SQL Code Smells

Once you've done a number of SQL code-reviews, you'll be able to spot signs in the code that indicate all might not be well. These 'code smells' are coding styles that, while not bugs, suggest design problems with the code.

Kent Beck and Massimo Arnoldi seem to have coined the term 'Code Smell' in the 'Once And Only Once' page of www.C2.com, where Kent also said that code 'wants to be simple'. Kent Beck and Martin Fowler expand on the issue of code challenges in their essay 'Bad Smells in Code', published as Chapter 3 of the book *Refactoring: Improving the Design of Existing Code* (ISBN 978-0201485677).

Although there are generic code smells, SQL has its own particular habits that will alert the programmer to the need to refactor code. (For grounding in code smells in C#, see 'Exploring Smelly Code' and 'Code Deodorants for Code Smells' by Nick Harrison.) Plamen Ratchev's wonderful article Ten Common SQL Programming Mistakes' lists some of these code smells along with out-and-out mistakes, but there are more. The use of nested transactions, for example, isn't entirely incorrect, even though the database engine ignores all but the outermost, but their use does flag the possibility the programmer thinks that nested transactions are supported.

If you are moving towards continuous delivery of database applications, you should automate as much as possible the preliminary SQL code-review. It's a lot easier to trawl through your code automatically to pick out problems, than to do so manually. Imagine having something like the classic 'lint' tools used for C, or better still, a tool similar to Jonathan 'Peli' de Halleux's Code Metrics plug-in for .NET Reflector, which finds code smells in .NET code.

One can be a bit defensive about SQL code smells. I will cheerfully write

very long stored procedures, even though they are frowned upon. I'll even use dynamic SQL on occasion. You should use code smells only as an aid. It is fine to 'sign them off' as being inappropriate in certain circumstances. In fact, whole classes of code smells may be irrelevant for a particular database. The use of proprietary SQL, for example, is only a code smell if there is a chance that the database will be ported to another RDBMS. The use of dynamic SQL is a risk only with certain security models. Ultimately, you should rely on your own judgment. As the saying goes, a code smell is a hint of possible bad practice to a pragmatist, but a sure sign of bad practice to a purist.

In describing all these code-smells in a booklet, I've been very constrained on space to describe each code-small. Some code-smells would require a whole article to explain them properly. Fortunately, SQL Server Central and Simple-Talk have, between them, published material on almost all these code smells, so if you get interested, please explore these essential archives of information.

Problems with Database Design

Packing lists, complex data, or other multivariate attributes into a table column

It is permissible to put a list or data document in a column only if it is, from the database perspective, 'atomic', that is, never likely to be shredded into individual values; in other words, it is fine as long as the value remains in

the format in which it started. You should never need to split an 'atomic' value. We can deal with values that contain more than a single item of information: We store strings, after all, and a string is hardly atomic in the sense that it consists of an ordinally significant collection of characters or words. However, the string shouldn't represent a list of values. If you need to parse the value of a column to access values within it, it is likely to need to be normalised, and it will certainly be slow. Occasionally, a data object is too complicated, peripheral, arcane or ephemeral to be worth integrating with the database's normalised structure. It is fair to then take an arm's-length approach and store it as XML, but in this case it will need to be encapsulated by views and table-valued functions so that the SQL Programmer can easily access the contents.

Using inappropriate data types

Although a business may choose to represent a date as a single string of numbers or require codes that mix text with numbers, it is unsatisfactory to store such data in columns that don't match the actual data type. This confuses the presentation of data with its storage. Dates, money, codes and other business data can be represented in a human-readable form, the 'presentation' mode, they can be represented in their storage form, or in their data-interchange form. Storing data in the wrong form as strings leads to major issues with coding, indexing, sorting, and other operations. Put the data into the appropriate 'storage' data type at all times.

Storing the hierarchy structure in the same table as the entities that make up the hierarchy

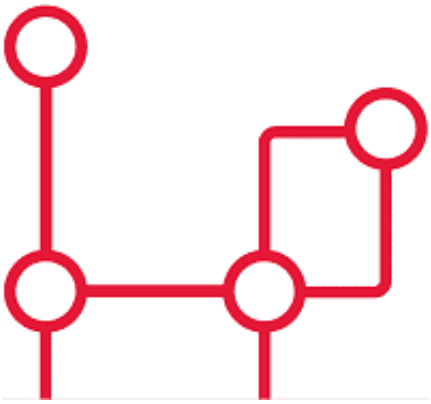
Self-referencing tables seem like an elegant way to represent hierarchies. However, such an approach mixes relationships and values. Real-life hierarchies need more than a parent-child relationship. The 'Closure Table' pattern, where the relationships are held in a table separate from the data, is much more suitable for real-life hierarchies. Also, in real life,

relationships tend have a beginning and an end, and this often needs to be recorded. The HIERARCHYID data type and the common language runtime (CLR) **SqlHierarchyId** class are provided to make tree structures represented by self-referencing tables more efficient, but they are likely to be appropriate for only a minority of applications.

Using an Entity Attribute Value (EAV) model

The use of an EAV model is almost never justified and leads to very tortuous SQL code that is extraordinarily difficult to apply any sort of constraint to. When faced with providing a 'persistence layer' for an application that doesn't understand the nature of the data, use XML instead. That way, you can use XSD to enforce data constraints, create indexes on the data, and use XPath to query specific elements within the XML. It is then, at least, a reliable database, even though it isn't relational!

Using a polymorphic association



Sometimes, one sees table designs which have 'keys' that can reference more than one table, whose identity is usually denoted by a separate column. This is where an entity can relate to one of a number of different entities according to the value in another column that provides the identity of the entity. This sort of relationship cannot be subject to foreign key constraints, and any joins are difficult for the query optimizer to provide good plans for. Also, the logic for the joins is likely to get complicated. Instead, use an intersection table, or if you are attempting an object-

oriented mapping, look at the method by which SQL Server represents the database metadata by creating an 'object' supertype class that all of the individual object types extend. Both these devices give you the flexibility of design that polymorphic associations attempt.

Creating tables as 'God Objects'

'God Tables' are usually the result of an attempt to encapsulate a large part of the data for the business domain in a single wide table. This is usually a normalization error, or rather, a rash and over-ambitious attempt to 'denormalise' the database structure. If you have a table with many columns, it is likely that you have come to grief on the third normal form. It could also be the result of believing, wrongly, that all joins come at great and constant cost. Normally they can be replaced by views or table-valued functions. Indexed views can have maintenance overhead but are greatly superior to denormalisation.

Contrived interfaces

Quite often, the database designer will need to create an interface to provide an abstraction layer, between schemas within a database, between database and ETL process, or between a database and application. You face a choice between uniformity, and simplicity. Overly complicated interfaces, for whatever reason, should never be used where a simpler design would suffice. It is always best to choose simplicity over conformity. Interfaces have to be clearly documented and maintained, let alone understood.

Using command-line and OLE automation to access server-based resources

In designing a database application, there is sometimes functionality that cannot be done purely in SQL, usually when other server-based, or network-based, resources must be accessed. Now that SQL Server's integration with PowerShell is so much more mature, it is better to use that, rather than xp_cmdshell or sp_OACreate (or similar), to access the file system or other server-based resources. This needs some thought and planning: You should also use SQL Agent jobs when possible to schedule your server-related tasks. This requires up-front design to prevent them becoming unmanageable monsters prey to ad-hoc growth.

Problems with Table Design

Using constraints to restrict values in a column

You can use a constraint to restrict the values permitted in a column, but it is usually better to define the values in a separate 'lookup' table and enforce the data restrictions with a foreign key constraint. This makes it much easier to maintain and will also avoid a code-change every time a new value is added to the permitted range, as is the case with constraints.

