dbs41.1 DATA SCRIPTING CONVENTION

logo_provfin

Author:

Dale Stewart - Aug 2016

# DOCUMENT CONTROL

## CHANGE HISTORY

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Status** | **Author** | **Comments** |
| 1 | 01/08/2014 | DRAFT | Dale Stewart | Initial Draft |
|  |  |  |  |  |
|  |  |  |  |  |

## DOCUMENT STAKEHOLDERS

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Position** | **Purpose** | **Date signed off** |
| David Green |  |  |  |
| Jon Eden |  |  |  |
| Atif Hakim |  |  |  |
| Jamie Dolan |  |  |  |
| Dipesh Mistry |  |  |  |
| Dale Stewart |  |  |  |
| David Bennet |  |  |  |
| Melissa Maluila |  |  |  |
| Russ Bagley |  |  |  |
| Kevin Whittington |  |  |  |
| Jon Bairstow |  |  |  |
| Ahmed Akhbar |  |  |  |

Table of Contents

[1 DOCUMENT CONTROL - 2 -](#_Toc461198627)

[1.1 CHANGE HISTORY - 2 -](#_Toc461198628)

[1.2 DOCUMENT STAKEHOLDERS - 2 -](#_Toc461198629)

[3 INTRODUCTION - 6 -](#_Toc461198630)

[3.1 CONVENTION OBJECTIVES - 6 -](#_Toc461198631)

[3.2 SCOPE - 6 -](#_Toc461198632)

[4 DATABASES - 7 -](#_Toc461198633)

[4.1 META DATA - 7 -](#_Toc461198634)

[5 READABILITY - 8 -](#_Toc461198635)

[5.1 OVERALL STRUCTURE - 8 -](#_Toc461198636)

[5.2 IDENTIFIERS - 9 -](#_Toc461198637)

[5.2.1 CASE (CAPITALIZATION) - 9 -](#_Toc461198638)

[5.2.2 PREFIXES AND SUFFIXES - 9 -](#_Toc461198639)

[5.2.3 VERBS - 10 -](#_Toc461198640)

[5.2.4 COLUMNS AND VARIABLES - 10 -](#_Toc461198641)

[5.2.5 NAME, TYPE, FLAG - 10 -](#_Toc461198642)

[5.3 TABLES - 11 -](#_Toc461198643)

[5.3.1 META DATA - 11 -](#_Toc461198644)

[5.3.2 CREATING TABLES - 11 -](#_Toc461198645)

[5.3.3 INDEXES - 11 -](#_Toc461198646)

[5.4 DATA MANIPULATION STATEMENTS - 12 -](#_Toc461198647)

[5.4.1 ERROR HANDLING - 12 -](#_Toc461198648)

[5.5 DATABASE SECURITY - 13 -](#_Toc461198649)

[5.6 FORMATING - 13 -](#_Toc461198650)

[5.6.1 QUOTES - 13 -](#_Toc461198651)

[5.6.2 PARENTHESES - 13 -](#_Toc461198652)

[5.6.3 WHITESPACE - 13 -](#_Toc461198653)

[5.6.4 COMMENTS - 13 -](#_Toc461198654)

[5.6.5 DML STATEMENTS (SELECT, INSERT, UPDATE, DELETE) - 13 -](#_Toc461198655)

[5.6.6 SELECT - 14 -](#_Toc461198656)

[5.6.7 INSERTS - 14 -](#_Toc461198657)

[5.6.8 UPDATES - 14 -](#_Toc461198658)

[5.7 TRANSACTIONS - 14 -](#_Toc461198659)

[5.8 CONTROL OF FLOW STATEMENTS - 14 -](#_Toc461198660)

[5.9 CURSORS - 15 -](#_Toc461198661)

[5.10 LINKED SERVERS - 15 -](#_Toc461198662)

[6 SCHEMA SCRIPTS - 16 -](#_Toc461198663)

[6.1.1 ROW COUNT VERIFICATION - 16 -](#_Toc461198664)

[7 ERROR NOTIFICATION - 17 -](#_Toc461198665)

[7.1 EMAIL - 17 -](#_Toc461198666)

[7.2 INSTANT MESSENGER - 17 -](#_Toc461198667)

[7.3 INCIDENT MANAGEMENT SYSTEM - 17 -](#_Toc461198668)

[7.4 LOG TABLE\WINDOWS EVENT LOG\LOG FILE - 17 -](#_Toc461198669)

[8 METHODOLOGY - 18 -](#_Toc461198670)

[8.1 DEVELOPMENT ENVIRONMENTS - 18 -](#_Toc461198671)

[8.2 DATA TYPE SELECTION - 18 -](#_Toc461198672)

[8.3 CLASS MODEL - 18 -](#_Toc461198673)

[8.4 CHANGE CONTROL - 18 -](#_Toc461198674)

[8.5 DEVELOPER INDEMNITY - 19 -](#_Toc461198675)

[8.5.1 UNIT TEST PLANS - 19 -](#_Toc461198676)

[8.5.2 CODE REVIEW - 19 -](#_Toc461198677)

[8.5.3 EVALUATION - 19 -](#_Toc461198678)

[8.5.4 DEVELOPMENT PROCESS - 20 -](#_Toc461198679)

[8.6 RELEASE PROCEDURE - 21 -](#_Toc461198680)

[9 QUALITY CHECKS - 22 -](#_Toc461198681)

[9.1 TESTING - 22 -](#_Toc461198682)

[9.1.1 TEMPLATE - 22 -](#_Toc461198683)

[9.2 DEVELOPMENT GUIDELINES - 23 -](#_Toc461198684)

[9.2.1 ADDING COLUMNS WITH DEFAULTS - 23 -](#_Toc461198685)

[9.2.2 ALWAYSON - 23 -](#_Toc461198686)

[9.2.3 BULK LOADS\DISABLING INDEXES - 23 -](#_Toc461198687)

[9.2.4 CASE CLAUSES - 23 -](#_Toc461198688)

[9.2.5 CLUSTERED INDEXES - 23 -](#_Toc461198689)

[9.2.6 COMMON LANGUAGE RUNTIME - 23 -](#_Toc461198690)

[9.2.7 CONTINUOUS DEPLOYMENT - 23 -](#_Toc461198691)

[9.2.8 DEPRECATED\DISCONTINUED FEATURES - 23 -](#_Toc461198692)

[9.2.9 EXISTS VS IN - 24 -](#_Toc461198693)

[9.2.10 EXTENDED PROCEDURES - 24 -](#_Toc461198694)

[9.2.11 FUNCTIONS IN JOINS - 24 -](#_Toc461198695)

[9.2.12 HARD CODED IDENTITY VALUES - 24 -](#_Toc461198696)

[9.2.13 INDEX FILL FACTOR - 24 -](#_Toc461198697)

[9.2.14 LARGE UPDATES\DELETES - 24 -](#_Toc461198698)

[9.2.15 LAST PAGE CONTENTION - 25 -](#_Toc461198699)

[9.2.16 LOGICAL REPRESENTATION - 25 -](#_Toc461198700)

[9.2.17 MULTI STATEMENT\INLINE FUNCTIONS - 25 -](#_Toc461198701)

[9.2.18 (N)VARCHAR(MAX) - 25 -](#_Toc461198702)

[9.2.19 PREVENTING DEADLOCKS - 25 -](#_Toc461198703)

[9.2.20 PRIMARY FILE GROUPS - 25 -](#_Toc461198704)

[9.2.21 READ COMMITTED SNAPSHOT AND SNAPSHOT ISOLATION - 26 -](#_Toc461198705)

[9.2.22 SPARSE COLUMNS - 26 -](#_Toc461198706)

[9.2.23 SUPPORTING DELTA LOADS - 26 -](#_Toc461198707)

[9.2.24 TABLE COMPRESSION - 26 -](#_Toc461198708)

[9.2.25 TABLE VARIABLES\HASH TABLES - 26 -](#_Toc461198709)

[9.2.26 TABLE VALUE PARAMETERS - 26 -](#_Toc461198710)

[9.2.27 TRUSTED FOREIGN KEYS - 27 -](#_Toc461198711)

[9.2.28 UNICODE DATA TYPES - 27 -](#_Toc461198712)

[9.2.29 VERSION CONTROL - 27 -](#_Toc461198713)

[9.2.30 VIRTUALISATION SUPPORT - 27 -](#_Toc461198714)

[9.2.31 XML AND SEMI-STRUCTURED DATA - 27 -](#_Toc461198715)

[10 REFERENCES - 28 -](#_Toc461198716)

[10.1 BESPOKE STANDARDS - 28 -](#_Toc461198717)

[10.1 BEST PRACTICES - 28 -](#_Toc461198718)

[10.1 BOOKS - 28 -](#_Toc461198719)

[10.1 DBS STANDARD OPERATING PROCEDURES - 28 -](#_Toc461198720)

[10.1 FUNCTIONS - 28 -](#_Toc461198721)

[10.1 INDEXES - 28 -](#_Toc461198722)

[10.1 NOTATION - 28 -](#_Toc461198723)

[10.2 STANDARDS ORGANISATIONS - 28 -](#_Toc461198724)

[10.3 STANDARDS DOCUMENTS - 28 -](#_Toc461198725)

[10.4 TESTING - 28 -](#_Toc461198726)

[11 APPENDIX A - 29 -](#_Toc461198727)

[11.1 ALTER TABLE - 29 -](#_Toc461198728)

[11.2 CREATE STORED PROCEDURE - 29 -](#_Toc461198729)

[11.3 CREATE TABLE - 32 -](#_Toc461198730)

[11.4 DML - 32 -](#_Toc461198731)

[11.4.1 DELETES - 32 -](#_Toc461198732)

[11.4.2 INSERT - 33 -](#_Toc461198733)

[11.4.3 SELECT - 33 -](#_Toc461198734)

[11.4.4 UPDATE - 34 -](#_Toc461198735)

[11.5 INDEXES - 34 -](#_Toc461198736)

[11.6 OBJECT META DATA - 34 -](#_Toc461198737)

[11.7 BULK DELETE - 35 -](#_Toc461198738)

[11.8 STANDARD SUPPORT SCRIPT - 36 -](#_Toc461198739)

# INTRODUCTION

This document will provide a framework to aid in optimal usability of Microsoft SQL Server schema scripts and stored procedures developed for applications by defining a reasonable, consistent and effective coding style.

The framework will serve to improve the application without unnecessary impact on development and unnecessary controls on personal coding preferences. For these reasons the framework will focus on identifier naming conventions that are intended to be used by all developers, general style guidelines indicating the preferred format and usage of SQL language components, and a definition of the database development methodology.

The identifier segment of the standard will formalize naming conventions. All schema scripts and stored procedures will conform to all elements of this area of the document.

The general guideline will include SQL statement formatting and outlines for solutions to more complex components within scripts and stored procedures. This instrument is intended as a best practice model consistent with the identified architecture and requirements. It may be necessary to adapt the guideline to specific solutions within the application. While the guidelines are not mandatory, adherence will aid in the ultimate success of the application and ease of maintenance. All developers will be expected to reasonably defend any deviation from the guideline.

The combination of conformance to standards and development within the stated guidelines will be measured and assessed in the context of the methodology. The methodology will present structure and consistency through clearly defined requirement specifications, change control procedures, code review, testing, controlled iterative development cycles, and regular developer evaluations.

## CONVENTION OBJECTIVES

The subject of database scripting is incredibly broad, it is possible for this document to become exhaustive and unusable. In an effort to make the document relevant the following objectives form the spirit of each section:

* Improve
  + readability
  + success rate of code deployment
  + maintainability
  + consistency
  + reuse of code
  + usage of source control systems
* Provide a method for measuring
  + code quality
  + deployment success
* Prevent
  + functional failures in production
  + common performance issues

## SCOPE

This document will focus specifically on TSQL scripts for use on SQL Server database engines. Conventions around SQL Server Analysis Services and SQL Server Integration Services will be discussed in a different document but within the same series of conventions.

