SQL SERVER METADATA

SUCCINCTLY

BY **JOSEPH D. BOOTH**

SUCCINCTLY EBOOK SERIES

Syncfusion®

SQL Server Metadata Succinctly

By
Joseph D. Booth

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, JavaScript, Visual C#, and the .NET Framework. He has also worked in various database platforms, including DBASE, Paradox, Oracle, and SQL Server.

He is the author of *GitHub Succinctly*, *Accounting Succinctly*, *Regular Expressions Succinctly*, *Visual Studio Add-Ins Succinctly*, and *Natural Language Processing Succinctly* from Syncfusion, as well as six books on Clipper and FoxPro programming, network programming, and client/server development with Delphi. He has also written several third-party developer tools, including CLIPWKS, which allows developers to programmatically create and read native Lotus and Excel spreadsheet files from Clipper applications.

Joe has worked for a number of companies, including Yprime, Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market-research software), PEPSys (industrial-distribution software), and a key contributor to AccuBuild (accounting software for the construction industry).

He has a background in accounting (*Accounting Succinctly*), having worked as a controller for several years in the industrial distribution field, but his real passion is computer programming.

In his spare time, Joe is an avid tennis player. He also practices yoga and martial arts, and plays with his first granddaughter, Blaire. You can visit his website for more information.

Chapter 1 Introduction

In the classic *Star Trek* episode called "The Enterprise Incident," the crew steals the Romulan cloaking device and attempts to integrate it into their ship. By reading the device's metadata, the crew was able to figure out how to make the device work and successfully evade those nasty Romulans. While it seems unlikely that the technology would be similar, most devices provide "metadata" about themselves.

Metadata

Metadata is very common in most digital devices and files. Pictures taken by a digital camera can have over 400 tags of metadata associated with the photo, including type of digital device used, where and when the photo was taken, and so on. Your phone call log also has metadata, like where you called from, who you called, and how long you talked. You can visit this website (or search for "photo metadata" on Google) to get a sense of the data that is stored.

Privacy concerns aside, it is clear that metadata is very prevalent in the digital world.

SQL metadata

Microsoft SQL Server (and other SQL systems) also provide a large amount of data about their servers, users, tables, and stored procedures. In this book, we are going to explore that metadata and provide example scripts and queries to learn a lot of information about your SQL environment.

Information schema

Information schema views are a series of SQL-92 ANSI standard views that are generally found in most SQL systems (Oracle being a notable exception). These views provide information about tables, columns, views, and stored procedures, and for the most part, the queries using these views will work across database platforms.

Note: The ANSI standard is not specific to Microsoft, so some of the field and view names might not use the same terminology that SQL Server uses.

System and data management views

The system and data management views in SQL Server provide the same information the information schema views do, and a whole lot more information that is unique to SQL Server. If you look behind the scenes at the views in **`INFORMATION_SCHEMA`**, you will discover they are wrapper views to system tables.

Code Listing 1: Object Definition for information_schema.tables

```
---------------------------------------------------
-- Script: Peek_Definition.sql
-- Look at definition of information_schema view
--------------------------------------------------
DECLARE @srcCode VARCHAR(max)
SELECT @srcCode =
OBJECT_DEFINITION(object_id('INFORMATION_SCHEMA.tables'))

PRINT @srcCode

-- The following result would be displayed by the PRINT statement

CREATE VIEW INFORMATION_SCHEMA.TABLES
AS
SELECT
	DB_NAME()			AS TABLE_CATALOG,
	s.name				AS TABLE_SCHEMA,
	o.name				AS TABLE_NAME,
	CASE o.type
		WHEN 'U' THEN 'BASE TABLE'
		WHEN 'V' THEN 'VIEW'
	END				AS TABLE_TYPE
FROM
	sys.objects o LEFT JOIN sys.schemas s
	ON s.schema_id = o.schema_id
WHERE
	o.type IN ('U', 'V')
```

The system tables provide much more information, but the information schema views are closer to the ANSI standard, and generally can be used across database products.

Note: The SQL code in this book was tested using SQL 2017, and most of the views and functions will work on older versions of SQL, as well. Scripts that don't work on older versions will be noted.

Summary

In this book, we will start with the information schema views and provide some handy queries about the server's data tables. We will then explore some of the views that are unique to Microsoft and provide very useful information about SQL Server.

Chapter 2 Information Schema

In most relational databases, the information schema (**`INFORMATION_SCHEMA`**) is an ANSI-standard set of read-only views that provide information about the tables, views, etc., in the database. SQL Server implements these views, although some of the terminology is slightly different, as shown in Table 1.

Table 1: Naming differences

SQL Server	Information schema
Database	Catalog
User-defined data type	Domain

The information schema views will use the catalog and domain terminology to meet the ANSI standard. However, to the Microsoft SQL developer, the database and the user-defined data type are the more common terms. You can find these views under the System Views list in SQL Server Management Studio (SSMS).

Note: Although the information schema views are available in each database, the table catalog (which is always the current database) is returned in the various views. This will come in handy if you need to write a procedure to iterate all databases on a server.

Tables

The **`Tables`** view holds four columns.

Table 2: Tables columns

Column name	Description
`TABLE_CATALOG`	Catalog (or database name)
`TABLE_SCHEMA`	Schema (or owner) of the table
`TABLE_NAME`	Actual name of the table
`TABLE_TYPE`	`BASE TABLE`: Physical table `VIEW`: View

The first three columns are generally used as identifiers (with different field names) for almost all information schema views. Use this view to identify the schema that the tables are owned by and whether it is a physical table or view.

Tip: SQL Server allows you not to specify the table owner schema when referencing a table. This can cause some subtle bugs when a table occurs in multiple schemas. The following query can identify table names that occur in more than one schema.

```
SELECT Table_Name,count(*) as Owners
FROM [INFORMATION_SCHEMA].[tables]
GROUP BY Table_Name HAVING count(*)>1
```

Table constraints

Constraints are rules applied to columns in a database table. They generally limit the type of data that can be added to a column. The **`TABLE_CONSTRAINTS`** view lists the tables and constraints on those tables. The columns are shown in Table 3.

Table 3: TABLE_CONSTRAINTS columns

Column name	Description
`CONSTRAINT_CATALOG`	Catalog (or database name) for the constraint.
`CONSTRAINT_SCHEMA`	Schema (or owner) of the table.
`CONSTRAINT_NAME`	Name of the constraint itself.
`TABLE_CATALOG`	Catalog (or database name) for the table the constraint applies to.
`TABLE_SCHEMA`	Schema (or owner) of the table.
`TABLE_NAME`	Actual name of the table.
`CONSTRAINT_TYPE`	One of four values: `CHECK`: Expressions to limit column content. `FOREIGN KEY`: Reference to another table's primary key. `PRIMARY KEY`: Primary key for table. `UNIQUE`: Values in this column must be unique.
`IS_DEFERRABLE`	In SQL Server, always returns NO.
`INITIALLY_DEFERRED`	In SQL Server, always returns NO.

The **`CHECK`** constraint definitions are found in the **`CHECK_CONSTRAINTS`** view, which includes the **`CHECK_CLAUSE`** column to show the definition of the constraint.

The remaining constraint types (**FOREIGN KEY**, **PRIMARY KEY**, and **UNIQUE**) reference tables and columns and can be found in the **CONSTRAINT_COLUMN_USAGE** view. You can combine these tables to create a query that returns the constraint type and name, and an expression column showing the expression or key columns. Code Listing 2 shows the query.

Code Listing 2: Constraint definition query

```
---------------------------------------------
-- Script: Constraint_Definitions.sql
-- Show various table constraints
---------------------------------------------
SELECT constraint_type AS TYPE,TC.constraint_name AS NAME
  CASE
    WHEN tc.constraint_type = 'CHECK'
    THEN cc.check_clause
    ELSE cu.table_name+'.'+cu.column_Name COLLATE DATABASE_DEFAULT
  END AS EXPRESSION
FROM INFORMATION_SCHEMA.table_constraints tc
LEFT JOIN INFORMATION_SCHEMA.check_constraints cc
     ON   cc.CONSTRAINT_NAME=tc.CONSTRAINT_NAME
          AND tc.CONSTRAINT_TYPE='CHECK'
LEFT JOIN INFORMATION_SCHEMA.constraint_column_usage cu
     ON cu.constraint_name=tc.CONSTRAINT_NAME
ORDER BY constraint_type,tc.constraint_name
```

This will return a table with the following sample data.

Table 4

Type	Name	Expression
CHECK	CK_Base_Location_Latitude	([Latitude]>=(-90) AND [Latitude]<=(90))
UNIQUE	UK_User_Id	Users.User_ID
PRIMARY KEY	PK_Client_ID	Clients.ID

Note: In this query, we are only joining on the constraint name for simplicity. If you have multiple schemas in your database, you should include schema names as part of your join condition.

Columns

The columns view begins with the table information (catalog, schema, and table name) followed by all the columns in the table and information about those columns. Some of the key column fields are shown in Table 5.

Table 5: Column fields

Column name	Description
COLUMN_NAME	Column name in the table.
ORDINAL_POSITION	Column position.
COLUMN_DEFAULT	The default value for the column or NULL.
IS_NULLABLE	Can the column value be NULL (YES or NO)?
DATA_TYPE	String description of the column data type.
CHARACTER_MAXIMUM_LENGTH	Length of the column, if a character column.
NUMERIC_PRECISION	Length for numeric columns.

Note: Certain column data types (such as bit and uniqueidentifer) don't have size information in the table. If a character column is defined as MAX, the character length column will contain a -1.

Domains

Domains (or "user-defined data types" in SQL Server parlance) are useful concepts that make it easier to maintain columns by creating your own column type. For example, you might want to create a domain to set the properties you want all phone number fields to use. The domains view contains most of the same column definitions as the columns view.

Table 6: Domain fields

Column name	Description
DOMAIN_NAME	Name of the domain.
DATA_TYPE	Data type of this domain.
DOMAIN_DEFAULT	The default value for the column or NULL.
CHARACTER_MAXIMUM_LENGTH	Length of the column, if a character column.
NUMERIC_PRECISION	Length for numeric columns.
NUMERIC_SCALE	Scale of the numeric data.
DOMAIN_DEFAULT	Text of the default constraint for this domain.

column_domain_usage

The **column_domain_usage** view provides a list of all tables that reference one of the user-defined columns (domains).

Table 7: Column Domain Usage fields

Column name	Description
DOMAIN_CATALOG	Name of the database the domain is in.
DOMAIN_SCHEMA	Name of the schema of the domain.
DOMAIN_NAME	Actual name of domain (link to Domains view).
TABLE_CATALOG	Name of database that uses the domain.
TABLE_SCHEMA	Name of the schema using the domain.
TABLE_NAME	Actual table name using the domain.
COLUMN_NAME	Column name using the domain.

Domain queries

You can perform some basic queries to view your user-defined types and the tables that use them. Code Listing 3 simply shows the user-defined types and displays the size information in a nicely formatted way.

Code Listing 3: List user-defined types

```
---------------------------------------------
-- Script: List_User_Defined_Types.sql
-- Show existing user-defined data types
---------------------------------------------
SELECT domain_schema,domain_name,
CASE
WHEN data_type='DECIMAL' THEN data_type+'('+CAST(Numeric_Precision as
varchar(3))+'.'+CAST(Numeric_Scale as varchar(3))+')'
WHEN data_type IN ('char','varchar','nchar','nvarchar')
     THEN
       CASE
            WHEN character_maximum_length<0 THEN data_type+'(MAX)'
            ELSE data_type+'('+
                 CAST(Character_Maximum_Length as Varchar(3))+')'
      END
ELSE data_type
END AS RootDateType
FROM INFORMATION_SCHEMA.domains
ORDER BY domain_schema,domain_name
```

Code Listing 4 shows a similar result but includes the count of columns that are referencing the domain.

Code Listing 4: Domain Usage Counts

```
---------------------------------------------
-- Script: Domains_Usage_Counts.sql
-- Show existing user-defined data types
```

```
---------------------------------------------
SELECT d.domain_schema,d.domain_name,
CASE
WHEN data_type='DECIMAL' THEN data_type+'('+
     CAST(Numeric_Precision as varchar(3))+'.'+
     CAST(Numeric_Scale as varchar(3))+')'
WHEN data_type IN ('char','varchar','nchar','nvarchar')
     THEN
       CASE
            WHEN character_maximum_length<0 THEN data_type+'(MAX)'
            ELSE data_type+'('+
                 CAST(Character_Maximum_Length as Varchar(3))+')'
      END
ELSE data_type
END AS RootDateType,
count(du.column_name) as ColumnsUsing
FROM INFORMATION_SCHEMA.domains d
LEFT JOIN INFORMATION_SCHEMA.column_domain_usage du on
d.domain_name=du.domain_name
GROUP BY d.domain_schema,d.domain_name,data_type,
character_maximum_length,numeric_precision,numeric_scale
ORDER BY d.domain_schema,d.domain_name
```

You can view user-defined types in SQL Server Management Studio (SSMS) by using the Programmability menu of the object inspector, as shown in Figure 1.

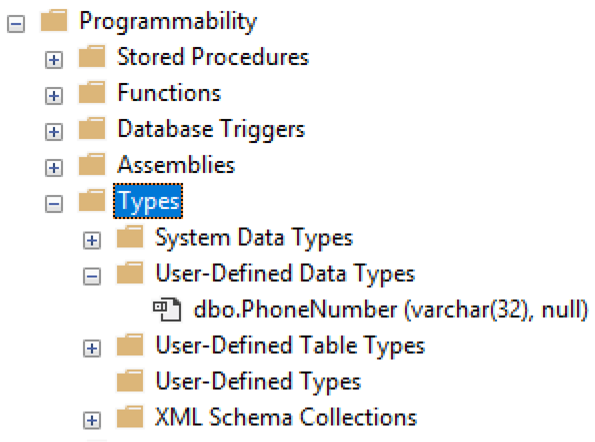


Figure 1: Object inspect user defined types

Routines

The **`ROUTINES`** view returns over 50 columns about the various stored procedures and functions in the database. It includes the source code and a flag indicating if the routine updates or simply reads the data. Some representative columns from the view are shown in Table 8.

Table 8: Routine fields

Column name	Description
`ROUTINE_CATALOG`	Name of the database the routine is in.
`ROUTINE_SCHEMA`	Name of the schema the routine is in.
`ROUTINE_NAME`	Name of the view.
`ROUTINE_TYPE`	FUNCTION or PROCEDURE.
`DATA_TYPE`	If a FUNCTION, this indicates the return type.
`ROUTINE_DEFINITION`	SQL Source code of the routine.
`SQL_DATA_ACCESS`	MODIFIES (routine updates data) or READS (only reads data).

Note: The routine definition column only contains 4,000 bytes of the routine. If you are writing code to get the complete routine body, consider using SP_HelpText instead.

Routine columns

If the routine is a table-valued function (it returns a table rather than a single value), the **`ROUTINE_COLUMNS`** view returns the column information for the returned table.

