Notes on Optimising Microsoft SQL Server Performance

By Robert Moller

Table of Contents

Hard Drive RAID Modes

RAID 10 - To optimise SQL performance data should be placed in RAID 10.

RAID Type	Minimum Disks	Disk Utilisation	Fault Tolerance	Information and Usage
0	2	100%	None	Temporary tables (TempDB). Excellent read/write performance.
1	2	50%	Single-drive failure.	Data duplicated across both drives. Databases requiring good performance. Good read/write speeds.
5	3	66%+	Single-drive failure.	1 parity disk and 2 or more data disks. Databases requiring average performance. Average read/write speeds. Higher CPU load due to parity calculation.
6	4	50%+	Two drive failures.	2 parity disks and 2 or more data disks. Databases tolerating low performance. Low read/write speeds. Higher CPU load due to 2 parity calculations.
10	**4**	**50%**	**Up to one disk in each sub array.**	**Data duplicated across 2 or more drives. Databases requiring good performance. Good read and average write speed.**

Hard Drive Types

Mechanical – Still recommended for databases even though replacement for SSDs were less frequent. SSDs however have higher incidence of uncorrectable errors. Recommended use of enterprise level (SAS) drive.

SSD Sata – Provides up to 6bB/s speed – if drives in RAID mode this will present performance bottleneck. Only recommended for read-only databases and data can be easily recovered if lost. Recommended for operating system.

SSD NVMe – Not limited by SATA constraints – requires RAID card which supports this mode. Only recommended for read-only databases and data can be easily recovered if lost. Recommended for operating system.

15,000 RPM – Recommended speed for mechanical drives for databases. NAS may use 10,000 RPM for extra reliability.

Toshiba 14/16TB SAS Mechanical Drive – Low number of failures than HGST, Seagate and WDC.

Reference: Drive Failure Rate in Data Centres - https://www.backblaze.com/b2/hard-drive-test-data.html

RAID Controller Types

Write Caching – Writing to disk provides greatest performance penalty. Implement **battery backed** up RAID controller as this provides write caching. Below is a list battery backed up write cache RAID controllers from Dell.

Series 10

PERC H840 Adaptor – 12Gb/s SAS – Cache 8GB NVRAM (AU$1000)
PERC H740P Adaptor – 12GB/s SAS - 6Gb/s SATA – Cache 8GB NVRAM (AU$600)

Series 9

PERC H830 Adaptor – 12 GB/s SAS – Cache 2GB NVRAM (AU$600)
PERC H730P Adaptor – 12Gb/s SAS – 6Gb/s SATA – 2GB NVRAM (AU$400)
PERC H730 Adaptor – 12Gb/s SAS – 6Gb/s SATA – 1GB NVRAM (AU$300)

Volume Separation for SQL Data Files

If possible place the database, log and tempdb on separate physical hard drives. The operating system and database should be on different volumes and ideally using different controllers to maximise IO operations.

Function	Controller	RAID Type	Disks	Drive Type	Description
Operating System	1	2	2	SSD	Operating system drive
SQL Data File (.mdf)	2	10	4	Mechanical	SQL Data Fie
SQL Log File (.ldf)	Not 2	2 or 10	2 or 4	Mechanical	SQL Log File
TempDB	Not 2	2 or 10	2 or 4	SSD	Temporary database

Database and Log Files Initial Size

Initial Database Size – In environments where database data is stored in separate drives configure initial database size to be 80% of dedicated hard drive space. Example: Hard drive is 200GB, initial database size is 160GB.

File Growth Settings – Don't use percentage, use 1024MB or more to reduce delays when auto-sizing. This applies for both database and log file. File growth is a last resort when database is running out of space.

Log File Size - Log file size should be 20-30% of initial database size. Example: Database is 100GB, log file initial size to 30GB. Set auto-growth to a minimum of 1024MB.