Not using referential integrity constraints

One way in which SQL Server maintains data integrity is by using constraints to enforce relationships between tables. The query optimizer can also take advantage of these constraints when constructing query

plans. Leaving the constraints off in support of letting the code handle it or avoiding the overhead is a common code smell. It's like forgetting to hit the 'turbo' button.

Enabling NOCHECK on referential integrity constraints

Some scripting engines disable referential integrity during updates. You must ensure that WITH CHECK is enabled or else the constraint is marked as untrusted and therefore won't be used by the optimizer.

BP019- Foreign key is disabled

Using too many or too few indexes

A table in a well-designed database with an appropriate clustered index will have an optimum number of non-clustered indexes, depending on usage. Indexes incur a cost to the system since they must be maintained if data in the table changes. The presence of duplicate indexes and almost-duplicate indexes is a bad sign. So is the presence of unused indexes. SQL Server lets you create completely redundant and totally duplicate indexes. Sometimes this is done in the mistaken belief that the order of 'included' (non-key) columns is significant. It isn't!

Not choosing the most suitable clustered index for a table

You can only have one clustered index on a table, of course, and this choice has a lot of influence on the performance of queries, so you should take care to select wisely. Are you likely to select predominately single values, unsorted or sorted ranges? Are you predominately using one particular index other than your primary key? Is the table experiencing many more reads than writes, with queries that make reference to columns that aren't part of the primary key? Are you typically selecting

ranges within a particular category? Are your WHERE clauses returning many rows? These ways that the table participates in frequently-used queries are likely to be better accommodated by a clustered index.

For your clustered index, you are likely to choose a 'narrow' index which is stored economically because this value has to be held in every index leaf-level pointer. This can be an interesting trade-off because the clustered index key is automatically included in all non-clustered indexes as the row locator so non-clustered indexes will cover queries that need only the non-clustered index key and the clustered index key.

BP001 Index type is not specified

Not explicitly declaring which index is the clustered one.

The primary key is the usual, but not the only, correct choice to be the clustered index. It is seldom wrong to assign a clustered index to a primary key. It is just a problem if your choice of primary key is a 'fat key' without natural order that doesn't work well as a clustered index, or if there is a much better use in the table for that clustered index, such as supporting range scans or avoiding sorts on a frequently-queried candidate key.

A Clustered index shouldn't necessarily be frittered away on a surrogate primary key, based on some meaningless ever-increasing integer. Do not be afraid to use the clustered index for another key if it fits better with the way you query the data, and specifically how the table participates in frequently-used queries.You can only have one clustered index on a table, of course, and this choice has a lot of influence on the performance of queries, so you should take care to select wisely. The primary key is the usual, but not the only, correct choice.

Misusing NULL values

The three-value logic required to handle NULL values can cause a

problems in reporting, computed values and joins. A NULL value means 'unknown', so any sort of mathematics or concatenation will result in an unknown (NULL) value. Table columns should be nullable only when they really need to be. Although it can be useful to signify that the value of a column is unknown or irrelevant for a particular row, NULLs should be permitted only when they're legitimate for the data and application, and fenced around to avoid subsequent problems.

Using temporary tables for very small result sets

Temporary tables can lead to recompiles, which can be costly. Table variables, while not so useful for larger data sets (approximately 15000 rows or more), avoid recompiles and are therefore preferred in smaller data sets. Even with huge tables, they can perform well, but only when a clustered index column is used, and when the nature of the processing makes an index seek very effective.

Table variables behave like other variables in their scoping rules. Once out of scope, they are disposed of without the developer needing to do any explicit cleanup. These are much easier to work with, and pretty secure, and they trigger fewer recompiles in the routines where they're used than if you were to use temporary tables. Table variables require less locking resources as they are 'private' to the batch or process that created them. Transaction rollbacks do not affect them because table variables have limited scope and are not part of the persistent database.

Where the number of rows in the table is moderate, you can even use them in joins as 'Heaps', unindexed. Beware, however, since, as the number of rows increase, joins on Table Variable heaps can easily become saddled by very poor execution plans, and this must be cured either by adding constraints (UNIQUE or PRIMARY KEY) or by adding the OPTION (RECOMPILE) hint if this is impossible. Occasionally, the way that the data is distributed prevents the efficient use of Table Variables, and this will require using a temporary table instead.

See:

1. Table Variables: an empirical approach.
2. Temporary Tables in SQL Server

ST011 – Consider using table variable instead of temporary table

ST012 – Consider using temporary table instead of table variable

Creating a table without specifying a schema

If you're creating tables from a script, they must, like views and routines, always be defined with two-part names. It is possible for different schemas to contain the same table name, and there are some perfectly legitimate reasons for doing this. Don't rely on dbo being the default schema for the login that executes the create script: The default can be changed.

The user of any database is defaulted to the 'dbo' schema, unless explicitly assigned to a different schema. Unless objects are referenced by schema as well as name, they are assumed by the database engine to be in the user's default schema, and if not there, in the dbo schema.

Most tables should have a clustered index

SQL Server storage is built around the clustered index as a fundamental part of the data storage and retrieval engine. The data itself is stored with the clustered key. All this makes having an appropriate clustered index a vital part of database design. The places where a table without a clustered index is preferable are rare; which is why a missing clustered index is a common code smell in database design.

A 'table' without a clustered index is actually a heap, which is a particularly bad idea when its data is usually returned in an aggregated form, or in a sorted order. Paradoxically, though, it can be rather good for implementing a log or a 'staging' table used for bulk inserts, since it is read very infrequently, and there is less overhead in writing to it. A table with a non-clustered index , but without a clustered index can sometimes perform well even though the index has to reference individual rows via a Row Identifier rather than a more meaningful clustered index. The arrangement can be effective for a table that isn't often updated if the table is always accessed by a non-clustered index and there is no good candidate for a clustered index.

Using the same column name in different tables but with different data types

Any programmer will assume a sane database design in which columns with the same name in different tables have the same data type. As a result, they probably won't verify types. Different types is an accident waiting to happen.

Defining a table column without explicitly specifying whether it is nullable

In a CREATE TABLE DDL script, a column definition that has not specified that a column is NULL or NOT NULL is a risk. The default nullability for a database's columns can be altered by the 'ANSI_NULL_DFLT_ON' setting.

Therefore one cannot assume whether a column will default to NULL or NOT NULL. It is safest to specify it in the column definition for noncomputed columns, and it is essential if you need any portability of your table design. Sparse columns must always allow NULL.

BP014 NOT NULL option is not specified in CREATE/DECLARE TABLE statement (registered once per table)

Trying to add a NOT NULL column without default value to a table with data

Adding a NOT NULL column without a DEFAULT value to an existing table with data in it will fail because SQL Server has no way of adding that column to existing rows, because there must be a value in the column.

EI028 Adding NOT NULL column without default value

Creating dated copies of the same table to manage table sizes

Now that SQL Server supports table partitioning, it is far better to use partitions than to create dated tables, such as Invoices2012, Invoices2013, etc. If old data is no longer used, archive the data, store only aggregations, or both.

Problems with Data Types

Using VARCHAR(1), VARCHAR(2), etc.

Columns of a short or fixed length should have a fixed size because variable-length types have a disproportionate storage overhead. For a large table, this could be significant. The narrow a table, the faster it can be accessed. In addition, columns of variable length are stored after all columns of fixed length, which can have performance implications. For short strings, use a fixed length type, such as CHAR, NCHAR, and BINARY.

BP009 Avoid var types of length 1 or 2

SR0009: Avoid using types of variable length that are size 1 or 2

Declaring var type variables without length

An VARCHAR, VARBINARY or NVARCHAR that is declared without an explicit length is shorthand for specifying a length of 1. Is this what you meant or did you do it by accident? Much better and safer to be explicit.

Using deprecated language elements such as the TEXT/NTEXT data types

There is no good reason to use TEXT or NTEXT. They were a first, flawed attempt at BLOB storage and are there only for backward compatibility. Likewise, the WRITETEXT, UPDATETEXT and READTEXT statements are also deprecated. All this complexity has been replaced by the VARCHAR(MAX) and NVARCHAR(MAX) data types, which work with all of SQL Server's string functions.

