# SQL Standards

The purpose of SQL Standards is to standardize a database environment, thru all environments, eventually being deployed to a production environment.

Deployable SQL code can be stored procedures, DTS packages, SQL jobs, replication filters, or SQL embedded in client objects.

It is expected that all new SQL code and database will adhere to these standards; existing code can be retrofitted to meet the standards to whatever degree possible, as modifications to that code are required.

## General rules

These general rules will help your database be performant, standard, future-proof and portable.

* Always use primary keys and use INT IDENTITY for primary keys

SQL Server will perform best (lookups and joins) when a small numerical value is used for the primary key/clustered index.

CREATE TABLE TableExample

(

TableExampleID INT IDENTITY (1,1)

)

DECLARE @TableExample TABLE

(

TableExampleID INT IDENTITY (1,1)

)

Exceptions:

For potentially large data sets, there may be conditions where BIGINT may be needed for the primary key.

If you think you need a GUID (UNIQUEIDENTIFIER) for a primary key, investigate if a GUID can be added as a separate attribute instead and keep the primary key as spelled out above.

If you determine you absolutely need to use a GUID for a PK, don't put a clustered index on them.

* Create foreign keys

A relational database works best when referential integrity is enforced.

This will help minimize data anomalies.

* Commenting your code

Create comments in the header of objects such as views, stored procedures, UDFs.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author: Jane Doe

Create Date: 01/01/2023

Purpose: Do a thing that needs be done

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

* Don’t immediately delete database objects

Rename objects and then delete the objects after they have been in production for at least an acceptable period.

Example: [OldTable] becomes [OldTable\_DeleteAfter01\_01\_2023]

* Ensure all changes are backwards compatible

Adding objects is usually a backwards compatible operation but changing objects can cause problems.

Always make changes with the assumption that existing applications will continue to depend on existing database objects for a period after an upgrade has taken place.

* Ensure all schema and data change scripts can be rerun.

All changes should be done through a schema update script or data change script.

These scripts must be re-runnable.

For example, if you are inserting data, you should check to make sure the data isn't already present, or always delete the existing data. Schema changes should only run if the changes aren't already present.

* Protect sensitive data.

Defining what is considered ‘sensitive data’ will be a chore and needs be done.

Data should be consumable by appropriate parties and only appropriate parties.

Sensitive data should never be easily readable (cleartext) and should always be stored as encrypted data.

Using encryption at the application layer is acceptable.

Relying on file system encryption only protects the data if hardware is compromised. This is not sufficient protection for data.

Using encryption at the database layer (TDE, Column-Level Encryption) may be incompatible with 3rd party software.

* Use a semicolon to terminate SQL statements.

Although the lack of semicolons is completely forgivable, it helps to understand more complicated code if individual statements are terminated.

With one or two exceptions, such as delimiting the previous statement from a CTE, using semicolons is currently only a decoration, though it is a good habit to adopt to make code more future-proof and portable.

* Avoid Implicit conversions

Implicit conversions can have unexpected results, such as truncating data or reducing performance. It is not always clear in expressions how differences in data types are going to be resolved.

If data is implicitly converted in a join operation, the database engine is more likely to build a poor execution plan.

You should explicitly define your conversions to avoid unintentional consequences.

## Data Types

Always use appropriate data types. Storing data in the wrong form leads to major issues with coding, indexing, sorting, and other operations. Put the data into the appropriate ‘storage’ data type.

* Use DateTime2(2)

DateTime2(2) uses less space than its predecessor DateTime, while allowing more precision. The parameter (2) allows for more specificity as rounded milliseconds.

* Never use float

Float is a rounded value and will produce inaccurate results.

The FLOAT (8 byte) and REAL (4 byte) data types are suitable only for specialist scientific use since they are approximate types with an enormous range.

The DECIMAL type is an exact data type and has an impressive range from -10^38+1 thru 10^38-1

* Don’t put delimited lists inside of columns. Maintain the atomicity and purpose of the data type.

Example: A column of UserIDs: 123, 124, 125

* Bit (Boolean) columns should start with “is”

This allows for clearer understanding of the purpose of the column.