Database and Log File Creation Example

```
use master;
go
create database sales
on
( name = sales_dat,
  filename = 'E:\DATA\MDF\sales.mdf',
  size = 100GB,
  maxsize = 200GB, -- 80% drive size
  filegrowth = 1GB
log on
```

```
( name = sales_log,
  filename = 'F:\DATA\LDF\sales.ldf,
  size = 30GB,
  maxsize = 100GB, -- 80% drive size
  filegrowth = 1GB );
go
```

Database Pre-Sizing for Large Insert Operation

If large insert operation is required it is faster to resize database then allow it to auto-grow to suit the size.

```
use master;
go
alter database sales
modify file
    ( name = sales_dat,
      size = 100GB );
go
```

Auto-growth Notification

Auto-growth notification can be sent to the application event log.

Get Global Unique Identifier (GUID) for databases in question

```
SELECT d.name, drs.database_guid, d.group_database_id
FROM sys.databases d
JOIN sys.database_recovery_status drs
ON d.database_id = drs.database_id
```

We get the database GUID and added to the code below...

Database Growth Notification

```
create event notification data_file_growth_notification
on server
with fan_in
for data_file_auto_grow
to server 'NotifyAutoGrow', '<GUID obtained in first step>';
```

Log File Growth Notification

```
create event notification log_file_growth_notification
on server
for log_file_auto_grow
to server 'NotifyAutoGrow', '<GUID obtained in first step>';
```

Database Fragmentation

Database fragmentation occurs both on mechanical and solid state hard drives. It is more of a concern on mechanical hard drives due to delay in head movements when reading data.

Database File Fragmentation

contig – Microsoft provides a tool to analyse and defrag files. It is available from:

http://technet.microsoft.com/en-us/sysinternals/bb897428

To make the database contiguous type:

```
C:\> contig sales.mdf
```

Index Fragmentation

Index fragmentation can be checked using the avg_fragmentation_in_percent and avg_page_space_used_in_percent.

```
use sales
go
select OBJECT_NAME( ps.object_id ) as TableName,
i.name AS IndexName,
ips.index_type_desc as IndexType,
index_level as IndexLevel,
ips.avg_fragmentation_in_percent as AvgFragPerc,
ips.avg_page_space_used_in_percent as AvgPagePerc,
ips.page_count as PageCount
from sys.dm_db_partition_stats ps
inner join sys.indexes i on ps.object_id = i.object_id AND ps.index_id = i.index_id
cross apply sys.dm_db_index_physical_stats( DB_ID(), ps.object_id, ps.index_id, null, 'DETAILED'
)  ips order by ips.avg_fragmentation_in_percent desc
```

Average Fragmentation Percent – (AvgFracPerc) – If it is above 10% this index should be de-fragmented. Defragmenting the a clustered index will also increase page space used percentage.

5% to 30% fragmentation - use the 'alter index reorganize' statement.

```
alter index PK_tb_lookup_system on sales.dbo.tb_lookup_system reorganize
```

More than 30% fragmentation - use the 'alter index reorganize' statement.

```
alter index PK_tb_lookup_system on sales.dbo.tb_lookup_system reorganize
```

More Info: https://blog.sqlauthority.com/2010/01/12/sql-server-fragmentation-detect-fragmentation-and-eliminate-fragmentation/

Checking for Adequate Database Space

```
use sales
go
select name as FileName, size/128 - cast( fileproperty(name, 'SpaceUsed' ) as int)/128 as
AvailableSpace from sys.database_files
```

The command above will give the names of the mdf and ldf files from MS SQL Server perspective. The name of files can be used as inputs to shriking the database file

Shrinking Database Size

Database should be shrunk as a result of large table drop operation.

Note: Databases require some available free space for regular day-to-day operations. If the database is shrunk but its size immediately grows, then the shrink command is taking too much working space from the database.