DEP002 WRITETEXT,UPDATETEXT and READTEXT statements are deprecated.

Using MONEY data type

The MONEY data type confuses the storage of data values with their display, though it clearly suggests, by its name, the sort of data held. Using the DECIMAL data type is almost always better.

Using FLOAT or REAL data types

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 (-1.79E+308 to -2.23E-308, 0 and 2.23E-308 to

1.79E+308, in the case of FLOAT). Any other use needs to be regarded as suspect, and a FLOAT or REAL used as a key or found in an index needs to be investigated. The DECIMAL type is an exact data type and has an impressive range from -10^38+1 through 10^38-1. Although it requires more storage than the FLOAT or REAL types, it is generally a better choice.

Mixing parameter data types in a COALESCE expression

The result of the COALESCE expression (which is shorthand for a CASE statement) is the first non-NULL expression in the list of expressions provided as arguments. Mixing data types can result in errors or data truncation.

Using DATETIME or DATETIME2 when you're concerned only with the date

Even with data storage being so cheap, a saving in a data type adds up and makes comparison and calculation easier. When appropriate, use the DATE or SMALLDATETIME type. Narrow tables perform better and use less resources

Using DATETIME or DATETIME2 when you're merely recording the time of day

Being parsimonious with memory is important for large tables, not only to save space but also to reduce I/O activity during access. When appropriate, use the TIME or SMALLDATETIME type. Queries too are generally simpler on the appropriate data type.

Using sql_variant inappropriately

The sql_variant type is not your typical data type. It stores values from a

number of different data types and is used internally by SQL Server. It is hard to imagine a valid use in a relational database. It cannot be returned to an application via ODBC except as binary data, and it isn't supported in Microsoft Azure SQL Database.

The length of the VARCHAR, VARBINARY and NVARCHAR datatype in a CAST or CONVERT clause wasn't explicitly specified

When you convert a datatype to a varchar, you do not have to specify the length. If you don't do so, SQL Server will use a Varchar length sufficient to hold the string. It is better to specify the length because SQL Server has no idea what length you may subsequently need.

BP008 CAST/CONVERT to var types without length

Storing a duration rather than a point in time

This takes some programmers by surprise. Although it is possible to store a time interval in a table it is not generally a good idea. A time interval is the difference between the start and end of a period of time. You may want to measure this in all sorts of ways, (milliseconds? Quarters?

weeks?) and you may subsequently need to deal with all sorts of queries that have to work out what the status was at a particular time (e.g. how many rooms were booked at a particular point in time). By storing the time period as the start and end date-and-time, you leave your options open. If you store the time interval (in what? Seconds?) and maybe the start DateTime, you make subsequent queries more difficult. It is possible to use a TIME data type if the duration is less than 24 hours, but this is not what the type is intended for, and can be the cause of confusion for the next person who has to maintain your code. They will display very oddly depending on the representation of the time-of-day you use and wrap around every 24 hours!

Using VARCHAR(MAX) or NVARCHAR(MAX) when it isn't necessary

VARCHAR types that specify a number rather than MAX have a finite maximum length and can be stored in-page, whereas MAX types are treated as BLOBS and stored off-page, preventing online re-indexing. Use MAX only when you need more than 8000 bytes (4000 characters for NVARCHAR, 8000 characters for VARCHAR).

Using VARCHAR rather than NVARCHAR for anything that requires internationalisation, such as names or addresses

You can't require everyone to stop using national characters or accents any more. The nineteen-fifties are long gone. Names are likely to have accents in them if spelled properly, and international addresses and language strings will almost certainly have accents and national characters that can't be represented by 8-bit ASCII!

Declaring VARCHAR, VARBINARY and NVARCHAR datatypes without explicit length

An NVARCHAR that is declared without an explicit length is shorthand for specifying a length of 1. Is this what you meant or did you do it by accident? Much better to be explicit.

BP007 Declaring var type variables without length

Problems with expressions

Excessive use of parentheses

Some developers use parentheses even when they aren't necessary, as a safety net when they're not sure of precedence. This makes the code more difficult to maintain and understand.

Using functions such as 'ISNUMERIC' without additional checks

Some functions, such as ISNUMERIC, are there to tell you in very general terms whether a string can be converted to a number without an error. Sadly, it doesn't tell you what kind of number. (Try SELECT isNumeric(','); or SELECT ISNUMERIC('4D177'); for example.) This causes immense confusion. The ISNUMERIC function returns 1 when the input expression evaluates to a valid numeric data type; otherwise it returns 0. The function also returns 1 for some characters that are not numbers, such as plus (+), minus (-), and valid currency symbols such as the dollar sign ($). This is legitimate because these can be converted to numbers, but counter-intuitive. Unfortunately, most programmers want to know whether a number is a valid quantity of money, or a float, or integer. Use a function such as TRY_CAST() and TRY_CONVERT() that is appropriate for the data type whose validity you are testing. E.g. select try_convert(int,'12,345') or select try_convert(float,'5D105')

EI029 Avoid using ISNUMERIC() function

Injudicious use of the LTRIM and RTRIM functions

These don't work as they do in any other computer language. They only trim ASCII space rather than any whitespace character. Use a scalar user-defined function instead.

Using DATALENGTH rather than LEN to find the length of a string.

Although using the DATALENGTH function is valid, it can easily give you the wrong results if you're unaware of the way it works with the CHAR, NCHAR, or NVARCHAR data types.

Not using 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. When developing code, it is usual add clauses on the end of statements, and in these circumstances, semicolons can be a considerable irritation because they trigger etrrors when they become embedded.

Relying on data being implicitly converted between types

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. More often then not, you should explicitly define your conversions to avoid unintentional consequences.

SR0014: Data loss might occur when casting from {Type1} to {Type2}

Using the @@IDENTITY system function

The generation of an IDENTITY value is not transactional, so in some circumstances, **@@IDENTITY** returns the wrong value and not the value from the row you just inserted. This is especially true when using triggers that insert data, depending on when the triggers fire. The **SCOPE_IDENTITY** function is safer because it always relates to the current batch (within the same scope). Also consider using the IDENT_CURRENT function, which returns the last IDENTITY value regardless of session or scope. The **OUTPUT** clause is a better and safer way of capturing identity values.

Usage of @@identity

Using BETWEEN for DATETIME ranges

You never get complete accuracy if you specify dates when using the

BETWEEN logical operator with DATETIME values, due to the inclusion of both the date and time values in the range. 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) and store this in a column (or columns), then indexed and used 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, or via a trigger.

Using SELECT * in a batch

Although there is a legitimate use in a batch for IF EXISTS (SELECT * FROM ...) or SELECT count(*), any other use is vulnerable to changes in column names or order. SELECT * was designed for interactive use, not as part of a batch. It assumes certain columns in a particular order, which may not last. Also, results should always consist of just the columns you need. Plus, requesting more columns from the database than are used by the application results in excess database I/O and network traffic, leading to slow application response and unhappy users.

Asterisk in select list

INSERT without column list

The INSERT statement need not have a column list, but omitting it assumes certain columns in a particular order. It likely to cause errors if the table in to which the inserts will be made is changed, particularly with table variables where insertions are not checked. Column lists also make code more intelligible

See: SR0001: Avoid SELECT * in a batch, stored procedures, views, and table-valued functions

ORDER BY clause with constants

The use of constants in the ORDER BY is deprecated for removal in the future. They make ORDER BY statements more difficult to understand.

BP002 ORDER BY clause with constants

Difficulties with Query Syntax

Creating UberQueries (God-like Queries)

Always avoid overweight queries (e.g., a single query with four inner joins, eight left joins, four derived tables, ten subqueries, eight clustered GUIDs, two UDFs and six case statements).