The following key areas will be considered:

* Readability
  + Structure
  + Formatting
  + Control of Flow
  + Data Manipulation Statements
* Methodology
  + Development
  + Change Control
  + Release Procedure
* Quality Checks
  + Data maintenance and house keeping
  + Retention periods
  + Source Control
  + Templating
* Validity
  + Front end data validation
  + Data structure
  + Data type selection
  + Indexing
  + Virtualization support
* Testing
  + Throughput\Benchmarking
  + Extreme value
  + Unit testing
  + Smoke testing
  + System

# DATABASES

## META DATA

|  |  |  |  |
| --- | --- | --- | --- |
| **Tag** | **Description** | **Format** | **IsMandatory** |
| Access\_Owner | Person or persons responsible for authorising access to the data | Person name or team name | TRUE |
| Actual\_DR\_Tier | The disaster recovery tier the data currently resides in | Number between 1-7 | TRUE |
| Actual\_HA\_Tier | The availability tier the data currently resides in | Number between 1-4 | TRUE |
| Backup\_Location | The folder path or suitable reference to where backups are retained for the data | Plain text | TRUE |
| Backup\_Schedule | How frequently the data is backed up | <Backup\_Schedule><Full>Weekly,Sunday</Full><Diff>Daily  </Diff><Log>30min</Log></Backup\_Schedule> | TRUE |
| Business\_Owner | The business stakeholder of the data. This is usually a manager of head of function for the business area | Person name or team name | TRUE |
| Can\_Be\_Overwritten | Only to be used for development and test databases. This indicates the database can be restored over with new data | TRUE\FALSE |  |
| Change\_Request\_No | The change request number pertaining to release of the database and any subsequent changes to the database after initial release | Number |  |
| CName | DNS alias used to access the database server. | Plain text | TRUE |
| Data\_Owner | The business stakeholder of the data. This is usually a manager of head of function for the business area | Person name or team name | TRUE |
| Data\_Quality\_Grade | More relevant for derived data from production systems, however this | See DataQuality |  |
| Data\_Sensitivity\_Level | The sensitivity of the data according to all relevant internal and external policies and acts | PUBLIC\INTERNAL\CONFIDENTIAL\HIGHLY\_CONFIDENTIAL | TRUE |
| Description | A description of the function of the database and how it pertains the system it supports | Plain text | TRUE |
| Desired\_DR\_Tier | The disaster recovery tier the data should reside in | Number between 1-7 |  |
| Desired\_HA\_Tier | The availability tier the data should reside in | Number between 1-4 |  |
| Last\_Release\_Date | The last date any type of release or hotfix was applied to the database. This does not include patching of the database engine itself | Datetime |  |
| Maintenance\_Window | An agreed period when downtime or sub-optimal performance is acceptable to carry out essential maintenance tasks (such as reindexing) | Example: 9-5, Every Monday 10-12 | TRUE |
| Originating\_Server | Intended for use after migration of the database from another server | Plain text |  |
| Project\_Manager | The name or names of the key project managers | Person name or team name |  |
| Project\_Owner | The sponsor of the project | Person name or team name |  |
| Recovery\_Point\_Objective | The maximum amount of data loss for a system or database before it poses a significant risk to the business | 00Days:00Hours:00Minutes:00Seconds | TRUE |
| Recovery\_Time\_Objective | The maximum amount of time a system or database can be "unavailable" before it poses a significant risk to the business | 00Days:00Hours:00Minutes:00Seconds | TRUE |
| Source\_Control\_Location | Link or relevant reference to source control | Plain text |  |
| Support\_Hours | The hours the database has been agreed to be support in | Example: 9-5 or 247 | TRUE |
| Support\_Response\_Time | The response time agreed by the support department for the database | 00Days:00Hours:00Minutes:00Seconds | TRUE |
| Support\_Service | Service(s) supported by the database | Plain text |  |
| Third\_Party\_Contact | Person name or team name | Plain text |  |
| Third\_Party\_Product | Supplier of the database or system | Plain text |  |
| Version | The version number of the database | Major.Minor.Build or relevant version reference | TRUE |

# READABILITY

## OVERALL STRUCTURE

The diagram below s the desired structure of support scripts supplied to the DBA team. Each high level section is discussed in more detail throughout this document.



## IDENTIFIERS

### CASE (CAPITALIZATION)

* Use all upper case for table and view names
* Use mixed case for column names and variables
* Use mixed case for stored procedure name
* Use lower case for other names except use the same case as indicated above where a table or column is used in another object’s name

### PREFIXES AND SUFFIXES

* Use the following standard prefixes for database objects:

|  |  |  |  |
| --- | --- | --- | --- |
| **Object type** | **Prefix** | **Example** | **Context** |
| Primary key Clustered | **pkc\_** | **Pkc\_Table\_Column** | Index |
| Primary key Nonclustered | **pkn\_** | **pkn\_Table\_Column\_List** | Index |
| Index Clustered | **cix\_** | **ixc\_Table\_Column** | Index |
| Index Nonclustered | **ncix\_** | **ixn\_Table\_Column\_List** | Index |
| Foreign key | **fk\_** | **fk\_This\_Table\_TB\_Pkey\_Table** | Table Object |
| Unique Constraint | **unq\_** | **unq\_Table\_Column\_List** | Table Object |
| Check Constraint | **chk\_** | **chk\_Table\_Column** | Table Object |
| Column Default | **dft\_** | **dft\_Table\_Column\_List** | Table Object |
| Passed Parameter | **@p** | **@pPassedVariableName** | Script |
| Local Variable | **@** | **@VariableName** | Script |
| Table | **\*** | **Table\_Name**(see detail below) | Database Object |
| View | **vw\_** | **VW\_NET\_Active\_Units** | Database Object |
| User Defined Scalar Function | **udf\_** | **udf\_return\_value\_name** | Database Object |
| User Defined Table Function | **udt\_** | **udt\_Table\_Name** | Database Object |
| User Defined CLR Scalar Function | **udcf\_** | **udcf\_return\_value\_name** | Database Object |
| User Defined CLR Table Function | **udct\_** | **udct\_return\_value\_name** | Database Object |
| Stored Procedure | **usp\_** | **usp\_Verb\_Business\_Name**(see detail below) | Database Object |
| In Memory Table | **imt\_** | **imt\_Table\_Name** | Database Object |
| In Memory Stored Procedure | **imsp\_** | **imsp\_Verb\_Business\_Name** (see detail below) | Database Object |
| Clustered Columnstore Index | **ccix\_** | **ccix\_Table\_Name** | Index |
| Nonclustered Columnstore index | **nccix\_** | **nccix\_Table\_Name** | Index |
| Server Trigger | **ustr\_** | **ustr\_Trigger\_Name** | Server Object |
| DatabaseTrigger | **udtr\_** | **udtr\_Trigger\_Name** | Database Object |
| Synonym | **syn\_** | **syn\_Referenced\_Table** | Database Object |
| Symmetric Key | **smk\_** | **smk\_Symmetric\_Key\_Name** | Database Object |
| Asymmetric Key | **asmk\_** | **asmk\_Asymmetric\_Key\_Name** | Database Object |
| Certificate | **crt\_** | **crt\_Certificate\_Name** | Database Object |
| Sequence | **seq\_** | **seq\_Sequence\_Name** | Database Object |

### VERBS

Where practical, the first identifier in any stored code should follow the allowed verbs in the latest version of Powershell. Adopting this naming style should allow for SQL Server and powershell code to be easier to understand for engineers.

The list below represents the allowed verbs as of version 4:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Add | Edit | Move | Restore | Uninstall |
| Approve | Enable | New | Resume | Unlock |
| Assert | Enter | Open | Revoke | Unprotect |
| Backup | Exit | Optimize | Save | Unpublish |
| Block | Expand | Out | Search | Unregister |
| Checkpoint | Export | Ping | Select | Update |
| Clear | Find | Pop | Send | Use |
| Close | Format | Protect | Set | Wait |
| Compare | Get | Publish | Show | Watch |
| Complete | Grant | Push | Skip | Write |
| Compress | Group | Read | Split |  |
| Confirm | Hide | Receive | Start |  |
| Connect | Import | Redo | Step |  |
| Convert | Initialize | Register | Stop |  |
| ConvertFrom | Install | Remove | Submit |  |
| ConvertTo | Invoke | Rename | Suspend |  |
| Copy | Join | Repair | Switch |  |
| Debug | Limit | Request | Sync |  |
| Deny | Lock | Reset | Test |  |
| Disable | Measure | Resize | Trace |  |
| Disconnect | Merge | Resolve | Unblock |  |
| Dismount | Mount | Restart | Undo |  |

### COLUMNS AND VARIABLES

All column names and variable names should follow the suffix structure below. Any suffixes not shown should use whole words and conform with the [Enterprise Data Standards](file:///\\hosfixity\devdata$\DBA\005%20Projects\041%20SQL%20Standards\Enterprise%20Data%20Standards.docx).

|  |  |  |
| --- | --- | --- |
| **Object Type** | **Suffix** | **Example** |
| Account | **\_Acct** | **Process\_Corp\_Acct** |
| Address | **\_Addr** | **Contact\_Addr** |
| Amount | **\_Amt** | **Total\_Credit\_Amt** |
| Balance | **\_Bal** | **Available\_Bal** |
| Date or Datetime | **\_Dt** | **Active\_Dt, @Archive\_Dt** |
| Description | **\_Descr** | **Product\_Descr** |
| Date of Birth | **\_DOB** | **Alternate\_DOB** |
| Indicator | **\_Ind** | **Net\_Gross\_Ind** |
| Line(*n*) | **\_Ln(*n*)** | **Address\_Ln2, @pOrderLn** |
| Number | **\_Nbr** | **Bank\_Nbr** |
| Record identifier/identity | **\_Id** | **Entity\_Id, @pEntityId** |
| National Insurance Number | **\_NI** | **Customer\_NI** |
| Card Transactions | **\_Tran** | **Daily\_Nbr\_Tran** |
| Post Code | **\_PC** | **Recipient\_PC** |

### NAME, TYPE, FLAG

* Stored procedure names ought to reflect the name of the primary data source, the action the procedure accomplishes, and the audience that uses the procedure
* Picking a table name can be a subjective, especially when the procedure spans multiple tables. Generally, the best table name is the one that represents most of the data or the primary join table. The longest, most inconvenient procedure name is usually the "right" one
* The audience might typically be one of the following:
  + Import
  + Export
  + Custom
  + Operations
  + Report
  + System (not directly available to the front end)
* Avoid abbreviations where possible though acronyms that have meaning to the business are acceptable
* Conform with [Enterprise Data Standards](file:///\\hosfixity\devdata$\DBA\005%20Projects\041%20SQL%20Standards\Enterprise%20Data%20Standards.docx)

## TABLES

### META DATA

Where practicable, all table objects within the database should have the following meta data tags completed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Level** | **Tag** | **Description** | **Format** | **IsMandatory** |
| Table | Access\_Owner | Person or persons responsible for authorising access to the data | Person name or team name | TRUE |
| Table | Business\_Owner | The business stakeholder of the data. This is usually a manager of head of function for the business area | Person name or team name | TRUE |
| Table | Can\_Be\_Overwritten | Only to be used for development and test databases. This indicates the database can be restored over with new data | TRUE\FALSE |  |
| Table | Change\_Request\_No | The change request number pertaining to release of the database and any subsequent changes to the database after initial release | Number |  |
| Table | Data\_Owner | The business stakeholder of the data. This is usually a manager of head of function for the business area | Person name or team name | TRUE |
| Table | Data\_Quality\_Grade | More relevant for derived data from production systems, however this | See DataQuality |  |
| Table | Data\_Sensitivity\_Level | The sensitivity of the data according to all relevant internal and external policies and acts | PUBLIC\INTERNAL\CONFIDENTIAL\HIGHLY\_CONFIDENTIAL | TRUE |
| Table | Description | A description of the function of the database and how it pertains the system it supports | Plain text | TRUE |
| Table | Last\_Release\_Date | The last date any type of release or hotfix was applied to the database. This does not include patching of the database engine itself | Datetime |  |
| Table | Version | The version number of the database | Major.Minor.Build or relevant version reference | TRUE |

### CREATING TABLES

All user objects **should not** be created on the PRIMARY filegroup. The PRIMARY filegroup should be reserved for system objects only. This practice enables the DBA team to empty data files for transition to new drives while the database is online. If user objects reside in the PRIMARY filegroup this is not possible because system data pages cannot be moved to another data file once they have been created.

When naming physical database objects (after data modelling), please observe the following conventions:

* Object names should include underscores between words
* Name all tables in the singular form (**Customer** not **Customers**)
* If a column references an ID in another table, use the full table name. For example, use **Title\_ID** in table **Author** to reference column **ID** or **Title\_ID** in table **Title**
* Explicitly name constraints. A table or column constraint name should include the table name(s) that it references. Separate each table in a constraint name with two underscores (\_\_) to differentiate from the single underscore (\_) that may be within the table or column’s name
* A foreign key name will identify the target table participating in a foreign key, the column(s) involved in the relationship. The foreign key column (table where constraint is attached) appears first

**fk\_Educator\_Id\_\_User\_\_Id**

* Avoid rules, database level defaults that must be bound, or user defined data types. While these are legitimate database constructs, opt for constraints and column defaults to hold the database consistent for development and conversion coding
* Never use a SQL Server reserved word as an identifier name. (Refer to SQL Server Books on Line for a complete listing)
* Unique constraints are preferred over unique indexes for performance reasons, though if a table’s data is accessed by the non-primary key unique column(s) a unique index may be appropriate
* **ALTER TABLE** should be used in scripts that will be used for upgrading existing table, not when creating tables. This maintains any object specific permissions applied to the table.

### INDEXES

* Explicitly name all indexes and include the table name and all indexed columns in index order

**{N}CIX\_{column name}[\_\_{column\_name}[..]]**

Examples:

**NCIX\_Name**

**CIX\_Itinirary\_Id\_\_Active\_Dt**

**NCIX\_Instructor\_Id**

* Unless there is a documented benefit, do not specify a fill factor when creating an index
* Always specify the filegroup when creating indexes. This should ideally be different to the filegroup for the data

## DATA MANIPULATION STATEMENTS

Below is a template for use when creating a new stored procedure. At first the template can seem quite lengthy, however there are various sections that are pertinent to deployment as well as the functional section of the code.