To shrink database to minimum size type

```
dbcc shrinkdatabase( sales )
```

To shrink database allowing a certain percentage of free space enter the following command. Percentage is a number after the database name

```
dbcc shrinkdatabase( sales, 10 )
```

To shrink database file elements (mdf or ldf) to initial size type:

```
use sales
go
dbcc shrinkfile( sales_mdf )
```

To shrink database file elements (mdf or ldf) to a specific size in MB type:

```
use sales
go
dbcc shrinkfile( sales_mdf, 1 )
```

Checking for Adequate Database Space

```
use sales
go
select name as FileName, size/128 - cast( fileproperty(name, 'SpaceUsed' ) as int)/128 as
AvailableSpace from sys.database_files
```

Tables Indexes

Primary Key Index

All primary keys should be indexed. Primary key indexes can be added at table creation or added after

Primary key at table creation

```
create table tb_users
(
  usr_id_pk bigint         identity( 1, 1 ) not null,
  usr_name  varchar( 100 ) collate latin_general_ci_as null,

  constraint pk_tb_users primary key clustered
  (
    usr_id_pk
  )
  on [ primary ],

)
on [ primary ]
```

Primary key added after

```
alter table dbo.tb_users
add constraint pk_tb_users primary key clustered ( usr_id_pk )
go
```

Other indexes

Tables should have additional indexes to optimise joined and search operations:

- Additional indexes can be added for:
- Frequently used columns in search criteria
- Foreign key columns
- Columns used in the ORDER BY clause

Additional indexes creation example

Recommended format:

```
create index <table name><column name> on <table name>(<column name>) on [primary]
```

Example:

```
create index ix_tb_products_prd_code on tb_products( prd_code ) on [primary]
```

If a join results in a lot of hits and filtering needs to be done to on additional columns as part of WHERE clauses then a compound index can be created with the additional columns in question. This is known as a covered index.

```
create index ix_tb_products_prd_code on tb_products( prd_code ) include ( prd_category,
prd_section ) on primary
```

Conventions and Styles

Database objects should have unique names. This helps improve readability. It is not necessary to use upper for statements and operators, this is just a carryover in style due to memory limitations of computers in the 70s.

Use of Upper and Lower Cases

Common syntax style – commands and operators are uppercase, table names and columns in lower case

```
CREATE TABLE [tb_users] (
  [usr_id_pk] [int] IDENTITY(1,1) NOT NULL,
  [usr_name]  [varchar] (15) COLLATE Latin1_General_CS_AS NULL,
  CONSTRAINT [PK_tb_users] PRIMARY KEY  CLUSTERED
        (
                [usr_id_pk]
        )  ON [PRIMARY] ,
) ON [PRIMARY]
```

More readable syntax style – all values in lower case, the square brackets are used to delimit identifiers, this is only necessary if the column name uses a reserved word or has spaces. Example: user id pk instead of usr_id_pk. The primary key word at the end of the declaration needs to be enclosed in brackets. Collation (sort order) uses latin general case-insensitive (CI), accent- insensitive (AI), that is an e does not have precedence over n è. C style syntax structure emulated below:

```
create table tb_users
(
  usr_id_pk int            identity( 1, 1 ) not null,
  usr_name  varchar ( 15 ) collate latin1_general_ci_ai null,

  costraint pk_tb_users primary key clusted
  (
    usr_id_pk
  )
  on [ primary ],

)
on [ primary ]
```

NULL Usage

A null represents missing or absent data. It is a source of confusion among developers. An easier approach is have a empty string, or in case of integer a default value indicating not available. The example below uses the constraint key word to define default values. For gender, there can be 3 values, ‘M’, ‘F’ or ‘X’, the last value indicates unknown.

Default Constraints for String Values

Recommended format for constraint naming

```
constraint df_<table name>_<column name> default ( <default value> )
```

Example:

```
constraint df_tb_users_usr_name default ( ‘’ )
```

Redesigned table

```
create table tb_users
(
  usr_id_pk [ int ] identity( 1, 1 ) not null,
```

```
  usr_name  [ varchar ] ( 15 ) collate latin1_general_ci_ai not null constraint
df_tb_users_usr_name default (''),
  usr_gender [ char ] ( 1 ) collate latin1_general_ci_ai not null constraint
df_tb_users_usr_gender default ('X'),

  costraint pk_tb_users primary key clusted
  (
    usr_id_pk
  ) on [ primary ],

) on [ primary ]
```

