119 SQL CODE SMELLS

Contents

Introduction

Once you've done a number of SQL code-reviews, you'll be able to identify 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 'CodeSmell' 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 119 code-smells in a booklet, I've been very constrained on space to describe each code smell. 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.

- ***Phil Factor***, *Contributing Editor*

Problems with Database Design

1 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, as long as the value remains in the format in which it started.

We store strings, after all, and a string is hardly atomic since it consists of an ordinally significant collection of characters or words. A list or XML value stored in a column, whether by character map, bitmap or XML data type, can be a useful temporary expedient during development, but the column will likely need to be normalized if values will have to be shredded.

A related code smell is:

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.

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

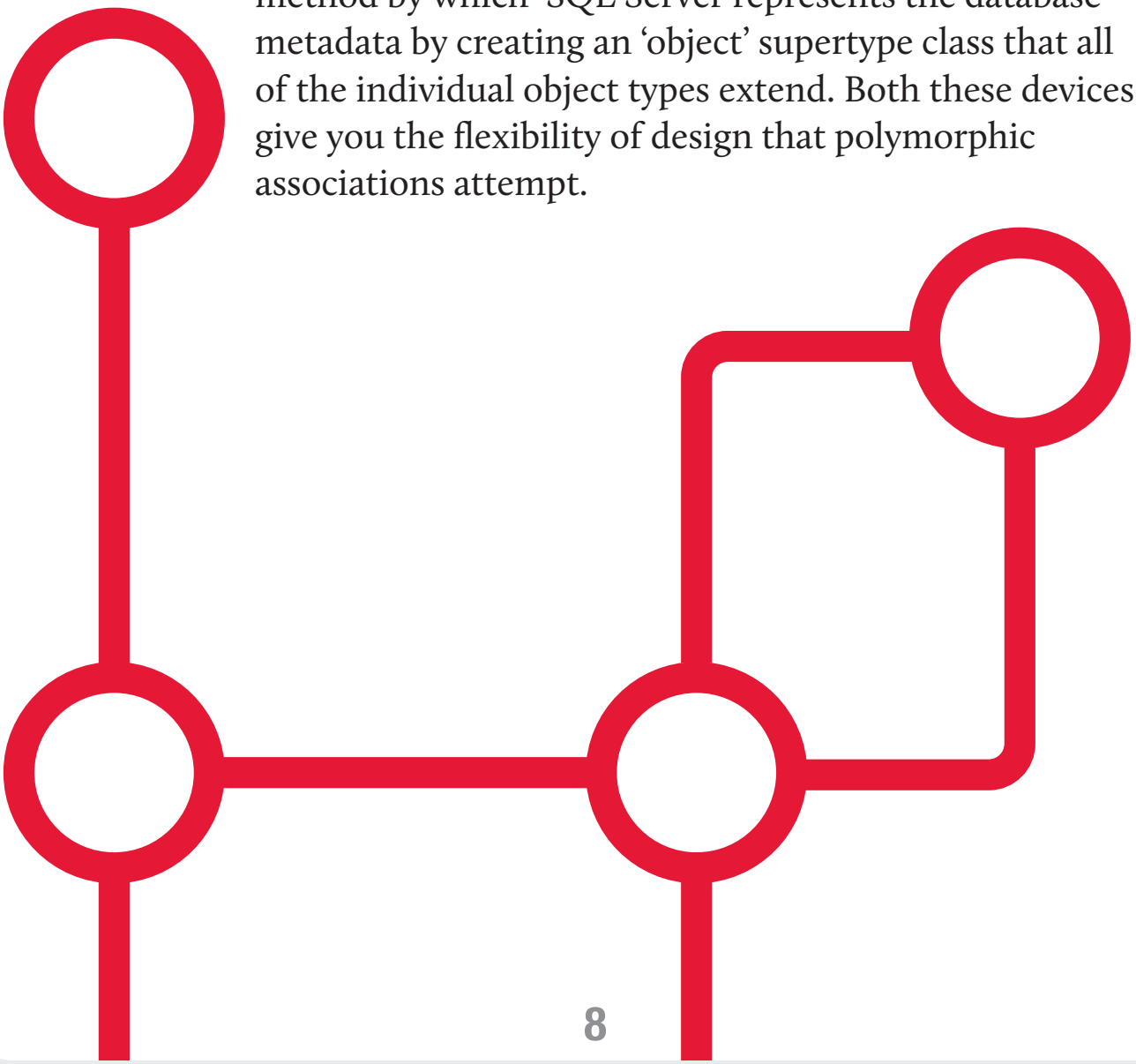
Self-referencing tables seem like an elegant way to represent hierarchies. However, such an approach mixes relationships and values. Real-life hierarchies need more than a parent-child relationship. The 'Closure Table' pattern, where the relationships are held in a table separate from the data, is much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

3 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!