Table 9: Routine column fields

Column name	Description
`TABLE_CATALOG`	Name of the database the function is in.
`TABLE_SCHEMA`	Name of the schema the function is in.
`TABLE_NAME`	Name of the function.
`COLUMN_NAME`	Column name from returned table.
`ORDINAL_POSITION`	Sequence of columns in returned table.
`DATA_TYPE`	Data type of the returned table column.
`CHARACTER_MAXIMUM_LENGTH`	Size of column (-1 max).
`IS_NULLABLE`	YES or NO, can column contain NULL.

Code Listing 5 shows the code used to identify the type of routine and whether it updates data.

Code Listing 5: Detailed routine types

```
---------------------------------------------
-- Script: Routine_type_details.sql
-- Type and data access of routines
---------------------------------------------
SELECT r.routine_name,
CASE
WHEN rc.table_name is null AND r.routine_type='FUNCTION' THEN 'Scalar
Function'
WHEN rc.table_name is not null and r.routine_type='FUNCTION' THEN 'Table
Valued Function'
ELSE 'Stored procedure'
END AS RoutineType,
r.sql_data_access
FROM INFORMATION_SCHEMA.routines r
LEFT JOIN
(SELECT DISTINCT table_name FROM INFORMATION_SCHEMA.routine_columns) rc
ON rc.table_name=r.routine_name
ORDER BY RoutineType,routine_name
```

Parameters

The **PARAMETERS** view returns many columns about the various input and output parameters to all the stored procedures and functions in the routines view. Table 10 lists some of the common columns in the view.

Table 10: Parameters fields

Column name	Description
SPECIFIC_CATALOG	Name of the database the routine parameter is in.
SPECIFIC_SCHEMA	Name of the schema the parameter is in.
SPECIFIC_NAME	Name of the routine.
ORDINAL_POSITION	Sequence number of the parameter.
PARAMETER_MODE	IN or OUT.
PARAMETER_NAME	Name of the parameter (if MODE = IN).
DATA_TYPE	Data type of the parameter.
CHARACTER_MAXIMUM_LENGTH	Size of the parameter (-1 = MAX).

Routine queries

Code Listing 6 is a query that returns the routine name and type, and a column containing all the parameters used by the routine. The following table shows a sample of the output.

Table 11: Sample routine query output

Routine name	Type	Parameters
CheckAuditTableColumns	PROC	@debug tinyint
CheckDataPointTableTypeExists	PROC	No parameters

Routine name	Type	Parameters
`CheckForCalculations`	`PROC`	`@ClientID int @VisitId int`

Code Listing 6 can provide a simple documentation overview of the database code.

Code Listing 6: Routine and parameters

```
---------------------------------------------
-- Script: Routine_and_parameter_details.sql
-- Routine names and parameters
---------------------------------------------
SELECT r.routine_schema,r.routine_name,left(routine_type, 4) AS [type],
    CASE left(routine_type,1)
    WHEN 'P' THEN IsNull(px.column_names,'No parameters')
    ELSE IsNull('(returns)=>' + px.column_names,'No parameters')
    END AS Parameters
    FROM INFORMATION_SCHEMA.routines r
    LEFT JOIN
    (SELECT specific_schema, specific_name,
         LEFT(column_names, LEN(column_names )) AS column_names
    FROM INFORMATION_SCHEMA.parameters AS extern
    CROSS APPLY
    (  SELECT parameter_name + ' '+
    CASE
    WHEN DATA_TYPE like '%varchar%'
        AND CHARACTER_MAXIMUM_LENGTH <0 THEN DATA_TYPE+'(MAX) '
    WHEN CHARACTER_MAXIMUM_LENGTH >0 THEN  DATA_TYPE+'('+
CAST(CHARACTER_MAXIMUM_LENGTH as varchar)+') '
    ELSE DATA_TYPE+'  '
    END
    FROM INFORMATION_SCHEMA.parameters AS intern
    WHERE extern.specific_name = intern.specific_name
    FOR XML PATH('') ) pre_trimmed(column_names)
    GROUP BY specific_schema,specific_name, column_names
    ) px on px.specific_schema=r.routine_schema AND
      px.specific_name=r.specific_name
ORDER BY r.routine_schema, r.routine_name
```

Views

The **`VIEWS`** view holds details about the views defined in the database. Notice that the column names are called **`TABLE`**, even though the content is the view itself. The view definition column contains a SQL Create View script to create the view. This column is limited to 4,000 bytes, which should be enough for all but the most complex views.

Table 12: View fields

Column name	Description
`TABLE_CATALOG`	Name of the database the view is in.
`TABLE_SCHEMA`	Name of the schema the view is in.
`TABLE_NAME`	Name of the view.
`VIEW_DEFINITION`	The source code for the view.
`IS_UPDATABLE`	YES or NO, can data manipulation operations be performed on the view?
`CHECK_OPTION`	Will return "`CASCADE`" if view was created using the `WITH CHECK OPTION`, "`NONE`" if the option was not applied when the view was created.

View table usage

The **`VIEW_TABLE_USAGE`** view provides details on which tables are referenced in which views. Table 13 lists the fields in this view.

Table 13: View table usage fields

Column name	Description
`VIEW_CATALOG`	Name of the database the view is in.
`VIEW_SCHEMA`	Name of the schema the view is in.
`VIEW_NAME`	Name of the view.
`TABLE_CATALOG`	Name of the database the referenced table is in.
`TABLE_SCHEMA`	Name of the schema the referenced table is in.
`TABLE_NAME`	Name of the table referenced by the view.

View column usage

The **`VIEW_COLUMN_USAGE`** view provides details on which tables and column are referenced in which views. Table 14 lists the fields in this view.

Table 14: View column usage fields

Column name	Description
`VIEW_CATALOG`	Name of the database the view is in.
`VIEW_SCHEMA`	Name of the schema the view is in.
`VIEW_NAME`	Name of the view.
`TABLE_CATALOG`	Name of the database the referenced table is in.

Column name	Description
TABLE_SCHEMA	Name of the schema the referenced table is in.
TABLE_NAME	Name of the table referenced by the view.
COLUMN_ NAME	Name of the column in the table referenced by the view.

View queries

Code Listing 7 contains a query that will display each view name and the tables that it references.

Code Listing 7: Views and referenced tables

```
---------------------------------------------
-- Script: Views_and_Table_Usage.sql
-- Views and the tables referenced
---------------------------------------------
SELECT v.table_schema,v.table_name AS 'ViewName',
       IsNull(xx.RefObjects,'') AS 'References'
       FROM INFORMATION_SCHEMA.views v
       LEFT JOIN(Select distinct ST2.view_schema + '.' +
                 ST2.view_name as ViewRollup,
       ltrim(substring((            Select ', ' + ST1.table_name
AS[text()]
       FROM INFORMATION_SCHEMA.view_table_usage  ST1
       WHERE ST1.view_schema + '.' + ST1.view_name = ST2.view_schema +
                               '.' + ST2.view_name
       ORDER BY ST1.view_schema + '.' + ST1.view_name
       FOR XML PATH('')        ), 2, 8000))[RefObjects]
       FROM INFORMATION_SCHEMA.view_table_usage ST2
       ) xx ON xx.ViewRollup = v.table_schema + '.' + v.table_name
ORDER BY v.table_schema,v.table_name
```

Tips and tricks

There are some SQL queries using information schema views that can be helpful when working with a SQL database.

Find which tables contain a column name

If you are working in an application that accesses SQL data, you can use the following query to determine all tables and views in which a column is used.

Code Listing 8: Find a column

```
---------------------------------------------
-- Script: Find_Column.sql
-- Search for a column by name
---------------------------------------------
SELECT table_schema,table_name
FROM INFORMATION_SCHEMA.columns
WHERE column_name='IsSupported'
ORDER BY table_schema,table_name
```

Same name, different types and sizes

In some databases, developers might have created a field in several tables, but the type or size information is different. This can cause some subtle, hard-to-find bugs in application code. For example, if a C# string variable reads a **`varchar`** column, the value is trimmed, but if it reads a **`char`** column, it will contain the trailing spaces, which could cause some unexpected comparison results.

This query identifies columns with the same name, but different types or sizes.

Code Listing 9: Same name, different type or size

```
---------------------------------------------------------
-- Script: Mismatched_Columns.sql
-- Columns with same name, but different types/sizes
---------------------------------------------------------
SELECT column_name,
CASE
WHEN min(data_type)<>max(data_type) THEN 'Type mismatch'
ELSE ''
END AS TypeError,
CASE
WHEN min(character_maximum_length) <> max(character_maximum_length) THEN
'Size differences'
ELSE ''
END AS SizeError,count(*) AS NumTables
FROM INFORMATION_SCHEMA.columns
GROUP BY column_name
HAVING (min(data_type)<>max(data_type) )
OR (min(character_maximum_length) <> max(character_maximum_length))
```

Tip: You can test collation sequence, nullable, etc., by changing your HAVING conditions in the query.

Similarly named columns

Often, over time and multiple developers, column names might have slight naming variations that can create application errors. This query finds all columns that contain the text “phone”.

Code Listing 10: Similar column names

```
---------------------------------------------
-- Script: Similarly_named_Column.sql
-- Look for columns with similar names
---------------------------------------------
SELECT table_schema,table_Name,column_Name
FROM INFORMATION_SCHEMA.COLUMNS
WHERE column_Name LIKE '%phone%'
```

In this example database, we found **`phone`** and **`phonenumber`** were both used to hold a phone number field for an organization.

Routines and parameters

You can check your stored procedures and functions and see what parameters they expect (in order) by using the following query.

Code Listing 11: Routines and parameters summary

```
---------------------------------------------
-- Script: Routines_summary.sql
-- Routines and parameters
---------------------------------------------
SELECT r.routine_type,r.specific_name,rc.parameter_name
FROM INFORMATION_SCHEMA.routines r
JOIN INFORMATION_SCHEMA.parameters rc ON r.specific_name=rc.specific_name
WHERE rc.parameter_mode='IN'
ORDER BY r.specific_name,rc.ordinal_position
```

Summary

The information schema views are a handy and generic (SQL-92 standard) way to identify your database structure and look for potential problem areas.

Note: While Oracle doesn't implement the information schema, it does have views to provide the same information. For example, the ALL_TABLE view is like the TABLES view, and ALL_TAB_COLUMNS view is like the COLUMNS view.

No matter which database you use, having programmable access to the underlying structures can help you to understand the database and solve problems. If you want more information about the information schema views in SQL Server, you can visit this website. In this chapter, we only focused on some columns in the views; this site will provide complete details of all the columns in the views.

Chapter 3 Server Information

In this chapter, we will explore some of the hardware, operating system, and SQL version information, as well as the various configuration options of the server. All this information is available through various views in the `sys` schema. Most of the information comes from the subset called dynamic management views within the `sys` schema. These views begin with **dm_** and provide a lot of information about the SQL environment.

Note: Your account may not have permission to access the various views described in this chapter, particularly if you are on a remote server. You will generally need the VIEW SERVER STATE permission for most of these queries.

Host version

You can determine the operating system information and version number by using the newly added **`sys.dm_os_host_info`** view (SQL 2017). Note that SQL Server does not return as much information if SQL is running on a Linux host.

Table 15: dm_os_host_info

Column name	Description
`host_platform`	Windows or Linux.
`host_distribution`	Description of the operating system.
`host_release`	Version number on Windows OS, empty on Linux.
`host_service_pack_level`	Service pack level (Windows) or empty (Linux).
`host_sku`	Window stock keeping unit (maps to a Windows product version) NULL on Linux systems • 4 is Enterprise Edition • 7 is Standard Server Edition • 8 is Datacenter Server Edition • 10 is Enterprise Server Edition • 48 is Professional Edition
`os_language_version`	Windows Locale identifier (LCID) of operating system.

You can find out more about the Windows version information at this website.

The locale ID can be found here.

If you are using an older version of SQL, the **`dm_os_windows_info`** view provides similar information (without the platform or distribution columns). Table 16 lists the columns in that view.

Table 16: dm_os_windows_info

Column name	Description
`windows_release`	Version number of Windows OS
`windows_service_pack_level`	Service pack level
`windows_sku`	Window stock keeping unit (maps to a Windows product version)
`windows_os_language_version`	Windows Locale identifier (LCID)

SQL Server version information

There are a variety of ways to determine which version of SQL is being run.

@@version

The simplest approach is to use the **`@@version`** global variable. This will return a string containing the version and copyright information.

```
Microsoft SQL Server 2017 (RTM-CU14-GDR) (KB4494352) - 14.0.3103.1 (X64)
Mar 22 2019 22:33:11
Copyright (C) 2017 Microsoft Corporation
Developer Edition (64-bit) on Windows Server 2016 Datacenter 10.0 <X64>
(Build 14393: ) (Hypervisor)
```

If you want to report on this data, you can save it to a variable and split the variable on the linefeed (**`char(10)`**) character.

xp_msver

Another option is to use the extended stored procedure **`xp_msver`** in the master database. This returns a four-column table containing version and copyright information about the server. Table 17 shows some sample rows from this procedure.

Table 17: Sample xp_msver result

Index	Name	Internal_Value	Character_Value
1	ProductName	NULL	Microsoft SQL Server

2	ProductVersion	917504	14.0.3103.1
3	Language	1033	English (United States)
4	Platform	NULL	NT x64

SERVERPROPERTY

While the previous two approaches show the version number and description, you might want to simply get a numeric indication of the version number (for example, your procedure only runs on a version of the server). For this approach, you can use the **`SERVERPROPERTY`** function and get the **`ProjectMajorVersion`**, as shown in the following.

```
select SERVERPROPERTY('ProductMajorVersion')
```

Table 18 shows the mapping between the product version and the SQL version.

Table 18: Project Major Version

Major version ID	SQL version
8	SQL 2000
9	SQL 2005
10	SQL 2008
10.5	SQL 2008 R2
11	SQL 2012
12	SQL 2014
13	SQL 2016
14	SQL 2017
15	SQL 2019

We will cover the **`SERVERPROPERTY`** function in more detail later in this chapter.

CLR version information

You can use the **`sys.dm_clr_properties`** view to determine what version of the Common Language Runtime (CLR) is installed on the SQL Server box. The view returns three rows with a name and value set of columns. Table 19 shows the CLR properties.

Table 19: CLR properties

Name	Value
`directory`	C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
`version`	v4.0.30319
`State`	CLR is initialized

The **`dm_clr_loaded_assemblies`** view will show all CLR assemblies running on the SQL server. SQL will generally keep these assemblies loaded for performance reasons but can unload them in a memory pressure situation.

Memory

SQL Server is designed to manage memory itself, rather than require administrators to allocate the memory. Basically, SQL will greedily take as much memory as it can get but will release memory to the operating system if the OS is needy (low memory situation). You can use the **`dm_os_sys_memory`** view to query the amount of memory on the server. Code Listing 12 is an example query.

Code Listing 12: Server memory

```sql
----------------------------------------------------
-- Script: Server_memory.sql
-- Reports memory usage on the server
----------------------------------------------------
SELECT
       [total_physical_memory_kb] / 1024 as TotalPhysMemoryMB
      ,[available_physical_memory_kb]/ 1024 as AvailMemoryMB
      ,[total_page_file_kb]/1024 as PageFileTotalMB
      ,[available_page_file_kb]/1024 as PageFileAvailMB
      ,[system_memory_state_desc]
  FROM [sys].[dm_os_sys_memory]
```

The **`system_memory_state_desc`** fields indicate whether memory is high (SQL can keep using it) or low (SQL needs to release some to operating system). Ideally, there are not a lot of memory-intensive processes running on the SQL server machine.