When creating a new stored procedure, the following conventions should be adhered to:

**Do not**:

* use temporary stored procedures
* define default values for parameters. If a default is needed, the front end will supply the value (applies only to stored procedures used with applications)
* use output parameters. Any information returned to the client should be done via a result set
* create stored procedures that return multiple result sets
* use the encryption option except as otherwise noted above
* use the recompile option
* use ‘\*’ wildcards, enumerate all column lists

**Avoid**:

* Calling nesting stored procedures, this can make error handling unpredictable. If procedures must be nested, fully qualify the procedure name
* Making cross database calls from views or stored procedures
* Using system stored procedures in stored procedure code. This will optimize performance as the system will never have to search for the system procedure.
* Dynamic SQL
* The use of CURSORs. There are a finite number of situations where CURSORs are relevant, these are:
  + Iterating through all databases
  + Executing a stored procedure for each row (even then a WHILE loop should be considered)
  + When parallelism is required for instances when a recursive CTE has been implemented (which are single threaded)

**Always**:

* Use ALTER PROCEDURE. This prevents any explicitly defined security permission Scripts should be structured so that procedures are dropped and recreated
* Specify the schema when creating a procedure script
* Specify the schema for table and stored procedure references in all stored procedure code
* Place set statements before any executing code in the procedure
* Place all declare statements before any other code in the procedure. This can aid in execution plan reuse
* Fully qualify all system stored procedures used in a stored procedure. This will optimize performance as the database engine doesn’t have to search for the system procedure
* In order to capture two or more SQL Server global variables pertaining to the same statement, declare a variable for each global to be identified and assign all in a single select immediately after the statement (capturing global variables one at a time will produce erroneous results):

DECLARE @RowCount INT

, @Error INT

DELETE FROM MyDB.dbo.MyTable

WHERE id NOT IN

(

SELECT Id

FROM MyDB.dbo.MyTable

)

SELECT @Rowcount = @@ROWCOUNT

, @Error = @@ERROR

* Use all UPPER case for system names, statements, variables, and functions:
  + Reserved words (**BEGIN, END, TABLE, CREATE, INDEX, GO, IDENTITY**)
  + Built-in types (**CHAR, INT, VARCHAR**)
  + System functions and stored procedures (**CAST, SELECT, CONVERT**)
* User all lower case for the following script objects:
  + System and custom extended stored procedures (**xp\_cmdshell**)
  + System and local variables (**@@error, @@identity, @value**)
  + References to system table names (**syscolumns**)
  + Table alias name within a SQL statement

### ERROR HANDLING

* Avoid abbreviations other than the specified prefixes and postfixes in error messages
* Use system messages stored in syscomments. Use the following outline for messages:

{fully qualified procedure name} : {message}

Example

MyDatabase.dbo.MyStoredPorcedure : A strange error has occurred?

* Error messages should be added to the system using the following outline:

sp\_addmessage msg\_id, severity, {message text}[,{language}[, 'with\_log'[,'replace']]]

* Capture the fully qualified procedure name by inserting the following code once at the beginning of each procedure that might raise the error:

DECLARE @sProcedureName VARCHAR(255);

SELECT @sProcedureName = DB\_NAME()

+ '.'

+ USER\_NAME(OBJECTPROPERTY(@@PROCID,'OwnerId')) + '.'

+ OBJECT\_NAME(@@PROCID)

* Assign error message numbers based in the following
  + Reserved 50000 thru 50999
  + General errors  51000 thru 51099
  + Import errors 52000 thru 52099
  + Export errors 53000 thru 53099
  + Services errors 58000 thru 58099
  + DbChangeControl Message 59000 thru 59099

## DATABASE SECURITY

* Use an NT login to access the server from the application

CREATE LOGIN [MyDomain\SomeUser] FROM WINDOWS;

* Permit the NT login only in application databases where access is required.

USE {database];

CREATE USER [MyDomain\SomeUser] FROM LOGIN [MyDomain\SomeUser];

* Place users within roles or groups to give them access to data

USE {database};

ALTER ROLE {role} ADD MEMBER [MyDomain\SomeUser];

* Grant access to data to groups or roles through execution of stored procedures

USE MyDB;

GRANT EXECUTE ON MyProcedure TO 'ApproptriateRole';

* Do not use SQL Server logins, including the **sa** login
* Do not grant database access to specific users, ALWAYs use active directory security groups (with the exception of service accounts)
* If object permissions are necessary, grant the permission to a role or group and add the correct users to the group or role
* Ideally stored procedures should insert, update or delete data. Where this method has not been used the reason for deviating from this best practice should be recorded and communicated to the DBA team
* Do not grant execute permission for any stored procedures that the user does not need to execute
* Encrypt procedures that may reveal database user permissions or passwords or otherwise compromise security if viewed. Do not encrypt procedures that do not meet these criteria

## FORMATING

### QUOTES

* Use single quote characters to delimit strings. Nest single quotes to express a single quote or apostrophe within a string

SET @sExample = 'Bill''s example'

### PARENTHESES

* Use parenthesis to increase readability, especially when working with branch conditions or complicated expressions

IF

(

(SELECT 1 WHERE 1 = 2) IS NOT NULL

)

BEGIN

END;

* Always use BEGIN…END blocks for all conditional code segments
* Limit the length of lines in all source code to 80 characters. If possible, try to leave all code viewable without the need to horizontally scroll an 800 x 600 IDE window using a 12 pitch Courier New font
* Indent one tab when indentation is required
* When nesting parentheses use new lines to separate each nest level

### WHITESPACE

* Use spaces so that expressions read like sentences: **fillfactor = 25**, not**fillfactor=25**
* Do not use white space in identifiers

### COMMENTS

* Use single-line comment markers where needed (--). Reserve multi-line comments (/\*..\*/) for blocking out sections of code
* Comment only where the comment adds value. Don't over-comment, and try to limit comments to a single line. Choose identifier names that are self-documenting whenever possible. An overuse of multi-line comments may indicate a design that is not elegant

### DML STATEMENTS (SELECT, INSERT, UPDATE, DELETE)

* Fully qualify all table references with the schema name
* Use ANSI join syntax

SELECT c.[Name]

, a.[Description]

FROM [User].[dbo].[TB\_ADDRESS] a

INNER JOIN [VIOLATIONS].[dbo].[TB\_INCIDENT] i

ON a.[Id] = [i.Address\_Id]

* Use ANSI operators

=, >, <, <>, IN, EXISTS, NOT, LIKE, IS NULL, AND, OR

* A correlated subquery using **exists** or **not exists** is preferred over the equivalent **in** or **not in** subquery due to performance degradation potential in some cases using **not in**.
* Avoid the use of cross joins if possible
* When a result set is not needed, use syntax that does not return a result set

IF EXISTS(SELECT 1 FROM EQUIPMENT.dbo.TB\_LOCATION WHERE [Type] = 50)

rather than:

IF ((SELECT COUNT(Id) FROM EQUIPMENT.dbo.TB\_LOCATION WHERE [Type] = 50) > 0)

* If more than one table is involved in a FROM clause, each column name must be qualified using either the complete table name or an alias. The alias is preferred
* Do not use the **identitycol** or **rowguidcol** to get the identity value of recenting inserted records
* Always use column names in an order by clause. Avoid positional references

### SELECT

* Do not use a select statement to create a new table (by supplying an into table that does not exist)
* When returning a variable or computed expression, always supply a friendly alias to the client

SELECT @@IDENTITY AS Exam\_Id

, (@pointsReceived / @pTotalPoints) as Average

* Opt for more descriptive alias (**SELECT @@identity AS UserId** is preferred over **SELECT @@identity AS Id)**
* Each column in the select list should appear on its own line. Each unrelated constraint within the where clause should appear on its own line
* When dealing with select statements used as conditions or subqueries, use more convenient formatting than the outline above where necessary. One line select statements are fine as long as they are easy enough to read
* Dependent constraints within the where clause should appear together offset by parenthesis. Use additional indentation if necessary

SELECT t.TASK\_ID

FROM Task.dbo.TASK t

INNER JOIN Task.dbo.ENROLLMENT et

ON t.TASK\_ID = et.TASK\_ID

WHERE et.MEMBER\_ID = @pMemberId

AND

(

(t.DUE\_DT <= @pStartDate)

OR (t.DUE\_DT >= @pEndDate)

OR (et.COMPLETED\_FLAG = 1)

)

* Variable values assigned by SELECT statements are processed in order. Therefore a variable can be reused

### INSERTS

* Always list column names within an insert statement. Never perform inserts based on column position alone
* Do not call a stored procedure during an insert as in:

INSERT Subscribe

EXECUTE Subscribers\_Buildnew\_System

* Place each column name and value on its own line and indent both so they match as shown
* Provide an inline comment to explain any hardcoded value

### UPDATES

* Each column in the SET clause should be on a new line with a preceding comma
* When more than 5000 rows of data will be updated a batch process must be employed (this is to prevent the transaction log from becoming full)

## TRANSACTIONS

When defining any new script or procedure, it is important to understand when transactions start and finish. Often the database engine is left encapsulate statements in a transaction (implicit transactions) which is usually sufficient for 80% of solutions.

This can become problematic when procedures or scripts become more complex. It is a generally accepted best practice to always define the transaction scope, or the unit of work the database engine should perform. The unit of work should include all statements

Mention isolation levels

* If an OLEDB client will manage a transaction always use

SET XACT\_ABORT ON;

to manage a connection.

If a transaction is necessary for a multi-statement operation, and the code will not be managed by an OLEDB client connection, use:

BEGIN TRANSACTION {trans name}

{statements}

IF {error}

BEGIN

COMMIT TRANSACTION {trans name}

END

ELSE

BEGIN

ROLLBACK TRANSACTION {trans name}

END

Distributed transactions should be avoided due to the additional overhead of the Microsoft Distributed Transaction Coordinator. The possibility of transaction escalation to a distributed transaction should be assessed when any of the following statements are true:

* + Within a .NET transaction:
    - A separate connection is made to two or more databases on the same or different servers and\or instances
    - A connection is made to one or more databases and the file system is manipulated
    - A connection is made to one or more databases and the registry is manipulated

## CONTROL OF FLOW STATEMENTS

* Use the following outline for if statements:

IF ({condition})

BEGIN

{statement}

.

.

.

{statement}

END

ELSE

BEGIN

IF ({condition})

BEGIN

{statement}

END

ELSE

BEGIN

{statement}

END

END;

* Use the following outline for while statements:

WHILE ({condition})

BEGIN

{statement}

.

.

.

{statement}

END

* The case construct is not equivalent to a switch in sequential processing. The SQL case statement is used to conditionally define a result set not to implement alternate processing
* Use the following outlines for case statements

SELECT CASE [{column or variable}]

WHEN {value | expression} THEN {result if this value}

WHEN {value | expression} THEN {result if this value}

ELSE {default result}

END

Example:

SELECT u.ID

, CASE u.[Type]

WHEN 10 then eu.Email

WHEN 20 then pu.Email

ELSE 'none provided'

END

FROM WebSite.dbo.Users u

INNER JOIN Users.dbo.TB\_ELECTRICIAN\_USER eu

ON u.ID = eu.ID

INNER JOIN Users.dbo.TB\_PARENT\_USER pu

ON u.ID = pu.ID

WHERE u.DISPLAY\_NAME = 'Smith, Bob'

* Nesting of CASE statements is not recommended, instead consider using lookup or reference tables to achieve value substitution

## CURSORS

The use of cursors is strongly discouraged. Nearly all DML operations can be performed using a SET based approach.

Cursors should only be considered in the following circumstances:

* Iterating through all databases is required
* Single row DML operations need to be performed
* Execution of a stored procedure based on each row of an object

Cursors are explicitly prohibited when used to return data to an application.

## LINKED SERVERS

The use of linked servers is explicitly prohibited. Where linked server objects already exist they will remain, however only in exceptional circumstances where no other viable alternative can be found can a linked server object be considered. Any solution requiring a linked server to be created have at least two of the following characteristics:

* Software supplied by a third party where the third party company:
  + Cannot or will not change the solution without significant investment
  + Is no longer in operation
  + Has technically proven the use of linked servers is more performant than other solutions
* Is isolated from all other database solutions
* Has a written performance degradation acceptance document from the users of the system
* Does not interact directly with ANY element of production processes or servers

Consider the following alternatives before opting for linked servers:

* ETL – moving the data to a more suitable location
* Replication – keeping point in time or real time read only copies of the data on a secondary server
* Virtualisation – using an abstraction layer to access the data

# SCHEMA SCRIPTS

For the purposes of deployment, scripts are supplied to the DBS team for review and execution first on pre-production, then following successful deployment, production systems. The table below describes the naming convention expected for the following deployment styles:

|  |  |  |
| --- | --- | --- |
| **Script type** | **Prefix** | **Example** |
| Schema deployment script | **dep\_** | **dep\_Calender.sql** |
| Conversion script | **conv\_** | **conv\_Schedule.sql** |
| Rollback script | **rbk\_** | **rbk\_Schedule.sql** |
| DACPAC |  | **<Database\_Name>.dacpac** |
| BACPAC |  | **<Database\_Name>.bacpac** |

* **GO** is the standard TSQL batch separator. Do not write scripts that depend on another separator. **GO** should appear on its own line of the script.
* Further, all pertinent blocks of code should be separated using the single line comment character repeated 80 times with an immediate comment explaining what the block of code does in the context of the whole script:

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

--<description of block of code>

* Each **dep\_** script should define only one table. The **dep\_** includes all constraints, keys, and indexes for the table. A **dep\_** script only creates an object. A **dep\_** script must not drop statements
* **Proc\_** and **conv\_** scripts should be able to be run multiple times against the same database with no adverse effect and no errors. Always check for the existence of each object before creating it again to avoid meaningless errors in the scripts output stream
* An **rbk\_** script must accompany every **conv\_** script. As with the **conv\_** script, the **rbk\_** script must be written so that it will execute multiple times with no adverse effects and no errors. The **rbk\_** script will reverse or rollback all changes applied in the **conv\_** such that full *a priori* functionality exists in all databases touched by the **conv\_** script

All files should be suffixed with the database version and file creation date and time as below:

**\_<Database\_Version>\_<File\_Creation\_DateTime>**

The file creation datetime should be expressed in the ANSI format of **yyyymmddhhmmss**. For example:

**proc\_Calendar\_v3\_23\_20160825131800.sql**

* Save all scripts using the **.sql**extension

### ROW COUNT VERIFICATION

A scenario seen many times by database administration teams is accidently missing a WHERE clause on UPDATE and DELETE statements. The impact of such a seemingly minor error can be quite large. In an effort to guard against this, all adhoc scripts must contain a section that verifies the number of rows affected by the DML statement, checked against a manually entered expected number.