4 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.



5 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 'denormalize' 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 denormalization.

6 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 processes, 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.

7 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

8 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.

9 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.

10 Leaving referential integrity constraints disabled

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.

11 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.

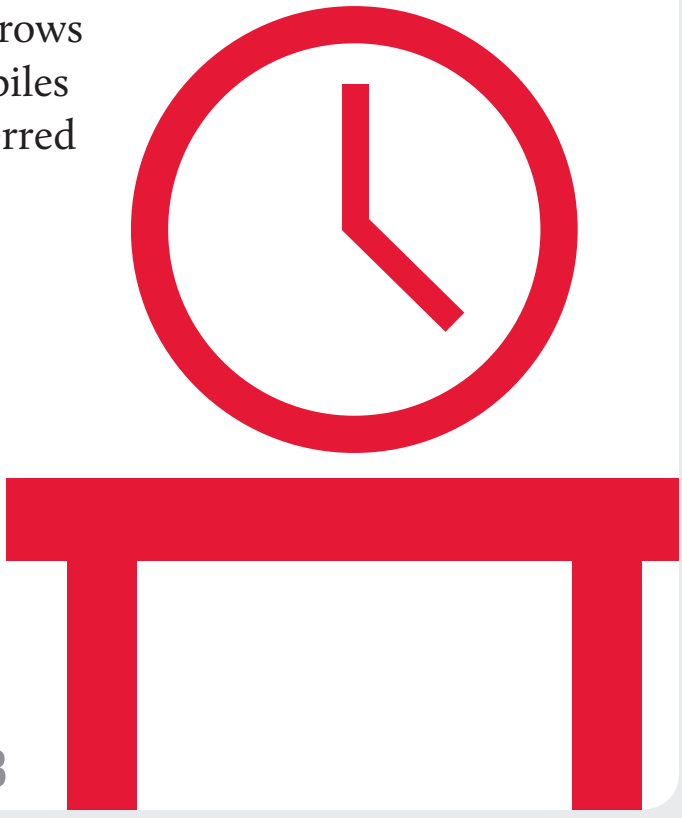
SQL Server lets you create completely redundant and totally duplicate indexes. Sometimes this is done in the mistaken belief that the order of 'included' (non-key) columns is significant. It isn't!

12 Misusing NULL values

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

13 Using temporary tables for very small resultsets

Temporary tables can lead to recompiles, which can be costly. Table variables, while not useful for larger data sets (approximately 75 rows or more), avoid recompiles and are therefore preferred in smaller data sets.



14 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.

15 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.

16 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.

17 Defining a table column without explicitly specifying whether it is nullable

The default nullability for a database's columns can be altered as a 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, and it is essential if you need any portability of your table design.

18 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

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

Columns of short or fixed length should have a fixed size since variable-length types have a disproportionate storage overhead. For a large table, this could be significant.

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

20 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.

21 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.

22 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, 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.

23 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.

24 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.

25 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.

26 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.

27 Using the TIME data type to store a duration rather than a point in time

Durations are best stored as a start date/time value and end date/time value. This is especially true given that you usually need the start and end points to calculate a duration. 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 cause confusion for the next person who has to maintain your code.

28 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).

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

You can't require everyone to stop using national characters or accents any more. The 1950s 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!

Problems with expressions

30 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.

31 Using 'broken' functions such as 'ISNUMERIC' without additional checks

Some functions, such as ISNUMERIC, are entirely useless and are there only for backward compatibility - or possibly as a joke. (Try `Select isNumeric(',')`, for example.) Use a function that is appropriate for the data type whose validity you are testing.

32 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.

33 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.

