARSE functions, which behave

the same as their counterparts when the conversion is valid, but return a

NULL instead of failing when the conversion isn’t valid. As an example, run

the following code to try and convert two strings to dates using the

CONVERT function:

SELECT CONVERT(DATE, '14/02/2017', 101) AS col1,

CONVERT(DATE, '02/14/2017', 101) AS col2;

The first value doesn’t convert successfully and therefore this code fails.

Use the TRY\_CONVERT function instead of CONVERT, like so:

SELECT TRY\_CONVERT(DATE, '14/02/2017', 101) AS col1,

TRY\_CONVERT(DATE, '02/14/2017', 101) AS col2;

This time the code doesn’t fail, but the first expression returns a NULL, as

the following output shows:

col1 col2

---------- ----------

NULL 2017-02-14

Lastly, the FORMAT function is an alternative to the CONVERT function

when you want to format an input expression of some type as a character

string, but instead of using cryptic style numbers, you specify a .NET format

string and culture, if relevant. For instance, to format an input date and time

value, such as now, as a character string using the form ‘yyyy-MM-dd’, use

the expression: FORMAT(SYSDATETIME(), ‘yyyy-MM-dd’).

### Date and time functions

T-SQL supports a number of date and time functions that allow you to

manipulate your date and time data.

**Current date and time**

One important category of functions is the category that returns the current

date and time.

The functions in this category are GETDATE, CURRENT\_TIMESTAMP,

GETUTCDATE, SYSDATETIME, SYSUTCDATETIME, and

SYSDATETIMEOFFSET. GETDATE is T-SQL–specific, returning the

current date and time in the SQL Server instance you’re connected to as a

DATETIME data type. CURRENT\_TIMESTAMP is the same, only it’s

standard, and hence the recommended one to use. SYSDATETIME and

SYSDATETIMEOFFSET are similar, only returning the values as the more

precise DATETIME2 and DATETIMEOFFSET types (including the time

zone offset from UTC), respectively. Note that there are no built-in functions

to return the current date and the current time. To get such information,

simply cast the SYSDATETIME function to DATE or TIME, respectively.

For example, to get the current date, use CAST(SYSDATETIME() AS

DATE). The GETUTCDATE function returns the current date and time in

UTC terms as a DATETIME type, and SYSUTCDATETIME does the same,

only returning the result as the more precise DATETIME2 type.

**Date and time parts**

This section covers date and time functions that either extract a part from a

date and time value (like DATEPART) or construct a date and time value

from parts (like DATEFROMPARTS).

Using the DATEPART function, you can extract from an input date and

time value a desired part, such as a year, minute, or nanosecond, and return

the extracted part as an integer. For example, the expression

DATEPART(month, ‘20170212’) returns 2. T-SQL provides the functions

YEAR, MONTH, and DAY as abbreviations to DATEPART, not requiring

you to specify the part. The DATENAME function is similar to DATEPART,

only it returns the name of the part as a character string, as opposed to the

integer value. Note that the function is language dependent. That is, if the

effective language in your session is us\_english, the expression

DATENAME(month, ‘20170212’) returns ‘February’, but for Italian, it

returns ‘febbraio.'

T-SQL provides a set of functions that construct a desired date and time

value from its numeric parts. You have such a function for each of the six

available date and time types: DATEFROMPARTS,

DATETIME2FROMPARTS, DATETIMEFROMPARTS,

DATETIMEOFFSETFROMPARTS, SMALLDATETIMEFROMPARTS,

and TIMEFROMPARTS. For example, to build a DATE value from its parts,

you would use an expression such as DATEFROMPARTS(2017, 02, 12).

Finally, the EOMONTH function computes the respective end of month

date for the input date and time value. For example, suppose that today was

February 12, 2017. The expression EOMONTH(SYSDATETIME()) would

then return the date ‘2017-02-29’. This function supports a second optional

input indicating how many months to add to the result (or subtract if

negative).

**Add and diff functions**

T-SQL supports addition and difference date and time functions called

DATEADD and DATEDIFF.

DATEADD is a very commonly used function. With it, you can add a

requested number of units of a specified part to a specified date and time

value. For example, the expression DATEADD(year, 1, ‘20170212’) adds

one year to the input date February 12, 2017.

DATEDIFF is another commonly used function; it returns the difference in

terms of a requested part between two date and time values. For example, the

expression DATEDIFF(day, ‘20160212’, ‘20170212’) computes the

difference in days between February 12, 2016 and February 12, 2017,

returning the value 366. Note that this function looks only at the parts from

the requested one and above in the date and time hierarchy—not below. For

example, the expression DATEDIFF(year, ‘20161231’, ‘20170101’) looks

only at the year part, and hence returns 1. It doesn’t look at the month and

day parts of the values.

The DATEDIFF function returns a value of an INT type. If the difference

doesn’t fit in a four-byte integer, use the DATEDIFF\_BIG function instead.

This function returns the result as a BIGINT type.

### Character functions

T-SQL was not really designed to support very sophisticated character string

manipulation functions, so you won’t find a very large set of such functions.

This section describes the character string functions that T-SQL does support,

arranged in categories.

**Concatenation**

Character string concatenation is a very common need. T-SQL supports two

ways to concatenate strings—one with the plus (+) operator, and another with

the CONCAT function.

Here’s an example for concatenating strings in a query by using the +

operator:

SELECT empid, country, region, city,

country + N', ' + region + N', ' + city AS location

FROM HR.Employees;

When any of the inputs is NULL, the + operator returns a NULL. If you want to substitute

a NULL with an empty string, there are a number of ways for you to do this

programmatically. One option is to use ISNULL or COALESCE functions to

replace a NULL with an empty string. For example, in this data, only region

can be NULL, so you can use the following query to replace a comma plus

region with an empty string when region is NULL:

SELECT empid, country, region, city,

country + ISNULL(N', ' + region, N'') + N', ' + city AS

location

FROM HR.Employees;

Another option is to use the CONCAT function which, unlike the +

operator, substitutes a NULL input with an empty string. Here’s how the

query looks:

SELECT empid, country, region, city,

CONCAT(country, N', ' + region, N', ' + city) AS

location

FROM HR.Employees;

**Substring extraction and position**

This section covers functions that you can use to extract a substring from a

string, and identify the position of a substring within a string.

With the SUBSTRING function, you can extract a substring from a string

given as the first argument, starting with the position given as the second

argument, and a length given as the third argument. For example, the

expression SUBSTRING(‘abcde’, 1, 3) returns ‘abc’. If the third argument is

greater than what would get you to the end of the string, the function doesn’t

fail; instead, it just extracts the substring until the end of the string.