Disk usage

You can use the **`xp_fixedDrives`** stored procedure or, starting with SQL 2017, the new Dynamic Management view called **`dm_os_enumerate_fixed_drives`**. The stored procedure returns two columns: the drive letter and megabytes free. The new view returns the drive path and drive type (usually fixed or network), and the bytes free. Code Listing 13 uses the new view.

Code Listing 13: Drive information

```
---------------------------------------------
-- Script: Basic_Drive_info.sql
-- Show free space on drives
---------------------------------------------
if object_id('sys.dm_os_enumerate_fixed_drives') is not null
      SELECT fixed_drive_path,drive_type_desc,
              free_space_in_bytes/(1024*1024) as MB_Free
      FROM sys.dm_os_enumerate_fixed_drives
else
      exec xp_fixedDrives
```

You can combine this view with the **`sys.master_files`** view to determine where the various databases reside. Code Listing 14 shows the **master** and **tempdb** databases, as well as the database and log files for the current database.

Code Listing 14: Where databases are stored

```
----------------------------------------------------
-- Script: Detailed_Drive_info.sql
-- Show free space and system and database files
----------------------------------------------------
IF object_id('sys.dm_os_enumerate_fixed_drives') is not null
      SELECT mf.type_desc,mf.name,mf.physical_name,
                  fd.drive_type_desc,fd.free_space_in_bytes/(1024*1024)
as MB_Free
      FROM sys.master_files mf
      JOIN sys.dm_os_enumerate_fixed_drives fd
             on substring(mf.physical_name,1,3)=fd.fixed_drive_path
      WHERE database_id in (1,2,db_id())
ELSE
      BEGIN
            create table #tmpDrives (drive char(1),free_space_in_bytes
bigint)
            INSERT INTO #tmpDrives
            exec xp_fixeddrives
            SELECT mf.type_desc,mf.name,mf.physical_name,
                  'FIXED' as
drive_type_desc,round((fd.free_space_in_bytes*1.0)/1024,0) as MB_Free
            FROM sys.master_files mf
            JOIN #tmpDrives fd on
substring(mf.physical_name,1,1)=fd.drive
            WHERE database_id in (1,2,db_id())
            DROP TABLE #tmpDrives
      END
```

You should expect to see that the log and data files are stored on separate drives, and that the **tempdb** files (could be multiple files) are on their own drive, as well. While this hard drive configuration could vary, those are the generally recommended guidelines for performance purposes.

Enumerate file system

SQL Server 2017 added a new table-valued function called **dm_os_enumerate_file_system.** This function takes two parameters, the starting folder, and a search pattern. For example, to find out if any new DLLs were added recently, you could run the following SQL command.

```
SELECT * from sys.dm_os_enumerate_filesystem('c:\windows\system32\','*.dll')
WHERE creation_time>=dateadd(m,-3,getDate())
```

Registry information

SQL Server uses the system registry of the server machine to hold several settings, such as the SQL image, startup parameters, or port. You can use the **dm_server_registry** view to peek at these registry settings.

```
SELECT * FROM [sys].[dm_server_registry]
```

Table 20 shows some of the values this view returns.

Table 20: dm_server_registry

registry_key	value_name	value_data
HKLM...\MSSQLSERVER	ObjectName	YPRIMECLOUD\svc_sql_az-irtsqldev
HKLM...\MSSQLSERVER	ImagePath	C:\Program Files\...\sqlservr.exe
HKLM...\SQLSERVERAGENT	ObjectName	svc_sql_az-irtsqldev@yprimecloud.local
HKLM...\SQLSERVERAGENT	ImagePath	C:\Program Files\...\SQLAGENT.EXE
HKLM...\SQLSERVERAGENT	DependOnService	MSSQLSERVER
HKLM\... CurrentVersion	CurrentVersion	14.0.1000.169
HKLM\... \Tcp	TcpDynamicPorts	1434

Databases on the server

A typical SQL server has multiple databases on it, some required by the server and those for your application data. The **sys.databases** view provides information about all the databases installed on the server. Code Listing 15 is a query that returns basic database names and version information.

Code Listing 15: Server databases

```
---------------------------------------------
-- Script: Server_databases.sql
-- Server databases and versions
---------------------------------------------
SELECT database_id,[name],create_date,
CASE compatibility_level
WHEN 80 THEN 'SQL 2005'
when 90 then 'SQL 2005'
when 100 then 'SQL 2008'
when 110 then 'SQL 2012'
when 120 then 'SQL 2014'
when 130 then 'SQL 2016'
when 140 then 'SQL 2017'
when 150 then 'SQL 2019'
else 'Unknown version'
end as SQL_Level
from sys.databases
order by database_id
```

The first four databases (**master**, **tempdb**, **model**, and **msdb**) are SQL internal databases.

Files for current database

The SQL tables are stored in a physical disk file (MDF files), and you can determine the files that are holding the current database tables using the **`sys.database_files`** view. Code Listing 16 shows a query to return the database file names.

Code Listing 16: Database files

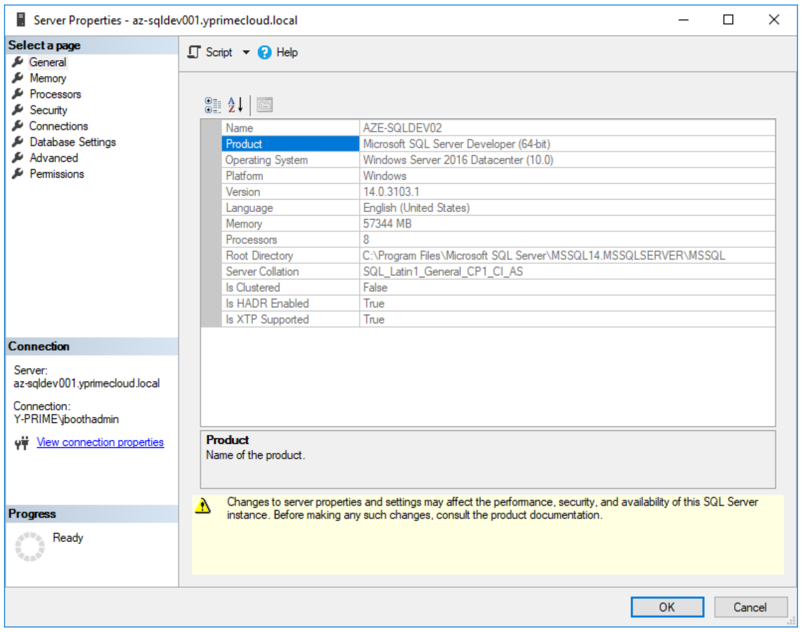
```
---------------------------------------------
-- Script: Database_files.sql
-- Files for current database
---------------------------------------------
SELECT type_desc,name,physical_name
FROM sys.database_files
```

The **`type_desc`** will be either **`ROWS`** or **`LOGS`**. The logs and rows should be on separate drives for better performance.

Configuration

The **`sys.configurations`** view has key fields of a name, value, and description. Many of the configuration options are represented by a named row in this view. You can visit the Microsoft website to determine the usage of the various settings.

You can also view most of the configuration information in SSMS by opening the **Properties** dialog on the server name, as shown in Figure 2.



Name	AZE-SQLDEV02
Product	Microsoft SQL Server Developer (64-bit)
Operating System	Windows Server 2016 Datacenter (10.0)
Platform	Windows
Version	14.0.3103.1
Language	English (United States)
Memory	57344 MB
Processors	8
Root Directory	C:\Program Files\Microsoft SQL Server\MSSQL14.MSSQLSERVER\MSSQL
Server Collation	SQL_Latin1_General_CP1_CI_AS
Is Clustered	False
Is HADR Enabled	True
Is XTP Supported	True

Product
Name of the product.

Changes to server properties and settings may affect the performance, security, and availability of this SQL Server instance. Before making any such changes, consult the product documentation.

Server:
az-sqldev001.yprimecloud.local

Connection:
Y-PRIME\jboothadmin

Ready

Figure 2: Server properties

Reading configuration data

You can run queries against the **`sys.configurations`** view to get the values of the settings. For example, the following SQL query checks whether CLR (common language runtime assemblies) are allowed on this server.

```
SELECT * FROM sys.configurations WHERE [name]='clr enabled'
```

You can also use the **`sp_configure`** stored procedure to look at any of the settings. You specify the setting name or leave off the parameter to see all settings available.

```
EXEC sp_configure 'clr enabled'
```

Advanced settings

Many of configuration settings are considered "advanced" as determined by the **`Is_advanced`** flag in the **`sys.configurations`** view. For example, the **`xp_cmdshell`** allows users to issue operating system commands on the server. By default, SQL Server is configured with this option disabled.

Updating configuration data

Configuration data is updated using the **`sp_configure`** stored procedure. It gets passed two parameters: the configuration name and new value. For example, the following command will enable CLR assemblies.

```
EXEC sp_configure 'clr enabled',1
```

SQL administrators will generally use scripts of **`sp_configure`** commands to configure the server. The Microsoft defaults are generally set toward a minimum machine, so the server will run even in lower memory/hardware configurations. Open connections allowed, server memory and query memory are often customized to get better performance based on knowledge of your server's hardware.

Note: Updating server configuration is the realm of experienced SQL administrators. In a well-designed system, developers would not have access to update the configuration. While you can use the configuration view to see how the server is set up, leave the configuration changes to the administrators.

SERVERPROPERTY

The **`SERVERPROPERTY`** function also provides a good deal of information about the server. It takes a single parameter, the property name, and returns the current property value. For example, the following code snippet shows the edition of SQL Server being run.

```
select SERVERPROPERTY('edition') as ServerEdition
```

Table 21: SERVERPROPERTY

ServerEdition
Developer Edition (64-bit)

The Microsoft website provides details as to the various server property parameters.

You can use the **`SERVERPROPERTY`** function to put together a detailed list of server information, as shown in Code Listing 17.

Code Listing 17: Server property snapshot

```
---------------------------------------------
-- Script: ServerProperties.sql
-- Reporting server information
---------------------------------------------
select 'SQL Server: ' as Label,
SERVERPROPERTY('ProductVersion') as Version,
SERVERPROPERTY('edition') as ServerEdition,
SERVERPROPERTY('productLevel') as ServerEdition,
SERVERPROPERTY('MachineName') as Machine,
CASE
SERVERPROPERTY('IsIntegratedSecurityOnly')
WHEN 0 Then 'SQL and Windows logins'
ELSE 'Windows Authentication'
END as AuthMode
```

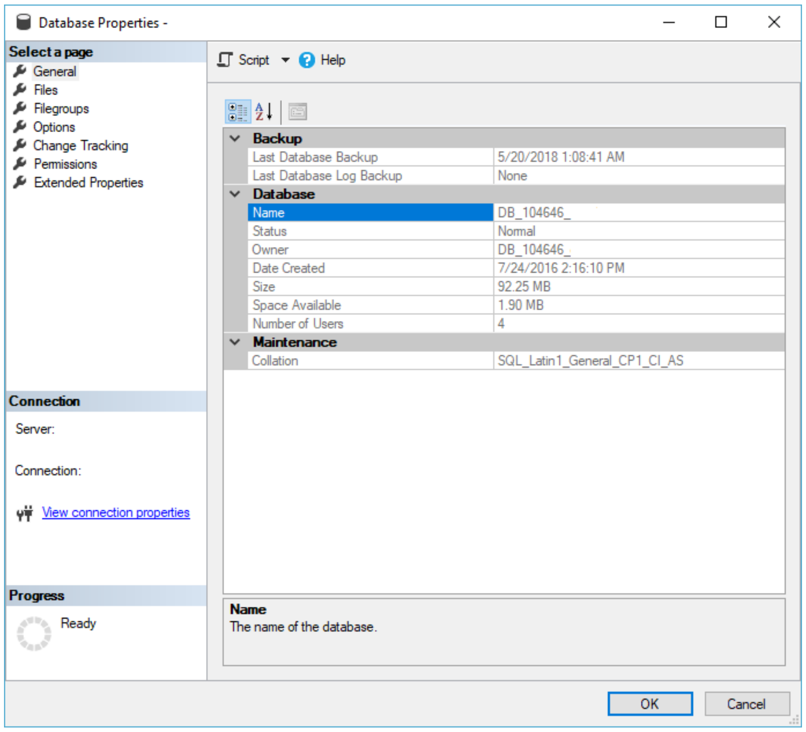
Summary

In this chapter we covered a few of the dynamic management views you can use to explore the details of your SQL Server installation. There are over 100 different views to provide all sorts of server information. Hopefully, this chapter whetted your appetite to explore them further.

We will cover some additional views in later chapters and discuss indexing and performance.

Chapter 4 Database Properties

Within a SQL Server instance, there can be any number of databases containing related tables and objects. In this chapter, we will explore how to use the SQL views to determine information about the individual databases. Figure 3 shows the database properties table from SSMS.



Backup	
Last Database Backup	5/20/2018 1:08:41 AM
Last Database Log Backup	None
Database	
Name	DB_104646_
Status	Normal
Owner	DB_104646_
Date Created	7/24/2016 2:16:10 PM
Size	92.25 MB
Space Available	1.90 MB
Number of Users	4
Maintenance	
Collation	SQL_Latin1_General_CP1_CI_AS

Name
The name of the database.

Figure 3: Database properties

Note: Your account may not have permission to access the various views described in this chapter. You will generally need the VIEW DATABASE STATE permission for most of these queries.

Sys.databases

The primary view for database information is the **`sys.databases`** view. This view holds a row of information for each database in the server. You can restrict it to just the current database by filtering to the current database ID, as shown in the following code.

```
SELECT * from sys.databases
WHERE database_id=db_id()
```

The function **`db_id()`** returns the numeric ID of the current database (or takes a parameter of database name and returns the database ID).

DATABASEPROPERTYEX()

The SQL function **`DATABASEPROPERTYEX ()`** provides additional information about any database in the system. It takes two parameters: the database name (using **`db_name()`** for current database) and the property you want to view.

General information

The General tab of the Properties page provides some simple status information, such as database size and date of the last backup. This information can be assembled using the **`sys.databases`** view and the **`DATABASEPROPERTYEX()`** function.

Basic information

Code Listing 18 uses the **`sys.databases`** view and **`DATABASEPROPERTYEX()`** function to duplicate much of the information from the General tab. In the following query, we are filtering to just the current database on the **`WHERE`** clause. If you remove the **`WHERE`** clause, you can obtain the general information for all databases to which you have access.

Code Listing 18: Database information

```
---------------------------------------------
-- Script: DatabaseInformation.sql
-- Some basic database information
---------------------------------------------
SELECT      db.name AS databaseName,
            DATABASEPROPERTYEX(db.name, 'Status') AS 'Status',
            su.Name AS 'Owner',
            db.create_date AS 'Date Created',
            CONVERT(VARCHAR,mf.size*8/1024)+' MB' AS [Total disk space]
FROM sys.databases db
JOIN (select database_id, sum(size) AS Size
      from sys.master_files group by database_id) mf ON
db.database_id=mf.database_id
LEFT join sys.sql_logins  su ON su.sid=db.owner_sid
WHERE db.database_Id=db_Id()
```

Table 22 shows the sample output.