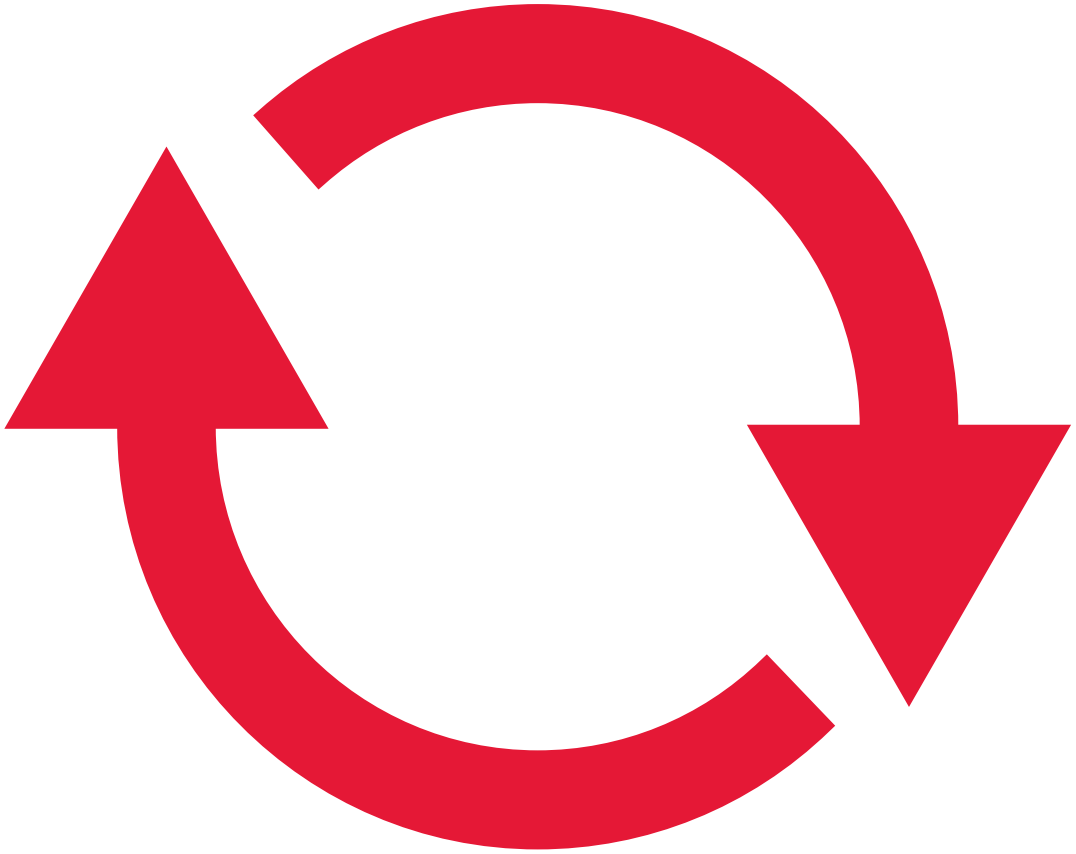
34 Not using a semicolon to terminate SQL statements

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.

35 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 than not, you should explicitly define your conversions to avoid unintentional consequences.

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



36 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.

See: SR0008: Consider using SCOPE_IDENTITY instead of @@IDENTITY

37 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.

38 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. 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.