The LEFT and RIGHT functions extract a requested number of characters

from the left and right ends of the input string, respectively. For example,

LEFT(‘abcde’, 3) returns ‘abc’ and RIGHT(‘abcde’, 3) returns ‘cde’.

The CHARINDEX function returns the position of the first occurrence of

the string provided as the first argument within the string provided as the

second argument. For example, the expression CHARINDEX(‘ ‘,’Inigo

Montoya’) looks for the first occurrence of a space in the second input,

returning 6 in this example. Note that you can provide a third argument

indicating to the function the position where to start looking.

You can combine, or nest, functions in the same expression. For example,

suppose you query a table with an attribute called fullname formatted as

‘<first> <last>’, and you need to write an expression that extracts the first

name part. You can use the following expression:

LEFT(fullname, CHARINDEX(' ', fullname) - 1)

T-SQL also supports a function called PATINDEX that, like

CHARINDEX, you can use to locate the first position of a string within

another string. But whereas with CHARINDEX you’re looking for a constant

string, with PATINDEX you’re looking for a pattern. The pattern is formed

very similar to the LIKE patterns that you’re probably familiar with, where

you use wildcards like % for any string, \_ for a single character, and square

brackets ([]) representing a single character from a certain list or range.

As an

example, the expression PATINDEX(‘%[0-9]%’, ‘abcd123efgh’) looks for

the first occurrence of a digit (a character in the range 0–9) in the second

input, returning the position 5 in this case.

**String length**

T-SQL provides two functions that you can use to measure the length of an

input value—LEN and DATALENGTH.

The LEN function returns the length of an input string in terms of the

number of characters. Note that it returns the number of characters, not bytes,

whether the input is a regular character or Unicode character string. For

example, the expression LEN(N’xyz’) returns 3. If there are any trailing

spaces, LEN removes them.

The DATALENGTH function returns the length of the input in terms of

number of bytes. This means, for example, that if the input is a Unicode

character string, it will count 2 bytes per character. For example, the

expression DATALENGTH(N’xyz’) returns 6. Note also that, unlike LEN,

the DATALENGTH function doesn’t remove trailing spaces.

**String alteration**

T-SQL supports a number of functions that you can use to apply alterations to

an input string. Those are REPLACE, REPLICATE, and STUFF.

With the REPLACE function, you can replace in an input string provided

as the first argument all occurrences of the string provided as the second

argument, with the string provided as the third argument. For example, the

expression REPLACE(‘.1.2.3.’, ‘.’, ‘/’) substitutes all occurrences of a dot (.)

with a slash (/), returning the string ‘/1/2/3/’.

The REPLICATE function allows you to replicate an input string a

requested number of times. For example, the expression REPLICATE(‘0’,

10) replicates the string ‘0’ ten times, returning ‘0000000000’.

The STUFF function operates on an input string provided as the first

argument; then, from the character position indicated as the second argument,

deletes the number of characters indicated by the third argument. Then it

inserts in that position the string specified as the fourth argument. For

example, the expression STUFF(‘,x,y,z’, 1, 1, ‘’) removes the first character

from the input string, returning ‘x,y,z’.

**Formatting**

This section covers functions that you can use to apply formatting options to

an input string. Those are the UPPER, LOWER, LTRIM, RTRIM, and

FORMAT functions.

The first four functions are self-explanatory (uppercase form of the input,

lowercase form of the input, input after removal of leading spaces, and input

after removal of trailing spaces). Note that there’s no TRIM function that

removes both leading and trailing spaces; to achieve this, you need to nest

one function call within another, as in RTRIM(LTRIM(<input>)).

As mentioned earlier, with the FORMAT function, you can format an input

value based on a .NET format string. I demonstrated an example with date

and time values. As another example, this time with numeric values, the

expression FORMAT(1759, ‘0000000000’) formats the input number as a

character string with a fixed size of 10 characters with leading zeros,

returning ‘0000001759’. The same thing can be achieved with the format

string ‘d10’, meaning decimal value with 10 digits, with the expression

FORMAT(1759, ‘d10’).

### System functions

System functions return information about various aspects of the system.

**The @@ROWCOUNT and ROWCOUNT\_BIG functions**

The @@ROWCOUNT function is a very popular function that returns the

number of rows affected by the last statement that you executed. It’s very

common to use it to check if the previous statement affected any rows by

checking that the function’s result is zero or greater than zero. For example,

the following code queries the input employee row, and prints a message if

the requested employee was not found:

DECLARE @empid AS INT = 10;

SELECT empid, firstname, lastname

FROM HR.Employees

WHERE empid = @empid;

IF @@ROWCOUNT = 0

PRINT CONCAT('Employee ', CAST(@empid AS VARCHAR(10)), ' was not found.');

**Arithmetic operators and aggregate functions**

T-SQL supports the four classic arithmetic operators + (add), - (subtract), \*

(multiply), / (divide), as well as the fifth operator % (modulo). The last

computes the remainder of an integer division. T-SQL also supports

aggregate functions, which you apply to a set of rows, and get a single value

back.

**Arithmetic operators**

For the most part, work with these arithmetic operators is intuitive. They

follow classic arithmetic operator precedence rules, which say that

multiplication, division and modulo precede addition and subtraction. To

change precedence of operations, use parentheses because they precede

arithmetic operators. For example, consider the following expression:

SELECT 2 + 3 \* 2 + 10 / 2;

It is equivalent to the following expression:

SELECT 2 + (3 \* 2) + (10 / 2);

The result of this expression is 13.

If you want to evaluate the operations from left to right, you need to use

parentheses as follows:

SELECT ((2 + 3) \* 2 + 10) / 2;

This expression evaluates to 10.

The data types of the operands in an arithmetic computation determine the

data type of the result. If the operands are integers, the result of arithmetic

operations is an integer. With this in mind, consider the following expression:

SELECT 9 / 2;

With integer division, the result of this expression is 4 and not 4.5.

Obviously, when using constants, you can simply specify numeric values

instead of integer values to get numeric division; however, when the

operands are integer columns or parameters, but you need numeric division,

you have two options. One option is to explicitly cast the operands to a

numeric type with the appropriate precision and scale as follows:

DECLARE @p1 AS INT = 9, @p2 AS INT = 2;

SELECT CAST(@p1 AS NUMERIC(12, 2)) / CAST(@p2 AS

NUMERIC(12, 2));

The result of this expression is 4.500000000000000. The operation here is division. The applicable formula

to calculate the precision here is p1 - s1 + s2 + max(6, s1 + p2 + 1), which

when applied to our inputs results in 27. The formula for the scale is max(6,

s1 + p2 + 1), which in this case results in 15.