Table 22: Sample database information

databaseName	Status	Owner	Date created	Total disk space
JDB Demo	ONLINE	sa	2019-07-31 21:27	144 MB

Backup information

The backup information is retrieved from the **`backupset`** table in the MSDB (Microsoft Database) database. We can retrieve the backup information for both the data and the logs using Code Listing 19.

Code Listing 19: Database backup information

```
---------------------------------------------
-- Script: BackupInfo.Sql
-- Backup information for current database
---------------------------------------------
SELECT
case type
when 'D' then 'Database'
when 'L' then 'Logs'
end as BackupType,
max(backup_finish_date) as LastBackup
FROM msdb.dbo.backupset
WHERE database_name=db_name()
GROUP BY database_name,type
```

There is other information available in the **`backupset`** table, such as the backup size, whether it is encrypted or not, recovery model, compressed size, etc. Although we are only interested in the backup dates, you might find occasional need to access the other fields.

Size information

The size of any database can be retrieved easily from the **`sys.master_files`** view, but the actual use (needed to compute space available) requires a bit more effort. Code Listing 20 provides this information.

Code Listing 20: Size information for database

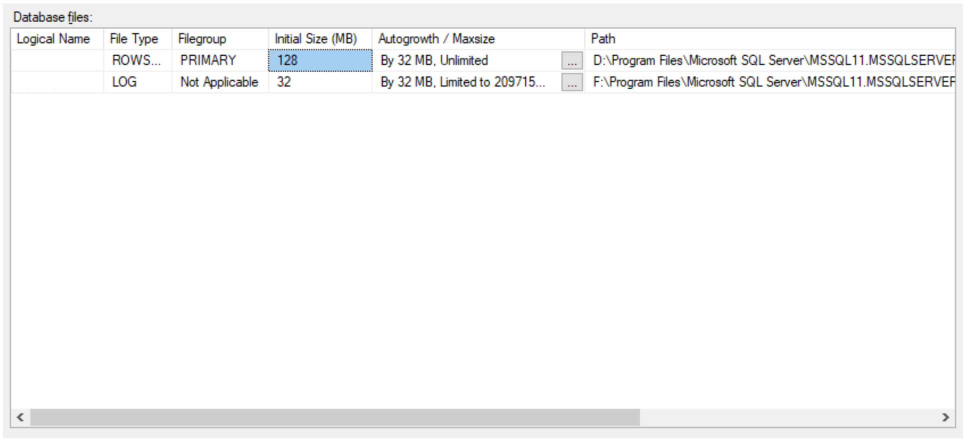
```
---------------------------------------------
-- Script: DBSizeInfo.Sql
-- Size information for the current database
---------------------------------------------
SELECT   db_name() as database_name,
   ltrim(str((CASE
         WHEN sf.dbsize >= pt.reservedpages
   THEN (convert(DECIMAL(15,2),sf.dbsize) -
         convert(DECIMAL(15,2),pt.reservedpages)) * 8192 / 1048576
         ELSE 0 END),15,2) + ' MB') as 'Space Available'
FROM (
      SELECT
sum(convert(BIGINT, CASE WHEN sf.STATUS & 64 = 0 THEN size ELSE 0 END))
as dbSize
      FROM dbo.sysfiles sf
      ) AS sf,
      ( SELECT reservedpages = sum(a.total_pages)
            FROM sys.partitions p
            INNER JOIN sys.allocation_units a
ON p.partition_id = a.container_id
      ) AS pt
```

The **`sp_spaceused`** stored procedure returns this information as well, but it currently returns two result sets, so you cannot execute it to a temporary table.

Note: ***SQL 2016 added a new parameter, @oneResultSet, which allows sp_spaceused to return a single result if set to 1.***

Files

The Files tab on the database properties shows basic information about the files and growth on the database.



Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize		Path
	ROWS...	PRIMARY	128	By 32 MB, Unlimited	...	D:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVEF
	LOG	Not Applicable	32	By 32 MB, Limited to 209715...	...	F:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVEF

Figure 4: Files information

The information about the database files and growth is available from the **`sysFiles`** view. Code Listing 21 converts the data to mimic the Files tab available from the SSMS Properties tab.

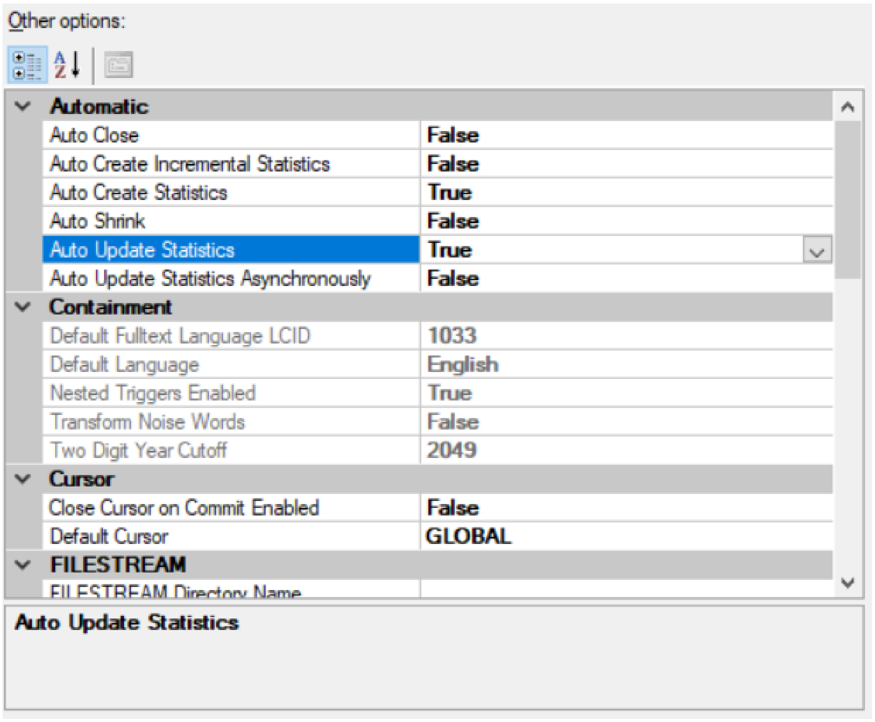
Code Listing 21: File information for current database

```
---------------------------------------------
-- Script: DBFileInfo.Sql
-- File information for current database
---------------------------------------------
SELECT sf.[name] AS Logical_Name,
      CASE
      WHEN Status = 2 THEN 'ROWS Data'
      WHEN Status = 66 THEN 'LOG'
      END AS FileType,
      IsNull(fg.type_desc,'Not Applicable') AS FileGroup,
      size*8.0/1024 as 'Current Size (MB)',
      growth*8.0/1024 as 'Autogrowth (MB)',
      CASE
      WHEN maxsize = 0 THEN 'No growth'
      WHEN maxsize < 0 THEN 'Unlimited'
      WHEN maxsize*8.0/1024 >=cast(268435456/8 as bigint) THEN '2
TerraBytes'
      ELSE CAST(Round(maxsize*8.0/1024,0) as varchar(20))+'MB'
      END as MaxSize,
      FileName AS 'Path'
FROM sys.sysfiles sf
LEFT join sys.filegroups fg on fg.data_space_id=sf.fileid
```

Note that sizes are expressed in pages (one page is 8K bytes), so we multiply the size by 8 to get the pages converted to bytes, and then divide by 1,024 to show the results in megabytes.

Options

Many of the database flags and options shown on the Options tab (Figure 5) can be obtained via the **DATABASEPROPERTYEX()** SQL function.



Other options:

Automatic	
Auto Close	False
Auto Create Incremental Statistics	False
Auto Create Statistics	True
Auto Shrink	False
Auto Update Statistics	True
Auto Update Statistics Asynchronously	False
Containment	
Default Fulltext Language LCID	1033
Default Language	English
Nested Triggers Enabled	True
Transform Noise Words	False
Two Digit Year Cutoff	2049
Cursor	
Close Cursor on Commit Enabled	False
Default Cursor	GLOBAL
FILESTREAM	
FILESTREAM Directory Name	

Auto Update Statistics

Figure 5: Database options

In addition to the **DATABASEPROPERTYEX** function, most of these bit fields also exist in the **sys.database** view. Table 19 lists the key column in **sys.databases** to see the various configuration options.

Automatic settings

Be aware that the automatic settings can impact server performance. For example, if the last user logs out and the database closes automatically, the next user will experience a slight delay as the database gets re-opened.

Table 23: Automatic settings

Column name	Description
`is_auto_close_on`	Will database close after last user logs off?
`is_auto_create_stats_incremental_on`	Rather than full scan, statistics are only created for table partitions that might have changed, which can increase performance while creating statistics.
`is_auto_create_stats_on`	Should the optimizer create statistics on columns that do not have statistics as they are used in a query?
`is_auto_update_stats_on`	If SQL detects certain data modifications that indicate statistics might be out of date, it updates them.
`is_auto_update_stats_async_on`	When on, the query is run first, and then stats are updated. When off, outdated stats are updated first.

If you do not automatically update stats, you should put a job or plan in to do so. Outdated statistics can cause the query optimizer not to generate an optimal plan, and the query will run slower than it could.

Tip: If you are unfamiliar with statistics, imagine a table holding a list of students, containing name, gender, and GPA. If a query was run to determine female students with a 3.0 or better GPA, the optimizer would likely search the GPA column first, and then consider gender. This assumes that GPA 3.0 would return fewer records to check gender against. However, if the system was aware that this was an all-male school with only four female students, it would likely check gender first, and then GPA. Statistics provide the information to allow the query optimizer to create the best query plan.

ANSI and NULL settings

These options can impact the way SQL queries and stored procedures operate and can be different between databases on the same server. If you are writing stored procedures, be sure you review these options to make sure your code takes them into account. Table 24 lists some of the settings that impact ANSI and NULL behavior in each database.

Table 24: ANSI and NULL options

Column name	Description
`is_ansi_nulls_on`	When ON, you need the IS NULL operator to test for NULL value (ANSI Standard). When OFF, = NULL will find null values.
`is_ansi_warnings_on`	When ON, any aggregate function (`SUM`, `COUNT`, etc.) will issue a warning if NULL values appear.
`is_arithabort_on`	If ON, a query will abort for overflow or div by zero errors.
`is_concat_null_yields_null_on`	If ON, string + NULL results in NULL value.
`is_numeric_roundabort_on`	If ON, a loss of numeric precision will return an error, otherwise the value will be rounded, and no warning returned.

For a programming example, the code in Code Listing 22 will return different results depending on the setting of **`is_concat_null_yields_null_on`.** (Note that this example assumes the database has a table called **`site`**.)

Code Listing 22

```
---------------------------------------------
-- Script: TellUserAboutMissingName.sql
-- Warn user to assign a name to the site
---------------------------------------------
SELECT  IsNull('Site Name:'+name,'Please assign a name to site '+
  cast(siteNumber as varchar)) as Msg
FROM site
```

Since these settings can change the behavior of SQL queries, it is at a minimum a good idea to check them in the database you are writing code in. Even better, set them to your expected value for the procedure and then restore them when done.

Note: In a future version of SQL Server, you will not be able to change the value of CONCAT equals NULL; it will also default to ON.

If you find some inconsistent behavior between databases, be sure to check these options that impact how the server operates.

Other database properties

There are a fair number of other properties that appear on the Properties tab and can be read from the databases view. Table 25 lists three of these.

Table 25: Other options

Column name	Description
`user_access_desc`	`SINGLE_USER`, `RESTRICTED_USER`, `MULTI_USER`
`is_read_only`	Is database in read-only mode?
`default_language_name`	Default language (English) for the current database.

The best way to get familiar with them is to look at the database properties tab in SSMS and find the corresponding option in the **`sys.databases`** view.

DATABASEPROPERTYEX()

In addition to the databases view, you can also use the SQL **`DATABASEPROPERTYEX()`** function to get a lot of the configuration options, as well. The function takes two parameters: the name of the database and the property you want to look at.

Many of the properties provide the same information as the **`sys.database`** view. A few sample properties are shown in Table 26.

Table 26: DATABASEPROPERTYEX() parameters

Column name	Description
`LCID`	Locale identifier for Windows
`Collation`	Collation name for the database
`IsNullConcat`	How is NULL concatenation handled?

You can see the list of all parameters on the Microsoft website.

Comparing two databases

Sometimes it is useful to compare properties between databases, and **`DATABASEPROPERTYEX`** can be very handy for that. Code Listing 23 shows code that compares two databases for a list of property values.

Code Listing 23: Compare database properties

```sql
---------------------------------------------
-- Script: CompareDBProperties.sql
-- Compare some properties between databases
---------------------------------------------
DECLARE @targetDB varchar(128)
SET @targetDB=''        -- SET NAME HERE
```

```
CREATE TABLE #tmpProps (PropertyName varchar(100))
INSERT INTO #tmpProps values ('LCID'),('Collation'),
 ('UserAccess'),('IsArithmeticAbortEnabled'),
('IsNullConcat')
SELECT PropertyName,
       DATABASEPROPERTYEX(db_name(),PropertyName) as CurrentDB_Property,
         DATABASEPROPERTYEX(@targetDB,PropertyName) as TargetDB_Property
FROM #tmpProps
DROP TABLE #tmpProps
```

You can adjust the contents of the **`#tmpProps`** table to see the properties you want to compare.

Database permissions

There are two system views that you can use to determine who has various types of permissions for your database.

Database permissions

This view lists all the database permissions, whether they've been granted or denied, and the group, role, user, etc., who was granted the permission and who granted the permission. Table 27 shows the fields in the view.

Table 27: Permissions view

Column name	Description
`class`	Number class value.
`class_desc`	Description of the class value (`DATABASE`, `OBJECT_OR_COLUMN`, `SCHEMA`, etc.).
`major_id`	0 for the database itself; >0 for user objects; <0 for system objects You can use the `OBJECT_NAME()` function to get the object name associated with the `major_id` field.
`minor_id`	Most often 0, or the column ID number of a table/view object.
`grantee_principal_id`	User/role/group who has been granted or denied permission.
`grantor_principal_id`	User/role/group who granted or denied the permission to the grantee.
`type`	Short code for type of permission.
`permission_name`	Full name of permission, `UPDATE`, `SELECT`, etc.

Column name	Description
`state`	D – Deny R – Revoke G – Granted W – Grant with option to grant
`state_desc`	Full name of state code

database_principals

This view lists the various database users, roles, groups, etc., that have access to the database. Table 28 lists the key columns needed to show permissions within the database.

Table 28: Database principals

Column name	Description
`Name`	Object name (role, user, group).
`principal_id`	ID for object, used to link with permissions table (grantor and grantee).
`type`	One-letter code for type object R – Database role S – SQL user G – Windows group See complete list of types here.
`type_desc`	Full name of type code.
`create_date`	Date this object was created.
`modify_date`	Date object was modified.
`authentication_type`	0 – None 1 – Instance 2 – Database (SQL Login) 3 – Windows Auth

We can join the **`permissions`** and **`principals`** views to explore who has what permissions in our database. The next section lists a few sample scripts to explore permissions.