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

Difficulties with Query Syntax

39 Creating UberQueries (God-like Queries)

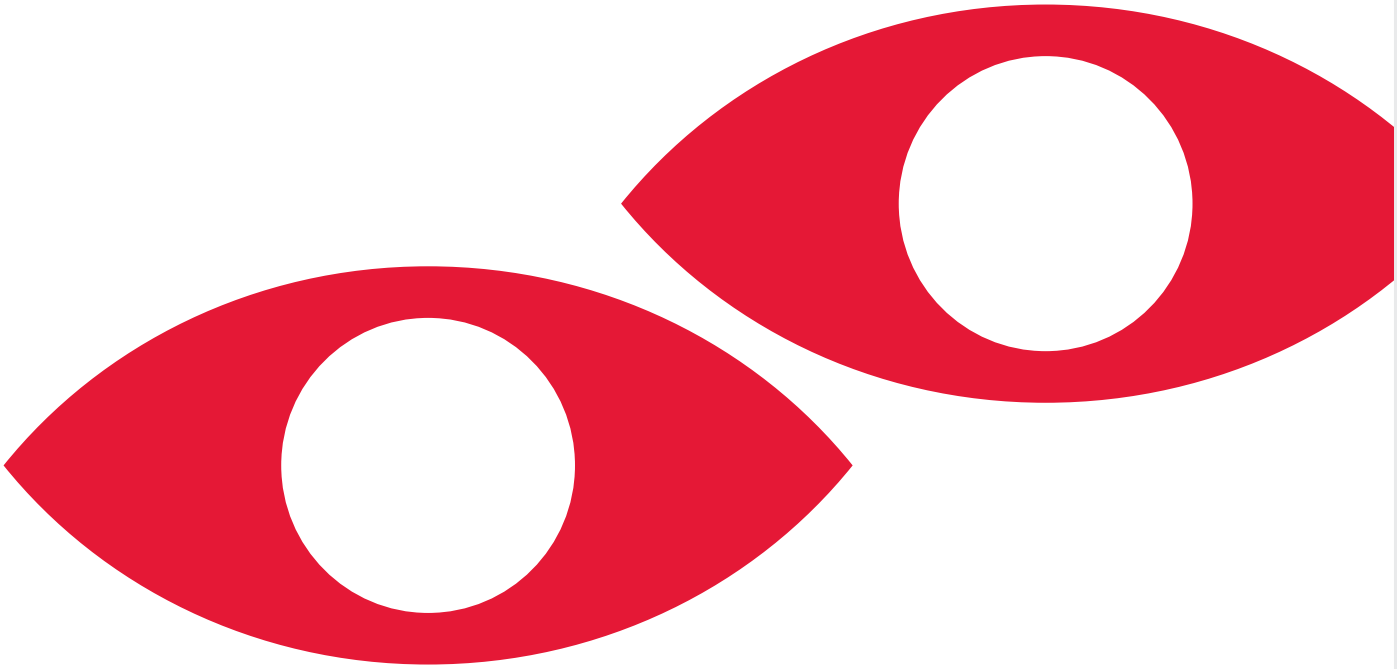
Always avoid overweight queries (e.g., a single query with four inner joins, eight left joins, four derived tables, ten subqueries, eight clustered GUIDs, two UDFs and six case statements).

40 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.

41 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'.



42 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.

See: SR0010: Avoid using deprecated syntax when you join tables or views

43 Using correlated subqueries instead of a JOIN

Correlated subqueries, queries that run against each row 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.

44 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.

45 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 optimized. Using user-defined functions in the same way will lead to very slow queries since they perform much like correlated subqueries.

46 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.

47 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.

48 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.

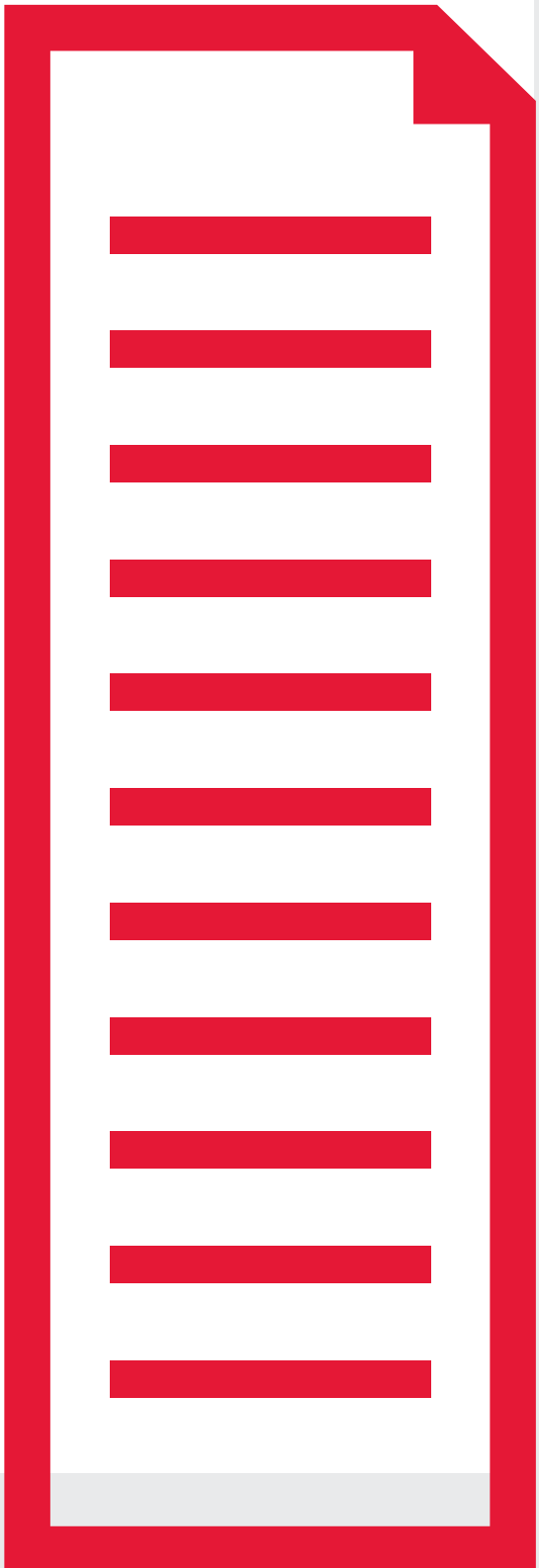
49 Including complex conditionals in the WHERE clause