This forces the developer of the script to consider how many rows are being altered before the script is run, drawing attention the necessity of a WHERE clause. Obviously with production, the number or rows can change before the script is run (after it has been through test cycles). In these circumstances a tolerance of 5% can be included in the script (so checks in the script for the row count allow a range rather than specific number of rows).

# ERROR NOTIFICATION

## EMAIL

When it is necessary to originate an administrative email message within a stored procedure, always reference an address associated to a sysoperators member. This will enable the same code to be used in development and testing without sending misleading messages to production personnel because the same operator can be configured to have a different email address in different software lifecycle environments. Members are added to sysoperators through the stored procedure **sp\_addoperator** (see Books on Line) or through the Enterprise Manager.

## INSTANT MESSENGER

Following the same messaging construct as that with emails, and providing the relevant integration layer is available (this could be in the form of a CLR function of external script), urgent messages can be sent to support personnel using instant messages. Currently the platform is Skype for business, however most instant messenger platform have an API which supports integration with monitoring applications. Some other messenger style platforms that could be used for immediate messaging are:

* Yammer
* SMS
* WhatsApp

## INCIDENT MANAGEMENT SYSTEM

In addition to notification by email or instant message, it may be prudent to log errors in an incident management system (currently RemedyForce). If the integration layer to the incident management system allows, the incident number should be included in the communication to the operational team. Whilst this option is recommended, combining integration layers with new developments may be beyond the skill level of some developers.

## LOG TABLE\WINDOWS EVENT LOG\LOG FILE

As with most front end applications, logging to a table or event log enables operational teams to subscribe to events generated by the process or procedure. Monitoring products such as SCOM can then be employed to watch for specific events defined by the procedure author which are bespoke to the process or application.

For example, if the database for a web application is responding slowly (for example 5 seconds per query), custom alerts can be sent to the event log so an engineer can investigate. An error of this type would not be visible to the users of the application (beyond the poor response time), however it would enact support operations more immediately.

# METHODOLOGY

## DEVELOPMENT ENVIRONMENTS

The tier 4 database environment consists of the following developments classes:

1. Development
2. Unit Test
3. System Test
4. UAT
5. PreProd (QA)

All development should move through these environments sequentially. If the developer or project have need to skip environments this should be communicated to the operational teams at least 2 full business days prior to deployment.

All database developers should use centralised dev\test labs for any “real life” business data. Local desktops should only be used if they are properly licensed for the database engine and the data being used is either obfuscated live data or generated.

All changes should be checked into Team Foundation Server or other appropriate source control system (should TFS be replaced). Network shares should be used as working directories, not the local file system.

System test is intended as a front end/middle ware/application server development tool. No database changes should be introduced from Unit Test to the Integration environment without moving through the change control system. If revisions or additional changes are needed, the source from the change control system must be checked out and the revisions completed and tested in the Unit Test environment. This will provide a greater level of stability for software developers than a chaotic environment where schema might change unbeknownst to an application developer in the middle of a test cycle.

Changes will be introduced to environments beyond **System Test** only via tested scripts and within the context of the company wide change control policy and procedures.

## DATA TYPE SELECTION

Incorrectly selected data types can make the most eloquently designed databases perform like a notepad and pen. Equally, correct use of data types can make accidental databases perform well enough to be functional.

The ramifications of data type selection are often overlooked, for example the use of DECIMAL(22,6) is pervasive within the company for values such as “Total Amount Payable” and “First Instalment Value”. This data type supports a value 20 digits to the left of the decimal point (assuming we are still storing a monetary value with a least two digits after the point) or a maximum value of 999 quintillion (3.87 million times the net worth of the world (2015)).

This particular data type requires 13 bytes of storage, which may not seem like a lot, however consider an agreement history table where this has been used 15 times in the table. Then consider this table has 850 million rows. Suddenly 13 bytes that could have been expressed using only 5 bytes equates to difference in storage of 94GB (((((13-5)\*15)\*850,000,000)/1024/1024/1024).

In summary, when selecting data types use the following as a guideline:

* Choose the smallest possible data type to reduce the storage footprint
* Use signed integer values for join attributes
* Use **SMALLDATETIME** for capturing the current date
* Use **DECIMAL (9,4)** for monetary values
* Consider data types can be increased in SQL Server without validation
* Do not use deprecated data types (**TEXT**, **NTEXT**, **IMAGE**)
* Do not use **NVARCHAR** or **NCHAR** unless working with languages other than English
* Avoid using **VARCHAR**(MAX) to prevent using overflow (LOB) data pages

## CLASS MODEL

The class model, or logical design of the database should, firstly, conform to the [Enterprise Data Standards](file:///C:\Users\higginp\AppData\Local\Microsoft\Windows\Temporary%20Internet%20Files\Content.Outlook\U8305IRN\Enterprise%20Data%20Standards.docx). In addition to the standards laid out in the EDS document, the class model should aid any interested party in discovering the following aspects of the data design:

* Associations (Relationships between classes)
* Direction of dependency (an arrow)
* Roles
* Aggregation
* Stored Procedures (expressed as operations)
* Parameters
* What is Returned

It is the purpose of the class model to form as the basis for all further generations of the database modelling process.

## CHANGE CONTROL

Introduction of changes to any SQL Server object can be in any combination of the following formats:

* DACPAC
* BACPAC
* SQL Script

All SQL Server DML, DDL creation and changes will be scripted and the scripts will be the vehicle to introduce additions or changes to the Integration environment and beyond. Changes may be implemented in the Unit Test environment using GUI tools or ad hoc techniques at the pleasure of the developer. Such changes will need to be scripted and the scripts tested before migrating them to the Integration environment. For this reason, it will generally be more efficient to always script changes and avoid SSMS or other non-scripted techniques.

A consistent configuration and strategy for moving objects through the change control system will assure clean and consistent versioning of the database schema and scripts. A general overview of the change control steps includes:

* Existing objects scripts are checked out of source control and the latest version is moved to the working directory. When checked out, include a brief description of the development project in the source control checkout dialog’s comment box for each object checked out
* The objectives are implemented in the Unit Test Environment
* Test plans are prepared/modified and executed for all stored procedures and scripts
* Return all objects to the source control system

All related objects should be kept checked out until all objectives are met for the development project. This will avoid conflicts created where incomplete development projects are included in release version.

In an effort to reduce errors, omissions, and surprises when changes are implemented into a production environment the Database Team will adhere to the following procedure at all times. These procedures will not only help protect The organization from the hazards of missed manual steps, but enable any member on the database team to provide support for any change and help us as we work to establish a standard for our Source control implementation. This procedure has the endorsement of QA and Production Operations.

All database changes will have a fully endorsed and properly numbered CR (Change Request) before the change is introduced to production. No Exceptions

All database changes included in a CR will be introduced via a single script with a .sql extension. The script will be tested before the change is delivered to QA. The script will be the method of delivery of the change to the designated PreProd server and then to all Production servers. This script will be stored in the Source control repository at the project path $/PROJECT/Database/Production/<server name>/Conversions/<CR number>/<CR number>.sql at the time the project is delivered to QA. The CR number will be added to the comments Source control attribute of the QA script once it is known.

A rollback procedure for all database changes will be outlined on the CR and will be available and fully tested as a single script named as the CR with an extension of .rbk when the change is delivered to PreProd. It will then be the option of Quality Assurance to test the rollback procedure as deemed appropriate. This script will be stored at the project path $/PROJECT/Database/Production/<server name>/Conversions>/<CR number>/<CR number>.rbk at the time the project is delivered to QA. The CR number will be added to the comments Source control attribute of the script once it is known

All database changes will include documentation as to recoverability. In general, this will mean that a Word document covering the steps needed to re-implement the changes should a new server need to be built or an existing server be lost or other considered scenarios where the changes need to be recreated will be produced and placed in the above mentioned Source control Project and any necessary scripts for this recovery process will be added or modified in support of the document

All database changes will be verified by a member of the database team and/or subjected to the database team's code review process prior to delivery to QA

## DEVELOPER INDEMNITY

Often the developer(s) of a particular release are cited as the reason for a failed release, when in reality there are other factors (such as differences in test environments) may have caused a failed release.

This can slow down finding the root cause of an issue because the release is assumed to have made the system malfunction. As well as allowing common flaws to be discovered before progression to production systems, providing detailed test evidence can help support engineers discount certain failure scenarios.

The sections below describe how the developer (and test teams) can best supply this information to the DBS team.

### UNIT TEST PLANS

All scripts and stored procedures must be tested before leaving the Unit test environment. A test plan is created or modified as needed and placed in the source control repository. The objective of the test plan is to document the procedure used to assure that every line of code does what it is supposed to do only when it is supposed to complete its instruction. There is a template for test plans at the test plans root of the Database Source control hierarchy. Any notes, observations, or deficiencies that should be recorded in the test plan.

The test document will define:

* The state of the database at the onset of the test cycle
  + Database Version (of user objects, from source control)
  + Log Sequence Number
  + Any static data requirements

Every line of code in the script or procedure must be executed in the test cycle. If revision is necessary, the test cycle should be restarted after changes are made. The default state will be that the database(s) will be freshly rebuilt and populated using generated (but static) test data, with a population level representative of the production system. At the beginning of each test cycle any further setup or configuration should be noted on the test document. Documentation of each individual test should (at a high level) have this structure at a minimum:

* All expected exit conditions within the script or procedure
* Each test required in the test cycle with expected result
* An explanation of any variance from the expected result
* The time test cycle completed and testers initial

### CODE REVIEW

The database team will conduct regular code reviews of all stored procedures and scripts. Code reviews are intended to improve the quality of the applications and as a learning experience for all database developers.

The code review will not be a tool used to evaluate any person, rather to focus on technical consideration of code. The review will not cover how a solution should be coded, rather it will examine the code to identify defects in coding standard compliance, logic, performance, portability, audit-ability, error handling, architectural compatibility, and security.

Code reviews will not have a static format, but will change as is deemed appropriate by the team.

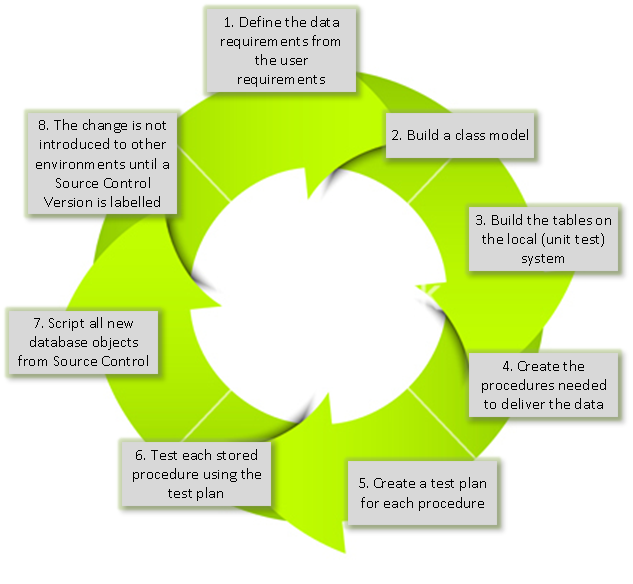
### EVALUATION

It is important for personal growth and for the efficacy of the team that all members are kept informed of their performance. An approved employee evaluation format that will follow issues, concerns and accolades around the coding standard will be included in the evaluation process.

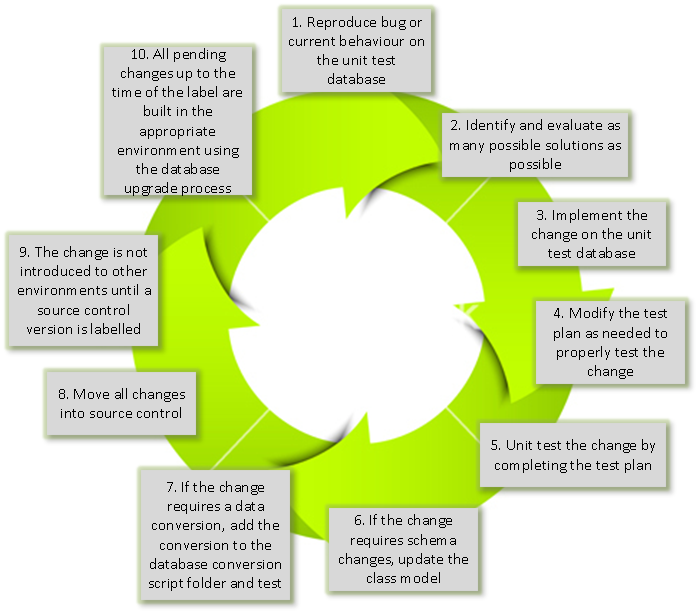
### DEVELOPMENT PROCESS

The diagrams in this section of the document are intended to represent development and bug fixing cycles that could be used. If another cycle is already in place this should be adhered to in place of this scheme. This section is only to provide a high level framework if one does not exist already.

For new developments (iterative process):



For bug fixes/enhancements (non-iterative process), follow this process:



## RELEASE PROCEDURE

When a development product has reached an appropriate point in its lifecycle, the guideline below should be used to ensure all relevant information is included in the change request.