Nesting views as if they were Russian dolls

Views are important for abstracting the base tables. However, they do not lend themselves to being deeply nested. Views that reference views that reference views that reference views perform poorly and are difficult to maintain. Recommendations vary but I suggest that views relate directly to base tables where possible.

Joins between large views

Views are like tables in their behaviour, but they can't be indexed to support joins. When large views participate in joins, you never get good performance. Instead, either create a view that joins the appropriately indexed base tables, or create indexed temporary tables to contain the filtered rows from the views you wish to 'join'.

Using the old Sybase JOIN syntax

The deprecated syntax (which includes defining the join condition in the WHERE clause) is not standard SQL and is more difficult to inspect and maintain. Parts of this syntax are completely unsupported in SQL Server 2012 or higher.

The “old style” Microsoft/Sybase JOIN style for SQL, which uses the =* and *= syntax, has been deprecated and is no longer used. Queries that use this syntax will fail when the database engine level is 10 (SQL Server 2008) or later (compatibility level 100). The ANSI-89 table citation list (FROM tableA, tableB) is still ISO standard for INNER JOINs only. Neither of these styles are worth using. It is always better to specify the type of join you require, INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER and CROSS, which has been standard since ANSI SQL-92 was published. While you can choose any supported JOIN style, without affecting the query plan used by SQL Server, using the ANSI-standard syntax will make your code easier to understand, more consistent, and portable to other relational database systems.

See: old-style join syntax

DEP017 NON-ANSI join (**== or =**) is used

ST001 Old-style join is used (...from table1,table2...)

Using correlated subqueries instead of a join

Correlated subqueries, queries that run against each returned by the main query, sometimes seem an intuitive approach, but they are merely disguised cursors needed only in exceptional circumstances. Window functions will usually perform the same operations much faster. Most usages of correlated subqueries are accidental and can be replaced with a much simpler and faster JOIN query.

Using SELECT rather than SET to assign values to variables

Using a SELECT statement to assign variable values is not ANSI standard SQL and can result in unexpected results. If you try to assign the result from a single query to a scalar variable, and the query produces several rows, a SELECT statement will return no errors, whereas a SET statement will. On the other hand, if the query returns no rows, the SET statement will assign a NULL to the variable, whereas SELECT will leave the current value of the variable intact.

Using scalar user-defined functions (UDFs) for data lookups as a poor man's join.

It is true that SQL Server provides a number of system functions to simplify joins when accessing metadata, but these are heavily optimised. Using user-defined functions in the same way will lead to very slow queries since they perform much like correlated subqueries.

Not using two-part object names for object references

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 name can be used in different schemas, so it pays to make your

queries unambiguous.

The complete name of any schema-based database object consists of up to four identifiers: the server name, database name, schema name, and object name. Only if you are calling a remote stored procedure would you need a fully qualified name consisting of all four identifiers. If you are calling a procedure in another database, you obviously need its database identifier in the name. Within a database, you only need the object name itself so long as the procedure is in the same schema. By specifying the schema, the database engine needs less searching to identify it. Even system stored procedures should be qualified with the 'sys' schema name. When creating a stored procedure as well, it is a good habit to always specify the parent schema.

It is a very good idea to get into the habit of qualifying the names of procedures with their schema. It is not only makes your code more resilient and maintainable, but as Microsoft introduces new features that use schemas, such as auditing mechanisms, you code contains no ambiguities that could cause problems.

See: procedures that lack schema-qualification

PE001/PE002 Schema name for procedure is not specified/Schema name for table or view is not specified

Using INSERT INTO without specifying the columns and their order

Not specifying column names is fine for interactive work, but if you write code that relies on the hope that nothing will ever change, then refactoring could prove to be impossible. It is much better to trigger an error now than to risk corrupted results after the SQL code has changed. Column lists also make code more intelligible

BP004 INSERT without column list

Using full outer joins unnecessarily.

It is rare to require both matched and unmatched rows from the two joined tables, especially if you filter out the unmatched rows in the WHERE clause. If what you really need is an inner join, left outer join or right outer join, then use one of those. If you want all rows from both tables, use a cross join.

Including complex conditionals in the WHERE clause

It is tempting to produce queries in routines that have complex conditionals in the WHERE clause where variables are used for filtering rows. Usually this is done so that a range of filtering conditions can be passed as parameters to a stored procedure or tale-valued function. If a variable is set to NULL instead of a search term, the OR logic or a COALESCE disables the condition. If this is used in a routine, very different queries are performed according to the combination of parameters used or set to null. As a result, the query optimizer must use table scans, and you end up with slow-running queries that are hard to understand or refactor. This is a variety of UberQuery which is usually found when some complex processing is required to achieve the final result from the filtered rows.

Mixing data types in joins or WHERE clauses

If you compare or join columns that have different data types, you rely on implicit conversions, which result in a poor execution plans that use table scans. This approach can also lead to errors because no constraints are in place to ensure the data is the correct type.

Assuming that SELECT statements all have roughly the same execution time

Few programmers admit to this superstition, but it is apparent by the strong preference for hugely long SELECT statements (sometimes called UberQueries). A simple SELECT statement runs in just a few milliseconds. A process runs faster if the individual SQL queries are clear enough to be easily processed by the query optimizer. Otherwise, you will get a poor query plan that performs slowly and won't scale.

Not handling NULL values in nullable columns

Generally, it is wise to explicitly handle NULLs in nullable columns, by using COALESCE to provide a default value. This is especially true when calculating or concatenating the results. (A NULL in part of a concatenated string, for example, will propagate to the entire string. Names and addresses are prone to this sort of error.)

SR0007: Use ISNULL(column, default_value) on nullable columns in expressions

Referencing an unindexed column within the IN predicate of a WHERE clause

A WHERE clause that references an unindexed column in the IN predicate causes a table scan and is therefore likely to run far more slowly than necessary.

See: SR0004: Avoid using columns that do not have indexes as test expressions in IN predicates

Using LIKE in a WHERE clause with an initial wildcard character

An index cannot be used to find matches that start with a wildcard character ('%' or '_'), so queries are unlikely to run well on large tables because they'll require table scans.

See: SR0005: Avoid using patterns that start with a '%' in LIKE predicates

Using a predicate or join column as a parameter for a user-defined function

The query optimizer will not be able to generate a reasonable query plan if the columns in a predicate or join are included as function parameters. The optimizer needs to be able to make a reasonable estimate of the number of rows in an operation in order to effectively run a SQL statement and cannot do so when functions are used on predicate or join columns.

Supplying object names without specifying the schema

Object names need only to be unique within a schema. However, when referencing an object in a SELECT, UPDATE, DELETE, MERGE or EXECUTE statements or when calling the OBJECT_ID function, the database engine can find the objects more easily found if the names are qualified with the schema name.

PE001/PE002 Schema name for procedure is not specified/Schema

name for table or view is not specified

Using '== NULL' or '<> NULL' to filter a nullable column for NULLs

An expression that returns a NULL as either the left value (Lvalue) or right value (Rvalue) will always evaluate to NULL. Use IS NULL or IS NOT NULL.

BP011 NULL comparison or addition/substring

Not using NOCOUNT ON in stored procedures and triggers

Unless you need to return messages that give you the row count of each statement, you should specify the NOCOUNT ON option to explicitly turn off this feature. This option is not likely to be a significant performance factor one way or the other. Whenever you execute a query, a short message is returned to the client with the number of rows that are affected by that T-SQL statement. When you use SET NOCOUNT ON, this message is not sent. This can improve performance by reducing network traffic slightly. It is best to use SET NOCOUNT ON in SQL Server triggers and stored procedures, unless one or more of the applications using the stored procedures require it to be OFF, because they are reading the value in the message.

