

T-SQL supports two “not equal to” operators: <> and !=. The former is standard and the latter is not. T-SQL supports multiple functions that convert a source value to a target type. Among them are the CAST and CONVERT functions. The former is standard

and the latter isn’t. The nonstandard CONVERT function has a style argument that CAST doesn’t support. Because CAST is standard, you should consider it your default choice for conversions. You should consider using CONVERT only when you need to rely on the style

argument. Yet another example of choosing the standard form is in the termination of T-SQL statements. According to standard SQL, you should terminate your statements with a semicolon. T-SQL currently doesn’t make this a requirement for all statements, only in cases where there would otherwise be ambiguity of code elements, such as in the WITH clause of a common table expression (CTE). You should still follow the standard and terminate all of your statements even where it is currently not required.

A relation in the relational model is what SQL calls a table. The two are not synonymous. A predicate is an expression that when attributed to some object, makes a proposition either true or false. For example, “salary greater than $50,000” is a predicate. You can evaluate

this predicate for a specific employee, in which case you have a proposition. For example, suppose that for a particular employee, the salary is $60,000. When you evaluate the proposition for that employee, you get a true proposition. In other words, a predicate is a parameterized proposition.

Remember that a relation has a heading and a body. The heading is a set of attributes and the body is a set of tuples. Remember from the definition of a set that a set is supposed to be considered as a whole. What this translates to in T-SQL is that you’re supposed to write queries that interact with the tables as a whole. You should try to avoid using iterative constructs like cursors and loops that iterate through the rows one at a time. You should also try to avoid thinking in iterative terms because this kind of thinking is what leads to iterative solutions. For people with a procedural programming background, the natural way to interact with data (in a file, record set, or data reader) is with iterations. So using cursors and other iterative constructs in T-SQL is, in a way, an extension to what they already know. However, the correct way from the relational model’s perspective is not to interact with the rows one at a time; rather, use relational operations and return a relational result. This, in T-SQL, translates to writing queries.

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;

Tatt: the order of rows is not guaranteed but order of columns is.

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;

T-SQL allows a SELECT list that looks 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 in the following.

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

T-SQL attempts to represent a relation with a table, a tuple with a row, and an attribute with a column;

Can you identify what the nonrelational aspects of the query are? Answer: The query doesn’t alias the expression YEAR(orderdate), so there’s no name for the result attribute. The query can return duplicates(but set should not have duplicates). The query forces certain presentation ordering to the result and uses ordinal positions in the ORDER BY clause( but set should not have any ordering).

SELECT custid, YEAR(orderdate)

FROM Sales.Orders

ORDER BY 1, 2;

Removing the non-relational aspects from the query you get:

SELECT distinct custid, YEAR(orderdate) AS ORderYear

FROM Sales.Orders;

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

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 >= '20030101'

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 2003. 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.

Attempting to refer in the WHERE clause to a column alias defined in the SELECT clause:

SELECT country, YEAR(hiredate) AS yearhired

FROM HR.Employees WHERE yearhired >= 2003;

This query fails with the following error.

Msg 207, Level 16, State 1, Line 3

Invalid column name 'yearhired'.

Group Phase: All expressions processed in subsequent phases must guarantee a single value per group. If you refer to an element from the GROUP BY list (for example, country), you already have such a guarantee, so such a reference is allowed. However, if you want to

refer to an element that is not part of your GROUP BY list (for example, empid), it must be contained within an aggregate function like MAX or SUM or COUNT etc.

The WHERE clause is evaluated before rows are grouped, and therefore is evaluated per row. The HAVING clause is evaluated after rows are grouped, and therefore is evaluated per group.

According to the relational theory, a set should be unordered, the rows distinct (no duplicates in result set), every column has distinct name.

Select Alias is only available in order by phase (which comes last after select).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 the following error.

Msg 207, Level 16, State 1, Line 1

Invalid column name 'yearhired'.