Another option is to multiply the first operand by a numeric constant, and

this way force implicit conversion of both the first and the second operands to

a numeric type as follows:

DECLARE @p1 AS INT = 9, @p2 AS INT = 2;

SELECT 1.0 \* @p1 / @p2;

### Aggregate functions

An aggregate function is a function that you apply to a set of rows and get a

single value back. T-SQL supports aggregate functions such as SUM,

COUNT, MIN, MAX, AVG and others.

Aggregate functions ignore NULL inputs when applied to an expression.

The COUNT(\*) aggregate just counts rows, and returns the result as an INT

value. Use COUNT\_BIG to return the row count as a BIGINT value. If you

want to apply an aggregate function to distinct values, add the DISTINCT

clause, as in COUNT(DISTINCT custid).

You can apply aggregate functions in explicit grouped queries as the

following example shows:

SELECT empid, SUM(qty) AS totalqty

FROM Sales.OrderValues

GROUP BY empid;

An aggregate function can also be applied as a scalar aggregate in an

implied grouped query. The presence of the aggregate function causes the

query to be considered a grouped one, as in the following example:

SELECT SUM(qty) AS totalqty FROM Sales.OrderValues;

This query returns the grand total quantity 51,317.

Like with arithmetic operators, also with aggregate functions like AVG,

the data type of the input determines the data type of the result. For instance,

the following query produces an integer average:

SELECT AVG(qty) AS avgqty FROM Sales.OrderValues;

The result of this average is the integer 61.

You can use the two aforementioned options that I described for arithmetic

operations to get a numeric average. Either explicitly cast the input to a

numeric type as follows:

SELECT AVG(CAST(qty AS NUMERIC(12, 2))) AS avgqty FROM

Sales.OrderValues;

Or implicitly as follows:

SELECT AVG(1.0 \* qty) AS avgqty FROM Sales.OrderValues;

This time you get the result 61.827710.

If you’re wondering why the scale of the result value here is 6 digits, the

AVG function is handled internally as a sum divided by a count. The scale of

the input expression (1.0 \* qty) is the sum of the scales of the operands (1 for

1.0 and 0 for the integer qty), which in our case is 1. The sum aggregate’s

scale is the maximum scale among the input values, which in our case is 1.

Then the scale of the result of the division between the sum and the count is

based on the formula max(6, s1 + p2 + 1), which in our case is 6.

## Modify data

The T-SQL support for data manipulation language (DML) includes both

statements that retrieve data (SELECT) and statements that modify data

(INSERT, UPDATE, DELETE, TRUNCATE TABLE, and MERGE). The

previous skills focused on data retrieval; this skill focuses on data

modification.

### Inserting data

T-SQL supports a number of different methods that you can use to insert data

into your tables. Those include statements like INSERT VALUES, INSERT

SELECT, INSERT EXEC, and SELECT INTO. This section covers these

statements and demonstrates how to use them through examples.

Some of the of the code examples in this section use a table called

Sales.MyOrders. Use the following code to create such a table in the sample

database TSQLV4:

USE TSQLV4;

DROP TABLE IF EXISTS Sales.MyOrders;

GO

CREATE TABLE Sales.MyOrders

(

orderid INT NOT NULL IDENTITY(1, 1)

CONSTRAINT PK\_MyOrders\_orderid PRIMARY KEY,

custid INT NOT NULL,

empid INT NOT NULL,

orderdate DATE NOT NULL

CONSTRAINT DFT\_MyOrders\_orderdate DEFAULT

(CAST(SYSDATETIME() AS DATE)),

shipcountry NVARCHAR(15) NOT NULL,

freight MONEY NOT NULL

);

Observe that the orderid column has an identity property defined with a

seed 1 and an increment 1. This property generates the values in this column

automatically when rows are inserted. As an alternative to the identity

property you can use a sequence object to generate surrogate keys.

Also observe that the orderdate column has a default constraint with an

expression that returns the current system’s date.

#### CREATE TABLE (Transact-SQL) IDENTITY (Property)

Syntax:

IDENTITY [ (seed , increment) ]

**Arguments**

seed  
Is the value that is used for the very first row loaded into the table.

increment  
Is the incremental value that is added to the identity value of the previous row that was loaded.

You must specify both the seed and increment or neither. If neither is specified, the default is (1,1).

**Remarks**

Identity columns can be used for generating key values. The identity property on a column guarantees the following:

* Each new value is generated based on the current seed & increment.
* Each new value for a particular transaction is different from other concurrent transactions on the table.

The identity property on a column does not guarantee the following:

* **Uniqueness of the value** - Uniqueness must be enforced by using a **PRIMARY KEY** or **UNIQUE** constraint or **UNIQUE** index.
* **Consecutive values within a transaction** - A transaction inserting multiple rows is not guaranteed to get consecutive values for the rows because other concurrent inserts might occur on the table. If values must be consecutive then the transaction should use an exclusive lock on the table or use the **SERIALIZABLE** isolation level.
* **Consecutive values after server restart or other failures** -SQL Server might cache identity values for performance reasons and some of the assigned values can be lost during a database failure or server restart. This can result in gaps in the identity value upon insert. If gaps are not acceptable then the application should use its own mechanism to generate key values. Using a sequence generator with the **NOCACHE** option can limit the gaps to transactions that are never committed.
* **Reuse of values** - For a given identity property with specific seed/increment, the identity values are not reused by the engine. If a particular insert statement fails or if the insert statement is rolled back then the consumed identity values are lost and will not be generated again. This can result in gaps when the subsequent identity values are generated.

These restrictions are part of the design in order to improve performance, and because they are acceptable in many common situations. If you cannot use identity values because of these restrictions, create a separate table holding a current value and manage access to the table and number assignment with your application.

If a table with an identity column is published for replication, the identity column must be managed in a way that is appropriate for the type of replication used. For more information, see [Replicate Identity Columns](https://docs.microsoft.com/en-us/sql/relational-databases/replication/publish/replicate-identity-columns?view=sql-server-ver15).

Only one identity column can be created per table.

**Example:**

USE AdventureWorks2012;

IF OBJECT\_ID ('dbo.new\_employees', 'U') IS NOT NULL

DROP TABLE new\_employees;

GO

CREATE TABLE new\_employees

(

id\_num int IDENTITY(1,1),

fname varchar (20),

minit char(1),

lname varchar(30)

);

INSERT new\_employees

(fname, minit, lname)

VALUES

('Karin', 'F', 'Josephs');

INSERT new\_employees

(fname, minit, lname)

VALUES

('Pirkko', 'O', 'Koskitalo');

#### INSERT VALUES

With the INSERT VALUES statement, you can insert one or more rows into

a target table based on value expressions. Here’s an example for a statement

inserting one row into the Sales.MyOrderValues table:

INSERT INTO Sales.MyOrders(custid, empid, orderdate,

shipcountry, freight)

VALUES(2, 19, '20170620', N'USA', 30.00);

Specifying the target column names after the table name is optional but

considered a best practice. That’s because it enables you to control the source

value to target column association, irrespective of the order in which the

columns were defined in the table.