The best approach, generally, is to prevent rowcount messages being sent, unless they are required, but the tricky part is accommodating legacy applications that use, and often misuse, these messages. Additionally, sending these messages can sometimes be a problem for asynchronous processing of procedures by intermediate layers of database applications such as ORMs. The rowcount messages are much slower to be transmitted to the client than the result of the stored procedure, and this can block threads.

See the SET NOCOUNT problem

PE009 No SET NOCOUNT ON before DML

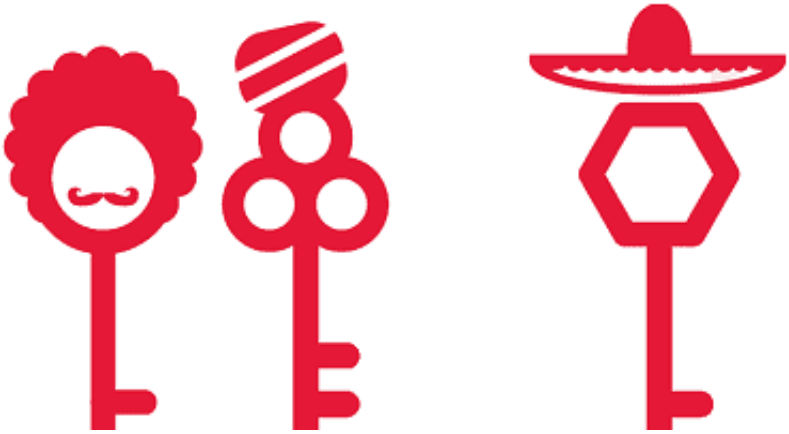
Using the NOT IN predicate in the WHERE clause

You're queries will often perform poorly if your WHERE clause includes a NOT IN predicate that references a subquery. The optimizer will likely have to use a table scan instead of an index seek, even if there is a suitable index. You can almost always get a better-performing query by using a left outer join and checking for a NULL in a suitable NOT NULLable column on the right-hand side.

Defining foreign keys without a supporting index

Unlike some relational database management systems (RDBMSs), SQL Server does not automatically index a foreign key column, even though an index will likely be needed. It is left to the implementers of the RDBMS as to whether an index is automatically created to support a foreign key constraint. SQL Server chooses not to do so, probably because, if the referenced table is a lookup table with just a few values, an index isn't useful. SQL Server also does not mandate a NOT NULL constraint on the foreign key, perhaps to allow rows that aren't related to the referenced table.

Even if you're not joining the two tables via the primary and foreign keys, with a table of any size, an index is usually necessary to check changes to PRIMARY KEY constraints against referencing FOREIGN KEY constraints in other tables to verify that changes to the primary key are reflected in the foreign key



Using a non-SARGable (Search ARGument..able) expression in a WHERE clause

In the WHERE clause of a query it is good to avoid having a column reference or variable embedded within an expression, or used as a parameter of a function. A column reference or variable is best used as a single element on one side of the comparison operator , otherwise it will most probably trigger a table scan, which is expensive in a table of any size.

See: SR0006: Move a column reference to one side of a comparison operator to use a column index

Including a deterministic function in a WHERE clause

If the value of the function does not depend on the data row that you wish to select, then it is better to put its value in a variable before the SELECT query and use the variable instead.

See: SR0015: Extract deterministic function calls from WHERE predicates PE017 Incorrect usage of const UDF

Using an unverified scalar user-defined function as a constant.

The incorrect use of a non-schema bound scalar UDF, as a global

database constant, is a major performance problem and must be winkled out of any production code. The problem arises because SQL Server doesn't trust non-schema verified scalar functions as being precise and deterministic, and so chooses the safest, though slowest, option when executing them. It's a slightly insidious problem because it doesn't really show its full significance in the execution plan, though an Extended Events session will reveal what is really going on.

See: Misuse of the scalar user-defined function as a constant (PE017)

PE017 Incorrect usage of const UDF

Using SELECT DISTINCT to mask a join problem

It is tempting to use SELECT DISTINCT to eliminate duplicate rows in a join. However, it's much better to determine why rows are being duplicated and fix the problem.

Using NOT IN with an expression that allows null values

If you are using a NOT IN predicate to select only those rows that match the results returned by a subquery or expression, make sure there are no NULL values in those results. Otherwise, your outer query won't return the results you expect. In the case of both IN and NOT IN, it is better to use an appropriate outer join.

PE019 Consider using NOT EXISTS instead of NOT IN (subquery)

See Consider using NOT EXISTS instead of NOT IN (subquery)

A DELETE statement has omitted that WHERE clause, which would delete the whole table

It is very easy to delete an entire table when you mean to delete just one

or more rows. There are occasionally good reasons for using DELETE to clear a table entirely. If you need to clear a table that is involved in replication or log shipping, or a table that has foreign key constraints that reference it, you have no choice. Otherwise, it is more usual to use the TRUNCATE TABLE statement that quickly deletes all records in a table by deallocating the data pages used by the table. The DELETE statement logs the deletions, aand establishes locks whereas the TRUNCATE statement only uses the transaction log to record the page deallocation. It also resets the IDENTITY back to the SEED, and the deallocated pages are recycled.

BP017 DELETE statement without WHERE clause

An UPDATE statement has omitted the WHERE clause, which would update every row in the table

It is very easy to update an entire table, over-writing the data in it, when you mean to update just one or more rows. At the console, Delete or Update statements should also be in a transaction so you can check the result before committing.

BP018 UPDATE statement without WHERE clause

Using a Common Table Expression (CTE) unnecessarily

CTEs are there to make SQL Statements clearer. They specify a temporary named result set, derived from a simple query and defined within the execution scope of a single SELECT, INSERT, UPDATE, or DELETE statement. They are convenient to use when an intermediate temporary result needs to be used more than once as a table-source within an expression. It is also useful for recursive statements because a common table expression can include references to itself. To use a CTE in other circumstances doesn't provide a performance benefit even though the

code may look neater. It is only useful if it makes the query easier to read and understand, and it compiles to the same query plan as the ordinary query. Otherwise it is unnecessary and may even provide extra overhead and slow down the performance.

Problems with naming

Excessively long or short identifiers

Identifiers should help to make SQL readable as if it were English. Short names like t1 or gh might make typing easier but can cause errors and don't help teamwork. At the same time, names should be names and not long explanations. Remember that these are names, not documentation. Long names can be frustrating to the person using SQL interactively, unless that person is using SQL Prompt or some other IntelliSense system, through you can't rely on it.

Using sp_ prefixes for stored procedures

The sp_ prefix has a special meaning in SQL Server and doesn't mean 'stored procedure' but 'special', which tells the database engine to first search the master database for the object.

EI024 Stored procedure name starts with sp_

'Tibbling' SQL Server objects with Reverse-Hungarian prefixes such as tbl_, vw_, pk_, fn_, and usp_

SQL names don't need prefixes because there isn't any ambiguity about what they refer to. 'Tibbling' is a habit that came from databases imported from Microsoft Access.

Using reserved words in names

Using reserved words makes code more difficult to read, can cause problems to code formatters, and can cause errors when writing code.

SR0012: Avoid using reserved words for type names

Including special characters in object names

SQL Server supports special character in object names for backward compatibility with older databases such as Microsoft Access, but using these characters in newly created databases causes more problems than they're worth. Special characters requires brackets (or double quotations) around the object name, makes code difficult to read, and makes the object more difficult to reference. Avoid particularly using any whitespace characters, square brackets or either double or single quotation marks as part of the object name.

R0011: Avoid using special characters in object names

Using numbers in table names

It should always serve as a warning to see tables named Year1, Year2, Year3 or so on, or even worse, automatically generated names such as tbl3546 or 567Accounts. If the name of the table doesn't describe the entity, there is a design problem

See: SR0011: Avoid using special characters in object names

Using square brackets unnecessarily for object names

If object names are valid and not reserved words, there is no need to use square brackets. Using invalid characters in object names is a code smell anyway, so there is little point in using them. If you can't avoid brackets, use them only for invalid names.