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.

50 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 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.



51 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.

52 Not handling NULL values in nullable columns

Generally, it is wise to explicitly handle NULLs in nullable columns, by using COALESCE or ISNULL 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.)

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

53 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

54 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 "%" in LIKE predicates

55 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.

56 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 statement - or when calling the OBJECT_ID function - the database engine can find the objects more easily if the names are qualified with the schema name.

57 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.

58 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.

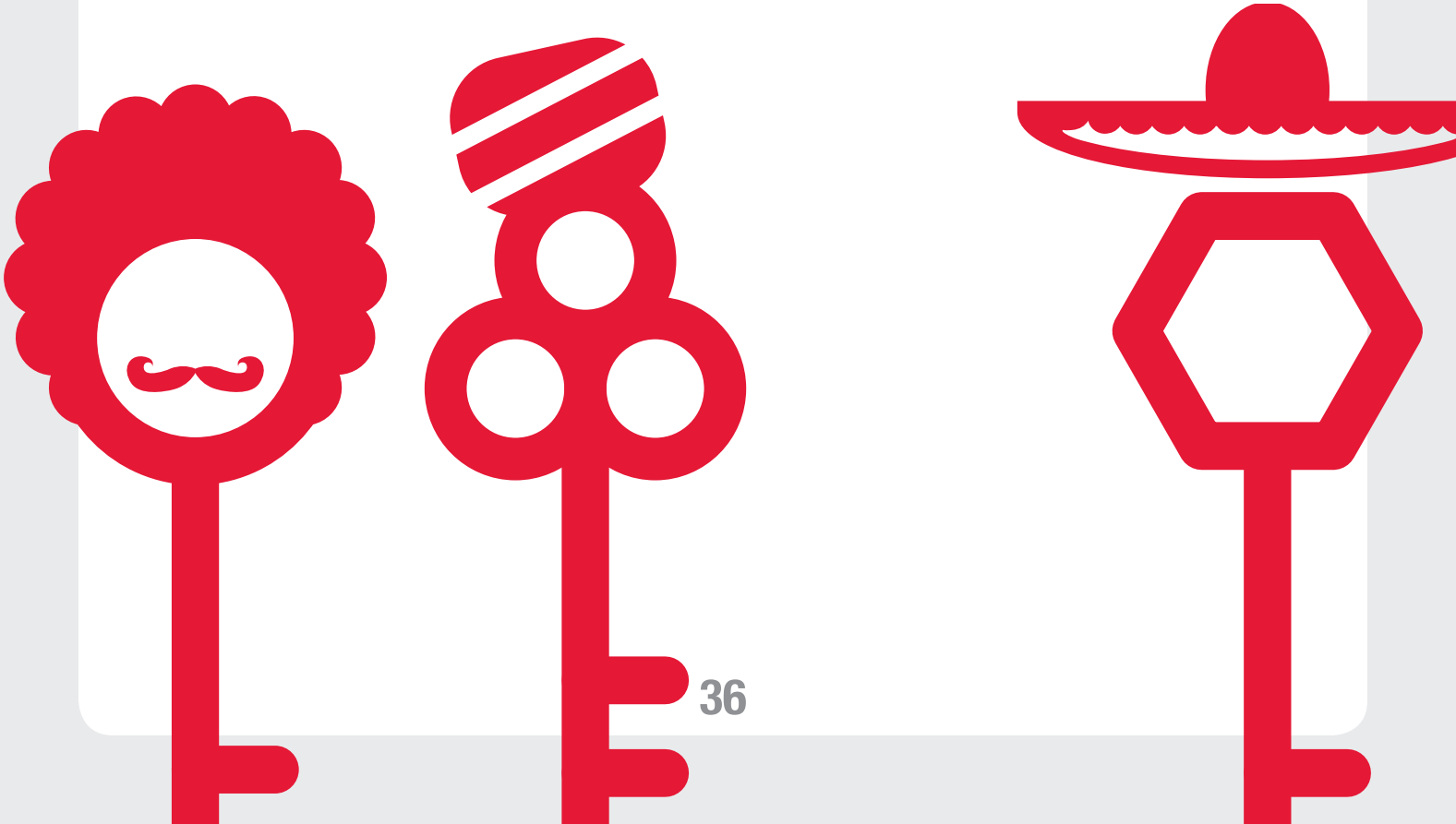
59 Using the NOT IN predicate in the WHERE clause