Without the target column list, you must specify the values in column

definition order. If the underlying table definition changes but the INSERT

statements aren’t modified accordingly, this can result in either errors, or

worse, values written to the wrong columns.

The INSERT VALUES statement does not specify a value for a column

with an identity property because the property generates the value for the

column automatically. Observe that the previous statement doesn’t specify

the orderid column. If you do want to provide your own value instead of

letting the identity property do it for you, you need to first turn on a session

option called IDENTITY\_INSERT, as follows:

SET IDENTITY\_INSERT <table> ON;

When you’re done, you need to remember to turn it off.

Note that in order to use this option, you need quite strong permissions;

you need to be the owner of the table or have ALTER permissions on the

table.

Besides using the identity property, there are other ways for a column to

get its value automatically in an INSERT statement. A column can have a

default constraint associated with it like the orderdate column in the

Sales.MyOrders table. If the INSERT statement doesn’t specify a value for

the column explicitly, SQL Server will use the default expression to generate

that value. For example, the following statement doesn’t specify a value for

orderdate, and therefore SQL Server uses the default expression:

INSERT INTO Sales.MyOrders(custid, empid, shipcountry,

freight)

VALUES(3, 11, N'USA', 10.00);

Another way to achieve the same behavior is to specify the column name

in the names list and the keyword DEFAULT in the respective element in the

VALUES list. Here’s an INSERT example demonstrating this:

INSERT INTO Sales.MyOrders(custid, empid, orderdate,

shipcountry, freight)

VALUES(3, 17, DEFAULT, N'USA', 30.00);

If you don’t specify a value for a column, SQL Server first checks whether

the column gets its value automatically—for example, from an identity

property or a default constraint. If that’s not the case, SQL Server checks

whether the column allows NULLs, in which case it assumes a NULL. If

that’s not the case, SQL Server generates an error.

The INSERT VALUES statement doesn’t limit you to inserting only one

row; rather, it enables you to insert multiple rows. Simply separate the rows

with commas, as follows:

INSERT INTO Sales.MyOrders(custid, empid, orderdate,

shipcountry, freight) VALUES

(2, 11, '20170620', N'USA', 50.00),

(5, 13, '20170620', N'USA', 40.00),

(7, 17, '20170620', N'USA', 45.00);

Note that the entire statement is considered one transaction, meaning that if

any row fails to enter the target table, the entire statement fails and no row is

inserted.

To see the result of running all INSERT examples in this section, query the

table by using the following query:

SELECT \* FROM Sales.MyOrders;

#### INSERT SELECT

The INSERT SELECT statement inserts the result set returned by a query

into the specified target table. As with INSERT VALUES, the INSERT

SELECT statement supports optionally specifying the target column names.

Also, you can omit columns that get their values automatically from an

identity property, default constraint, or when allowing NULLs.

As an example, the following code inserts into the Sales.MyOrders table

the result of a query against Sales.Orders returning orders shipped to

customers in Norway:

SET IDENTITY\_INSERT Sales.MyOrders ON;

INSERT INTO Sales.MyOrders(orderid, custid, empid,

orderdate, shipcountry, freight)

SELECT orderid, custid, empid, orderdate,

shipcountry, freight

FROM Sales.Orders

WHERE shipcountry = N'Norway';

SET IDENTITY\_INSERT Sales.MyOrders OFF;

The code turns on the IDENTITY\_INSERT option against Sales.MyOrders

in order to use the original order IDs and not let the identity property generate

those.

Setting IDENTITY\_INSERT to OFF causes the current identity value of

the table to be set to the current maximum value in the identity column. In

our example, the current identity value was set to 11015. If you now add

another row to the table, the order ID will be set to 11016.

#### INSERT EXEC

With the INSERT EXEC statement, you can insert the result set (or sets)

returned by a dynamic batch or a stored procedure into the specified target

table. Much like the INSERT VALUES and INSERT SELECT statements,

INSERT EXEC supports specifying an optional target column list, and allows

omitting columns that accept their values automatically.

To demonstrate the INSERT EXEC statement, the following example uses

a procedure called Sales.OrdersForCountry, which accepts a ship country as

input and returns orders shipped to the input country. Run the following code

to create the Sales.OrdersForCountry procedure:

DROP PROC IF EXISTS Sales.OrdersForCountry;

GO

CREATE PROC Sales.OrdersForCountry

@country AS NVARCHAR(15)

AS

SELECT orderid, custid, empid, orderdate, shipcountry,

freight

FROM Sales.Orders

WHERE shipcountry = @country;

GO

Run the following code to invoke the stored procedure with Portugal as the

input country, and insert the result of the procedure into the Sales.MyOrders

table:

SET IDENTITY\_INSERT Sales.MyOrders ON;

INSERT INTO Sales.MyOrders(orderid, custid, empid,

orderdate, shipcountry, freight)

EXEC Sales.OrdersForCountry

@country = N'Portugal';

SET IDENTITY\_INSERT Sales.MyOrders OFF;

Here as well, the code turns on the IDENTITY\_INSERT option against the

target table so that the INSERT statement can specify the values for the

identity column instead of letting the property assign those.

INSERT EXEC works even when the source dynamic batch or stored

procedure has more than one query. But that’s as long as all queries return

result sets that are compatible with the target table definition.

#### SELECT INTO

The SELECT INTO statement involves a query (the SELECT part) and a

target table (the INTO part). The statement creates the target table based on

the definition of the source and inserts the result rows from the query into

that table. The statement copies from the source some aspects of the data

definition like the column names, types, nullability, and identity property, in

addition to the data itself. Certain aspects of the data definition aren’t copied

like indexes, constraints, triggers, permissions, and others. If you want to

include these aspects, you need to script them from the source and apply

them to the target.

The following code shows an example for a SELECT INTO statement that

queries the Sales. Orders table returning orders shipped to Norway, creates a

target table called Sales.MyOrders, and stores the query’s result in the target

table:

DROP TABLE IF EXISTS Sales.MyOrders;

SELECT orderid, custid, orderdate, shipcountry, freight

INTO Sales.MyOrders

FROM Sales.Orders

WHERE shipcountry = N'Norway';

As mentioned, the SELECT INTO statement creates the target table based

on the definition of the source. You don’t have direct control over the

definition of the target. If you want target columns to be defined different

than the source, you need to apply some manipulation.