Default Constraints for Date Values

Achieving a non-null default date value is harder as the base date time for Microsoft SQL is 1900-01-01. If dates are used, for example, as part of product lead times extra processing required.

```
constraint df_tb_users_usr_active_date default ( '1900-01-01' )
```

Redesigned table

```
create table tb_users
(
  usr_id_pk [ int ] identity( 1, 1 ) not null,
  usr_name  [ varchar ] ( 15 ) collate latin1_general_ci_ai not null constraint
df_tb_users_usr_name default (''),
  usr_gender [ char ] ( 1 ) collate latin1_general_ci_ai not null constraint
df_tb_users_usr_gender default ('X'),
  usr_active_date [ datetime ] not null constraint df_users_usr_active_date default ( '1900-01-
01' ),

  costraint pk_tb_users primary key clusted
  (
    usr_id_pk
  ) on [ primary ],

) on [ primary ]
```

Object Naming Prefixes

Spaces should not be used in objects, Example, use sales_2019 instead of sales 2019. If the later were used values would not be enclosed in brackets, [sales 2019].

Do not re-use same column names for different tables. If values are re-used, the table name prefix would need to included in an inner join to avoid ambiguous values. Prefixing column names with 3-character table acronym also improves readability.

```
select s.id, r.id
from sales s
inner join regions r
on sales.id = regions.id
```

Using 3-character table acronym prefix avoids needing to add table alias to every column

```
select sal_id, rgn_id
from sales s
inner join regions r
on sales.sal_id = regions.reg_id
```

Object Type	Prefix or Suffix	Examples
Database	None	abc_sales
Table	tb	tb_sales tb_regions
Columns	Use 3 letter acronym representing table name Examples: reg = regions cus = customers prd = productions	reg_id_pk cus_address_1 prd_vendor_code
Indices	Ix	Ix_tb_customer_cus_id_pk Index is ix_<table name>_<column name>
Constraints	df	df_tb_customer_cus_name Constraint is df_<table name>_<column name>
Primary Key	pk	cus_id_pk reg_name_pk
Foreign Key	fk Note: A primary key may also be a foreign key, in this case use 'pkfk' prefix	reg_code_fk reg_code_pkfk
Stored Procedures	usp Note: MS SQL will first search in the system database for any stored procedures beginning with 'sp' which slows down queries.	usp_customer_list
Functions	fn	fn_calculate_payment

Stored Procedure Naming

Stored procedures should follow a pseudo-object appended by method approach. This is the concept using in object orientated programming (OOP) and helps group functions within a similar row in the database.

For a customer database there will be a few common functions such as: add a new customer, delete customer, update customer, list customers, etc. Examples:

```
usp_customer_add
usp_customer_delete
usp_customer_update
usp_customer_list
usp_customer_check_exists
usp_customer_view
```

Stored Procedure Design

There are several design features to improve stored procedure speed and security.

Set no count on – When fetching data from table MS SQL Server will also return a count of records matching criteria. On occasion this is useful to obtain this total if user requires it. It howevers, slows down queries and adds to network traffic, so unless needed the statement 'set no count on' should be used,

```
alter procedure dbo.usp_customer_list
as
begin
  set no count on;
  select * from tb_customer;
end
```

Fully qualified name – To optimise performance call stored procedures using fully qualified name. This name is comprised of:

```
<server name>.<database name>.<owner name>.<object name>
```

Instead of using

```
dbo.usp_customer_list
```

Use instead (with server name):

Get server name...

```
select name from sys.servers
```

Use fully qualified name. The server name needs to be in quotes

```
"WIN-SERV-4\PRDSALES".sales.dbo.usp_customerlist
```

On smaller environments the server name can be dropped:

```
sales.dbo.usp_customerlist
```

Sp_ prefix – Stored procedures prefixed with 'sp_' are used as system stored procedures. If a user-created stored procedure uses this name MS SQL Server will first search the master database then current database. If the system and user-created stored procedure has the same name the user-create SP will never get executed, use instead 'usp_' prefix.