Your 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.

60 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



61 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

62 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

63 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.

64 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.

Problems with naming

65 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. Long names can be frustrating to the person using SQL interactively, unless that person is using SQL Prompt or some other IntelliSense system, though you can't rely on it.

66 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.

67 '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.

68 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.

See: SR0012: Avoid using reserved words for type names

69 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 require brackets (or double quotations) around the object name, make code difficult to read, and make 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.

See: SR0011: Avoid using special characters in object names

70 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

71 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.

72 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

73 Including few or no comments

Being antisocial is no excuse. Neither 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.

74 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.

75 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.

76 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.

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

79 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.

80 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.

81 Unnecessarily using stored procedures or 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 table valued functions.

82 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.

83 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.

84 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 optimized to perform set-based operations rapidly. Don't fight it!

85 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 cause the first procedure to generate an error.

86 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

87 Specifying parameters by order rather than 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.

88 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

89 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.

90 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.

91 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.

92 Using the ISNUMERIC function

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 ($).

93 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.

94 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.

95 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.

Using SET ROWCOUNT to specify how many rows should be returned

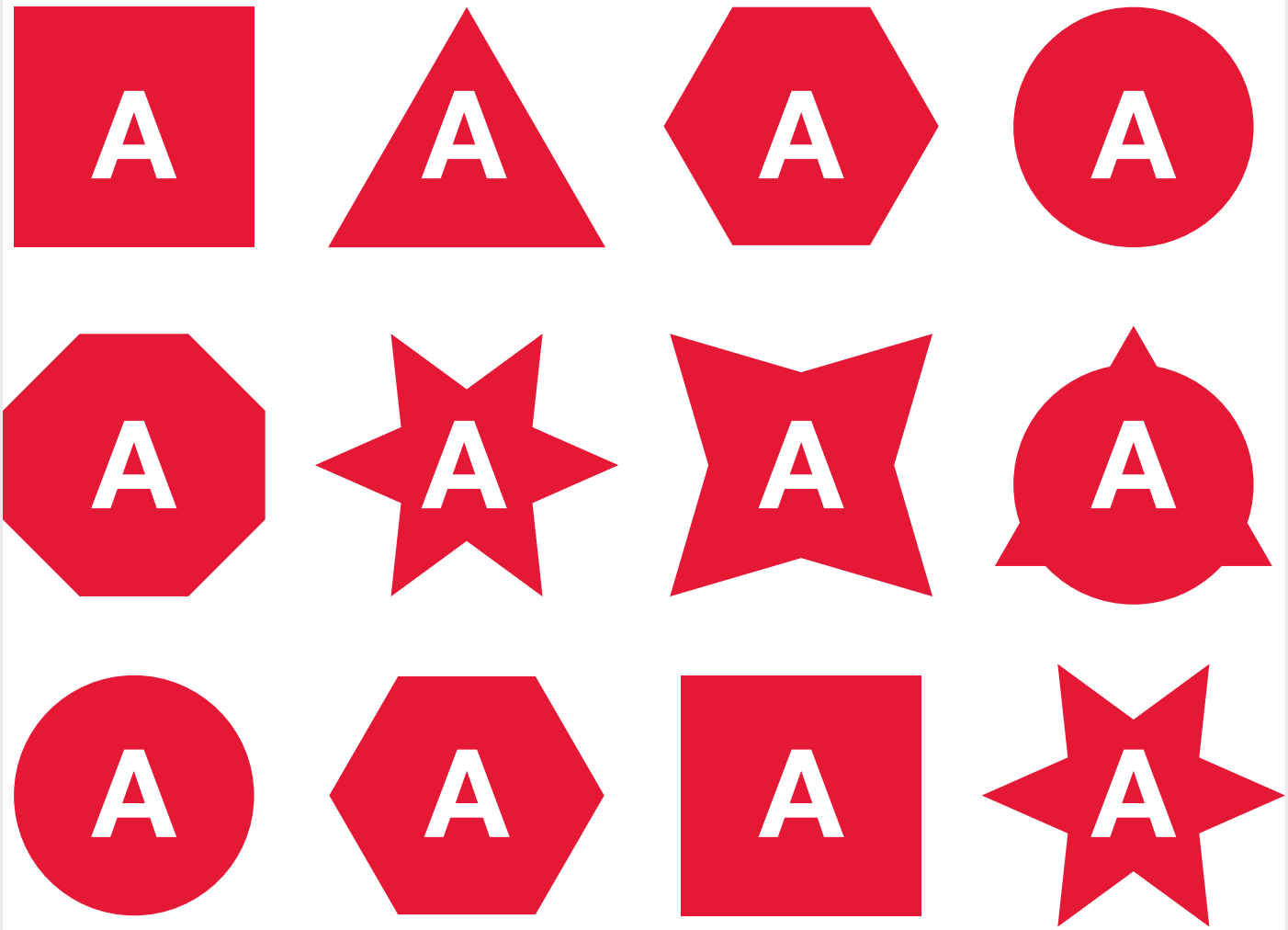
We had to use this option until the TOP clause (with ORDER BY) was implemented. The TOP option is much easier for the query optimizer.

Using TOP 100% 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.

98 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.



99 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.