For example, the source orderid column has an identity property, and

hence the target column is defined with an identity property as well. If you

want the target column not to have the property, you need to apply some kind

of manipulation, like orderid + 0 AS orderid. Note that after you apply

manipulation, the target column definition allows NULLs. If you want the

target column to be defined as not allowing NULLs, you need to use the

ISNULL function, returning a non-NULL value in case the source is a

NULL. This is just an artificial expression that lets SQL Server know that the

outcome cannot be NULL and, hence, the column can be defined as not

enabling NULLs. For example, you could use an expression such as this one:

ISNULL(orderid + 0, -1) AS orderid.

Similarly, the source custid column is defined in the source as allowing

NULLs. To make the target column be defined as NOT NULL, use the

expression ISNULL(custid, -1) AS custid.

If you want the target column’s type to be different than the source, you

can use the CAST or CONVERT functions. But remember that in such a

case, the target column definition enables NULLs even if the source column

disallowed NULLs, because you applied manipulation to the source column.

As with the previous examples, you can use the ISNULL function to make

SQL Server define the target column as not enabling NULLs. For example, to

convert the orderdate column from its source type DATETIME to DATE in

the target, and disallow NULLs, use the expression

ISNULL(CAST(orderdate AS DATE), ‘19000101’) AS orderdate.

To put it all together, the following code uses a query similar to the

previous example, only defining the orderid column without the identity

property as NOT NULL, the custid column as NOT NULL, and the orderdate

column as DATE NOT NULL:

DROP TABLE IF EXISTS Sales.MyOrders;

SELECT

ISNULL(orderid + 0, -1) AS orderid, -- get rid of

-- identity property

-- make column NOT NULL

ISNULL(custid, -1) AS custid, -- make column NOT NULL

empid,

ISNULL(CAST(orderdate AS DATE), '19000101') AS

orderdate,

shipcountry, freight

INTO Sales.MyOrders

FROM Sales.Orders

WHERE shipcountry = N'Norway';

Remember that SELECT INTO does not copy constraints from the source

table, so if you need those, it’s your responsibility to define them in the

target. For example, the following code defines a primary key constraint in

the target table:

ALTER TABLE Sales.MyOrders

ADD CONSTRAINT PK\_MyOrders PRIMARY KEY(orderid);

One of the benefits of using SELECT INTO is that when the database’s

recovery model is not set to full, but instead to either simple or bulk logged,

the statement uses an optimized logging mode. This can potentially result in a

faster insert compared to when full logging is used.

Also, remember that SELECT INTO involves both creating a table and

populating it with data. This means that both the metadata related to the target

table and the data are exclusively locked until the SELECT INTO transaction

finishes. As a result, you can run into blocking situations due to conflicts

related to both data and metadata access.

Generate new table without data, just the structure of another table.

SELECT TOP 0

Order\_Id = ordered

, Cust\_Id = custid

, Order\_Date = orderdate

, Ship\_Country = shipcountry

, freight

INTO Sales.MyOrders

FROM Sales.Orders

When you are done, run the following code for cleanup:

DROP TABLE IF EXISTS Sales.MyOrders;

### UPDATE statement

T-SQL supports the standard UPDATE statement, which enables you to

update existing rows in a table. The standard UPDATE statement has the

following form:

UPDATE <target table>

SET <col 1> = <expression 1>,

...,

<col n> = <expression n>

WHERE <predicate>;

You specify the target table name in the UPDATE clause. If you want to

filter a subset of rows, you indicate a WHERE clause with a predicate. Only

rows for which the predicate evaluates to true are updated. Rows for which

the predicate evaluates to false or unknown are not affected. An UPDATE

statement without a WHERE clause affects all rows. You assign values to

target columns in the SET clause. The source expressions can involve

columns from the table, in which case their values before the update are used.

As an example, you modify rows in the Sales.MyOrderDetails table

representing order lines associated with order 10251. First, query those rows

to examine their state prior to the update:

SELECT \*

FROM Sales.MyOrderDetails

WHERE orderid = 10251;

You get the following output:

orderid productid unitprice qty discount

----------- ----------- --------------------- ------ -----

----

10251 22 16.80 6 0.050

10251 57 15.60 15 0.050

10251 65 16.80 20 0.000

The following code demonstrates an UPDATE statement that adds a five

percent discount to these order lines:

UPDATE Sales.MyOrderDetails

SET discount += 0.05

WHERE orderid = 10251;

Notice the use of the compound assignment operator discount += 0.05.

This assignment is equivalent to discount = discount + 0.05. T-SQL supports

such enhanced operators for all binary assignment operators: += (add), -=

(subtract), \*= (multiply), /= (divide), %= (modulo), &= (bitwise and), |=

(bitwise or), ^= (bitwise xor), += (concatenate).

Query again the order lines associated with order 10251 to see their state

after the update:

SELECT \*

FROM Sales.MyOrderDetails

WHERE orderid = 10251;

You get the following output showing an increase of five percent in the

discount:

orderid productid unitprice qty discount

----------- ----------- --------------------- ------ -----

----

10251 22 16.80 6 0.100

10251 57 15.60 15 0.100

10251 65 16.80 20 0.050

Use the following code to reduce the discount in the aforementioned order

lines by five percent:

UPDATE Sales.MyOrderDetails

SET discount -= 0.05

WHERE orderid = 10251;

These rows should now be back to their original state before the first

update.

**UPDATE based on join**

Standard SQL doesn’t support using joins in UPDATE statements, but TSQL

does. The idea is that you might want to update rows in a table, and

refer to related rows in other tables for filtering and assignment purposes.

As an example, suppose that you want to add a five percent discount to

order lines associated with orders placed by customers from Norway. The

rows you need to modify are in the Sales.MyOrderDetails table. But the

information you need to examine for filtering purposes is in rows in the

Sales.MyCustomers table. In order to match a customer with its related order

lines, you need to join Sales.MyCustomers with Sales.MyOrders, and then

join the result with Sales.MyOrderDetails. Note that it’s not sufficient to

examine the shipcountry column in Sales. MyOrders; instead, you must

check the country column in Sales.MyCustomers.

Based on your knowledge of joins, if you wanted to write a SELECT

statement returning the order lines that are the target for the update, you

would write a query like the following one:

SELECT OD.\*

FROM Sales.MyCustomers AS C

INNER JOIN Sales.MyOrders AS O

ON C.custid = O.custid

INNER JOIN Sales.MyOrderDetails AS OD

ON O.orderid = OD.orderid

WHERE C.country = N'Norway';

This query identifies 16 order lines, all currently with a discount value of

0.000.

In order to perform the desired update, simply replace the SELECT clause

from the last query with an UPDATE clause, indicating the alias of the table

that is the target for the

UPDATE (OD in this case), and the assignment in the SET clause, as

follows:

UPDATE OD

SET OD.discount += 0.05

FROM Sales.MyCustomers AS C

INNER JOIN Sales.MyOrders AS O

ON C.custid = O.custid

INNER JOIN Sales.MyOrderDetails AS OD

ON O.orderid = OD.orderid

WHERE C.country = N'Norway';

Note that you can refer to elements from all tables involved in the

statement in the source expressions, but you’re allowed to modify only one

target table at a time. Rerun the SELECT query to examine the affected order

lines, and you find that they now have a discount value of 0.050.