* Use VARCHAR (X) instead or NVARCHAR (X)

Only use NVARCHAR when internationalization is truly needed for the column.

* Use of VARCHAR (Max) or NVARCHAR (Max)

Using VARCHAR (max) or NVARCHAR (max) treats columns as BLOBS and stores off-page, preventing online reindexing.   
Only use MAX when you need more than 8000 bytes (4000 characters for NVARCHAR, 8000 characters for VARCHAR).

* Be explicit on the size for VARCHAR values

A VARCHAR that is declared without an explicit length is specified as a length of 1.

Be explicit with the size.

* Use VARCHAR (50) when a variable text column is needed

For consistency’s sake, use 50 as the standard size.

There will be times when you need more than 50, and these can be tackled when encountered.

Also, use Varchar instead of NVARCHAR. If an international character set is needed (as mentioned previously), the column may be changed as needed.

* Use a TINYINT or SMALLINT where it makes sense

If you have a small range of valid numbers, using a smaller data type will minimize storage requirements and help improve I/O performance.

## General Naming Conventions

* Avoid using numbers; if a number is absolutely required, spell it out as a word.
* Database and object names should not use keywords

Example Names: Update, User, Object

* Database names should be singular, not plural and should be named after the context they are persisting
* Table names should be singular, not plural

Avoid adding s/es to the end of table names

Use the singular form (i.e., the Loan table and not the Loans table).

* Associative Tables should use the names of their foreign keys

In order to resolve many-to-many relationships, an associative table should be used.

Example:

Table: [VehicleOwner]

Table: Vehicle Associative Table: [VehicleOwnerVehicle]

* Column Names should provide a clear, concise description of the attribute in mixed case, without prefixes or underscores.
* ‘Camel capitalization’ (i.e., mixed case) should be used on all identifiers (for example, FirstName, LogDetail, AcctSysId, LkupCountryCode, @ErrNum, etc).
* Don't start stored procedures with "sp"

"sp" is reserved for system stored procedures and can lead to conflict.

“usp” should be avoided.

* Objects should be explicitly named

Autogenerated names will be confusing and lead to less readability of the database system for future usage.

Follow these guidelines for naming objects

* + Primary Keys

PK\_ + Name of Table

PK\_TableName

* + Foreign Keys

FK\_ + TableName + ‘\_’ + Column Name (first column name required, rest optional)

FK\_ChildTable\_ParentTable

FK\_ChildTable\_ParentTable\_ColumnName

FK\_DimMember\_DimDate\_SignUpDate

* + Default Constraints

DF\_ + TableName + ‘\_’ + Column Name

DF\_FGMarketDataChange\_DateCreated

DF\_DimDate\_IsWorkday

DF\_ReimbursementAccount\_ZeroBalanceAtEndOfPlanYear

* + Indexes

Index names should be readable to know what is contained within the index.

Include as much detail as possible, so that the purpose of the index is readily identifieable.

IX\_ + TableName\_ + ColumnName\_ + ColumnName(s)\_

IX\_Employee\_EmployeeName

IX\_Employee\_EmployeeName\_Position

IX\_ + TableName\_ + FunctionalName

IX\_Employee\_EmployeeDashboard

IX\_ + TableName\_ + ColumnName(s)\_ + INC\_ + Included ColumnName(s)\_

IX\_Employee\_EmployeeName\_\_INC\_\_Field\_Field2\_Field3

IX\_Employee\_EmployeeName\_\_INC\_\_Many

Filtered Index Example:

FIX\_ + TableName\_ + ColumnName\_ + ColumnName(s)\_

FIX\_Employee\_EmployeeName

Columnstore Index Example:

CIX\_ + TableName\_ + ColumnName\_ + ColumnName(s)\_

CIX\_Employee\_EmployeeID

* Avoid acronyms for object names

Object names should be descriptive.

Naming of objects with acronyms should usually be avoided.

The exception would be allowed if the name is universally understood.