100 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.

101 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.

102 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.

103 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.

104 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 syntax. The COMPUTE and COMPUTE BY operations were devised for printed summary results. The ROLLUP clause is a better alternative.

105 Using numbers in the ORDER BY clause to specify column order

It is certainly possible to specify non-negative 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.

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

Column references should be two-part names when there is any chance of ambiguity due to column names being duplicated in other tables. 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.

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.

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 initialize 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.

112 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.

113 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.

114 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.

Security Loopholes

115 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 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.

116 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.

117 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.

118 Using dynamic SQL without the 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.

Using dynamic SQL with the possibility of SQL injection

SQL injection can be used not only from an application but also by a 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, it can be accessed by a malicious user. 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`.

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**.

redgate

Avoid code smells with these database delivery tools from Red Gate

SQL Source Control

Plug your source control system into SQL Server Management Studio.

SQL Compare

SQL Compare is the industry-standard tool for synchronizing SQL Server environments.

SQL Test

Run unit tests inside SQL Server Management Studio. Use the command line to automate testing as part of your CI process.

SQL Data Generator

Generate realistic test data. Use the command line alongside SQL Test as part of your CI process.

Database Delivery Patterns & Practices Library

Keep on learning with the Database Delivery Patterns & Practices Library on Simple-Talk.

Manual

No source control, or manual, ad hoc source control

Source control

Automated source control solution

Continuous integration

Source control and CI for the database

Continuous delivery

Source control, CI, and automated deployment

Monitoring

Drift checking and performance monitoring

Find out how you can make your database delivery process more efficient. Start reading today

redgate

SQL Enlight

Refactoring and static code analysis tool for SQL Server

SQL Enlight provides a fast, automated way to ensure that your T-SQL source code fits to your predefined design and style guidelines as well as best coding practices. With its help, you can easily analyze, identify and quickly resolve potential design and performance pitfalls in your SQL Server databases.

redgate