To get the previous order lines back to their original state, run an UPDATE

statement that reduces the discount by five percent:

UPDATE OD

SET OD.discount -= 0.05

FROM Sales.MyCustomers AS C

INNER JOIN Sales.MyOrders AS O

ON C.custid = O.custid

INNER JOIN Sales.MyOrderDetails AS OD

ON O.orderid = OD.orderid

WHERE C.country = N'Norway';

**Nondeterministic UPDATE**

You should be aware that the proprietary T-SQL UPDATE syntax based on

joins could be nondeterministic. The statement is nondeterministic when

multiple source rows match one target row. Unfortunately, in such a case,

SQL Server doesn’t generate an error or even a warning. Instead, SQL Server

silently performs a nondeterministic UPDATE where it arbitrarily chooses

one of the source rows.

As an example, the following query matches customers with their related

orders, returning the customers’ postal codes, as well as shipping postal codes

from related orders:

SELECT C.custid, C.postalcode, O.shippostalcode

FROM Sales.MyCustomers AS C

INNER JOIN Sales.MyOrders AS O

ON C.custid = O.custid

ORDER BY C.custid;

This query generates the following output:

custid postalcode shippostalcode

----------- ---------- --------------

1 10092 10154

1 10092 10156

1 10092 10155

1 10092 10154

1 10092 10154

1 10092 10154

2 10077 10182

2 10077 10181

...

Each customer row is repeated in the output for each matching order. This

means that each customer’s only postal code is repeated in the output as

many times as the number of matching orders. It’s important for the purposes

of this example to remember that there is only one postal code per customer.

The shipping postal code is associated with an order, so as you can realize,

there can be multiple distinct shipping postal codes per customer. With this in

mind, consider the following UPDATE statement:

UPDATE C

SET C.postalcode = O.shippostalcode

FROM Sales.MyCustomers AS C

INNER JOIN Sales.MyOrders AS O

ON C.custid = O.custid;

There are 89 customers that have matching orders—some with multiple

matches. SQL Server doesn’t generate an error though; instead it arbitrarily

chooses for each target row which source row is to be considered for the

update, returning the following message:

(89 row(s) affected)

Query the rows from the Sales.Customers table after the update:

SELECT custid, postalcode

FROM Sales.MyCustomers

ORDER BY custid;

This generated the following output on one system, but your results could

be different:

custid postalcode

----------- ----------

1 10154

2 10182

...

(91 row(s) affected)

Note that the table has 91 rows, but because only 89 of those customers

have related orders, the previous UPDATE statement affected 89 rows.

As to which source row gets chosen for each target row, the choice isn’t

exactly random, but arbitrary; in other words, it’s optimization-dependent. At

any rate, you do not have any logical elements in the language to control this

choice. The recommended approach is simply not to use such

nondeterministic UPDATE statements, rather have logic in your solution to

break ties.

For example, suppose that you want to update the customer’s postal code

with the shipping postal code from the customer’s first order (based on the

sort order of orderdate, orderid). You can achieve this using the following

code:

UPDATE C

SET C.postalcode = A.shippostalcode

FROM Sales.MyCustomers AS C

CROSS APPLY (SELECT TOP (1) O.shippostalcode

FROM Sales.MyOrders AS O

WHERE O.custid = C.custid

ORDER BY orderdate, orderid) AS A;

The APPLY operator applies a subquery that identifies its most recent order.

SQL Server generates the following message:

(89 row(s) affected)

Query the Sales.MyCustomers table after the update:

SELECT custid, postalcode

FROM Sales.MyCustomers

ORDER BY custid;

You get the following output:

custid postalcode

----------- ----------

1 10154

2 10180

...

(91 row(s) affected)

If you want to use the most-recent order as the source for the update,

simply use descending sort order in both columns: ORDER BY orderdate

DESC, orderid DESC.

**UPDATE with a variable**

Sometimes you need to modify a row and also collect the result of the

modified columns into variables. You can handle such a need with a

combination of UPDATE and SELECT statements, but this would require

two visits to the row. T-SQL supports a specialized UPDATE syntax that

allows achieving the task by using one statement and one visit to the row.

As an example, run the following query to examine the current state of the

order line associated with order 10250 and product 51:

SELECT \*

FROM Sales.MyOrderDetails

WHERE orderid = 10250

AND productid = 51;

This code generates the following output:

orderid productid unitprice qty discount

----------- ----------- --------------------- ------ -----

----

10250 51 42.40 35 0.150

Suppose that you need to modify the row, increasing the discount by five

percent, and collect the new discount into a variable called @newdiscount.

You can achieve this using a single UPDATE statement, as follows.

DECLARE @newdiscount AS NUMERIC(4, 3) = NULL;

UPDATE Sales.MyOrderDetails

SET @newdiscount = discount += 0.05

WHERE orderid = 10250

AND productid = 51;

SELECT @newdiscount;

As you can see, the UPDATE and WHERE clauses are similar to those

you use in normal UPDATE statements. But the SET clause uses the

assignment @newdiscount = discount += 0.05, which is equivalent to using

@newdiscount = discount = discount + 0.05. The statement assigns the result

of discount + 0.05 to discount, and then assigns the result to the variable

@newdiscount. The last SELECT statement in the code returns the new

discount 0.200.

When you’re done, issue the following code to undo the last change:

UPDATE Sales.MyOrderDetails

SET discount -= 0.05

WHERE orderid = 10250

AND productid = 51;

**UPDATE all-at-once**

Earlier in the book as part of the discussion about logical query processing I

explained that expressions that appear in the same logical phase are treated as

a set, in an *all-at-once* manner. The all-at-once concept also has implications

on UPDATE statements. To demonstrate those implications, this section uses

a table called T1. Use the following code to create the table T1 and insert a

row into it:

DROP TABLE IF EXISTS dbo.T1;

CREATE TABLE dbo.T1

(

keycol INT NOT NULL

CONSTRAINT PK\_T1 PRIMARY KEY,

col1 INT NOT NULL,

col2 INT NOT NULL

);

INSERT INTO dbo.T1(keycol, col1, col2) VALUES(1, 100, 0);

Next, examine the following code but don’t run it yet:

DECLARE @add AS INT = 10;

UPDATE dbo.T1

SET col1 += @add, col2 = col1

WHERE keycol = 1;

SELECT \* FROM dbo.T1;

Can you guess what should be the value of col2 in the modified row after

the update? If you guessed 110, you were not thinking of the assignments as a

set, all-at-once. All assignments use the original values of the row as the

source values, irrespective of their order of appearance. So the assignment

col2 = col1 doesn’t get the col1 value after the change, but rather before the

change—namely 100. To verify this, run the previous code.