1. Test evidence
   1. Gather the test evidence in the test plan template ensuring all expected and actual outcomes have been completed
2. Create the deployment package
   1. Version label in source control
   2. Application and database scripts
   3. Deployment guide
3. Check access
   1. Ensure the production environment has the required firewall and security permissions to function
4. Raise change request
5. Negotiate with release teams a suitable deployment window
6. Assign appropriate assessors
7. Submit for approval
8. Implement change

# QUALITY CHECKS

## TESTING

There are many different forms of testing, this document is not intended to be an exhaustive list of the tests that could be performed to ensure system stability. In the interest of assigning context to the testing template which accompanies this document, some high level test types will be discussed however the extent of testing and therefore the inclusion of various types of testing will have to be determined based on project budget and business risk appetite.

Some basic tests that are suggested to provide confidence in the unit being deployed are:

* Substitute all input parameters with the maximum value for the data type
* Non-functional\performance testing
  + Using applications such as SQLQueryStress, HammerDB andDistributed Relay to put the server under load using typical statements from the application
* Synthetic defect injection
  + Purposefully injecting defects to validate the test routines spot them
* System test
  + Running the process or system from end to end, showing all components function as intended

### TEMPLATE

A template is provided with this document to standardise the format of testing evidence. The format of the template is:

* Test Type
  + Compatibility
  + Conformance
  + Functional
  + Load
  + Performance
  + Regression
  + Stress
  + System
  + Unit
  + UAT
* Item to test\not to test
  + Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested
* Risks\Issues
  + Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans
* Test Approach
  + - Describe the overall testing approach to be used to test the project’s product. If there is a code solution to facilitate the test, provide the code in the next column. Some high level items the approach might include are:
      * Syntax
      * Description of Functionality
      * Arguments for Tests
      * Expected Output
      * Specific Exclusions
      * Dependencies
      * Success/Failure Criteria for Test Cases
      * Pass/Fail Criteria for the Complete Test Cycle
* Test Code
  + Provide any code snippets that will perform the test (if any)
* Regulatory Criteria
  + Describe any regulations or mandates that the system must be tested against
* Criteria for Pass
  + Describe the criteria used to determine if a test item has passed or failed its test
* Entry\Exit Criteria
  + Describe the entry and exit criteria used to start testing and determine when to stop testing
* Test Deliverable
  + Describe the deliverables that will result from the testing process (documents, reports, charts, etc.)
* Suspension Criteria
  + Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing
* Environmental\Staffing\Training Needs
  + Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).
* Test Result
  + Supply the result of the test based on the criteria for pass. Express this value as PASS\FAIL

## DEVELOPMENT GUIDELINES

### ADDING COLUMNS WITH DEFAULTS

Since SQL Server 2012, adding non-nullable columns to a table will no longer invoke an update to the table with the default value. This is now a meta-data operation (see sys.tables, columns has\_default and default\_value). The database engine now interprets any rows without a defined value as having the value of the default at run time.

### ALWAYSON

In SQL Server 2012, a new enterprise feature was introduced called AlwaysOn. This is a meld of three existing technologies, Windows Server clustering and SQL Server Database Mirroring and snapshot isolation.

SQL Server has been cluster aware from a great deal of time, however standalone instances of SQL Server previously had no “knowledge” of the cluster. A new resource type to support AlwaysOn is created in the Windows cluster when AlwaysOn is enabled. This allows SQL Server to integrate with the failover cluster for management of the listener. Effectively this works the same as a standard failover cluster in regard to the DNS name to connect with and the way that is moved between servers.

In addition, database mirroring is used to synchronise transactions on the two (or more) servers participating in the availability group. When a failover event occurs the direction of the mirror is reversed.

To support read only copies on the secondary server, SQL Server uses read committed snapshot isolation to allow a user to select data whilst simultaneously mirroring is writing to the database. This works by storing copies of the data being read by the user in Tempdb. One important consequence of this is, when a DDL statement is issued on the primary node (such as an index rebuild), this can in some cases cause select statements to fail when they attempt to issue a schema stability lock on objects in the read only copy. This is not a bug as the schema stability lock is designed to ensure the schema does not change while the SELECT statement is running.

### BULK LOADS\DISABLING INDEXES

Bulk loading data into SQL Server can be expedited in numerous ways. One common method is to disable non-clustered indexes on the table (sometimes even removing the clustered index can be of benefit). With the indexes disabled the database engine does not need to maintain the data pages as each write is made to the tables primary data pages. It also avoids the potential of page splits if the data isn’t insert in the sort order of the index (which would be nearly impossible for tables with multiple indexes).

As well as non-clustered indexes being disabled, it may be beneficial to drop the clustered index (do not disable the clustered index, this will make the table inaccessible). This is largely for the same reasons as disabling the non-clustered indexes.

### CASE CLAUSES

It is often necessary for simple data substitution to be included in views and stored procedures. Whilst a few simple CASE clauses do not cause much concern, as the complexity and nesting level increases the performance of the statement can be negatively affected.

In place of using CASE clauses, alternatives such as lookup or reference tables perform must better, more noticeably if a CASE clause has been used on a JOIN predicate or a WHERE clause.

### CLUSTERED INDEXES

In most circumstances tables should have a clustered index. In only a few scenarios is the absence of a clustered index beneficial. A table with no clustered index is often referred to a heap object.

Since SQL Server 2012 heaps can be rebuilt in nearly the same way indexes can be rebuilt (ALTER TABLE <table> REBUILD), however the issue with space reclamation still remains a problem with heaps. When deciding on columns to include in a clustered, assess carefully whether the primary key is the correct column(s) to use. In some circumstances other columns may offer better performance.

### COMMON LANGUAGE RUNTIME

Common Language Runtime or CLR, is a framework within SQL Server to allow user developed assemblies to be imported and referenced as functions or stored procedures. Whilst the bulk of business logic should ideally happen outside of the database, CLR allows a developer to create code that makes use of features in .NET that would be difficult of have security complications using TSQL.

For example, .NET code is typically much better at handling XML than the inbuilt SQL Server XML functions. .NET is also typically better at iterating through “rows” of data. For a TSQL statement looping is often referred to as RBAR, or row by agonising row, because the database engine is designed to work in a purely set based manner.

Consider using a CLR function or stored procedure when any of the following are true:

* Handling CSV, XML or JSON, particularly shredding into a normalised format
* String comparisons
* Calculating hash values (HASHBYTES calculates hash values differently to the cryptography assembly in .NET)

### CONTINUOUS DEPLOYMENT

Continuous deployment is the practice of moving through a cycle of development that is promoted to the production system many times a day. There are obvious concerns with this practice, particularly with regard to deployments causing data corruption.

When designing new solutions, give careful regard to generalising the system so most functions are driven from configuration. Consider also how new versions will be applied, can new features be installed “at the side” of the existing process?

For more information on continuous deployment, please refer to this article: <https://www.infoq.com/news/2014/03/etsy-deploy-50-times-a-day>.

### DATA MAINTENANCE AND HOUSE KEEPING

As an organisation, a lot of data is processed. Not all of this can be retained indefinitely, if not only because an ever expanding infrastructure would be too expensive to maintain, but also because legislation exists that mandates data (about any particular person) isn’t kept indefinitely.

With this in mind, data maintenance and\or archiving procedures should be included in **every** new system. If a system will never expand beyond its initial disk allocation, this should be recorded in the design under a section about data maintenance.

### DEPRECATED\DISCONTINUED FEATURES

When each new version of SQL Server is released, a certain number of features become deprecated while others are added to a list of features that will be deprecated in future. Below are some of the features that have been deprecated from SQL Server 2005 to SQL Server 2014:

TSQL

* In SQL Server 2005, trailing spaces specified in the first input parameter to the REPLACE function are trimmed when the parameter is of type**char**. For example, in the statement SELECT '<' + REPLACE(CONVERT(char(6), 'ABC '), ' ', 'L') + '>', the value 'ABC ' is incorrectly evaluated as 'ABC'. In SQL Server 2008, trailing spaces are always preserved. For applications that rely on the previous behavior of the function, use the RTRIM function when specifying the first input parameter for the function. For example, the following syntax will reproduce the SQL Server 2005 behavior SELECT '<' + REPLACE(RTRIM(CONVERT(char(6), 'ABC ')), ' ', 'L') + '>'.
* The GROUP BY clause cannot contain a subquery in an expression that is used for the group by list
* To prevent nondeterministic behavior, the OUTPUT clause cannot reference a column from a view or inline table-valued function when that column is defined by one of the following methods:
  + A subquery.
  + A user-defined function that performs user- or system-data access, or is assumed to perform such access.
  + A computed column that contains in its definition a user-defined function that performs user- or system-data access.
* The target table of the OUTPUT INTO clause cannot have any enabled triggers
* You cannot specify the READPAST hint under Snapshot Isolation
* WITH APPEND clause on triggers
* SET ROWCOUNT for INSERT, UPDATE, and DELETE statements
* Use of \*= and =\* join predicates
* COMPUTE / COMPUTE BY
* The old style syntax for RAISERROR (Format: RAISERROR integer string) syntax is deprecated
* FASTFIRSTROW hint
* Ability to return result sets from triggers
* HOLDLOCK table hint without parenthesis

XQuery

* Datetime support:
  + Values without timezone are validated.
  + The provided timezone or the absence of a timezone is preserved.
  + The internal storage representation is modified.
  + Resolution of stored values is increased.
  + Negative years are disallowed.
* In SQL Server 2005, steps in an XQuery or XPath expression that begin with a colon (':') are allowed. In SQL Server 2008, this usage is disallowed because it does not conform to XML standards

Plan Guides

In SQL Server 2008, if a plan guide cannot be honored, the query compiles using a different plan and no error is returned. In SQL Server 2005, an error is raised and the query fails.

### EXISTS VS IN

Whilst not always the case, it is often the opinion of database professionals using EXISTS is faster than IN. The key difference between these clauses is, for EXISTS, the database engine only needs to checks whether a value is present at all. Once the condition of the EXISTS statement is found to be true, the database engine stops searching. With IN however the engine must continue to find all results from the sub-query or list of values until the whole data set has been searched (which realistically should be satisfied by an index, however there is still additional load).

Obviously the performance of these two methods and the difference between them depends very much on the data. A table with only a few hundred rows will not yield much difference in processing time.

### EXTENDED PROCEDURES

Extended procedures are intended for use only by database administration teams. Use of extended procedures in any user objects is prohibited.

### FUNCTIONS IN JOINS

Where possible it is advisable to avoid using system functions in join clauses. Issuing this type of statement forces the data engine to evaluate each row against the function (as the outcome of the function is unknown until is it run). Because the optimiser does not know the result of the function until it has run, it is unable to use any indexes for seek operations. If any index does cover all the columns in the query, it will be scanned, if not, a table scan will be used.

### HARD CODED IDENTITY VALUES

When retrieving values from a lookup table or similar data structure, it is sometimes the choice of the developer to reference the primary key value of an entry. Sometimes this is because of a clustered index on the primary key. The problem with this practice is it assumes the ID values will never be changed. This is often the case, however were the table properly indexed to support WHERE clauses on other fields in the table, this would not become a concern.

It is best practice to not reference ID fields in application code (this includes ETL packages) but instead to use another identifying attribute.

### INDEX FILL FACTOR

Index fill factor describes the percentage of each data page of indexes. Ordinarily the default value can be used for a particular system. However, when dealing with high throughput OLTP systems or any application that has a large number of small non-sequential writes.

In these cases, reducing the fill factor may be beneficial as it allows data to be written into the empty portion of the data page without the need for a new index page to be created (and therefore a page split to occur).

### LARGE UPDATES\DELETES

After a new system has been procured and deployed, usually it’s data footprint is quite small. Issuing updates and deletes largely doesn’t pose much of a risk. However as systems age and inevitably grow, the same UPDATE and DELETE statements can become less manageable. All DML statements generate transaction logs, either on a temporary basis for database in SIMPLE recovery, or until they are backed up in the case of FULL and BULK LOGGED recovery modes.

When archiving of the data is considered, it is likely the data will be copied to another location or format and the original data deleted. This DELETE operation is one of the most likely operations to fill the transaction log in a single transaction. In these circumstances examine whether undergoing the same process in batches would be more suitable. An example batch script is included in the appendix (BULK DELETE).

### LAST PAGE CONTENTION

When data is written sequentially to a table with a clustered index on the sequential attribute (such as an identity column), data is added to the last data page until it is full, then a new page is created. Because all new data is being added to one 8Kb page at a time the number of writes is limited to how fast the storage system can write one block (which may be larger than 8Kb). At a certain volume of transactions blocking will start to occur as the storage system competes with the volume of writes.

Though it may seem counter-intuitive, using a random guid for the clustered index column spreads the load of writes across multiple 8Kb pages, which in test scenarios can increase write throughput by as much as 400%. It is worth nothing this type of design is only relevant for high volumes of transactions and where the fill factor of the clustered index has been tuned to account time period between index rebuilds (otherwise page splits begin to occur which would negatively impact performance).

### LOGICAL REPRESENTATION

Though this activity is normally part of the data design process, the prevalence of ELT (Extract Load Transform) processes in Provident meant including a section on logically representing data in views relevant. This does not mean views are the answer to all data movement tasks, however when designing and new data movement solution, the developer should evaluate how best to represent the data to the interfacing package. Though slightly out of scope of this document, transformation of the data whilst being processed in the data movement package is strongly encouraged over UPDATE statements on otherwise “static” data.