SR0011: Avoid using special characters in object names

Using system-generated object names, particularly for constraints

This tends to happen with primary keys and foreign keys if, in the data definition language (DDL), you don't supply the constraint name. Auto-generated names are difficult to type and easily confused, and they tend to confuse SQL comparison tools. When installing SharePoint via the GUI, the database names get GUID suffixes, making them very difficult to deal with.

Problems with routines

Including few or no comments

Being antisocial is no excuse. Either is being in a hurry. Your scripts should be filled with relevant comments, 30% at a minimum. This is not just to help your colleagues, but also to help you-in-the-future. What seems obvious today will be as clear as mud tomorrow, unless you comment your code properly. In a routine, comments should include intro text in the header as well as examples of usage.

You have a stored procedure that does not return a result code

When you use the EXECUTE command to execute a stored procedure, or call the stored procedure from an application, an integer is returned that can be assigned to a variable. It is generally used to communicate the success of the operation. It provides a very useful way of reacting to problems in a process and can make a batch process far less convoluted

BP016 Return without result code

Excessively 'overloading' routines

Stored procedures and functions are compiled with query plans. If your routine includes multiple queries and you use a parameter to determine which query to run, the query optimizer cannot come up with an efficient execution plan. Instead, break the code into a series of procedures with one 'wrapper' procedure that determines which of the others to run.

Creating routines (especially stored procedures) as 'God Routines' or 'UberProcs'

Occasionally, long routines provide the most efficient way to execute a process, but occasionally they just grow like algae as functionality is added. They are difficult to maintain and likely to be slow. Beware particularly of those with several exit points and different types of result set.

Creating stored procedures that return more than one result set

Although applications can use stored procedures that return multiple result sets, the results cannot be accessed within SQL. Although they can be used by the application via ODBC, the order of tables will be significant and changing the order of the result sets in a refactoring will then break the application in ways that may not even cause an error, and will be difficult to test automatically from within SQL.

Creating a Multi-statement table-valued function, or a scalar function when an inline function is possible

Inline table-valued Functions run much quicker than a Multi-statement table-valued function, and are also quicker than scalar functions. Obviously, they are only possible where a process can be resolved into a single query.

Too many parameters in stored procedures or functions

The general consensus is that a lot of parameters can make a routine unwieldy and prone to errors. You can use table-valued parameters (TVPs) or XML parameters when it is essential to pass data structures or lists into a routine.

Duplicated code

This is a generic code smell. If you discover an error in code that has been duplicated, the error needs to be fixed in several places. Although duplication of code In SQL is often a code smell, it is not necessarily so. Duplication is sometimes done intentionally where large result sets are involved because generic routines frequently don't perform well. Sometimes quite similar queries require very different execution plans. There is often a trade-off between structure and performance, but sometimes the performance issue is exaggerated. Although you can get a

performance hit from using functions and procedures to prevent duplication by encapsulating functionality, it isn't often enough to warrant deliberate duplication of code

High cyclomatic complexity

Sometimes it is important to have long procedures, maybe with many code routes. However, if a high proportion of your procedures or functions are excessively complex, you'll likely have trouble identifying the atomic processes within your application. A high average cyclomatic complexity in routines is a good sign of technical debt.

Using an ORDER BY clause within a view

You cannot use the ORDER BY clause without the TOP clause or the OFFSET ... FETCH clause in views (or inline functions, derived tables, or subqueries). Even if you resort to using the TOP 100% trick, the resulting order isn't guaranteed. Specify the ORDER BY clause in the query that calls the view.

EI030 Order by in view or single-statement TVF

Unnecessarily using stored procedures or multiline table-valued functions where a view is sufficient

Stored procedures are not designed for delivering result sets. You can use stored procedures as such with INSERT ... EXEC, but you can't nest INSERT ... EXEC so you'll soon run into problems. If you do not need to provide input parameters, then use views, otherwise use inline table valued functions.

Using Cursors

SQL Server originally supported cursors to more easily port dBase II

applications to SQL Server, but even then, you can sometimes use a WHILE loop as an effective substitute. However, modern versions of SQL Server provide window functions and the CROSS/OUTER APPLY syntax to cope with most of the traditional valid uses of the cursor.

You have not explicitly defined the scope of a cursor

When you define a cursor with the DECLARE CURSOR statement you can, and should, define the scope of the cursor name. GLOBAL means that the cursor name should be global to the connection. LOCAL specifies that the cursor name is LOCAL to the stored procedure, trigger, or batch containing the DECLARE CURSOR statement.

BP015 Scope of cursor (LOCAL/GLOBAL) is not specified

Overusing CLR routines

There are many valid uses of CLR routines, but they are often suggested as a way to pass data between stored procedures or to get rid of performance problems. Because of the maintenance overhead, added complexity, and deployment issues associated with CLR routines, it is best to use them only after all SQL-based solutions to a problem have been found wanting or when you cannot use SQL to complete a task.

Excessive use of the WHILE loop

A WHILE loop is really a type of cursor. Although a WHILE loop can be useful for several inherently procedural tasks, you can usually find a better relational way of achieving the same results. The database engine is heavily optimised to perform set-based operations rapidly. Don't fight it!

Relying on the INSERT … EXEC statement

In a stored procedure, you must use an INSERT … EXEC statement to retrieve data via another stored procedure and insert it into the table targeted by the first procedure. However, you cannot nest this type of statement. In addition, if the referenced stored procedure changes, it can case the first procedure to generate an error.

Executing stored procedure without getting result

If a stored procedure provides one or more result, the rows will be sent to the client. For large result sets the stored procedure execution will not continue to the next statement until the result set has been completely sent to the client. For small result sets the results will be spooled for return to the client and execution will continue. Within a batch, a stored procedure that returns a result should be called with INSERT …EXECUTE syntax.

EI025 Executing stored procedure without getting result

Forgetting to set an output variable

The values of the output parameters must be explicitly set in all code paths, otherwise the value of the output variable will be NULL. This can result in the accidental propagation of NULL values. Good defensive coding requires that you initialize the output parameters to a default value at the start of the procedure body.

See SR0013: Output parameter (parameter) is not populated in all code

paths

Specifying parameters by order rather by assignment, where there are more than four parameters

When calling a stored procedure, it is generally better to pass in parameters by assignment rather than just relying on the order in which the parameters are defined within the procedure. This makes the code easier to understand and maintain. As with all rules, there are exceptions: It doesn't really become a problem when there are less than a handful of parameters. Also, natively compiled procedures work fastest by passing in parameters by order.

EI018 Missing parameter(s) name in procedure call

Try to avoid using hardcoded references to other databases.

There is nothing wrong in executing procedures in other databases, but it is better to avoid hard-coding these references and use synonyms instead.

EI016 Reference to procedure in other database

Use of a Hardcoded current database name in a procedure call

You only need to specify the database when calling a procedure in a different database. It is better to avoid using hardcoded references to the current database as this causes problems if you later do the inconceivable by changing the databases name or cut-and-pasting a routine. There is no performance advantage whatsoever in specifying the current database if the procedure is in the same database.

EI017 Hardcoded current database name in procedure call

Setting the QUOTED_IDENTIFIER or ANSI_NULLS options inside stored procedures

Stored procedures use the SET settings specified at execute time, except for SET ANSI_NULLS and SET QUOTED_IDENTIFIER. Stored procedures that specify either the SET ANSI_NULLS or SET QUOTED_IDENTIFIER use the setting specified at stored procedure creation time. If used inside a stored procedure, any such SET command is ignored

MI008 QUOTED_IDENTIFIERS option inside stored procedure, trigger or function

Creating a routine with ANSI_NULLS or QUOTED_IDENTIFIER options set to OFF.