You get the following output:

keycol col1 col2

----------- ----------- -----------

1 110 100

When you’re done, run the following code for cleanup:

DROP TABLE IF EXISTS dbo.T1;

### **Deleting data**

T-SQL supports two statements that you can use to delete rows from a table:

DELETE and TRUNCATE TABLE.

As a reminder, the sample data involves the tables

Sales.MyCustomers, Sales.MyOrders, and Sales.MyOrderDetails, which are

initially created as copies of the tables Sales.Customers, Sales.Orders, and

Sales.OrderDetails, respectively. Use the following code to recreate tables

and repopulate them with sample data:

DROP TABLE IF EXISTS Sales.MyOrderDetails, Sales.MyOrders,

Sales.MyCustomers;

SELECT \* INTO Sales.MyCustomers FROM Sales.Customers;

ALTER TABLE Sales.MyCustomers

ADD CONSTRAINT PK\_MyCustomers PRIMARY KEY(custid);

SELECT \* INTO Sales.MyOrders FROM Sales.Orders;

ALTER TABLE Sales.MyOrders

ADD CONSTRAINT PK\_MyOrders PRIMARY KEY(orderid);

SELECT \* INTO Sales.MyOrderDetails FROM

Sales.OrderDetails;

ALTER TABLE Sales.MyOrderDetails

ADD CONSTRAINT PK\_MyOrderDetails PRIMARY KEY(orderid,

productid);

#### **DELETE statement**

With the DELETE statement, you can delete rows from a table. You can

optionally specify a predicate to restrict the rows to be deleted. The general

form of a DELETE statement looks like the following:

DELETE FROM <table>

WHERE <predicate>;

If you don’t specify a predicate, all rows from the target table are deleted.

As with unqualified updates, you need to be especially careful about

accidentally deleting all rows by highlighting only the DELETE part of the

statement, missing the WHERE part.

The following example deletes all order lines containing product ID 11

from the Sales.MyOrderDetails table:

DELETE FROM Sales.MyOrderDetails

WHERE productid = 11;

You get a message indicating that 38 rows were affected.

The tables used by the examples in this chapter are very small, but in a

more realistic production environment, the volumes of data are likely to be

much bigger. A DELETE statement is fully logged and as a result, large deletes can take a long time to complete, and much longer to roll back if you need to terminate them.

Such large deletes can cause the transaction log to increase in size

dramatically during the process. They can also result in lock escalation,

meaning that SQL Server escalates fine-grained locks like row or page locks

to a full-blown table lock. Such escalation can result in blocking access to all

table data by other processes.

To prevent the aforementioned problems from happening, you can split

your large delete into smaller chunks. You can achieve this by using a

DELETE statement with a TOP option that limits the number of affected

rows in a loop. Here’s an example for implementing such a solution:

WHILE 1 = 1

BEGIN

DELETE TOP (1000) FROM Sales.MyOrderDetails

WHERE productid = 12;

IF @@rowcount < 1000 BREAK;

END

As you can see, the code uses an infinite loop (WHILE 1 = 1 is always

true). In each iteration, a DELETE statement with a TOP option limits the

number of affected rows to no more than 1,000 at a time. Then the IF

statement checks if the number of affected rows is less than 1,000; in such a

case, the last iteration deleted the last chunk of qualifying rows. After the last

chunk of rows has been deleted, the code breaks from the loop. With this

sample data, there are only 14 qualifying rows in total. So if you run this

code, it is done after one round, break from the loop, and return. But with a

large number of qualifying rows, say, millions, you’d very likely be better off

with such a solution.

#### **TRUNCATE TABLE statement**

TRUNCATE TABLE is an optimized statement that deletes all rows from the

target table or partition. Unlike the DELETE statement, the TRUNCATE

TABLE statement doesn’t support a filter. Also, whereas the DELETE

statement is fully logged and therefore tends to be quite slow, the

TRUNCATE table statement uses an optimized logging mode and therefore

is significantly faster.

For example, the following statement truncates the table Sales.MyOrderDetails:

TRUNCATE TABLE Sales.MyOrderDetails;

Suppose that you had a partitioned table called MyTable and you wanted

to truncate partitions 1, 2 and 11 to 20. You would achieve this with the

following code:

TRUNCATE TABLE MyTable WITH ( PARTITIONS(1, 2, 11 TO 20)

);

Besides the performance difference and the fact that TRUNCATE TABLE

doesn’t support a filter, there are a few additional differences compared to the

DELETE statement:

* You cannot assign direct TRUNCATE TABLE permissions, rather at minimum you need ALTER permission on the target table. A common workaround is to place the TRUNCATE TABLE statement in a module, like a stored procedure, and assign the required permission to the module using the EXECUTE AS clause.
* If there’s a column with an identity property in the target table, DELETE doesn’t reset the property whereas TRUNCATE TABLE does.
* If there are any foreign keys pointing to the target table, a DELETE statement is supported as long as there are no related rows in the referencing table, but a TRUNCATE TABLE statement isn’t. You need to first drop the foreign keys, truncate the table, and then recreate the foreign keys.
* If there are any indexed views based on the table, a DELETE statement is supported whereas a TRUNCATE TABLE statement isn’t.

Clearly, if you need to delete all rows from a table or a partition but leave

the table definition in place, the recommended tool to use is the TRUNCATE

TABLE statement.

**DELETE based on a join**

Much like the proprietary syntax that T-SQL supports for an UPDATE

statement based on a join, T-SQL supports similar syntax for a DELETE

statement based on a join. The idea is to allow you to delete rows from one

table based on the presence of related rows in other tables, with the ability to

apply a filter predicate that is based on attributes in the related tables.

As an example, the following statement deletes orders placed by customers

from the US:

DELETE FROM O

FROM Sales.MyOrders AS O

INNER JOIN Sales.MyCustomers AS C

ON O.custid = C.custid

WHERE C.country = N'USA';

Notice that there are two FROM clauses. The second is mandatory and is

similar to the FROM clause in a SELECT statement. That’s where you apply

table operators like joins. The first FROM clause appears right after the

DELETE clause and is optional. That’s where you specify the target for the

delete. In our case it’s the alias O representing the Sales.MyOrders table.

When you’re done, run the following code for cleanup:

DROP TABLE IF EXISTS Sales.MyOrderDetails, Sales.MyOrders,

Sales.MyCustomers;

### **Merging data**

With the MERGE statement, you can merge data from a source table into a

target table. The statement has many practical uses in both online transaction

processing (OLTP) scenarios and in data warehousing ones. As an example

of an OLTP use case, suppose that you have a table that isn’t updated directly

by your application; instead, you get the delta of changes periodically from

an external system. You first load the delta of changes into a staging table,

and then use the staging table as the source for the merge operation into the

target.