Who can edit?

Code Listing 24 shows a list of all users, roles, groups, etc., that can manipulate data with your database.

Code Listing 24: Who can edit?

```
------------------------------------------------
-- Script: WhoCanEdit.SQL
-- List users, groups, roles with edit ability
------------------------------------------------
SELECT
pr.name,pr.type,p.permission_name
FROM sys.database_permissions p
JOIN sys.database_principals pr ON pr.principal_id=p.grantee_principal_id
WHERE permission_name IN ('DELETE','UPDATE','INSERT') AND state='G'
```

Who can select database objects?

You might want to see which groups and roles can select from various tables, including system metadata tables. Code Listing 25 provides this information.

Code Listing 25: Who and what can they select?

```
--------------------------------------------------------
-- Script: WhoHasSelectRights.sql
-- List users, groups, roles with selectable objects
--------------------------------------------------------
SELECT
pr.name,pr.type,p.permission_name,OBJECT_NAME(major_id) as AllowedView
FROM sys.database_permissions p
JOIN sys.database_principals pr ON pr.principal_id=p.grantee_principal_id
WHERE permission_name IN ('SELECT') AND p.state='G'
ORDER BY AllowedView
```

What can the public role do?

You might also want to see what rights the public role has, since all users inherit these rights. Code Listing 26 shows how to find what everyone can do.

Code Listing 26: Public role permissions

```
--------------------------------------------------------
-- Script: WhatCanPublicDo.SQL
-- List all permissions granted to public role
--------------------------------------------------------
SELECT
      p.permission_name,OBJECT_NAME(major_id) as AllowedView
FROM sys.database_permissions p
JOIN sys.database_principals pr on pr.principal_id=p.grantee_principal_id
```

```
WHERE pr.name = 'Public' and state='G' and class<>0
ORDER BY AllowedView
```

In a production environment, you should limit what the public role has access to. Even if just **SELECT** rights, a lot of the metadata could be exposed to the public role.

Summary

SQL provides a lot of information about the database you are working in. Every property that you can see on the Properties tab in SSMS can be pulled from system views or functions.

Chapter 5 Tables and Columns

Although we covered the information schema views back in Chapter 2, SQL Server provides additional views and procedures we can use to view our tables and columns in more depth. In this chapter, we will look at the **sys.tables** and various **sys** column views to provide some table and column analysis.

Note: Many of these views are derived from sys.objects. Every "object" in a SQL database (tables, procedures, triggers, etc.) has an object ID associated with it, and sys objects hold the information about the object. The OBJECT_NAME() SQL function returns an object name from the object_id parameter.

Tables

We can join the **sys.tables** view with other views to determine information about our table design. This allows us to perform additional table analysis beyond the basic information schema views. When referencing the **sys.tables** view, we can use the **object_schema_name** and **object_name** SQL functions to create a table name (schema + table name).

sys.identity_columns (starting with SQL 2008)

This table holds information about tables that have identity columns present.

Code Listing 27: Identity columns

```
---------------------------------------------
-- Script: Identity_columns.sql
-- List all identity columns
---------------------------------------------
SELECT
OBJECT_SCHEMA_NAME(st.object_id)+'.'+st.name AS [TableName],
ic.name AS KeyName,
t.name AS dataType,
ic.seed_value,ic.increment_value,
isNull(ic.last_value,0) AS Last_Value
FROM sys.tables st
JOIN sys.identity_columns ic ON ic.object_id=st.object_id
JOIN sys.types t ON t.system_type_id=ic.system_type_id
ORDER BY [TableName]
```

This listing shows tables that have an **Identity** column, along with the seed value and increment for the column. If the table has rows, the **Last_value** column will report the last seed number.

Tip: You can use the DBCC CHECKIDENT ('[TableName]', RESEED, 0) command to reset an identity column's value.

sys.computed_columns (starting with SQL 2008)

This view allows you to get the definitions for any computed columns in the database. The definition will be a SQL expression that is used to provide the column value.

Code Listing 28: Computed columns

```
---------------------------------------------
-- Script: Computed_Columns.sql
-- List all computed columns
---------------------------------------------
SELECT
OBJECT_SCHEMA_NAME(st.object_id)+'.'+st.name AS [TableName],
cc.name AS KeyName,
t.name AS dataType,
cc.definition AS [Column_Definition],cc.is_persisted
FROM sys.tables st
JOIN sys.computed_columns cc on cc.object_id=st.object_id
JOIN sys.types t on t.system_type_id=cc.system_type_id
ORDER BY [TableName]
```

Note: If a computed column is persisted, the value will be stored on disk, and can be used for indexing, checking constraints, etc. It will be updated when the data is updated. If it is not persisted, the value will be virtual and will be computed every time the column is referenced.

sys.default_constraints (starting with SQL 2008)

A database table may have a default value to provide during an **`INSERT`** when the corresponding field is NULL. The following script lists all the default constraints and the table and columns they are found in.

Code Listing 29: Default constraints

```
---------------------------------------------
-- Script: Default_Constraints.sql
-- List all default constraints
---------------------------------------------
SELECT object_schema_name(dc.parent_object_id)+'.'+
       object_name(dc.parent_object_id) as TableName,
 c.name as ColumnName,
 dc.definition
FROM sys.default_constraints AS dc
```

```
INNER JOIN sys.columns AS c
ON dc.parent_object_id = c.object_id
AND dc.parent_column_id = c.column_id
ORDER BY tableName,columnName
```

Note: If a column has a default of NewID() to generate a new UniqueIdentifier and this column is part of an index, you should consider using NewSequentialId() instead. Because NewID() generates a random unique identifier, it is likely to cause an index split, as it attempts to insert the record into the index. (The NewSequentialID() function generates a unique identifier higher than the prior one.) So while you can improve creation performance with NewSequentialID, it is also a privacy risk to use sequential (hence predictable) identifiers.

sys.index_columns (starting with SQL 2008)

This script will identify all the columns that are used as indexes in the various tables. This can be a handy way to determine whether a new search you want to add is already indexed in the table (improving performance).

Code Listing 30: Index columns

```
---------------------------------------------
-- Script: Index_columns.sql
-- List all indexed columns
---------------------------------------------
SELECT st.name as TableName
    ,i.name as IndexName
    ,COL_NAME(ic.object_id,ic.column_id) AS ColumnName
FROM sys.indexes AS i
INNER JOIN sys.index_columns AS ic ON i.object_id = ic.object_id AND
i.index_id = ic.index_id
JOIN sys.tables st on st.object_id=ic.object_id
ORDER BY [TableName],ic.index_column_id,ColumnName
```

You can add the **`WHERE`** expression **`ic.index_column = 1`** to find only those columns that are the first expression in the index. You could also specify all columns you need to retrieve to see whether there is a covering index for your search parameters.

sys.key_constraints (starting with SQL 2008)

This script is used to identify the columns that are being used as primary keys in a table. It shows the table and index name, as well as the column(s) making up the key.

Code Listing 31: Key constraints

```
---------------------------------------------
-- Script: Key_constraints.sql
-- List all key constraint columns
---------------------------------------------
select object_schema_name(tb.object_id)+'.'+tb.name as [TableName],
            object_name(kc.object_id) as IndexName,
            sc.name as ColumnName
from sys.tables tb
join sys.key_constraints kc on kc.parent_object_id=tb.object_id
join sys.index_columns ic
      on ic.object_id=kc.parent_object_id and
kc.unique_index_id=ic.index_id
join sys.columns sc on sc.object_id=ic.object_id and
ic.index_column_id=sc.column_id
where tb.type='U' and kc.type='PK'
order by tableName,ic.index_column_id
```

sys.check_constraints (starting with SQL 2008)

A check constraint is a SQL expression that is applied to a column to validate the type of data allowed in that column. For example, you might use the following **`LIKE`** expression to ensure a zip code field only contains five digits.

`zip LIKE '[0-9][0-9][0-9][0-9][0-9]'`

Code Listing 32 will list all columns in the database that have check constraints applied to them.

Code Listing 32: Check constraints

```
---------------------------------------------
-- Script: Check_constraints.sql
-- List all check constraints
---------------------------------------------
SELECT object_schema_name(dc.parent_object_id)+'.'+
       object_name(dc.parent_object_id) as TableName,
c.name as ColumnName,
dc.definition
FROM sys.check_constraints AS dc
INNER JOIN sys.columns AS c
ON dc.parent_object_id = c.object_id
AND dc.parent_column_id = c.column_id
ORDER BY tableName,columnName
```

sys.masked_columns (starting with SQL 2016)

In SQL 2016, a new feature called Dynamic Data Masking was added, which provides the ability to apply **`masked`** to a column, so users querying the data will not see the actual column contents. To create a masked column, you add the **`MASKED WITH`** (expression) to the table column. For example, to make an email column, you could use the following.

`Email VARCHAR(150) MASKED WITH (Function = ‘email()’)`

When the **`Email`** column appears in a query, it will be displayed as `jXXX@XXX.net`. It is a handy feature for simple security in a database. You can identify all the masked columns in a database using the following query.

Code Listing 33: Masked columns

```
---------------------------------------------
-- Script: Masked_Columns.sql
-- List all masked columns
---------------------------------------------
if SERVERPROPERTY('ProductMajorVersion')>='13'
      SELECT object_schema_name(tb.object_id)+'.'+tb.name as [TableName],
               c.name AS column_name,  c.masking_function
      FROM sys.masked_columns AS c
      JOIN sys.tables AS tb ON c.object_id = tb.object_id
      WHERE is_masked = 1
ELSE
      SELECT 'Requires SQL 2016 or higher'
```

Putting it all together

You can combine the various queries to produce a report of tables and columns in your system, along with column information. Code Listing 34 shows a table/column reporting script.

Code Listing 34: Table/column detail script

```
---------------------------------------------
-- Script: Column_Report.sql
-- List column details
---------------------------------------------
SELECT object_schema_name(tb.object_id)+'.'+
       object_name(tb.object_id) as TableName,
       c.name as ColumnName,
       isNull(pk.PK,'') as IsKey,
       isNull(ic.IdentityColumn,'') as Identity_Column,
       isNull(cc.CheckConstraint,'') as Check_Constraint,
       isNull(dc.DefaultConstraint,'') as Default_Constraint
FROM sys.tables as tb
JOIN sys.columns AS c on c.object_id=tb.object_id
LEFT JOIN (
select  ic.object_id,ic.index_column_id,'PRIMARY' as PK
      from sys.tables tb
      join sys.key_constraints kc on kc.parent_object_id=tb.object_id
      join sys.index_columns ic on ic.object_id=kc.parent_object_id
  and kc.unique_index_id=ic.index_id
      where tb.type='U' and kc.type='PK'
) pk on pk.object_id=tb.object_id and pk.index_column_id=c.column_id
LEFT JOIN (
      SELECT ic.object_id,ic.name,ic.name+' identity('+
                    cast(ic.seed_value as varchar(10))+','+
                  cast(ic.increment_value as varchar(10))+') ' as
IdentityColumn
      from sys.tables st
      join sys.identity_columns ic on ic.object_id=st.object_id
) ic ON tb.object_id=ic.object_id and c.name=ic.name
LEFT JOIN (
      SELECT      dc.parent_object_id,dc.parent_column_id,
                  dc.definition as CheckConstraint
      FROM sys.check_constraints AS dc
      INNER JOIN sys.columns AS c  ON dc.parent_object_id = c.object_id
AND dc.parent_column_id = c.column_id
) cc on cc.parent_object_id=tb.object_id and
cc.parent_column_id=c.column_id
LEFT JOIN (
      SELECT      dc.parent_object_id,dc.parent_column_id,
                  dc.definition as DefaultConstraint
      FROM sys.default_constraints AS dc
      INNER JOIN sys.columns AS c  ON dc.parent_object_id = c.object_id
AND dc.parent_column_id = c.column_id
) dc on dc.parent_object_id=tb.object_id and
dc.parent_column_id=c.column_id
ORDER BY tableName,column_id
```

When this script is run, it will produce a report of all table names and columns, and indicate which columns are primary keys, identity columns, constraints, and so on.

Searching for deprecated columns

In early versions of SQL, there were text and image columns called **`text`**, **`nText`**, and **`image`**. These columns (while still supported) were deprecated in SQL Server 2005. The following script allows you to search for deprecated column types and indicates the appropriate replacement column type.

Code Listing 35: Deprecated columns

```
---------------------------------------------
-- Script: Deprecated_columns.sql
-- List all columns with deprecated types
---------------------------------------------
select t.name,c.name as ColName,  'Deprecated: '+
      CASE
      WHEN tp.name = 'text' then 'Replace [text] with varchar(max)'
      WHEN tp.name = 'ntext' then 'Replace [ntext] with nvarchar(max)'
      WHEN tp.name = 'image' then 'Replace [image] with varbinary(max)'
      ELSE 'Table contains Text,nText, or Image fields' END as Msg
from sys.columns c
join sys.tables t on c.object_id=t.object_id
join sys.types tp on tp.user_type_id=c.user_type_id
```

```
where t.is_ms_shipped=0 and tp.name in ('text','ntext','image')
```

If you are using any of these column data types, you should plan on changing the data type to keep current with SQL Server.

Numeric columns

SQL Server and most database servers perform much better with integer values, rather than a numeric data type. If a decimal or numeric column has a scale of 0 (no decimal place), you should consider replacing that column with the equivalent integer column. Code Listing 36 will search for any numeric columns with a zero scale and suggest the equivalent integer column type.

Code Listing 36: Suggest integer columns

```
---------------------------------------------
-- Script: Suggest_integers.sql
-- Convert numeric columns to integers
---------------------------------------------
SELECT tb.table_schema, tb.table_name, tc.column_name as colname,
CASE
  WHEN numeric_precision <= 2
  THEN 'Convert '+tc.Data_type+'('+
       CAST(numeric_precision as varchar)+',0) to tinyint data type'
  WHEN numeric_precision <= 4
  THEN 'Convert '+tc.Data_type+
     '('+CAST(numeric_precision as varchar)+',0) to smallint data type'
  WHEN numeric_precision <= 9
  THEN 'Convert '+tc.Data_type+
     '('+CAST(numeric_precision as varchar)+',0) to int data type'
  WHEN numeric_precision <= 18
  THEN 'Convert '+tc.Data_type+
     '('+CAST(numeric_precision as varchar)+',0) to bigint data type'
  ELSE 'Consider using an integer data type'
END as Msg
FROM INFORMATION_SCHEMA.columns tc
JOIN INFORMATION_SCHEMA.tables tb ON
    tb.table_name = tc.table_name and tb.table_schema = tc.table_schema
WHERE tb.Table_Type='BASE TABLE'
      AND tc.data_type IN('numeric','decimal')
      AND tc.numeric_scale = 0
      AND tc.numeric_precision <= 18
ORDER BY tb.table_schema,tb.table_name,tc.column_name
```

Approximate column types

The **`float`** and **`real`** column types in a database are approximations, rather than exact values. Generally, graphic applications use floats for smaller storage requirement, and can accept the loss of precision. So, while a float or real data type might be necessary, you should review your usage to make sure it is necessary.