### MULTI STATEMENT\INLINE FUNCTIONS

A multi statement function is a SQL server function that can contain more than one statement, but can also contain more than one action. The example below shows what appears to be a single statement, however the function must declare the table variable used to return the data:

CREATE FUNCTION Util.MyMTVFunction (@Parameters INT)

RETURNS @FunctionResultTableVariable TABLE (N INT)

AS

BEGIN

INSERT INTO @FunctionResultTableVariable (N)

SELECT TOP (@Parameters) N

FROM Util.Tally

ORDER BY N;

RETURN;

END

GO

As an inline function this can be expressed as:

CREATE FUNCTION Util.MyITVFunction (@Parameters INT)

RETURNS TABLE

AS

RETURN

SELECT TOP (@Parameters) N

FROM Util.Tally

ORDER BY N;

GO

The key difference between these two methods is the multi statement function must be called for each row because the result cannot be assessed prior to it being run. The inline function however can be assessed, the statistics of the table it accesses can be incorporated in the execution plan. In this way inline functions behave more like views and are offered all the benefit of the optimizer “filling the gaps” that the multi statement function is not.

### (N)VARCHAR(MAX)

The MAX parameter of the VARCHAR and NVARCHAR data types were introduced to replace TEXT and NTEXT (which will be deprecated). When more the 8Kb needs to be stored in a single value, SQL Server adds a pointer to an overflow data page (LOB page). This overflow page has some unfortunate restrictions, such as:

* Cannot be compressed
* Aren’t “moved” with ONLINE index rebuilds

### PREVENTING DEADLOCKS

A deadlock is an event in a database engine when two or more connections request a lock on one or more resources in a sequence that means neither are able to obtain the lock level they require until the other connection completes. In these circumstances the database engine will rollback the transaction that requires the least “effort” to undo, and will then return a deadlock event.

Because code and object structure can be quite complex, there are many scenarios where deadlocks can occur. There are however a few common scenarios such as:

* Reader-Writer
  + Lock conversion: When an UPDATE statement is issued, the database engine will “search” for the
* Writer-Writer
* Key Lookup

Some ways to prevent deadlocks from occurring are:

* Adding WHERE clauses to UPDATE and DELETE statements to prevent table scans
* Adding supporting indexes so encourage index seeks (again reducing table scans)
* Using explicit transactions (BEGIN TRANSACTION) to
* Consider using DEADLOCK\_PRIORITY to force statements to take precedence over others (if feasible)
* Consider increasing the isolation level (with caution, while this may prevent deadlocks, it will decrease concurrency)
* Using SNAPSHOT isolation, this isolation level enables different transactions to maintain their own consistent version of the data for the duration of the transaction. This can cause addition load on Tempdb so should be carefully considered before implementation

### PRIMARY FILE GROUPS

The primary file group of a new database contains header pages that describe the database itself and hold the system objects.

A consequence of the primary filegroup containing system data pages is they cannot be moved to another file. Therefore, if a user data is stored here it also cannot be transferred (unless the clustered index is drop and recreated).

### READ COMMITTED SNAPSHOT AND SNAPSHOT ISOLATION

Since SQL Server 2005, snapshot isolation for transactions has been an available feature. This isolation level maintains copies of the data being used by a particular connection in Tempdb. This means each connection (or transaction) has its own private version of the data when the transaction began. Because Tempdb is used as the holding area for these rows, particular attention should be paid to tuning Tempdb correctly before enabling this feature.

### SPARSE COLUMNS

SPARSE columns were introduced into SQL Server in version 2008 R1. When an attribute of a table has 10% or less population, using SPARSE can significantly reduce the storage capacity required for the table. Give careful consideration when applying this feature as columns that contain more data and are set to SPARSE can use more disk space and also perform poorly.

### SUPPORTING DELTA LOADS

In nearly all cases, for business intelligence developers, it is necessary to identify “what has changed” since the last time the source system was observed. This allows the BI developer to take incremental feeds of data at regular intervals.

SQL Server offers several features that enable existing systems to mimic incremental data feeds even if it hasn’t been considered in the design. For example:

* Change Tracking
  + This implements a set of hidden tables that record which rows have been changed by the rowid. When data is requested the rows that have changed can be identified, however only the current value of the row can be obtained
* Change Data Capture
  + CDC is essentially mining of the live transaction. Changes are read from the transaction log in sequence, meaning the full history of changes to a row can be obtained. The danger with this feature is, if the data is collected from CDC the transaction log will continue to grow
* StreamAnalytics
  + Purchased by Microsoft and included as a separate installation file, StreamAnalytics is SQL Servers CEP (Complex Event Processing) engine. Instead of querying static data, these engines work by taking data in flight and passing it through “standing” queries, storing the results in a target data structure

All of these features of SQL Server usually work very if used properly. It would however be preferable to bake this type of functionality into the design of new systems. Inclusion of the following in the base design would make identification of changed data less arduous:

* Rowversion
  + This is an ever increasing number represented at as binary value. If an ETL process records the last rowversion is collected and then uses this as starting point in subsequent collections, new or changed rows can be identified. This method can also be used to assess differences between copies of the data
* Last Updated DateTime
  + It is fairly common to see date and time fields in tables. Often though the DateTime series is relevant to the data in the row (such as a transaction date). The record could be updated
* Rowhash
  + Adding this attribute to a table implies it is populated by either the front end application (which can be SSIS) or by a CLR function. This value represents a hash of the entire row. It allows the existence of the row to be easy tested against a target system. When combined with the primary key of a table, changed rows can be easily identified
* Logical Delete Date
  + In the same way some slowly changing dimensions work in data warehouses, rather than deleting records, logically removing them but leaving the row in place allows for data integration teams to collect all previous versions of the row then cleaning up when it has been collected
* GUID
  + A globally unique identifier uses quite a large data type (uniqueidentifier is 16 bytes) in SQL Server. Whilst this is a significant amount of space per row, if relevant, using a GUID and preserving it throughout ETL operations into target systems (which may require junction tables to represent where the data has been used for aggregations) allows the journey of data to be traced

### TABLE COMPRESSION

In SQL Server 2012, for enterprise editions table compression was introduced. The technology utilises Vertipaq compression, a feature initially introduced into Microsoft Excel to support pivoting of large amounts of data (PowerPivot), and is also the basis for Tabular SQL Server Analysis Services.

The first action of table compression is to assess whether Unicode data types, such as NVARCHAR can be reduced to VARCHAR (though this happens behind the scenes so the logical table structure isn’t altered). Then the data itself

### TABLE VARIABLES\HASH TABLES

If possible, the use of any temporary object is not advised. It is preferable to make use of sub-query, functions, common table expressions or views instead. It is the responsibility of the developer to explore all such options before considering using hash tables or table variables.

When the developer has researched all these options and decided the use of temporary objects would be beneficial, there are some important performance considerations that could vastly alter the run time and resource usage of a query. For example:

* If a hash table is created and the schema subsequently changed, the table is no longer eligible for reuse. SQL Server keeps the data the temp table held and if it is requested again, satisfies the request from the data already in TempDb
* Table variables do not store data in memory. It is held in TempDb in a similar way to hash tables
* The optimiser interprets table variables as only having one row. This is because table variables do not have statistics like hash tables
* Table variables should only be used for very small data sets
* Hash tables should be used where understanding the data distribution is relevant and\or where indexes are required
* **Global temp tables should never be used (##Table\_Name)**

### TABLE VALUE PARAMETERS

In SQL Server 2012 a new type of parameter for stored procedures and functions. A database type of table can be defined, which specifies the table structure in much the same way as table variables. An application can then reference this as a “structured” data type and pass a table (or object) as a parameter. This makes CRUD procedures much easier to implement. Rather than specifying a parameter for each target column having to execute the procedure for each row. Now multiple rows can be sent to a stored procedure from an application.

### TRUSTED FOREIGN KEYS

Foreign key constaints are used in SQL Server to enforce referential integrity across tables. When a new record is inserted into a table, the database engine checks the reference table for the existence of the key value. It is fairly common practice for foreign keys to be disabled when performing bulk loads into the table. When the foreign key is enabled after the load, unless specified to do so, SQL Server will not check the values in reference table. This leaves the foreign key in a untrusted state, meaning the query optimiser will adjust any future execution plans to perform the check (usually creating a plan with a table or index scan).

To prevent this happening, when enabling the foreign key use the following syntax:

ALTER TABLE dbo.Test WITH CHECK CHECK CONSTRAINT FK\_Test;

### UNICODE DATA TYPES

SQL Server supports Unicode characters with the data type NVARCHAR. In most circumstances the use of this data is not necessary, most solutions that are designed for operation within the UK do not make use of any Unicode characters. VARCHAR is sufficient for the majority of solutions.

The key disadvantage to NVARCHAR is each character requires 2 bytes of storage, whilst VARCHAR only requires 1 byte. Not only does this double the disk space required, but this also has the effect of doubling the number of disk reads required to access the data, double the memory to store the data for consumption by the client, and double the data to transmit over the network.

Not only does this expand the disk storage requirement, further it consumes the same space in memory, effectively meaning more memory is required to handle the same data (assuming NVARCHAR has been used incorrectly). In short more data has been moved around, meaning more resources are consumed in nearly every component of a server.

### VERSION CONTROL

The benefits of maintaining a version number for any particular solution can seem arduous, particularly ones not kept in a source control system. However, it does offer a qualitative method for assessing the deployment state of a solution. If objects within the solution are version controlled this offers even greater control over release granularity.

### VIRTUALISATION SUPPORT

Though data virtualisation has not yet been introduced into the production environment, it is feasible a product or suite of products will be used to abstract users and applications from the underlying databases and database engines.

This means consideration should be given to how abstraction products will interface with systems. What endpoints will it interface with? how will it discover meta data and present that to users? how easy is it to change the definition of the endpoint? All of these should be factored into the design of any new service or system, and effort to implement assessed for existing systems as they are redeveloped or improved upon.

### XML AND SEMI-STRUCTURED DATA

Where possible, storage of XML and other semi-structured data (such as JSON) should not be stored with a database. Obviously there are exceptions to this, such as small configuration databases, or where small strings relevant to the record are being stored (less than 4000 characters).

If semi-structured data cannot be avoided, consider using FileStream or FileTables to store the text in the file system rather than the database itself.

# REFERENCES

## BESPOKE STANDARDS

<http://www.sourceformat.com/pdf/sql-coding-standard-sqlserver.pdf>

<https://msdn.microsoft.com/en-us/library/aa260844(v=vs.60).aspx>

<http://www.nyx.net/~bwunder/dbChangeControl/standard.htm>

<http://www.slideshare.net/nCodeIT/database-development-coding-standards>

<https://msdn.microsoft.com/en-us/library/ms175874.aspx>

## BEST PRACTICES

<http://stackoverflow.com/questions/2065329/sql-server-in-vs-exists-performance>

<https://www.simple-talk.com/sql/database-administration/foreign-keys-and-their-states/>

<https://blogs.msdn.microsoft.com/sqlserverstorageengine/2006/11/13/enabling-vardecimal-storage-format/>

<https://msdn.microsoft.com/en-us/library/ms143179(v=sql.100).aspx>

<https://msdn.microsoft.com/en-us/library/ms143179(v=sql.105).aspx>

<https://www.brentozar.com/blitz/foreign-key-trusted/>

<http://sqlblog.com/blogs/hugo_kornelis/archive/2007/03/29/can-you-trust-your-constraints.aspx>

<https://www.simple-talk.com/sql/performance/sql-server-deadlocks-by-example/>

<https://www.infoq.com/news/2014/03/etsy-deploy-50-times-a-day>

## BOOKS

Code Complete by Steve McConnell (Microsoft Press, 1993).

## DBS STANDARD OPERATING PROCEDURES

SOP013 Database Meta Data ([\\hosfixity\devdata$\DBA\002 Documentation\001 Standard Operating Procedures](file:///\\hosfixity\devdata$\DBA\002%20Documentation\001%20Standard%20Operating%20Procedures))

## FUNCTIONS

<http://www.sqlservercentral.com/blogs/discussionofsqlserver/2012/02/15/comparing-inline-and-multistatement-table-valued-functions/>

## INDEXES

<http://www.sqlskills.com/blogs/kimberly/understanding-lob-data-20082008r2-2012/>

## NOTATION

<http://www.omg.org/spec/DMN/20151101/ch11example.xml>

## STANDARDS ORGANISATIONS

<https://www.oasis-open.org/>

<http://www.odbms.org/odmg-standard/>

<https://www.ansi.org/>

<http://it-cisq.org/>

## STANDARDS DOCUMENTS

<http://www.omg.org/spec/DMN/1.1/PDF/>

<http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45342>

<http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=16549>

<http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html>

## TESTING

<http://www.sei.cmu.edu/productlines/ppl/unit_test_plan_template.html>

<https://en.wikipedia.org/wiki/Test_plan>

[www.uml.org.cn/**test**/utp\_template.doc](http://www.uml.org.cn/test/utp_template.doc)

<http://www.softwaretestingstuff.com/2007/10/test-efficiency-vs-test-effectiveness.html>

# APPENDIX A

The scripts below can be used to create tables within an existing database. All scripts use template parameters in SQL Server Management Studio (Query 🡪 Specify Values for Template Parameters…).

## ALTER TABLE

* Use the following outline to drop an existing column or table constraint

IF (OBJECTPROPERTY(OBJECT\_ID('{constraint name}'),'IsConstraint') IS NOT NULL)

BEGIN

ALTER TABLE {table name} DROP CONSTRAINT {constraint name}

END;

* Use the following outline for adding or changing an existing column or table constraint. (Just drop it first if your changing it.)