Sp_executesql – Using sp_executesql is better than exec as the former has the ability to reuse cached query plans. If only the only the values change MS SQL Server can use the same query plan. If exec used a new query plan is created for each query using up MS SQL Server resources.

Example using exec

```
declare @SQL as nvarchar( 1000 );
set @SQL = 'select * from tb_customerlist where usp_cus_name = ''John''';
exec( @SQL );
```

Example using sp_executesql

```
declare @SQL as nvarchar( 1000 );
set @SQL = 'select * from test.dbo.tb_customer where cus_name = ''Adam''';
exec sp_executesql @SQL;
```

Parametised Queries

Also known as prepared statements. This allows pre-compiling SQL statements where only the parameters are supplied. SQL injection attacks can happen where the attacker places terminating quotes together with other SQL statements in place of a search value.

The code below is vulnerable to inject attack

```
use test
declare @sql varchar( 100)
declare @cust varchar( 100 )
set @cust = 'John'
set @sql = 'insert into tb_sales ( sal_customer ) values (''' + @cust + ''')'
exec( @sql )
```

We could modify @cust variable to something like

```
set @cust = 'John''); drop table tb_customers;--';
```

The above statement script contains two statements. It also artificially closes the true statement in the middle so extra statements can be added.

Parametised queries begin with statement ‘exec sp_executesql’. The attack parameter is merely added to the table. If in use stored procedure a function should be called to sanitize passed parameters, Ie. Xp_, drop,

```
declare @sql nvarchar( 100)
declare @cust nvarchar( 100 )
declare @params nvarchar( 100 )
--set @cust = N'John'
set @params = N'@param_cus_name varchar( 50 )'
set @cust = 'John''); drop table tb_products;--';
set @sql = N'insert into tb_customer ( cus_name ) values ( @param_cus_name )'
exec sp_executesql
  @sql,
  @params,
  @param_cus_name = @cust
```

Using Custom Errors

Getting existing error list – user messages start at 50,001 to 2,147,483,647

```
select * from master.dbo.sysmessages
```

Get language – use the alias column

```
select * from sys.syslanguages order by name
```

Adding error message

Format is

```
exec sp_addmessage
  @msgnum = <message number>, -- From 50,001 to 2,147,4873,647
  @severity = <severity level>, -- Recommended level 11 to 16 – can be corrected by user
  @ msgtext = "<text>",
  @lang="<message language>" – Get from sys.syslanguages table – use the alias column
```

Example – English message

```
exec sp_addmessage @msgnum = 50001, @severity = 16, @msgtext = 'Invalid parameter',
  @lang = 'us_english'
```

Example – Spanish message

```
exec sp_addmessage @msgnum = 50001, @severity = 16, @msgtext = 'Parametro invalido',
  @lang = 'spanish'
```