Code Listing 37 searches for float and real columns in your database tables.

Code Listing 37: Real and float columns

```
---------------------------------------------
-- Script: SearchFloatColumns.sql
-- Identify float and real columns
---------------------------------------------
SELECT schema_name(o.schema_id) AS SchemaName,
       o.name AS TableName,c.name AS columnName,
       t.Name AS ColumnType
FROM sys.all_columns c
JOIN sys.objects o ON c.object_id=o.object_id
JOIN sys.types t ON t.user_type_id=c.user_type_id
WHERE t.name IN ('float','real') AND o.type='U'
```

You can read this article to see if floating point arithmetic is necessary for your application.

Unexpected columns

There are often columns in a database table that hold standard information (such as phone numbers, email addresses, and state codes). Sometimes, these columns have unexpected sizes. (For example, one system used a **`varchar(max)`** to store phone numbers).

Max characters

Code Listing 38 shows all columns using **`varchar`** or **`nvarchar max`**, even though the column name suggests it is not needed. You should adjust the list of searched column names for more common suggestions, based on your knowledge of your application.

Code Listing 38: Max columns check

```
---------------------------------------------
-- Script: VarChar_Max_check.sql
-- Maximum columns that might not be needed
---------------------------------------------
SELECT schema_name(o.schema_id) AS SchemaName,
         o.name AS TableName,c.name AS columnName,
         t.Name+'(max)'  AS ColumnType
FROM sys.all_columns c
JOIN sys.objects o on c.object_id=o.object_id
JOIN sys.types t on t.user_type_id=c.user_type_id
WHERE t.name like '%varchar%' and c.max_length < 0
AND o.schema_id <> 4
AND (c.name LIKE '%phone%' or c.name LIKE '%address%')
```

Nondate columns

Another scenario seen in databases is columns that are holding dates, but not using a date column type. Code Listing 39 searches for such a column type.

Code Listing 39: Date columns using other data types

```
-------------------------------------------------------
-- Script: Date_columns_check.sql
-- Date values possibly stored in nondate columns
-------------------------------------------------------
SELECT schema_name(o.schema_id) AS SchemaName,
         o.name AS TableName,c.name AS columnName,
         t.Name AS ColumnType
FROM sys.all_columns c
JOIN sys.objects o on c.object_id=o.object_id
JOIN sys.types t on t.user_type_id=c.user_type_id
WHERE t.name not like '%date%'
AND (c.name LIKE '%date%')
AND o.schema_id <> 4
ORDER BY SchemaName,TableName,columnName
```

Note that in both these queries, we are filtering out **`schema_id 4`** (the **`sys`** schema).

Summary

You can use the various views to optimize your columns, hopefully identifying problematic columns and, where possible, simplifying the data types. SQL Server is a powerful tool, and by giving it the best column types and size, you can improve database integrity and performance.

Chapter 6 Performance

SQL Server is a dynamic system that is constantly running queries, scheduled jobs, and system maintenance. In this chapter, we are going to look at views and functions that allow us a peek into some of the processes and work happening on the server.

What is happening on the server?

The view **`sys.sysprocesses`** provides a list of all connections currently open on the server. Table 29 lists some of the columns you can use to query this view.

Table 29: Sys Processes

Column name	Description
`spid`	SQL Server session ID.
`kpid`	Windows thread ID.
`blocked`	spid of session blocking this process.
`waittime`	How long process has been waiting (milliseconds) or 0.
`lastwaittype`	String description of last wait encountered.
`dbid`	Database ID (use `db_name()` to see name) of database.
`cpu`	Cumulative CPU usage time for this process.
`physical_io`	Cumulative disk reads/writes.
`memusage`	Current number of memory pages allocated to process.
`login_time`	When this process was logged in.
`last_batch`	Last time a statement was run by process.
`open_tran`	Current number of open transactions used by this process.
`status`	String description of current status: • Running • Background • Runnable • Sleeping

`hostname`	Name of the workstation.
`program_name`	Name of the application.
`cmd`	Type of command being executed (`SELECT`, `DELETE`, etc.).
`nt_domain`	Windows domain, if using Windows authentication.
`nt_username`	Username, if Windows authentication or trusted connection.
`loginname`	User's login name.
`sql_handle`	Memory pointer to the currently executing command.
`stmt_start`	Offset into handle of current statement.
`stmt_end`	Ending offset for current statement.

The information in this table provides the ability to determine what exactly the server is doing, and who is doing it. Some example usages appear in the next few queries. Note that **`dbid`** of 1 through 4 are system databases, so activity in those databases is typically done by SQL Services. Database ID number 2 is **`tempdb`**, which might be worth checking out if you hit performance issues.

Who is running SQL Management Studio?

SQL Management Studio allows users to run queries, updates, etc., in a database. Typically, developers and database administrators will be using this tool. Any other users might be worth reviewing.

Code Listing 40: Who is using SSMS?

```
--------------------------------------------------------
-- Script: Find_SMSS.SQL
-- Find users running SQL Server Management Studio
--------------------------------------------------------
select loginame,login_time,cmd
from sys.sysprocesses
where dbid>4 and program_name like '%SQL Server Man%'
order by loginame
```

Note: We once had a user who didn't know the difference between NULL and "NULL", and set all of a particular field to "NULL" (string). It took a bit of digging to realize she had SSMS installed and ran the query, invalidating all the records. (She now has read-only access.)

Similarly, you can identify .NET applications by looking for a program name like **`'.Net%'`**.

Who is blocking others?

You can see who might be blocking other processes using the code in Code Listing 41.

Code Listing 41: Who is blocking?

```
--------------------------------------------------------
-- Script: WhoIsBlocking.sql
-- Report users blocking other users
---------------------------------------------------------
select 'Process '+str(sp.spid)+', user '+
       sp.loginame+' is being blocked by '+str(bl.spid)+
         ' user '+bl.loginame as BlockedMsg
from sys.sysprocesses sp
join sys.sysprocesses bl on sp.blocked=bl.spid
where sp.dbid>4 and sp.blocked <> 0
```

Who has open transactions?

If a user has a transaction open, the tables impacted within that transaction will block other update operations (and possibly select statements, depending on isolation level). You can use the code in Code Listing 42 to identify processes with open transactions.

Code Listing 42: Who has open transactions?

```
--------------------------------------------------------
-- Script: OpenTransactions.sql
-- Sessions with open transactions
---------------------------------------------------------
SELECT 'Process '+ltrim(str(sp.spid))+', user '+
       sp.loginame+' has '+str(sp.open_tran)+' open transactions'
FROM sys.sysprocesses sp
WHERE sp.dbid>4 AND sp.open_tran <> 0
```

You've seen from these examples that you can see what is happening on the server using the view, and possibly diagnose some sessions that could be impacting performance.

What are they doing?

The **`sql_handle`** column in the view provides the ability to see what is being done by the session. You can use the **`sys.dm_exec_sql_text`** table-valued function to look at the actual work being done. Code Listing 43 shows a simple example using the function to see what a **`spid`** is doing.

Code Listing 43: Look at entire query text

```
--------------------------------------------------------
-- Script: EntireQueryText.sql
-- Look at the content of a particular session
---------------------------------------------------------
```

```sql
SELECT sp.spid,sp.loginame, st.text
FROM sys.sysprocesses sp
CROSS APPLY sys.dm_exec_sql_text(sp.sql_handle) st
WHERE sp.dbid>4
AND sql_handle <> 0
AND spid = @YourSpid
```

This will return the entire code being executed by the session. You can also drill down further, if the statement starting offset is known. Code Listing 44 shows the statement being extracted from the full query text.

Code Listing 44: Extracting statement from query

```sql
--------------------------------------------------------
-- Script: ExtractStatement.sql
-- Look at the statement within the session
--------------------------------------------------------
select sp.spid,sp.loginame,
       case
         when sp.stmt_start >=0 and sp.stmt_end>0
         then substring(st.text,sp.stmt_start,(sp.stmt_end-
sp.stmt_start)+1)
         else st.text
         end as QueryText
from sys.sysprocesses sp
CROSS APPLY sys.dm_exec_sql_text(sp.sql_handle) st
where sp.dbid>4
and sql_handle <> 0
and sp.spid = @YourSpid
```

You can use the **dm_exec_sql_text** function to look at cached plans as well. The view **sys.dm_exec_cached_plans** holds query plans that SQL has cached. The **objType** column indicates the type of code, such as a trigger or ad hoc query. Code Listing 45 shows an example of how to look at the top 10 ad hoc queries based on project CPU usage.

Code Listing 45: Top 10 cached plans

```sql
--------------------------------------------------------
-- Script: Top10CachedPlans.sql
-- Top 10 cached plans by usage
--------------------------------------------------------
select top 10 cacheobjtype,objtype,st.text as Query,*
from sys.dm_exec_cached_plans cp
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
where objtype = 'Adhoc'
order by usecounts desc
```

Worst queries

SQL Server keeps query stats in a dynamic management view called **`dm_exec_query_stats`**. This view is very handy for looking at the queries that are potentially causing issues in your server. This view provides the pointer to the code (**`plan_handle`**) as well as counters for key values of the query. Table 30 shows some of the key fields in the view.

Table 30: dm_exec_query_stats

Field	Description
`plan_handle`	Binary pointer to query code.
`total_worker_time`	Time used by the CPU.
`total_physical_reads`	Disk reads performed by the query.
`total_physical_writes`	Disk writes performed.
`total_logical_reads`	Logical reads by query.
`total_logical_writes`	Logical writes by query.
`total_CLR_time`	Time spent in CLR procedures.
`total_elapsed_time`	Time (milliseconds) query takes.
`total_rows`	Number of rows query takes.

While we are looking at the total values, there are corresponding fields for **`last`**, **`min`**, and **`max`** values, as well.

Worst CPU usage

CPU time is measured by the **`worker_time`** value, so we can order by total worker time to identify those plans using a lot of CPU time.

Code Listing 46: Worst five plans based on CPU usage

```
--------------------------------------------------------
-- Script: WorstPlansByCPU.SQL
-- Worst 5 plans based on CPU usage
--------------------------------------------------------
SELECT TOP 5
      st.text as SrcCode,
      qp.query_plan,
      qs.execution_count,qs.last_execution_time
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.plan_handle) st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
```

```
ORDER BY total_worker_time DESC
```

Worst I/O

The input/output (I/O) totals indicate how often a query needs to read something from the disk. Ideally, in a query, you should read the minimum amount of data needed. When a query uses **`SELECT *`** or a lot of table scans, SQL is bringing back more data than is needed. For example, imagine a personnel table that includes a binary image of the person. If your code does a **`SELECT *`** from this table, but only displays the name and phone number, you've had SQL bring back extra data (the binary image, among other fields), when all it needed was two fields.

One of the statistics that tracks how much I/O a query uses is called logical reads.

Code Listing 47: Worst 5 by logical I/O

```
--------------------------------------------------------
-- Script: WorstPlansByIO.SQL
-- Worst 5 plans based on I/O
--------------------------------------------------------
SELECT TOP 5
      st.text as SrcCode,
      qp.query_plan,
      qs.execution_count,qs.last_execution_time
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.plan_handle) st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY total_logical_reads DESC
```

Note: Logical reads versus physical reads: A physical read means the data was pulled from the disk subsystem, while a logical read means the data could have been pulled from the disk. However, it might have come from the memory cache, instead. The amount of memory SQL has helps determine its cache content, so when optimizing a query, focus on logical reads, and let the hardware guys make sure your server has a lot of memory.

When exploring the query stats for optimization purposes, keep in mind that there can be many factors making up a good versus bad query. For example, imagine a query has a high number of logical reads, but returns very few rows. This would suggest bringing in extra fields that are not needed, so you might want to look for **`SELECT *`** statements, or tables with large **`varchar`** fields.

Also, pay attention to the execution count and last execution time. If you are looking to optimize queries, a query that is frequently and recently run should be more of a focus than a query that is run once every month.

Why is a query sometimes slow?

Sometimes a query runs fine, but occasionally will slow down. There can be other factors besides the query itself that can impact performance. Your query does not run in isolation; many other things can be happening on the server. Check these items out before focusing on the fast query itself.

Is another processing blocking your query?

You can investigate the **`sys.dm_os_waiting_tasks`** view for the blocked column of your target **`spid`**. If some other process is blocking your query, focus on the blocker first. Similarly, if another query has open transactions, that might be the culprit slowing your query down.

You can use the SQL global variable **`@@SPID`** to get your session ID in SQL.

Code Listing 48: Who might be blocking me?

```sql
--------------------------------------------------------
-- Script: WhoMightBeBlocking.SQL
-- Who might be blocking me?
---------------------------------------------------------
SELECT wait_type, blocking_session_id,p.program_name,p.loginame
FROM sys.dm_os_waiting_tasks wt
JOIN sys.sysprocesses p on p.sid=wt.blocking_session_id
WHERE  session_id=@@spid
```

Note: You might occasionally see a CXPACKET wait type, and it looks like you are blocking yourself. This can occur when SQL is using hyperthreading, and it broke your query into pieces for each processor to handle, Basically, one piece of your query is waiting for the other piece to complete. These waits will almost always clear themselves.

How busy is TempDB?

TempDB is a shared database file that all databases can use for sorting, temporary tables, etc. If TempDB is busy, or the data files making up TempDB have grown large, this can impact overall query performance. Code Listing 49 shows a sample query to check out TempDB usage.

Code Listing 49: How busy is TempDB?

```sql
--------------------------------------------------------
-- Script: TempDB_Usage.sql
-- Anything slowing tempDB?
---------------------------------------------------------
SELECT 'Blocked processes' as Msg,count(*) as Total
FROM sys.sysprocesses
WHERE db_name(dbid)='tempDB' AND blocked <>0
UNION
```

```
SELECT 'Waiting processes',count(*) as Total
FROM sys.sysprocesses
WHERE db_name(dbid)='tempDB' AND waittime >0
```

Autogrowth and other settings

There are configuration settings that allow the server to grow or shrink the database automatically and to update statistics. If this process gets started by the server, your query might slow down during this time frame. If the **`autogrowth`** parameter is too small, for example, the server may frequently slow down to grow the database. Code Listing 50 reports these potential settings.

Code Listing 50: Autogrowth settings

```
--------------------------------------------------------
-- Script: AutoGrowth.sql
-- Autogrowth settings that might impact performance
---------------------------------------------------------
SELECT
CASE
WHEN growth = 0 then 'No growth allowed'
WHEN status>100 then Cast(growth as varchar(3))+'% growth%'
ELSE cast((growth*8.0/1024) as varchar)+' MB'
END AS growth,
CASE
      WHEN maxsize = 0 THEN 'No growth'
      WHEN maxsize < 0 THEN 'Unlimited'
      WHEN maxsize*8.0/1024 >=cast(268435456/8 as bigint) THEN '2
TerraBytes'
      ELSE CAST(Round(maxsize*8.0/1024,0) as varchar(20))+'MB'
END AS MaxGrowth
FROM sys.sysfiles
```

Tables and indexes

Indexes are one of the key ways to increase performance in a SQL Server application. An index allows one or more columns in a database table to be maintained so that queries can use the smaller index to find the larger data row from the table. Developers will often create an index on the most commonly used fields to improve performance in their applications.