* Avoid using product-specific names, or names whose meaning is subject to change (Schema, DCN159)

## Recommendations

* Normalize to Third Normal Form (3NF)
* Choose appropriate file groups for objects.

Use of the Primary filegroup for new objects may be allowed.

Consider alternative file groups for better organization.

* INSERT column list.

The INSERT statement need not have a column list, but when you omit it, it assumes certain columns in a particular order.

It likely to cause errors if the table experiences changes in the schema.

Column lists also make code more intelligible.

* INSERT INTO column list

As with the INSERT statement, specifying the columns and their order in an INSERT INTO statement is advisable.

Column lists make code more intelligible.

* Default values

NULL should be used in most cases instead of an empty string, zero, MinDate, MaxDate

Use of an empty string, 0, or any other value is discouraged.

* Use WITH SCHEMABINDING in view declarations

The hint ‘WITH SCHEMABINDING’ helps avoid problems with underlying objects changing and breaking the view.

* Using BETWEEN for DATETIME ranges.

When using the BETWEEN logical operator with DATETIME values, due to the inclusion of both the date and time values in the range, you never get complete accuracy.

It is better to first use a date function such as DATEPART to convert the DATETIME value into the necessary granularity (such as day, month, year, day of year) to use as a filtering or grouping value.

This can be done by using a persisted computed column to store the required date part as an integer.

* Indexes
  + Don’t create unnecessary indexes

Index maintenance is unnecessary overhead if an index won’t be used.

* + Correct Index Design

Use the most selective columns first, except for columns that can’t be compared with an equality.

For example, a DateTime2(2) column will usually be used in a query using a range (>, <=, etc.). Date columns and similar columns should be listed last in an index.

Turn off Statistics updating and enable page compression.

Use the UNIQUE keyword when possible as this helps the execution engine and makes the index smaller.

Example Index

CREATE UNIQUE NONCLUSTERED INDEX [UX\_SampleTable\_SampleColumn] ON [dbo].[SampleTable]([SampleColumn] ASC, [AuditInfo\_CreatedAt] ASC) INCLUDE (SampleColumnAdditional) WITH (DATA\_COMPRESSION = PAGE, ONLINE = ON) ON INDEXES;

* Schemas

Database objects should use schemas to help with organization and security.

However, the default schema of dbo should be avoided, when possible, in favor of schemas that align with domains or product areas.

Views, stored procedures, tables, user-defined, functions and other database objects can all be put in these schemas.

* Multi-part object names

The compiler can interpret a two-part object name quicker than just one name.

This applies particularly to tables, views, procedures and functions.

The same object name can be used in different schemas, so it pays to make your queries unambiguous.

* Avoid complex conditionals in the WHERE clause.

While it may be tempting to produce queries in routines that have complex conditionals in the WHERE clause, where variables are used for filtering rows.

The query optimizer may use table scans and end up with slow running queries that are hard to understand or refactor.

* Avoid implicit casting

Implicit casting can lead to performance problems. If you know you need to cast, explicitly do it by using CONVERT or CAST.

* Avoid Select \*

Selecting all columns doesn’t perform as well as selecting only the columns that are needed.

Legitimate uses for ‘Select \*’ include

IF EXISTS (SELECT \* FROM … )

SELECT count(\*),

* Avoid Dynamic SQL

This recommendation doesn’t hold when using Entity Framewor.

Don’t explicitly use dynamic SQL when it can be avoided.

* Avoid ODBC functions such as { fn NOW() }, use TSQL versions instead

For example, use GetDate() or GetDateUTC()

* Avoid cursors and loops

It is rare that a cursor should exist for a production system or transactional work.

Usually there is a set-based solution to your problem or a recursive CTE.

* Avoid making all columns nullable

Null has a place and purpose, but don’t use it indiscriminately.

* Avoid subqueries

For performance and maintenance considerations, avoid inline subqueries, derived tables, & scalar subqueries.

Consider using Common Table Expressions (CTE’s) instead.

Consider using a table variable or temp table, depending on the number of rows being used.

* Avoid query hints

Databases grow and change, and your query hint will not always produce the intended result. Instead, make sure the appropriate indexes are in place.