As an example for a data warehousing scenario, suppose that you maintain

aggregated views of the data in your data warehouse. Using the MERGE

statement, you can apply changes that were applied to detail rows into the

aggregated form.

These are just a couple of typical use cases; there are many more. This

lesson describes the MERGE statement and its different options, and

demonstrates its use through examples.

**Using the MERGE statement**

With the MERGE statement, you can merge data from a source table or table

expression into a target table. This statement is mostly standard, with one

proprietary extension by Microsoft of a clause called WHEN NOT

MATCHED BY SOURCE. The general form of the MERGE statement is as

follows:

MERGE INTO <target table> AS TGT

USING <SOURCE TABLE> AS SRC

ON <merge predicate>

WHEN MATCHED [AND <predicate>] -- two clauses allowed:

THEN <action> -- one with UPDATE one with DELETE

WHEN NOT MATCHED [BY TARGET] [AND <predicate>] -- one

clause allowed:

THEN INSERT... –- if indicated, action must be INSERT

WHEN NOT MATCHED BY SOURCE [AND <predicate>] -- two

clauses allowed:

THEN <action>; -- one with UPDATE one with DELETE

The following are the clauses of the statement and their roles:

**MERGE INTO <target table>** This clause defines the target table for

the operation. You can alias the table in this clause if you want.

**USING <source table>** This clause defines the source table for the

operation. You can alias the table in this clause if you want. Note that

the USING clause is designed similar to a FROM clause in a SELECT

query, meaning that in this clause you can define table operators like

joins, refer to a table expression like a derived table or a common table

expression (CTE), or even refer to a table function like

OPENROWSET. The outcome of the USING clause is eventually a

table result, and that table is considered the source of the merge

operation.

**ON <merge predicate>** In this clause, you specify a predicate that

matches rows between the source and the target and defines whether a

source row is or isn’t matched by a target row. Note that this clause

isn’t a filter like the ON clause in a join.

**WHEN MATCHED [AND <predicate>] THEN <action>** This

clause defines an action to take when a source row is matched by a

target row. Because a target row exists, an INSERT action isn’t allowed

in this clause. The two actions that are enabled are UPDATE and

DELETE. If you want to apply different actions in different conditions,

you can specify two WHEN MATCHED clauses, each with a different

additional predicate to determine when to apply an UPDATE and when

to apply a DELETE.

**WHEN NOT MATCHED [BY TARGET] [AND <predicate>]**

**THEN <action>**

This clause defines what action to take when a source row is not

matched by a target row. Because a target row does not exist, the only

action allowed in this clause (if you choose to include this clause in the

statement) is INSERT. Using UPDATE or DELETE holds no meaning

when a target row doesn’t exist. You can still add an additional

predicate that must be true in order to perform the action.

**WHEN NOT MATCHED BY SOURCE [AND <predicate>] THEN**

**<action>**

This clause is a proprietary extension by Microsoft to the standard

MERGE statement syntax. It defines an action to take when a target

row exists, but it is not matched by a source row. Because a target row

exists, you can apply either an UPDATE or a DELETE, but not an

INSERT. If you want, you can have two such clauses with different

additional predicates that define when to use an UPDATE and when to

use a DELETE.

To demonstrate examples of the MERGE statement, this section uses the

Sales.MyOrders table and the Sales.SeqOrderIDs sequence. Use the

following code to create these objects.

**Click here to view code image**

DROP TABLE IF EXISTS Sales.MyOrders;

DROP SEQUENCE IF EXISTS Sales.SeqOrderIDs;

CREATE SEQUENCE Sales.SeqOrderIDs AS INT

MINVALUE 1

CACHE 10000;

CREATE TABLE Sales.MyOrders

(

orderid INT NOT NULL

CONSTRAINT PK\_MyOrders\_orderid PRIMARY KEY

CONSTRAINT DFT\_MyOrders\_orderid

DEFAULT(NEXT VALUE FOR Sales.SeqOrderIDs),

custid INT NOT NULL

CONSTRAINT CHK\_MyOrders\_custid CHECK(custid > 0),

empid INT NOT NULL

CONSTRAINT CHK\_MyOrders\_empid CHECK(empid > 0),

orderdate DATE NOT NULL

);

Notice that the sequence is defined to start with the value 1, and uses a

cache size of 10,000 for performance reasons. The cache size defines how

frequently to write a recoverable value to disk. To request a new key from the

sequence, you use the function NEXT VALUE FOR <sequence\_name>. Our

code defines a default constraint with the function call for the orderid column

to automate the creation of keys when new rows are inserted.

Suppose that you need to define a stored procedure that accepts as input

parameters attributes of an order. If an order with the input order ID already

exists in the Sales.MyOrders table, you need to update the row, setting the

values of the nonkey columns to the new ones. If the order ID doesn’t exist in

the target table, you need to insert a new row. Because this book doesn’t

cover stored procedures until Chapter 3, the examples in this section use local

variables for now. A MERGE statement in a stored procedure simply refers

to the procedure’s input parameters instead of the local variables.

The first things to identify in a MERGE statement are the target and the

source tables. The target is easy—it’s the Sales.MyOrders table. The source

is supposed to be a table or table expression, but in this case, it’s just a set of

input parameters making an order. To turn the inputs into a table expression,

you can define a derived table based on the VALUES clause, which is also

known as a table value constructor. The following MERGE statement updates

the target row if the source key exists in the target, and inserts a new row if it

doesn’t:

**Click here to view code image**

DECLARE

@orderid AS INT = 1, @custid AS INT = 1,

@empid AS INT = 2, @orderdate AS DATE = '20170212';

MERGE INTO Sales.MyOrders WITH (SERIALIZABLE) AS TGT

USING (VALUES(@orderid, @custid, @empid, @orderdate))

AS SRC( orderid, custid, empid, orderdate)

ON SRC.orderid = TGT.orderid

WHEN MATCHED THEN

UPDATE

SET TGT.custid = SRC.custid,

TGT.empid = SRC.empid,

TGT.orderdate = SRC.orderdate

WHEN NOT MATCHED THEN

INSERT VALUES(SRC.orderid, SRC.custid, SRC.empid,

SRC.orderdate);

Observe that the MERGE predicate compares the source order ID with the

target order ID. When a match is found (the source order ID is matched by a

target order ID), the MERGE statement performs an UPDATE action that

updates the values of the nonkey columns in the target to those from the

respective source row.

When a match isn’t found (the source order ID is not matched by a target

order ID), the MERGE statement inserts a new row with the source order

information into the target.