The reason that this isn’t allowed is that, conceptually, T-SQL evaluates all expressions that appear in the same logical query processing phase in an all-at-once manner. Note the use of the word conceptually. SQL Server won’t necessarily physically process all expressions at the same point in time, but it has to produce a result as if it did. This behavior is different than many other programming languages where expressions usually get evaluated in a left-toright order, making a result produced in one expression visible to the one that appears to its

right. But T-SQL is different.

A query may specify the TOP or OFFSET-FETCH filtering options. If it does, the same ORDER BY clause that is normally used to define presentation ordering also defines which rows to filter for these options. Tatt: TOP or OFFSET-FETCH need not necessarily be used with ORDER BY??? Even though they only make sense when used with ORDER BY as otherwise the order of rows in result set is not guaranteed. So doing a TOP 10 might result different result sets each time.

Tatt: DISTINCT clause has to appear before the first column is referred to in a SELECT list. So you can’t do SELECT E.Name, DISTINCT E.Age…; ????

Tatt: predicate is the condition and proposition is the instantiation of the condition. Means, proposition refers to the rows which satisfy that predicate.

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.

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. So the following won’t work:

SELECT H.firstname, HR.Employees.lastname

FROM HR.Employees AS H

But this will work:

SELECT H.firstname, lastname

FROM HR.Employees AS H

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 or just empid.

Using an \* is just a matter of laziness. You send more data than is needed over the network, and this can have a negative impact on the system’s performance.

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, and <alias> = <expression> as in employeeid = empid. Use the first form with the AS clause because it’s both standard and we find it to be the most readable.

A result with duplicates is considered nonrelational because relations—being sets—are not supposed to have duplicates. Therefore,

if duplicates are possible in the result, and you want to eliminate them in order to return a relational result, you can do so by adding a DISTINCT clause, as in the following.

SELECT DISTINCT country, region, city FROM HR.Employees;

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

Tatt: Remember SELECT 1 can be used to ping SQL server to test network performance. So that is non-standard SQL.

T-SQL supports both a standard form to delimit identifiers using double quotation marks, as in "Sales"."Orders", as well as a proprietary form using square brackets, as in [Sales].[Orders]. For example, an attribute called 2006 is considered an irregular identifier because it starts with a digit, and therefore must be delimited as "2006" or [2006].

When defining columns in tables, parameters in procedures and functions, and variables in T-SQL batches, you need to choose a data type for those. Much like a type is a constraint, NOT NULL is a constraint as well. If an attribute isn’t supposed to allow NULLs, it’s important to enforce a NOT NULL constraint as part of its definition.

Suppose you have an attribute representing test scores, which are integers in the range 0 to 100. Using an INT data type for this purpose is overkill. It would use 4 bytes per value, whereas a TINYINT would use only 1 byte, and is therefore the more appropriate type in this case. If the value is supposed to represent both date and time, you should consider DATETIME2 or SMALLDATETIME. The former requires storage between 6 to 8 bytes (depending on precision), and as an added value, provides a wider range of dates and improved, controllable precision. Be very careful with the imprecise types FLOAT and REAL.

Another important aspect in choosing a type has to do with choosing fixed types (CHAR, NCHAR, BINARY) vs. dynamic ones (VARCHAR, NVARCHAR, VARBINARY). Fixed types use the storage for the indicated size; for example, CHAR(30) uses storage for 30 characters, whether you actually specify 30 characters or less. This means that updates will not require the row to physically expand, and therefore no data shifting is required. So for attributes that get updated frequently, where the update performance is a priority, you should consider fixed types. Note that when compression is used—specifically row compression—SQL Server stores fixed types like variable ones, but with less overhead.

Variable types use the storage for what you enter, plus a couple of bytes for offset information (or 4 bits with row compression). So for widely varying sizes of strings, if you use variable types you can save a lot of storage. As already mentioned, the less storage used, the less there is for a query to read, and the faster the query can perform. So variable length types are usually preferable in such cases when read performance is a priority.