IF (OBJECTPROPERTY(OBJECT\_ID('<Constraint\_Name, sysname, Constraint\_Name>'),'IsConstraint') IS NULL)

BEGIN

ALTER TABLE [<Security\_Schema, sysname, dbo>].[<Table\_Name, sysname, Table\_Name>]

ADD CONSTRAINT [<Constraint\_Name, sysname, Constraint\_Name>]

DEFAULT <Constraint\_Value, sysname, Constraint\_Value>

FOR <Constraint\_Column, sysname, Constraint\_Column>

END;

* Use the following outline to drop an existing column

IF (COLUMNPROPERTY(OBJECT\_ID('{table name}'),{column name},'AllowsNull') IS NOT NULL)

BEGIN

ALTER TABLE {fully qualified table name} DROP {column name}

END;

* Use the following outline for adding a column

IF (COLUMNPROPERTY(OBJECT\_ID('{table name}'),{column name},'AllowsNull') IS NULL)

BEGIN

ALTER TABLE {fully qualified table name}

ADD {column name} {data type} {NULL | NOT NULL} [constraint {default name}

DEFAULT ({default value})]

END;

* Use the following outline for altering a default on an existing column. (Note that It may be necessary to remove foreign keys before altering a column and in a few cases it may be necessary to duplicate the table with the desired new structure and move the data to the new table)

IF (COLUMNPROPERTY(OBJECT\_ID('{table name}'),{column name},'AllowsNull') IS NOT NULL)

BEGIN

ALTER TABLE {fully qualified table name} [with nocheck]

ALTER COLUMN {column name} {new compatible data type}

END;

Alter table examples:

IF (COLUMNPROPERTY(OBJECT\_ID('{table name}'),{column name},'AllowsNull') IS NOT NULL)

BEGIN

ALTER TABLE dbo.TB\_DESIGNS DROP COLUMN Color\_Id

END;

Or

IF (OBJECTPROPERTY(OBJECT\_ID('fk\_Educator\_Id\_User\_Id'),'IsConstraint') IS NOT NULL)

BEGIN

ALTER TABLE dbo.TB\_EDUCATOR\_SCHOOL DROP CONSTRAINT fk\_Educator\_Id\_User\_Id

END

IF (OBJECTPROPERTY(OBJECT\_ID('fk\_Educator\_Id\_User\_Id'),'IsConstraint') IS NULL)

BEGIN

ALTER TABLE dbo.TB\_EDUCATOR\_SCHOOL ADD CONSTRAINT fk\_Educator\_Id\_User\_Id

FOREIGN KEY (Educator\_Id) REFERENCES TB\_EDUCATOR (User\_Id)

END

## CREATE STORED PROCEDURE

-- Drop stored procedure if it already exists

IF EXISTS

(

SELECT \*

FROM INFORMATION\_SCHEMA.ROUTINES

WHERE SPECIFIC\_SCHEMA = N'<Schema\_Name, sysname, Schema\_Name>'

AND SPECIFIC\_NAME = N'<Procedure\_Name, sysname, Procedure\_Name>'

)

DROP PROCEDURE <Schema\_Name, sysname, Schema\_Name>.<Procedure\_Name, sysname, Procedure\_Name>

GO

CREATE PROCEDURE <Schema\_Name, sysname, Schema\_Name>.<Procedure\_Name, sysname, Procedure\_Name>

<@param1, sysname, @p1> <datatype\_for\_param1, , int> = <default\_value\_for\_param1, , 0>,

<@param2, sysname, @p2> <datatype\_for\_param2, , int> = <default\_value\_for\_param2, , 0>

AS

/\*\*

\*-----------------------------------------------------------------------------\*

| <Procedure\_Name, sysname, Procedure\_Name> |

| |

| Version 1 <Author, sysname, Your Name> <Version\_1\_Date, sysname, 10/08/2016> |

| |

\*-----------------------------------------------------------------------------\*

COMPATIBLITY

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

<Minimum\_SQL\_Version, sysname, 2008> upwards

\*-----------------------------------------------------------------------------\*

BACKGROUND

----------

Describe the reason the script has been created. this section should include

as much information about the

\*-----------------------------------------------------------------------------\*

USAGE

-----

Give examples of how to execute the script. Are there any special conditions

that should exist beforehand (should the script test for those first?).

This section is particularly important when this code block is used in

stored procedures.

EXECUTE <Schema\_Name, sysname, Schema\_Name>.<Procedure\_Name, sysname, Procedure\_Name>

<value\_for\_param1, , 1>

, <value\_for\_param2, , 2>

GO

\*-----------------------------------------------------------------------------\*

VERSION 1

---------

List all features of the first version

VERSION 2

---------

list all changes made to the script since version 1

\*-----------------------------------------------------------------------------\*

REFERENCES

----------

Include any references to external sites (if code has been copied) or to

any other internal policies or guidelines

\*-----------------------------------------------------------------------------\*

PARAMETERS

----------

List all parameters in the stored procedure

\*-----------------------------------------------------------------------------\*

DEPENDENCIES

----------

List all scripts or code this script relies upon. For example this could

include scripts that need to be run before this one.

\*\*/

SET XACT\_ABORT ON;

SET NOCOUNT ON;

SET ANSI\_NULLS ON;

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

--Start of TRY block

BEGIN TRY

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

DECLARE @Variable\_1 VARCHAR

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

{body of procedure}

RETURN

END TRY

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

--Start of CATCH block

BEGIN CATCH

DECLARE @Event\_Data XML

, @Error\_Line INT = 0

, @Error\_Number INT = 0

, @Error\_Severity INT = 0

, @Error\_State INT = 0

, @Error\_Message VARCHAR(4000)

SET @Event\_Data = EVENTDATA()

SET @Error\_Line = ERROR\_LINE()

SET @Error\_Number = ERROR\_NUMBER()

SET @Error\_Severity = ERROR\_SEVERITY()

SET @Error\_State = ERROR\_STATE()

SET @Error\_Message = ERROR\_MESSAGE()

EXECUTE <Logging\_Proc\_Schema, sysname, dbo>.<Logging\_Proc\_Name, sysname, usp\_Log\_Error>

@Event\_Data = @Event\_Data

, @Error\_Line = @Error\_Line

, @Error\_Number = @Error\_Number

, @Error\_Severity = @Error\_Severity

, @Error\_State = @Error\_State

, @Error\_Message = @Error\_Message

END CATCH

GO

Example:

USE Subscriptions;

IF (OBJECTPROPERTY(OBJECT\_ID('dbo.UnsubscribeMagazine'),'IsProcedure') IS NOT NULL)

BEGIN

DROP PROCEDURE dbo.UnsubscribeMagazine

END;

GO

CREATE PROCEDURE dbo.usp\_UnsubscribeMagazine

(

@pMagazineId INT

, @pUserId INT

)

AS

/\*\*

\*-----------------------------------------------------------------------------\*

| usp\_UnsubscribeMagazine |

| |

| Version 1 Dale Stewart 01/09/2016 |

| |

\*-----------------------------------------------------------------------------\*

COMPATIBLITY

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

<Minimum\_SQL\_Version, sysname, 2008> upwards

\*-----------------------------------------------------------------------------\*

BACKGROUND

----------

Describe the reason the script has been created. this section should include

as much information about the

\*-----------------------------------------------------------------------------\*

USAGE

-----

Give examples of how to execute the script. Are there any special conditions

that should exist beforehand (should the script test for those first?).

This section is particularly important when this code block is used in

stored procedures.

EXECUTE dbo.usp\_UnsubscribeMagazine

@pMagazineId = 1

, @pUserId = 1

GO

\*-----------------------------------------------------------------------------\*

VERSION 1

---------

List all features of the first version

VERSION 2

---------

list all changes made to the script since version 1

\*-----------------------------------------------------------------------------\*

REFERENCES

----------

Include any references to external sites (if code has been copied) or to

any other internal policies or guidelines

\*-----------------------------------------------------------------------------\*

PARAMETERS

----------

List all parameters in the stored procedure

\*-----------------------------------------------------------------------------\*

DEPENDENCIES

----------

List all scripts or code this script relies upon. For example this could

include scripts that need to be run before this one.

\*\*/

SET XACT\_ABORT ON;

SET NOCOUNT ON;

SET ANSI\_NULLS ON;

BEGIN TRY

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

DELETE Subscriptions.dbo.TB\_MAILDROP

WHERE Magazine\_Id = @pMagazineId

AND User\_Id = @pUserId;

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

DELETE Magazines.dbo.TB\_SUBSCRIBERS

WHERE Magazine\_Id = @pMagazineId

AND User\_Id = @pUserId;

END TRY

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

--Start of CATCH block

BEGIN CATCH

DECLARE @Event\_Data XML

, @Error\_Line INT = 0

, @Error\_Number INT = 0

, @Error\_Severity INT = 0

, @Error\_State INT = 0

, @Error\_Message VARCHAR(4000)

SET @Event\_Data = EVENTDATA()

SET @Error\_Line = ERROR\_LINE()

SET @Error\_Number = ERROR\_NUMBER()

SET @Error\_Severity = ERROR\_SEVERITY()

SET @Error\_State = ERROR\_STATE()

SET @Error\_Message = ERROR\_MESSAGE()

EXECUTE <Logging\_Proc\_Schema, sysname, dbo>.<Logging\_Proc\_Name, sysname, usp\_Log\_Error>

@Event\_Data = @Event\_Data

, @Error\_Line = @Error\_Line

, @Error\_Number = @Error\_Number

, @Error\_Severity = @Error\_Severity

, @Error\_State = @Error\_State

, @Error\_Message = @Error\_Message

END CATCH

RETURN;

## CREATE TABLE

CREATE TABLE [<Database\_Name, sysname, your\_database\_name>].[<Security\_Schema, sysname, dbo>].[<Table\_Name, sysname, Table\_Name>]

(

[<Table\_Name, sysname, Table\_Name>\_Id] [<ID\_Data\_Type, sysname, INT>] IDENTITY(<ID\_Seed, INT, -2147483648>, <ID\_Increment, INT, 1>)

, [<Column\_Name\_1, sysname, Column 1 Name>] [<Data\_Type\_1, sysname, VARCHAR(50)>] {NULL | NOT NULL}

, [<Column\_Name\_2, sysname, Column 2 Name>] [<Data\_Type\_2, sysname, VARCHAR(50)>] {NULL | NOT NULL}

CONSTRAINT dft\_<Table\_Name, sysname, Table\_Name>\_\_<Column\_Name\_2, sysname, Column 2 Name>

DEFAULT (<Default\_Constraint\_Value, sysname, Default\_Constraint\_Value>)

, [<Column\_Name\_3, sysname, Column 3 Name>] [<Data\_Type\_3, sysname, VARCHAR(50)>] {NULL | NOT NULL}

, CONSTRAINT [pk{c | n}\_<Table\_Name, sysname, Table\_Name>\_\_<Primary\_Key\_Col, sysname, Primary\_Key\_Col>]

PRIMARY KEY {CLUSTERED | NONCLUSTERED} (<Primary\_Key\_Col, sysname, Primary\_Key\_Col>)

, CONSTRAINT [fk\_<Table\_Name, sysname, Table\_Name>\_to\_<FK\_Dest\_Table, sysname, FK\_Dest\_Table>]

FOREIGN KEY <FK\_Source\_Table\_Col\_List, sysname, Column\_Name\_1>

REFERENCES <FK\_Dest\_Table, sysname, FK\_Dest\_Table>(<FK\_Dest\_Table\_Col\_List, sysname, FK\_Dest\_Table\_Col\_List>)

, CONSTRAINT [unq\_<Table\_Name, sysname, Table\_Name>\_\_<Unique\_Key\_Col\_List, sysname, Column\_Name\_1, Column\_Name\_2, Column\_Name\_3>]

UNIQUE KEY (<Unique\_Key\_Col\_List, sysname, Column\_Name\_1, Column\_Name\_2, Column\_Name\_3>)

, CONSTRAINT [chk\_<Table\_Name, sysname, Table\_Name>\_\_<Check\_Expression\_Col\_List, sysname, Column\_Name\_1, Column\_Name\_2>]

CHECK (<Check\_Expression, sysname, Column\_Name\_1 <= Column\_Name\_2>)

)

ON ([<Data\_Filegroup, sysname, Not the PRIMARY filegroup>]);

Example:

CREATE TABLE [AdventureWorks].[dbo].[Sales]

(

[Sales\_Id] [INT] IDENTITY(-2147483648, 1)

, [Quantity] [SMALLINT] not null

, [Price] [MONEY] not null CONSTRAINT dft\_Sales\_\_Price DEFAULT (0)

, [SalesPerson\_ID] [INT] not null

, CONSTRAINT pkc\_Sales\_\_Sales\_ID PRIMARY KEY CLUSTERED (Sales\_ID)

, CONSTRAINT fk\_Sales\_\_SalesPerson\_ID\_\_to\_\_SalesPerson\_\_SalesPerson\_ID FOREIGN KEY [SalesPerson\_ID] REFERENCES SalesPerson([SalesPerson\_ID])

, CONSTRAINT chk\_Sales\_\_Price CHECK ([Price] > 0)

)

ON ([DATA])

## DML

### DELETES

* Use the following outline for simple delete statements. Format the where clause as described earlier.

DELETE FROM {database name}.{owner}.{table name}

WHERE {clause}

Example

DELETE FROM WebLog.dbo.TB\_ARTICLE\_STATISTICS

WHERE ARTICLE\_ID = @pArticleId

Use the following outline for table-driven delete statements. Use a subquery (formatted as described earlier) rather than using the TSQL extension form.