Removing error message

```
exec sp_dropmessage @msgnum = <message number>
```

Example

```
exec sp_dropmessage @msgnum = 50001
```

Getting Raiserror State – this helps find where error came from

```
create function [dbo].[fn_get_raiserror_state]
(
  @p_vchr_name varchar( 100 )
)
returns int
as
begin

  declare @int_return int

  select @int_return = case
    when @p_vchr_name = 'a' then 1
    when @p_vchr_name = 'b' then 2
    when @p_vchr_name = 'c' then 3
  else
    0
  end

  return @int_return
```

```
end

use test
go
declare @int_raiserror_state int
set @int_raiserror_state = dbo.fn_get_raiserror_state( 'b' )
print @int_raiserror_state

raiserror( 50001, -1, @int_raiserror_state );



BEGIN TRY
    -- Generate a divide-by-zero error.
    SELECT 1/0;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber,
        ERROR_SEVERITY() AS ErrorSeverity,
        ERROR_STATE() AS ErrorState,
        ERROR_PROCEDURE() AS ErrorProcedure,
        ERROR_LINE() AS ErrorLine,
        ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

Data Types and Sizes

Numeric Data Types

Data Type	Lowest Value	Highest Value	Memory Usage (Bytes)
bigint	−2^63 (−9,223,372, 036,854,775,808)	2^63−1 (−9,223,372, 036,854,775,807)	8
int	−2^31 (−2,147, 483,648)	2^31−1 (−2,147, 483,647)	4
smallint	−2^15 (−32,767)	2^15 (−32,768)	2
tinyint	0	255	1
bit	0	1	1 Can be compacted to 8 bits to use 1 byte if side-by-side
decimal	−10^38+1	10^381−1	5 to 17
numeric	−10^38+1	10^381−1	5 to 17
money	−922,337, 203, 685,477.5808	+922,337, 203, 685,477.5807	8
smallmoney	−214,478.3648	+214,478.3647	4
uniqueidentifier (GUID)			16

Bit Data Type – Placing bit columns site by site will optimise bit storage. Example: If 8 bit type columns are placed side by side then only 1 byte is used for all 8 columns. If columns are scattered in the table then 8 bytes are used.

Floating Point Data Types

Data Type	Lowest Value	Highest Value	Memory Usage (Bytes)
Float(n) n=1 to 24	−1.79E+308	1.79E+308	4 Precision 7 Digits
Float(n) n=25 to 53	−1.79E+308	1.79E+308	4 Precision 15 Digits
Real	−3.40E+38	3.40E+38	4 Precision 15 Digits

Date and Time Data Types

Data Type	Lowest Value	Highest Value	Memory Usage (Bytes)
datetime	1753-01-01 00:00:00.000	9999-12-31 23:59:59.999	8 Millisecond accuracy in increments of .000, .003, .007...
smalldatetime	1900-01-01 00:00	2079-06-06 23:59	4 Accuracy is 1 minute
date	0001-01-01	9999-12-31	3
time	00:00:00.0000000	23:59:59.9999999	5 Accuracy 100 nanoseconds
datetimeoffset	0001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999	10 Accuracy 100 nanoseconds
datetime2	0001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999	6 Accuracy 100 nanoseconds

Datetime2 – This data types offers better time accuracy and uses 2 bytes less than datetime.

Date – If time not required date type uses 5 bytes less than datetime. For convenience and simplicity it can be replaced with datetime2.

Character Data Types

Data Type	Minimum Length	Highest Length	Memory Usage (Bytes)
char(n)	0	4000	n bytes
nchar(n)	0	4000	n * 2 bytes Each character uses 2 bytes
varchar(n)	0	8000	n + 2 bytes Two bytes are used as length indicator
nvarchar(n)	0	2 ^ 31 = 2,147,483,648	n * 2 + 2 bytes Two bytes are used length indicator
text	0	2 ^ 31 – 1 = 2,147,483,647	n + 4 bytes
ntext	0	1,073,741,823	2 * string length

Varchar vs nvarchar – Only use nvarchar is international language is required as 2 bytes are used per character. For MS SQL Server 2019 this is not required as UTF-8 support is built in.

UTF-8 Support – MS SQL Server 2019 supports utf-8. This is better than using nvarchar as utf-8 only uses 2 bytes if character requires making it more compact.

See https://www.sqlshack.com/sql-varchar-data-type-deep-dive/

To enable UTF-8 support create (or modify) collation of char and varchar columns to a collation with _utf8 suffix. Example: Change latin_general_100_ci_as_sc to latin_general_100_ci_as_sc_utf8.

To check which collations are supported in Windows type:

```
select name, description from fn_helpcollations()
where name like '%UTF8';
```

Char – Character Type

Fixed length string. Use for codes which are known to have fixed length such as postcodes, zip codes, Ids. Sorts are faster using char. Index lookups on char are about 20% faster than varchar.

Varchar – Variable Character Type

Variable size character column. The temptation is to use varchar(255) for all character types. However, this will have an impact on operations that require additional resources such as sorts. MS SQL Server also has a maximum index length of 900 bytes for composite keys (multiple columns), so using 255 values only allows 3 columns to be part of the index.

nvarchar – Unicode Variable Character Type

Nvarchar is used where international language support is required. Values are stored in 2-bytes doubling storage requirement, MS SQL 2019 has UTF-8 support within varchar obviating the need to use nvarchar and saving memory as UTF-8 only uses 2 bytes (or more) if required. Do no use nvarchar if international language support is not required.

Variable size character column. The temptation is to use varchar(255) for all character types. However, this will have an impact on operations that require additional resources such as sorts. MS SQL Server also has a maximum index length of 900 bytes for composite keys (multiple columns), so using 255 values only allows 3 columns to be part of the index.

Recommended Sizes forDifferent DataTtypes

Information Type	Size
URL	255
Customer Names Street Names Email Company name Product Names	120
First Name Middle Name Last Name Product Codes Building Names	60
Suburb Names City Names	40 (Longest suburb in Australia is Mamungkukumpurangkuntjunya)
Honorific (Mr, Miss, Prof) Suburb Codes/ZIP Drivers License Medicare Card Private Insurance ID	20

See performance impact on for large varchars:
https://sqlperformance.com/2017/06/sql-plan/performance-myths-oversizing-strings

AWS Optimisations Tips:
https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-techniques-for-amazon-redshift/

Binary Data Types

Data Type	Minimum Length	Highest Length	Memory Usage (Bytes)
binary(n)	0	8000	n bytes
varbinary(n)	0	8000	n + 2 bytes
image(n)	0	2 ^ 31 – 1 = 2,147,483,647	n + 2 bytes Two bytes are used as length indicator

Image Data Type – This data type will be retired from MS SQL Server, use instead varbinary(max)

Users and Roles

Create users and assign to a custom role. Database security will be set through the role. The role never provides direct access to table commands such as SELECT, DROP, etc so the user is limited into what operations they can do to a table.

Creating Users

```
create login john with password = 'password';
go
```

Allow User to Access Selected Database

```
use sales;
go
exec sp_grantdbaccess @loginame = 'john';
go
```

Create User Level Role

Setting permissions via role is easier than setting permissions directly to the user as often different users share the login level. The role is in relation to the database so ensure the database is selected before assigning the role.

```
use sales;
go
create role users;
go
```

Add User to Role

The role membership is in relation to the database so ensure the database so select it before assigning role.

```
use sales;
go
exec sp_addrolemember @rolename = 'users,' @membername = 'john';
```

Assign Execute Permission to Selected Role

```
use Sales;
go
grant execute on object::usp_customer_list to users;
go
```

Checking via Logging in and Running Command

Access should be denied since user only had access via stored procedure. The message returned should be 'The SELECT permission was denied on the object 'tb_customer'...'

```
C:\ sqlcmd –S.\Sales –Ujohn –dSales –Q"select * from tb_customer;"
Password: <password>
```

Try and call authorise stored procedure, this should be allowed.

```
C:\ sqlcmd –S.\Sales –Ujohn –dSales –Q"exec dbo.usp_customer_list;"
Password: <password>
```

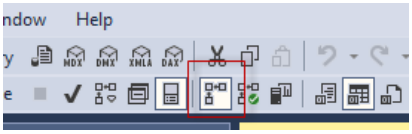
Finding Slow Queries

Queries performance can be measured from the table sys.dm_exec_query_stats

```
Select top 25
total_worker_time/execution_count AS Avg_CPU_Time
,Execution_count
,total_elapsed_time/execution_count as AVG_Run_Time
,total_elapsed_time
,(SELECT
SUBSTRING(text,statement_start_offset/2+1,statement_end_offset
) FROM sys.dm_exec_sql_text(sql_handle)
) AS Query_Text
from sys.dm_exec_query_stats
order by Avg_CPU_Time desc
```

SSMS Execution Plan

In Microsoft SQL Server Management Studio (SSMS) a query or stored procedure can be analysed by clicking the speed button Include Actual Execution Plan



After executing the query clicking on the 'Execution Plan' tab will show performance, check which indices are used and 'Actual Number of Rows Read' to check how optimised indices are for the select.