At the time the routine is created (parse time), both options should normally be set to ON. They are ignored on execution. The reason for keeping Quoted Identifiers ON is that it is necessary when you are creating or changing indexes on computed columns or indexed views. If set to OFF, then CREATE, UPDATE, INSERT, and DELETE statements on tables with indexes on computed columns or indexed views will fail. SET QUOTED_IDENTIFIER must be ON when you are creating a filtered index or when you invoke XML data type methods. ANSI_NULLS will eventually be set to ON and this ISO compliant treatment of NULLS will not be switchable to OFF.

DEP028 The SQL module was created with ANSI_NULLS and/or QUOTED_IDENTIFIER options set to OFF.

Updating a primary key column

Updating a primary key column is not by itself always bad in moderation.

However, the update does come with considerable overhead when maintaining referential integrity. In addition, if the primary key is also a clustered index key, the update generates more overhead in order to maintain the integrity of the table.

Overusing hints to force a particular behaviour in joins

Hints do not take into account the changing number of rows in the tables or the changing distribution of the data between the tables. The query optimizer is generally smarter than you, and a lot more patient.

Using the CHARINDEX function in a WHERE Clause

Avoid using CHARINDEX in a WHERE clause to match strings if you can use LIKE (without a leading wildcard expression) to achieve the same results.

Using the NOLOCK hint

Avoid using the NOLOCK hint. It is much better and safer to specify the correct isolation level instead. To use NOLOCK, you would need to be very confident that your code is safe from the possible problems that the other isolation levels protect against. The NOLOCK hint forces the query to use a read uncommitted isolation level, which can result in dirty reads, non-repeatable reads and phantom reads. In certain circumstances, you can sacrifice referential integrity and end up with missing rows or duplicate reads of the same row.

Using a WAITFOR DELAY/TIME statement in a routine or batch

SQL routines or batches are not designed to include artificial delays. If many WAITFOR statements are specified on the same server, too many

threads can be tied up waiting. Also, including WAITFOR will delay the completion of the SQL Server process and can result in a timeout message in the application. Sometimes, a transaction that is the victim of a deadlock can be re-executed after a very short delay, and you'll find a WAIT used for this, which is legitimate.

MI007 WAIT FOR DELAY/TIME used

Using SET ROWCOUNT to specify how many rows should be returned

We had to use this option until the TOP clause (with ORDER BY) was implemented. The TOP option is much easier for the query optimizer.

DEP014 SET ROWCOUNT option is deprecated.

Using TOP 100 PERCENT in views, inline functions, derived tables, subqueries, and common table expressions (CTEs).

This is usually a reflex action to seeing the error 'The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and common table expressions, unless TOP or FOR XML is also specified'. The message is usually a result of your ORDER BY clause being included in the wrong statement. You should include it only in the outermost query.

```
CREATE FUNCTION dbo.CurrencyTable(@Region VARCHAR(20) = '%')

--returns the currency for the region, supports wildcards

--SELECT * FROM dbo.CurrencyTable(DEFAULT) returns all

--SELECT * FROM dbo.CurrencyTable('%Slov%')

RETURNS TABLE

WITH SCHEMABINDING
```

```
AS

RETURN

  (

  SELECT TOP 100 PERCENT CountryRegion.Name AS country, Currency.Name
AS currency

    FROM Person.CountryRegion

      INNER JOIN Sales.CountryRegionCurrency

        ON CountryRegion.CountryRegionCode =
CountryRegionCurrency.CountryRegionCode

      INNER JOIN Sales.Currency

        ON CountryRegionCurrency.CurrencyCode = Currency.CurrencyCode

    WHERE CountryRegion.Name LIKE @Region

    ORDER BY Currency.Name

  );

);
```

Not specifying the Schema name for a procedure

Usually, performance is slightly better if you specify the schema, but in certain cases, you need versions of the same stored procedure to be different depending on the user role. You can put different SPs of the same name in different schemas. You then need to specify the stored procedure without the schema because sql server will then choose the stored procedure from the schema associated with the role of the user.

Duplicating names of objects of different types

Although it is sometimes necessary to use the same name for the same

type of object in different schemas, it is never necessary to do it for different object types and it can be very confusing. You would never want a SalesStaff table and SalesStaff view and SalesStaff stored procedure.

Using WHILE (not done) loops without an error exit

WHILE loops must always have an error exit. The condition that you set in the WHILE statement may remain true even if the loop is spinning on an error. You can create a deadlock by running a query that includes a WAITFOR statement within a transaction that also holds locks to prevent changes to the rowset that the WAITFOR statement is trying to access.

Using a PRINT statement or statement that returns a result in a trigger

Triggers are designed for enforcing data rules, not for returning data or information. Developers often embed PRINT statements in triggers during development to provide a crude idea of how the code is progressing, but the statements need to be removed or commented out before the code is promoted beyond development.

PE011 PRINT statement is used in trigger

SELECT statement in trigger that returns data to the client

Although it is possible to do, it is unwise. A trigger should never return data to a client. It is possible to place a SELECT statement in a trigger but it serves no practical useful purpose, and can have unexpected effects. A trigger behaves much like a stored procedure in that, when the trigger fires, results can be returned to the calling application. This requires special handling because these returned results would have to be handled in some way, and this would have to be written into every application in which modifications to the trigger table are allowed.

BP003 SELECT in trigger

Using TOP without ORDER BY

Using TOP without an ORDER BY clause in a SELECT statement is meaningless and cannot be guaranteed to give consistent results. because asking for the TOP 10 rows implies a certain order, and tables have no implicit logical order.

BP006 TOP without ORDER BY

Using a CASE statement without the ELSE clause

Always specify a default option even if you believe that it is impossible for that condition to happen. Someone might change the logic, or you could be wrong in thinking a particular outcome is impossible.

BP012 CASE without ELSE

Using EXECUTE(string)

Don't use EXEC to run dynamic SQL. It is there only for backward compatibility and is a commonly used vector for SQL injection. Use sp_executesql instead because it allows parameter substitutions for both inputs and outputs and also because the execution plan that

sp_executesql produces is more likely to be reused.

BP013 EXECUTE(string) is used

Using the GROUP BY ALL <column>, GROUP BY <number>, COMPUTE, or COMPUTE BY clause

The GROUP BY ALL <column> clause and the GROUP BY <number> clause are deprecated. There are other ways to perform these operations using the standard GROUP BY and GROUPING syntax. The COMPUTE and COMPUTE BY operations were devised for printed summary results. The ROLLUP and CUBE clauses are a better alternative.

Using numbers in the ORDER BY clause to specify column order

It is certainly possible to specify nonnegative integers to represent the columns in an ORDER BY clause, based on how those columns appear in the select list, but this approach makes it difficult to understand the code at a glance and can lead to unexpected consequences when you forget you've done it and alter the order of the columns in the select list.

DEP003 GROUP BY ALL clause is deprecated.

Using unnecessary three-part and four-part column references in a select list

Sometimes, when a table is referenced in another database or server, programmers believe that the two or three-part table name needs to be applied to the columns. This is unnecessary and meaningless. Just the table name is required for the columns. Three-part column names might be necessary in a join if you have duplicate table names, with duplicate column names, in different schemas, in which case, you ought to be using aliases. The same goes for cross-database joins.

DEP026 Three-part and four-part column references in SELECT list are deprecated.

Using RANGE rather than ROWS in SQL Server 2012

The implementation of the RANGE option in a window frame ORDER BY clause is inadequate for any serious use. Stick to the ROWS option whenever possible and try to avoid ordering without framing.

Doing complex error-handling in a transaction before the ROLLBACK command

The database engine releases locks only when the transaction is rolled back or committed. It is unwise to delay this because other processes may be forced to wait. Do any complex error handling after the ROLLBACK command wherever possible.