DELETE [from] {database name}.{owner}.{table name}

WHERE [NOT] EXISTS {correlated subquery expression}

Example:

DELETE WebLog.dbo.TB\_ARTICLE\_STATISTICS TAS

WHERE EXISTS

(

SELECT ID

FROM ARTICLES.dbo.TB\_EXPIRED

WHERE ARTICLE\_ID = TAS.ARTICLE\_ID

)

### INSERT

Use the following outline for inserts that move data from one table to another. Break and indent the column lists so they match. Apply the same formatting to the from clause as described in the select statement

INSERT INTO {schema}.{table name} | {alias}

(

{column name}

, {column name}

,[…]

)

SELECT

{column name}

, {column name}

, […]

FROM {clause}

WHERE {clause}

* Example:

INSERT INTO dbo.TB\_TRACTOR

(

Tractor\_Id

, Manufacturer\_Id

, Name

)

SELECT

Id

, @sBoltId

, 'plow bolts' -- name from vendor catalog

FROM dbo.HeavyDuty

WHERE Id = @pTractorId

INSERT [INTO] {database name}.{owner}.{table name}

(

{column name}

, {column name}

, …

)

VALUES

(

{value or variable} --{comment hard coded value}

, {value or variable} --{comment hard coded value}

, …

)

Example:

INSERT INTO Parts.dbo.TB\_TOASTER

(

TOASTER\_ID

, MANUFACTURER\_ID

, NAME

, NOTES

)

VALUES

(

1

, 1

, 'spring'

, 'cross sell handle latch'

)

### SELECT

SELECT {alias.Column\_Name}

,{alias.Column\_Name}

, ...

FROM {SchemaName}.{Table\_Name} {alias 1}

INNER JOIN {SchemaName}.{Table\_Name} {alias 2}

ON {alias 1}.{Column\_Name} = {alias 2}.{Column\_Name}

INNER JOIN ...

WHERE {constraint condition}

AND {constraint condition}

GROUP BY {column [list]}

HAVING {constraint condition}

ORDER BY {column [list]}

{union}

{next select statement}

Example:

SELECT

t.Task\_Id

, t.Course\_Id

, t.Due\_Dt

, t.Start\_Time

, t.End\_Time

, t.Name

, et.Completed\_Flag

, et.Completed\_Dt

FROM BusyWork.dbo.TB\_TASK t

INNER JOIN BusyWork.dbo.ENROLLMENTTASK et

ON t.Task\_Id = et.Task\_Id

WHERE t.Due\_Dt >= @pStartDate

AND t.Due\_Dt <= @pEndDate

AND et.Member\_Id = @pMemberId

ORDER BY t.Due\_Dt

, t.Start\_Time

### UPDATE

* Use the following outline for simple update statements. Format the where clause as described earlier.

UPDATE {schema}.{table name}

SET {column} = {expression}

,{column} = {expression}, […]

WHERE {where clause}

Example:

UPDATE Articles.dbo.TB\_STATISTICS

SET READ\_HITS = READ\_HITS + 1

, LAST\_READ\_DT = current\_timestamp

WHERE ARTICLE\_ID = @pArticleId

* Use the following outline for table-to-table update statements. Format the from and where clauses as described earlier.

UPDATE {database name}.{owner}.{table name}

SET {column} = {expression}

, {column} = {expression}

, …

FROM {clause}

WHERE {clause}

Example:

UPDATE PUBS.dbo.TB\_TITLES

SET Total\_Sales = t.Total\_Sales + s.Quantity

FROM Pubs.dbo.TB\_TITLES t

INNER JOIN Pubs.dbo.TB\_SALES s

ON t.Title\_Id = s.Title\_Id

## INDEXES

* Use the following outline for creating Indexes

CREATE {CLUSTERED | NONCLUSTERED} INDEX (CIX | NCIX | CCIX | NCCIX | \_{index name}

ON {fully qualified table name}

(

{column list}

)

{options}

ON ([FILEGROUP])

Example:

CREATE NONCLUSTERED INDEX NCIX\_Expire\_Dt

ON dbo.TB\_EVENT(Expire\_Dt)

ON ([INDEXES])

## OBJECT META DATA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tag** | **Description** | **Format** | | **IsMandatory** |
| Access\_Owner | Person or persons responsible for authorising access to the data | Person name or team name | TRUE | |
| Actual\_DR\_Tier | The disaster recovery tier the data currently resides in | Number between 1-7 | TRUE | |
| Actual\_HA\_Tier | The availability tier the data currently resides in | Number between 1-4 | TRUE | |
| Backup\_Location | The folder path or suitable reference to where backups are retained for the data | Plain text | TRUE | |
| Backup\_Schedule | How frequently the data is backed up | <Backup\_Schedule><Full>Weekly,Sunday</Full><Diff>Daily</Diff><Log>30min</Log></Backup\_Schedule> | TRUE | |
| Business\_Owner | The business stakeholder of the data. This is usually a manager of head of function for the business area | Person name or team name | TRUE | |
| Can\_Be\_Overwritten | Only to be used for development and test databases. This indicates the database can be restored over with new data | TRUE\FALSE |  | |
| Change\_Request\_No | The change request number pertaining to release of the database and any subsequent changes to the database after initial release | Number |  | |
| CName | DNS alias used to access the database server. | Plain text | TRUE | |
| Data\_Owner | The business stakeholder of the data. This is usually a manager of head of function for the business area | Person name or team name | TRUE | |
| Data\_Quality\_Grade | More relevant for derived data from production systems, however this | See DataQuality |  | |
| Data\_Sensitivity\_Level | The sensitivity of the data according to all relevant internal and external policies and acts | PUBLIC\INTERNAL\CONFIDENTIAL\HIGHLY\_CONFIDENTIAL | TRUE | |
| Description | A description of the function of the database and how it pertains the system it supports | Plain text | TRUE | |
| Desired\_DR\_Tier | The disaster recovery tier the data should reside in | Number between 1-7 |  | |
| Desired\_HA\_Tier | The availability tier the data should reside in | Number between 1-4 |  | |
| Last\_Release\_Date | The last date any type of release or hotfix was applied to the database. This does not include patching of the database engine itself | Datetime |  | |
| Maintenance\_Window | An agreed period when downtime or sub-optimal performance is acceptable to carry out essential maintenance tasks (such as reindexing) | Example: 9-5, Every Monday 10-12 | TRUE | |
| Originating\_Server | Intended for use after migration of the database from another server | Plain text |  | |
| Project\_Manager | The name or names of the key project managers | Person name or team name |  | |
| Project\_Owner | The sponsor of the project | Person name or team name |  | |
| Recovery\_Point\_Objective | The maximum amount of data loss for a system or database before it poses a significant risk to the business | 00Days:00Hours:00Minutes:00Seconds | TRUE | |
| Recovery\_Time\_Objective | The maximum amount of time a system or database can be "unavailable" before it poses a significant risk to the business | 00Days:00Hours:00Minutes:00Seconds | TRUE | |
| Source\_Control\_Location | Link or relevant reference to source control | Plain text |  | |
| Support\_Hours | The hours the database has been agreed to be support in | Example: 9-5 or 247 | TRUE | |
| Support\_Response\_Time | The response time agreed by the support department for the database | 00Days:00Hours:00Minutes:00Seconds | TRUE | |
| Support\_Service | Service(s) supported by the database | Plain text |  | |
| Third\_Party\_Contact | Person name or team name | Plain text |  | |
| Third\_Party\_Product | Supplier of the database or system | Plain text |  | |
| Version | The version number of the database | Major.Minor.Build or relevant version reference | TRUE | |

## BULK DELETE

USE [DW\_Statement\_Mart]

GO

CREATE PROCEDURE [Source].[Batch\_Delete\_Latest\_Transaction\_Old\_Rows]

(

@Batch\_Size SMALLINT

)

AS

DECLARE @ExpectedRowCount int = 0

, @ExpectedBatchCount int = 0

, @BatchCount int = 0

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

IF NOT EXISTS (SELECT OBJECT\_ID('Tempdb', '#RecordIdsToDelete'))

BEGIN

CREATE TABLE #RecordIdsToDelete

(

Row\_Hash VARCHAR(500) PRIMARY KEY CLUSTERED

, Deleted BIT DEFAULT 0

)

END

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

-- Insert the ids of the records to delete into the table variable

INSERT INTO #RecordIdsToDelete

(

Row\_Hash

)

SELECT T1.Row\_Hash

FROM [EDW\_DBO].[DM\_STAT\_LATEST\_TRANSACTION] T1 (NOLOCK)

LEFT OUTER JOIN [Source].[DM\_STAT\_LATEST\_TRANSACTION] T2 (NOLOCK)

ON T1.Row\_Hash = T2.Row\_Hash

WHERE t1.row\_hash IS NULL

SET @ExpectedRowCount = @@ROWCOUNT;

DECLARE @Delete\_Work TABLE

(

Row\_Hash VARCHAR(500)

)

SET @ExpectedBatchCount = CEILING(@ExpectedRowCount / @Batch\_Size)

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

WHILE (@BatchCount < @ExpectedBatchCount AND @@ERROR = 0)

BEGIN

BEGIN TRANSACTION

INSERT INTO @Delete\_Work

SELECT TOP (@Batch\_Size) Row\_Hash

FROM #RecordIdsToDelete

WHERE DELETED = 0

DELETE FROM [EDW\_DBO].[DM\_STAT\_LATEST\_TRANSACTION]

FROM [EDW\_DBO].[DM\_STAT\_LATEST\_TRANSACTION] SLT

INNER JOIN @Delete\_Work DW

ON DW.Row\_Hash = SLT.Row\_Hash

UPDATE #RecordIdsToDelete

SET Deleted = 1

FROM #RecordIdsToDelete rtd

INNER JOIN @Delete\_Work DW

ON DW.Row\_Hash = rtd.Row\_Hash

DELETE FROM @Delete\_Work

SET @BatchCount = @BatchCount + 1

COMMIT TRANSACTION

END

## STANDARD SUPPORT SCRIPT

/\*\*

\*-----------------------------------------------------------------------------\*

| Standard Support Script |

| |

| Version 1 Dale Stewart 13/11/2014 |

| Version 2 Dale Stewart 10/08/2016 |

| |

\*-----------------------------------------------------------------------------\*

COMPATIBLITY

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

2008 upwards

\*-----------------------------------------------------------------------------\*

BACKGROUND

----------

Describe the reason the script has been created. this section should include

as much information about the

\*-----------------------------------------------------------------------------\*

USAGE

-----

Give examples of how to execute the script. Are there any special conditions

that should exist beforehand (should the script test for those first?).

This section is particularly important when this code block is used in

stored procedures.

\*-----------------------------------------------------------------------------\*

VERSION 1

---------

List all features of the first version

VERSION 2

---------

list all changes made to the script since version 1

\*-----------------------------------------------------------------------------\*

REFERENCES

----------

Include any references to external sites (if code has been copied) or to

any other internal policies or guidelines

\*-----------------------------------------------------------------------------\*

PARAMETERS

----------

List a

\*-----------------------------------------------------------------------------\*

DEPENDENCIES

----------

List all scripts or code this script relies upon. For example this could

include scripts that need to be run before this one.

\*\*/

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

:setvar checkSQLCMD "checkSQLCMD"

GO

IF ('$(checkSQLCMD)' = '$' + '(checkSQLCMD)') RAISERROR ('This script must be run in SQLCMD mode.', 20, 1) WITH LOG

GO

SELECT 'This part executes only in SQLCMD mode!'

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

:setvar ServerName "<Target\_SQL\_Server, sysname, >"

:setvar DatabaseName "<Target\_Database, sysname, >"

:setvar CR "<Change\_Request\_No, sysname, >"

:connect $(ServerName)

USE $(DatabaseName)

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

SET XACT\_ABORT ON;

DECLARE @ExpectedRowCount INT

, @ActualRowCount INT

SET @ExpectedRowCount = <Expected\_Row\_Count, INT, >

SET @ActualRowCount = 0

BEGIN TRAN

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

--Take a backup copy of the data

SELECT \*

INTO <Data\_Backup\_Database, sysname, DBSAdmin>.dbo.CR$(CR)

FROM [<Security\_Schema, sysname, dbo>].[<Table\_Name, sysname, Table\_Name>]

WHERE <WHERE\_Condition\_Column, sysname, Column1> <WHERE\_Predicate, VARCHAR, => <WHERE\_Value, VARCHAR, 1>;

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

--perform data modification

/\*\*

DELETE FROM [<Security\_Schema, sysname, dbo>].[<Table\_Name, sysname, Table\_Name>]

WHERE <WHERE\_Condition\_Column, sysname, Column1> <WHERE\_Predicate, VARCHAR, => <WHERE\_Value, VARCHAR, 1>;

\*\*/

/\*\*

UPDATE [<Security\_Schema, sysname, dbo>].[<Table\_Name, sysname, Table\_Name>]

SET [<Security\_Schema, sysname, dbo>].[<Table\_Name, sysname, Table\_Name>]

WHERE <WHERE\_Condition\_Column, sysname, Column1> <WHERE\_Predicate, VARCHAR, => <WHERE\_Value, VARCHAR, 1>;

\*\*/

SET @ActualRowCount = @@ROWCOUNT

IF @ExpectedRowCount <> @ActualRowCount

BEGIN

RAISERROR ('Row count mismatch', 16, 1);

END;

COMMIT TRAN