Take the necessary time to ensure that you have optimized the SQL.

* Avoid Multi-statement User Defined Functions (UDFs)

A multi-statement function does not allow SQL Server to use statistics or operate in a set based manner.

Use an inline tabular UDF instead

|  |  |  |
| --- | --- | --- |
| EF Concern? | EF ok | General rules |
|  |  | ·        Always use primary keys and use INT IDENTITY for primary keys |
| x |  | ·        Create foreign keys |
| x |  | ·        Commenting your code |
| x |  | ·        Don’t immediately delete database objects |
| x | x | ·        Ensure all changes are backwards compatible |
| x |  | ·        Ensure all schema and data change scripts can be rerun. |
| n/a | n/a | ·        Protect sensitive data. |
| n/a | n/a | ·        Use a semicolon to terminate SQL statements. |
| x |  | ·        Avoid Implicit conversions |
|  |  |  |
|  |  | Data Types |
| x |  | ·        Use DateTime2(2) |
| x |  | ·        Never use float |
| x |  | ·        Don’t put delimited lists inside of columns. Maintain the atomicity and purpose of the data type. |
| x |  | ·        Bit (Boolean) columns should start with “is” |
| x |  | ·        Use VARCHAR (X) instead or NVARCHAR (X) |
| x |  | ·        Use of VARCHAR (Max) or NVARCHAR (Max) |
| x |  | ·        Be explicit on the size for VARCHAR values |
| x |  | ·        Use VARCHAR (50) when a variable text column is needed |
| x |  | ·        Use a TINYINT or SMALLINT where it makes sense |
|  |  |  |
|  |  | General Naming Conventions |
|  |  |  |
|  | x | ·        Avoid using numbers; if a number is absolutely required, spell it out as a word. |
|  | x | ·        Database and object names should not use keywords |
|  | x | ·        Database names should be singular, not plural and should be named after the context they are persisting |
|  | x | ·        Table names should be singular, not plural |
|  | x | ·        Associative Tables should use the names of their foreign keys |
|  | x | ·        Column Names should provide a clear, concise description of the attribute in mixed case, without prefixes or underscores. |
|  | x | ·        ‘Camel capitalization’ (i.e., mixed case) should be used on all identifiers (for example, FirstName, LogDetail, AcctSysId, LkupCountryCode, @ErrNum, etc). |
|  | x | ·        Don't start stored procedures with "sp" |
|  | x | ·        Objects should be explicitly named |
|  | x | o   Primary Keys |
|  | x | o   Foreign Keys |
|  | x | o   Default Constraints |
|  | x | o   Indexes |
|  | x | ·        Avoid acronyms for object names |
|  | x | ·        Avoid using product-specific names, or names whose meaning is subject to change (Schema, DCN159) |
|  |  |  |
|  |  |  |
|  |  | Recommendations |
|  |  |  |
|  | x | ·        Normalize to Third Normal Form (3NF) |
| x |  | ·        Choose appropriate file groups for objects. |
| x |  | ·        INSERT column list. |
| x |  | ·        INSERT INTO column list |
|  | x | ·        Default values |
|  | x | ·        Use WITH SCHEMABINDING in view declarations |
| x |  | ·        Using BETWEEN for DATETIME ranges. |
|  | x | ·        Indexes |
|  | x | o   Don’t create unnecessary indexes |
|  | x | o   Correct Index Design |
|  | x | ·        Schemas |
| x |  | ·        Multi-part object names |
| x |  | ·        Avoid complex conditionals in the WHERE clause. |
| x |  | ·        Avoid implicit casting |
| x |  | ·        Avoid Select \* |
| x |  | ·        Avoid Dynamic SQL |
| x |  | ·        Avoid ODBC functions such as { fn NOW() }, use TSQL versions instead |
| x |  | ·        Avoid cursors and loops |
| x |  | ·        Avoid making all columns nullable |
| x |  | ·        Avoid subqueries |
| x |  | ·        Avoid query hints |
| x |  | ·        Avoid Multi-statement User Defined Functions (UDFs) |