Use of BEGIN TRANSACTION without ROLLBACK TRANSACTION

ROLLBACK TRANSACTION rolls back a transaction to the beginning of it, or to a savepoint inside the transaction. You don't need a ROLLBACK TRANSACTION statement within a transaction, but if there isn't one, then it may be a sign that error handling has not been refined to production standards

EI019 BEGIN TRANSACTION without ROLLBACK TRANSACTION

Use of ROLLBACK TRANSACTION without BEGIN TRANSACTION

It is possible to have a ROLLBACK TRANSACTION within a block where there is no explicit transaction. This will trigger an error if the code is executed outside a transaction, and suggests that transactions are being

held open unnecessarily.

EI020 ROLLBACK TRANSACTION without BEGIN TRANSACTION

Not defining a default value for a SELECT assignment to a variable

If an assignment is made to a variable within a SELECT … FROM statement and no result is returned, that variable will retain its current value. If no rows are returned, the variable assignment should be explicit, so you should initialise the variable with a default value.

Not defining a default value for a SET assignment that is the result of a query

If a variable's SET assignment is based on a query result and the query returns no rows, the variable is set to NULL. In this case, you should assign a default value to the variable unless you want it to be NULL.

The value of a nullable column is not checked for NULLs when used in an expression

If you are using a nullable column in an expression, you should use a COALESCE or CASE expression or use the ISNULL(*column, default_value*) function to first verify whether the value is NULL.

Using the NULLIF expression

The NULLIF expression compares two expressions and returns the first one if the two are not equal. If the expressions are equal then NULLIF returns a NULL value of the data type of the first expression. NULLIF is syntactic sugar. Use the CASE statement instead so that ordinary folks can understand what you're trying to do. The two are treated identically.

Not putting all the DDL statements at the beginning of the batch

Don't mix data manipulation language (DML) statements with data definition language (DDL_ statements. Instead, put all the DDL statements at the beginning of your procedures or batches.

PE010 Interleaving DDL and DML in stored procedure/trigger.

Using meaningless aliases for tables (e.g., a, b, c, d, e)

Aliases aren't actually meant to cut down on the typing but rather to make your code clearer. To use single characters is antisocial.

Variable type is not fully compatible with procedure parameter type

A parameter passed to a procedure or function must be of a type that can be cast into the variable datatype declared for that parameter in the body of the routine. It should be of exactly the same type so as to avoid the extra processing to resolve an implicit conversion.

EI001 Incompatible variable type for procedure call

Literal type is not fully compatible with procedure parameter type

A parameter passed to a procedure can be a literal (e.g. 1,'03 jun 2017' or 'hello world') but it must be possible to cast it unambiguously to the variable datatype declared for that parameter in the body of the routine.

EI002 Incompatible literal type for procedure call

Subquery may return more than one row

A subquery can only be scalar, meaning that it can return just one value. Even if you correctly place just one expression in your select list, you must also ensure that just one row is returned. TOP 1 can be used if there is an ORDER BY clause

EI003 Non-scalar subquery in place of a scalar

A named parameter is not found in parameter list of a procedure

Parameters can be passed by position in a comma-delimited list, or by name, where order position isn't required. Any parameters that are specified by name must have the name identical to the definition for that procedure

EI004 Extra parameter passed

Use of the position notation after the named notation for parameters when calling a procedure

Parameters can be passed by position in a comma-delimited list, or by name, but it is a bad idea to mix the two methods even when it is possible. If a parameter has a default value assigned to it, it can be left out of the parameter list, and it is difficult to check whether the values you supply are for the parameters you intend.

EI005 Unnamed call after named call

Parameter is not passed to a procedure and no default is provided.

With procedures and functions, parameters can be assigned default values that are used when a value isn't passed for that parameter when calling the procedure. However, if a parameter isn't assigned a value and

there is no default provided it is an error. If you don't want to provide a value and a default is provided, use the DEFAULT keyword to specify that the default value should be used.

EI006 Required parameter is not passed

Procedure parameter is not defined as OUTPUT, but marked as OUTPUT in procedure call statement.

Output scalar parameters for procedures are passed to the procedure, and can have their value altered within the procedure. This allows procedures to return scalar output. The formal parameter must be declared as an OUTPUT parameter if the actual parameter that is passed had the OUTPUT keyword. This triggers an error.

EI007 Call parameter declared as output

Procedure parameter is defined as OUTPUT, but is not marked as OUTPUT in procedure call statement.

Output scalar parameters for procedures are passed to the procedure, and can have their value altered within the procedure. This allows procedures to return scalar output. However, the matching variable passed as the output parameter in the module or command string must also have the keyword OUTPUT. There is no error but the resultant value is NULL, which you are unlikely to want.

EI008 Call parameter is not declared as output

Number of passed parameters exceeds the number of procedure parameters

Parameters can be passed to procedures and functions in an ordered delimited list, but never more than the number of parameters. For a

function, this must have the same number of list members as the parameters. For a procedure you can have fewer if defaults are declared in parameters

EI009 Call has more parameters than required

Security Loopholes

Using SQL Server logins, especially without password expirations or Windows password policy

Sometimes you must use SQL Server logins. For example, with Microsoft Azure SQL Database, you have no other option, but it isn't satisfactory. SQL Server logins and passwords have to be sent across the network and can be read by sniffers. They also require passwords to be stored on client machines and in connection strings. SQL logins are particularly vulnerable to a brute-force attacks. They are also less convenient because the SQL Server Management Studio (SSMS) registered servers don't store password information and so can't be used for executing SQL across a range of servers. Windows-based authentication is far more robust and should be used where possible.

Using the xp_cmdshell system stored procedure

Use xp_cmdshell in a routine only as a last resort, due to the elevated

security permissions they require and consequential security risk. The xp_cmdshell procedure is best reserved for scheduled jobs where security can be better managed.

Authentication set to Mixed Mode

Ensure that Windows Authentication Mode is used wherever possible. SQL Server authentication is necessary only when a server is remote or outside the domain, or if third-party software requires SQL authentication for remote maintenance. Windows Authentication is less vulnerable, and avoids having to transmit passwords over the network or store them in connection strings.

Using dynamic SQL without the WITH EXECUTE AS clause

Because of ownership chaining and SQL injection risks, dynamic SQL requires constant vigilance to ensure that it is used only as intended. Use the EXECUTE AS clause to ensure the dynamic SQL code inside the procedure is executed only in the context you expect, and use loginless users with just the specific permissions required but no others in the EXECUTE AS clause.

Using dynamic SQL with the possibility of SQL injection

SQL injection can be used not only from an application but also by a database user who lacks, but wants, the permissions necessary to perform a particular role, or who simply wants to access sensitive data. If dynamic SQL is executed within a stored procedure, under the temporary EXECUTE AS permission of a user with sufficient privileges to create users, and it can be accessed by a malicious user, then suitable precautions must be taken to make this impossible. These precautions start with giving EXECUTE AS permissions only to WITHOUT LOGIN users

with least-necessary permissions, and using sp_ExecuteSQL with parameters rather than EXECUTE.

BP013 EXECUTE(string) is used

Acknowledgements

For a booklet like this, it is best to go with the established opinion of what constitutes a SQL Code Smell. There is little room for creativity. In order to identify only those SQL coding habits that could, in some circumstances, lead to problems, I must rely on the help of experts, and I am very grateful for the help, support and writings of the following people in particular.

- Dave Howard
- Merrill Aldrich
- Plamen Ratchev
- Dave Levy
- Mike Reigler
- Anil Das
- Adrian Hills
- Sam Stange
- Ian Stirk
- Aaron Bertrand
- Neil Hambly
- Matt Whitfield
- Nick Harrison
- Bill Fellows
- Jeremiah Peschka
- Diane McNurlan
- Robert L Davis
- Dave Ballantyne
- John Stafford
- Alex Kusnetsov

- Gail Shaw
- Jeff Moden
- Joe Celko
- Robert Young

And special thanks to our technical referees, Grant Fritchey and Jonathan Allen.