While indexes speed up performance during queries, they negatively impact table update operations. Every time a row is inserted, modified, or removed, the index needs to be updated to reflect the change. There needs to be a reasonable balance between indexes that are needed for querying performance, but without having too many indexes to slow down the CRUD (**`CREATE`**, **`READ`**, **`UPDATE`**, **`DELETE`**) operations.

Duplicate indexes

Sometimes, indexes get created that are duplicates (same columns) of other indexes associated with the table. The following script can be used to identify indexes that contain the exact same column. Note that we also check to make sure the duplicate is not only duplicated by column name, but also whether the key is descending.

Code Listing 51: Duplicated indexes

```
--------------------------------------------------------
-- Script: Duplicated_indexes.sql
-- List all indexes that contain same keys and order
---------------------------------------------------------
SELECT OBJECT_SCHEMA_NAME(tb.object_id)+'.'+tb.name AS [TableName],
      ix.name AS FirstIndex,
      Dupes.name AS SecondIndex,
      c.name AS ColumnName
FROM sys.tables AS tb
JOIN sys.indexes AS ix     ON tb.object_id = ix.object_id
JOIN sys.index_columns ic  ON ic.object_id = ix.object_id
                              AND ic.index_id = ix.index_id
                              AND ic.index_column_id = 1
JOIN sys.columns AS c      ON c.object_id = ic.object_id
                              AND c.column_id = ic.column_id
CROSS APPLY
(     SELECT ind.index_id,ind.name
      FROM sys.indexes AS ind
      JOIN sys.index_columns AS ico ON ico.object_id = ind.object_id
                                    AND ico.index_id = ind.index_id
                                    AND ico.index_column_id = 1
      WHERE ind.object_id = ix.object_id  AND ind.index_id > ix.index_id
                                          AND ico.column_id =
ic.column_id
AND ico.is_descending_key= ic.is_descending_key
) Dupes
ORDER BY [TableName],ix.index_id
```

If the script detects any duplicate indexes, they should be reviewed, and one of the indexes should be removed. You should not remove a clustered index if you have the choice, since the clustered index is generally the fastest indexing option.

Unused indexes

SQL Server has a very useful view, called **`sys.dm_db_index_usage_stats.`** This view keeps track of activity performed against an index file, such as when the index was last used (for a seek, scan, or lookup operation). By using this view, we can create a script to remove indexes that are not used.

Code Listing 52: Unused indexes

```
--------------------------------------------------------
-- Script: Unused_indexes.sql
-- List all indexes that have not been used
---------------------------------------------------------
SELECT OBJECT_SCHEMA_NAME(i.object_id)+'.'+
       OBJECT_NAME(i.OBJECT_ID) AS TableName,
       i.name AS UnusedIndexName, i.type_desc AS index_type
FROM sys.indexes AS i
LEFT JOIN sys.dm_db_index_usage_stats AS usage
            ON usage.OBJECT_ID = i.OBJECT_ID AND i.index_id =
usage.index_id
            AND usage.database_id = DB_ID()
WHERE OBJECTPROPERTY(i.object_id, 'IsIndexed') = 1
      AND usage.index_id IS NULL                -- No entry in usage table
      OR (usage.user_updates > 0                -- Updated by DML command
            AND usage.user_seeks = 0            -- But never used in a
query
            AND usage.user_scans = 0
            AND usage.user_lookups = 0)
GROUP BY i.object_id, i.name, i.type_desc
ORDER BY TableName
```

We are confirming that the table has at least one index (**`ObjectProperty`**). If the index never appears in the usages stats table, or appears (has been updated), but never used in a query, we consider the index unused. You can review the indexes reported and consider removing them to increase performance during your DML operations.

Note: Some indexes might be created in anticipation of how a user might be querying the data, and early in a deployment lifecycle, those indexes might not have been used. Be careful removing "unused" indexes, particularly on a new system.

Missing indexes

When SQL gets a query to run, it invokes the SQL optimizer to determine the most efficient way to run the query. The optimizer takes a lot of factors into consideration, such as which indexes are available, or the size of the tables. One of the first steps is to make a guess as to how much this query will cost (some internal measurement to the optimizer).

Tip: When we see cost times from the optimizer, we tend to want to associate them with real-world costs (time? dollars?). However, there is no real-world meaning to the value. Just know that the higher the cost, the longer the query will take, and more resources will be used.

After the optimizer runs the query, it looks to see if the query could be improved with some additional indexes. It assembles this information into a series of **`missing_index`** views. We can use these views to see if we might be able to improve performance with the addition of an index.

The following query looks at the assembled missing index data, and “suggests” how things can be improved via indexes.

Code Listing 53: Missing indexes

```
--------------------------------------------------------
-- Script: Missing_indexes.sql
-- List all indexes that are missing
---------------------------------------------------------
SELECT DMID.statement as [TableName],
      avg_user_impact,
      unique_compiles,
      equality_columns,
      inequality_columns,
      included_columns
FROM sys.dm_db_missing_index_groups            AS DMIG
INNER JOIN sys.dm_db_missing_index_group_stats AS DMIGS
                       ON DMIGS.group_handle = DMIG.index_group_handle
INNER JOIN sys.dm_db_missing_index_details     AS DMID
                       ON DMID.index_handle = DMIG.index_handle
WHERE database_id>=DB_ID()
ORDER BY avg_user_impact DESC,unique_compiles
```

Understanding the columns is the key to determining whether you should create the recommended index.

The **`TableName`** column is the table that was being queried when the missing index was detected. It includes the database name and schema name.

The **`avg_user_impact`** is a percentage guess as to how much the query would be improved if this index were added. However, it is a guess made by the optimizer, not a guaranteed improvement.

The **`unique_compiles`** is a guess as to the number of queries that would benefit from the addition of the index. Basically, the optimizer says, “I found a number of different queries that I think this index will help.” Table 31 shows some sample results.

Table 31: Sample “missing indexes”

TableName	avg_user_impact	unique_compiles	equality_columns
[dbo].[Device]	83.13	50	[AssignUserId]

This result says that this index would help 50 “queries” and improve them by over 80%. However, keep in mind that these statistics are gathered from the time SQL was last started. If you start SQL every day, 50 queries with an 80%+ improvement is worth considering. If SQL has been up for six months, or you see a small number of compiles or a low improvement percentage, you are probably safe ignoring the recommended index.

The next three columns indicate the fields SQL was using for which it felt an index would be helpful. The columns contain comma-delimited field names to assist in creating the index.

Equality columns

These fields contain the list of columns where SQL was looking for a match (for example, **`UserID = 50`**).

Inequality columns

These are fields where SQL was looking for anything other than a match, such as the following.

`LoginDate > ‘7/1/2019’` or **`IsActive <> 1`**

Included columns

These are columns that SQL recommends you *include* in the index, to help improve performance.

Keep in mind that there are limitations to the missing indexes system, and it is only intended to point a developer or administrator to an area to consider, not to fine-tune index configuration. Some of the limitations are:

- It can only gather 500 index groups.
- The index order is not suggested.
- If the query only contains inequality columns, the cost information is less accurate.
- Filtered indexes are not considered.

There are other tuning tools provided, and any experience reading execution plans can really help fine-tune your indexes and query performance. While the missing index tables might help identify which tables/columns to look at, a good developer or DBA is your best bet to optimize the index usage.

LIKE clause

The **`LIKE`** clause is a powerful SQL feature, allowing a person to find “matches” in a table, rather than exact values. However, it is possible to create a condition where SQL must use a table scan (slower) rather than any indexes to resolve the like expression. This can impact performance when applying the like clause to large tables.

If you were creating a system to allow people to search by last name, you might want to use **`LIKE`** as shown in the following.

```
   Beginning with => LIKE ‘Mc%’
   Ending with => LIKE ‘%son’
```

The first **`LIKE`** clause will use an index (assuming one exists on the last name column). However, the second clause will require a table scan, which can slow performance by quite a bit. To the user of the system, though, the difference might not be understandable as to why one search is quick and the other quite slow.

Simple solution

If you need to improve performance for the second scenario, you can take advantage of SQL's computed and persisted columns.

Add a computed and persisted column that consists of the **`REVERSE(last_name)`** column.

When the user wants to search for last names ending with a pattern, simply reverse their input string and add the wildcard to the end.

```
DECLARE @reversed varchar(32) = REVERSE(@SearchName)+'%'
```

This is a simple solution to address name searching and performance in large tables.

Summary

SQL Server provides the ability to review and analyze your table and index structure, to help eliminate potential development gotchas, and to improve performance. In this chapter, we used these views and functions to give you a heads-up on improving your database performance.

The Dynamic Management Views and functions are very thorough and helpful for exploring what is going on inside your server. We just touched upon a few of them here, but hopefully this chapter will encourage you to explore them further to better optimize your SQL server.

Chapter 7 Security

The data held by an organization is an important business asset, and it is the job of a SQL administrator to ensure this data is secure from hackers and other unauthorized users. In this chapter, we will focus on using the views and features to determine any areas where the SQL databases might be at risk.

Attack surface

The attack surface represents all the areas of a system that an unauthorized user can use to gain access to the system. As an administrator of a SQL server, it is necessary to defend all potential areas, since the hacker only needs to use one to perform nefarious deeds.

Demo databases

SQL Server provides two sample databases (**AdventureWorks** and **WorldWideImporters**). In this script, we will check to see if these sample databases are still installed on a server. It is not a good idea to leave unmonitored sample databases in a production server environment.

Note: Prior SQL versions had sample databases called Northwind and Pubs.

Code Listing 54: Script to check for sample databases

```
--------------------------------------------------------
-- Script: Sample_Databases.sql
-- Are demo/sample databases included on the server?
---------------------------------------------------------
SELECT name as [DatabaseName],
       'This is a demo database, consider removing it' as Msg
 FROM sys.databases
WHERE [name] in ('AdventureWorks', 'WorldWideImporters',
                 'Pubs', 'Northwind' )
```

A database administrator (DBA) should always be aware of what databases are installed on a production server. The following script can be used to identify new databases added within the past two weeks. A development database that gets installed on a production probably lacks security and could give an attacker a way to get to the production server.

Code Listing 55: Recently installed databases

```
---------------------------------------------------------
-- Script: Recently_Installed.sql
-- Any databases recently installed?
---------------------------------------------------------
SELECT name as [DatabaseName],create_date
```

```
FROM sys.databases
WHERE create_date >= dateadd(WEEK,-2,getDate())
```

If the list of databases is small, you could also write a script to identify any databases outside of the expected database list.

Guest user

Every SQL database has a guest user, and although it has minimum privileges, it still represents an attack surface. You cannot remove the guest user, but you can prevent it from being used to access a database. The following script identifies whether the guest account can connect to the current database.

Code Listing 56: Guest Login accounts

```
---------------------------------------------
-- Script: Guest_Logins.sql
-- Suggest removing GUEST login if enabled
---------------------------------------------
SELECT DISTINCT 'Consider disabling the GUEST account in '+db_Name() as
Msg
       FROM sys.database_principals dp
       INNER JOIN sys.server_permissions sp
                  ON dp.principal_id = sp.grantee_principal_id
WHERE name = 'guest' AND permission_name = 'CONNECT'
```

If you've identified that the guest user is still in the database, you can run the following SQL script to prevent the account from accessing the database.

`REVOKE CONNECT FROM guest`

SQL logins

In SQL Server, a login is an ID that allows you to connect to the server itself. This is separate from a user (which is an ID within a database). The **`sys.syslogins`** view contains all the logins on the server. Table 32 lists the key columns in the view.

Table 32: SQL logins view

Field	Description
`SID`	Security identifier.
`createdate`	Date login was added to the system.
`updatedate`	Date login was last updated.
`name`	Login name of the user.

Field	Description
`dbname`	Name of the default database when user connects.
`hasaccess`	Does account have access to the server?
`isntname`	1 = Windows user or group, 0 = SQL Server login
`isntgroup`	1 = A Windows group
`isntuser`	1 = A Windows user
`sysadmin`	1 = Member of sys admin role (can perform any server activity)
`securityadmin`	1 = Member of the security admin role (used to manage logins and properties)
`serveradmin`	1 = Member of server admin role (can change server-wide configuration and shut down the server)
`setupadmin`	1 = Member of setup admin role (add and remove remote servers)
`processadmin`	1 = Member of process admin role (can end processes running SQL Server)
`diskadmin`	1 = Member of disk admin role (manages disk files)
`dbcreator`	1 = Member of dbcreator (create, alter, drop, restore databases)
`bulkadmin`	1 = Member of bulk admin (can run the `BULK INSERT` statement)
`loginname`	Login name for this account.

Note: All SQL logins belong to the public role as well.

Windows users (**`isntuser`**) and Windows group (**`isntgroup`**) have their credentials managed by Active Directory. However, SQL logins (**`isntname=0`**) are managed by SQL. The **`sql_logins`** view is a list of just the SQL logins from the list of login accounts. We can use some SQL code to perform security checks on these logins. The **`sql_logins`** view adds fields for policy check, expiration check, and the password hash. Although we can't extract the password from the encrypted hash, we can make use of it to do some password checking.

Password policy

In general, the password policy and expiration date should be checked on all SQL logins. The password policy SQL Server uses is generally the same policy that Windows uses, including:

- Cannot contain username.

- Is at least eight characters long.
- Contains at least three of (upper case, lower case, digits, and special characters).

Code Listing 57 checks that the password policy and password expiration policy are set for SQL logins.

Code Listing 57: Check password and expiration

```sql
--------------------------------------------------
-- Script: CheckPasswordPolicy.sql
-- Identify accounts not using password policies
--------------------------------------------------
SELECT name,
CASE
WHEN is_policy_checked=0 and is_expiration_checked=0
THEN 'Password policy and expiration dates are not checked'
WHEN is_policy_checked=0 and is_expiration_checked=1
THEN 'Password policy is not checked'
WHEN is_policy_checked=1 and is_expiration_checked=0
THEN 'Password expiration is not checked'
END AS Msg
FROM sys.sql_logins
WHERE is_disabled=0
AND (is_policy_checked=0 or is_expiration_checked=0)
ORDER BY name
```

Duplicate passwords

In some environments, users might have multiple accounts, and use the same password for each login. This can create a situation where if an attacker can compromise a single account, they could gain access to multiple logins, potentially ones with more permissions and rights. Code Listing 58 provides a simple script to identify accounts that have duplicate passwords.

Code Listing 58: Duplicate password check

```sql
---------------------------------------------
-- Script: Duplicated_passwords.sql
-- Accounts using the same password
---------------------------------------------
SELECT [name] as Account,
        'have a duplicate passwords' as Msg
FROM sys.sql_logins
WHERE password_hash in
 (SELECT password_hash
  FROM sys.sql_logins
  GROUP BY[password_hash] HAVING count(*)> 1
)
```

In this code, we are grouping by the password hash field simply to check for any time a hash occurs more than once. We won't know the actual password, but we will know that multiple accounts are using the same one.

Blank passwords

If a password is blank, or the password is the same as the account name, it can easily be hacked. This simple script allows you to identify these risky accounts.

Code Listing 59: Blank passwords

```
---------------------------------------------
-- Script: Blank_Passwords.sql
-- Blank or passwords same as account name
---------------------------------------------
SELECT name as [AccoutName],
CASE
WHEN PWDCOMPARE('', password_hash) = 1
THEN 'This account has an empty password'
WHEN PWDCOMPARE(name, password_hash)= 1
THEN 'This accounts password is the same as the name'
END as Msg
FROM sys.sql_logins
WHERE(PWDCOMPARE('', password_hash) = 1 or
      PWDCOMPARE(name, password_hash) = 1)
```

Note that if password policy is enforced, this condition should never occur.

Common passwords

While you cannot expose account passwords using SQL Server, you can check a password against a text string. By creating a table of common passwords, you can compare accounts against this list and identify those accounts using simple passwords.

Code Listing 60: Common passwords

```
---------------------------------------------
-- Script: Common_Passwords.sql
-- Commonly used, simple passwords
---------------------------------------------
CREATE TABLE #passwords ( PasswordString varchar(32) )
INSERT INTO #passwords VALUES ('password'),('123456'),('qwerty'),
              ('Admin'), ('password1'), ('abc123')

SELECT name as [Account],
       p.PasswordString+' is not a secure password' as Msg
FROM sys.sql_logins l
JOIN #passwords p on (1=1)
WHERE  PWDCOMPARE(p.passwordString,l.Password_hash)=1
```

```
DROP TABLE #passwords
```

Note that this script will show the account and the bad password. You can change the `msg` string if you want to find the account, but not expose the actual password used.

Tip: There are many sites that have common password lists available for download, such as this one.

If password policy is enforced, your common password list should be adjusted to meet the password complexity rules (eight digits, upper, lower, and digit), but still common. For example, **Abcd1234** and **Password1** meet the complexity rules, but are simple passwords.

Recent accounts

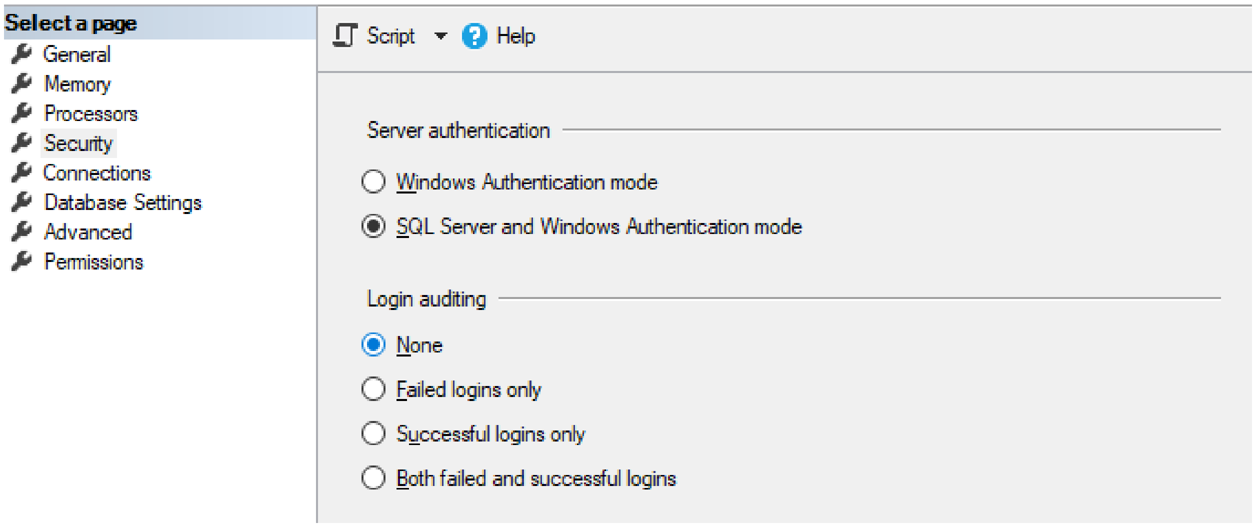
If a hacker gains access to your server, one thing they might do is create their own account in case the hacked account gets detected. This script reports any recently added or modified accounts, which possibly could be a sign of suspicious activity.

Code Listing 61: Recent accounts adds/updates

```
------------------------------------------------------
-- Script: Recent_Accounts.sql
-- Look for any recent accounts added or modified
------------------------------------------------------
SELECT name AS 'Account Name',
      'Login created recently' as Msg
 FROM master.sys.server_principals
 WHERE type LIKE 's' and datediff(d,create_date,getdate())< 14
 UNION
SELECT name,
      'Login modified recently' as Msg
            FROM master.sys.server_principals
            WHERE type LIKE 's' and datediff(d,modify_date,getdate())< 14
```

Auditing logins

The SQL Server Security menu at the server level allows you to specify authentication (either Windows or SQL and Windows) and the login auditing to use.


Select a page
- General
- Memory
- Processors
- Security
- Connections
- Database Settings
- Advanced
- Permissions

Script | Help

Server authentication

- ( ) Windows Authentication mode
- (●) SQL Server and Windows Authentication mode

Login auditing

- (●) None
- ( ) Failed logins only
- ( ) Successful logins only
- ( ) Both failed and successful logins

Figure 6: SQL Server security

The Windows authentication login is considered more secure because of its reliance on Active Directory and is recommended for that reason. In addition, your server should ideally audit all login attempts, but at a minimum, it should log failed logins. The following listing checks for the server authentication mode and the login level. This listing uses the **`SERVERPROPERTY`** function to determine the authentication mode.

Code Listing 62: Check authentication mode

```sql
------------------------------------------------------
-- Script: AuthenticationMode.sql
-- Check authentication mode
------------------------------------------------------
SELECT
CASE SERVERPROPERTY('IsIntegratedSecurityOnly')
WHEN 0 THEN 'SQL and Windows Authentication'
ELSE 'Windows Authentication only '
END as AuthMode
```

This listing uses the system stored procedure to read the audit level from the registry and returns both the audit level and a description.

Code Listing 63: Check login auditing level

```sql
---------------------------------------------
-- Script: Audit_Level.sql
-- Check on audit level of server
---------------------------------------------
DECLARE @auditLevel INT;
EXEC MASTER.dbo.xp_instance_regread N'HKEY_LOCAL_MACHINE',
       'Software\Microsoft\MSSQLServer\MSSQLServer', 'AuditLevel',
@AuditLevel OUTPUT;
SELECT
@auditLevel as AuditLevel,
CASE @auditLevel
WHEN 1 THEN 'No login auditing'
WHEN 2 THEN 'Failed Logins only'
WHEN 3 THEN 'Successful logins only'
ELSE 'All Logins'
END as AuditRules
```

SA account

The system administrator or SQL administrator (SA) account is a well-known and powerful account on a SQL server. If it is enabled, it must be protected with a strong password. The following listing checks the SA password against empty or **sa**, or any other weak passwords.

Code Listing 64: Check for weak SA password

```
--------------------------------------------------------
-- Script: Weak_SA_Password.sql
-- Report weak SA passwords
---------------------------------------------------------
CREATE TABLE #passwords ( PasswordString varchar(32) )
INSERT INTO #passwords VALUES ('password'),('123456'),('Qwerty'),
              ('Admin'), ('Password1'), ('abc123'),(''),('sa')

SELECT 'ERROR: '+p.PasswordString+' is not a secure password on the sa
account' as Msg
FROM sys.sql_logins l
JOIN #passwords p on (1=1)
WHERE  PWDCOMPARE(p.passwordString,l.Password_hash)=1 and [name]='sa'
DROP TABLE #passwords
```

You should add your own common password lists, particularly if you know passwords commonly used with your organization. This is a back door you do not want to leave unlocked.

Users

The **`sys.database_principals`** view shows the user with the current database. You can use this table to identify SQL logins and Windows logins that have access to the current database. Code Listing 65 lists various users in the database, their type, and what database roles they have.

Code Listing 65: Database users

```
---------------------------------------------------------
-- Script: Database_Users.sql
-- Users in the current database
---------------------------------------------------------
SELECT p.[name],p.type_desc,p.create_date,p.modify_date,rl.name as
RoleName
```

```
FROM sys.database_principals p
LEFT JOIN sys.database_role_members rm
      ON rm.member_principal_id=p.principal_id
LEFT JOIN sys.database_principals rl ON
rl.principal_id=rm.role_principal_id
WHERE p.type in ('S','U')
```

The **`Guest`**, **`sys`**, and **`INFORMATION_SCHEMA`** principals will appear in each database, although they are not accounts that can log in, and typically have limited rights. You should monitor any newly added users, and particularly those with write access to the database.

Public role in database

Every user in the database has access to the public role. If you grant the public role UPDATE, DELETE, and INSERT rights, or give the role rights to all tables and system stored procedures, you could be putting your server at risk of any user who manages to connect to it. Code Listing 66 lists some items to which the public role most likely should not have access.

Code Listing 66: Check public role

```
--------------------------------------------------------
-- Script: What_CanPublicRole_do.sql
-- Can the public role see all items
--------------------------------------------------------
SELECT      p.permission_name,OBJECT_NAME(major_id) as AllowedView
FROM sys.database_permissions p
JOIN sys.database_principals pr on pr.principal_id=p.grantee_principal_id
WHERE pr.name = 'Public' and state='G' and class<>0
AND object_name(major_id) like 'All[_]%' and p.permission_name='SELECT'
UNION
SELECT      p.permission_name,OBJECT_NAME(major_id) as AllowedView
FROM sys.database_permissions p
JOIN sys.database_principals pr on pr.principal_id=p.grantee_principal_id
WHERE pr.name = 'Public' and state='G' and class<>0
AND object_name(major_id) like 'SP[_]%' and p.permission_name='EXECUTE'
UNION
SELECT      p.permission_name,''
FROM sys.database_permissions p
JOIN sys.database_principals pr on pr.principal_id=p.grantee_principal_id
WHERE pr.name = 'Public' and state='G' and class<>0
AND p.permission_name in ('UPDATE','DELETE','INSERT')
ORDER BY p.permission_name,AllowedView
```

You can tweak this query to check the public role's access, but in general, the public role should be very limited, particularly on a production server.

What is on my server?

Ideally, SQL Server and its supporting tools are the only major applications that are running on your SQL Server hardware. However, it is possible that other programs are installed, and these could represent a security risk to the server.

Unneeded applications

While the Microsoft Office suite is a common product, I would not expect it to be available on my SQL server box. Code Listing 67 uses the new **`enumerate_filesystem`** function to look for Microsoft Office or Visual Studio. You can supplement the code to include your own list of applications that should not be installed on a production server.

Code Listing 67: Check for unneeded applications

```
---------------------------------------------------------
-- Script: Check_for_Applications.sql
-- See if any unexpected applications are installed
---------------------------------------------------------
IF SERVERPROPERTY('ProductMajorVersion') >= '14'
BEGIN
      SELECT file_or_directory_name,creation_time
      from sys.dm_os_enumerate_filesystem('C:\Program Files\','*.*') x
      WHERE x.is_directory=1 AND
            (file_or_directory_name LIKE 'Microsoft Office%' or
         file_or_directory_name LIKE 'Microsoft Visual Studio%')
END
ELSE
      SELECT 'Requires SQL 2017 or higher' as Msg
```

Newly added files

Code Listing 68 will check the Windows folders for any new .dll or .exe files added to the server.

Code Listing 68: Check for new .dll or .exe files

```
---------------------------------------------------------
-- Script: Check_ForNewFiles.sql
-- Check if any DLL or EXE were added to server recently
---------------------------------------------------------
DECLARE @numDays INT = -14

IF SERVERPROPERTY('ProductMajorVersion') >= '14'
BEGIN
      SELECT file_or_directory_name,creation_time
      from sys.dm_os_enumerate_filesystem('c:\windows\system32\','*.dll')
      WHERE creation_time>=dateadd(DAY,@numDays,getDate())
      UNION
      SELECT file_or_directory_name,creation_time
      from sys.dm_os_enumerate_filesystem('c:\windows\system32\','*.exe')
      WHERE creation_time>=dateadd(DAY,@numDays,getDate())
      ORDER BY creation_time DESC
END
ELSE
      SELECT 'Requires SQL 2017 or higher' as Msg
```

Summary

By reducing the potential surface area that a hacker can attack, you can improve the security of the company's data. You should also monitor user accounts—a powerful account with a simple password can be dangerous in the hands of an attacker. And finally, there are certain stored procedures that are very powerful, but should be carefully protected. Imagine the damage a hacker could do via **`xp_cmdshell`** and access to the operating system the server is running on.

These scripts should give you a starting point to monitor the security of your server and to take steps to keep the data protected from prying eyes and keyboards.

Summary

SQL Server is a very complex and powerful product, but it provides tremendous amounts of data about itself. You can use this data to improve your database design, increase performance, review security, and more. There are several groupings of these views.

INFORMATION_SCHEMA

Provides ANSI standard views for accessing database objects (tables, procedures, columns, etc.).

Microsoft sys schema

These views are unique to SQL Server and provide the content for **`INFORMATION_SCHEMA`** with additional information. All the information schema views pull their information from the **`sys`** schema.

Dynamic management views

These views are updated with information about how SQL is operating and can often offer performance and security information.

Dig in and explore the views we covered in this book (and a lot of other ones) to help you understand and optimize your SQL Server environment.

Appendix: Information Schema

The following is a listing of all the views in the information schema. When possible, these views should be your first resource for querying tables, columns, views, routines, and so on. Using these views allows you to create queries that should run on other SQL dialects, as well.

View	Description
`CHECK_CONSTRAINTS`	Check (column content) constraints.
`COLUMN_DOMAIN_USAGE`	Columns using a domain (user-defined type).
`COLUMN_PRIVILEGES`	Column-level privileges, if used.
`COLUMNS`	Every column in every table and view.
`CONSTRAINT_COLUMN_USAGE`	Columns used in constraints, link to domain or referential constraints.
`CONSTRAINT_TABLE_USAGE`	Tables used in constraints.
`DOMAIN_CONSTRAINTS`	Constraints for data types (domains) in database.
`DOMAINS`	User-defined types (domain in ANSI terminology).
`KEY_COLUMN_USAGE`	Each column used a key.
`PARAMETERS`	Parameters passed to functions or stored procedures.
`REFERENTIAL_CONSTRAINTS`	Foreign key constraint and cascade rules.
`ROUTINE_COLUMNS`	Columns returned from table-valued function.
`ROUTINES`	Stored procedures and user-defined functions.

View	Description
`SCHEMATA`	Each schema in the database.
`SEQUENCES`	Sequence objects in database.
`TABLE_CONSTRAINTS`	Constraints applied to a table (Check, PK, etc.).
`TABLE_PRIVILEGES`	Privileges assigned to users for table access.
`TABLES`	Schemas and tables/views.
`VIEW_COLUMN_USAGE`	What columns are used in a view.
`VIEW_TABLE_USAGE`	What tables are used by various views.
`VIEWS`	List of views in current database.