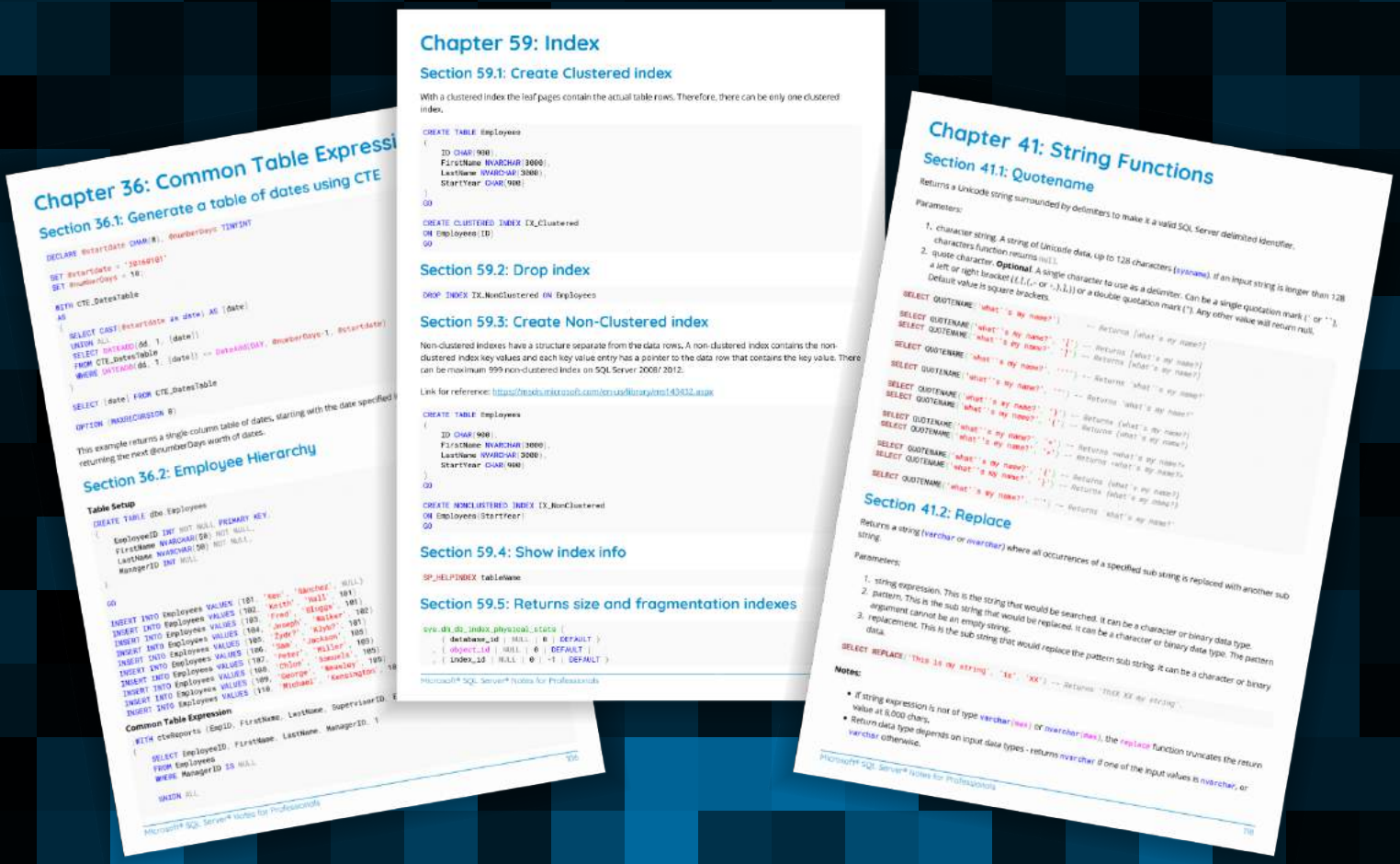
# Microsoft® SQL Server®

## Notes for Professionals




**200+ pages**

of professional hints and tricks

# Contents

# About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/MicrosoftSQLServerBook



Please send feedback and corrections to web@petercv.com

# Chapter 1: Getting started with Microsoft SQL Server

| Version | Release Date |
|---|---|
| SQL Server 2016 | 2016-06-01 |
| SQL Server 2014 | 2014-03-18 |
| SQL Server 2012 | 2011-10-11 |
| SQL Server 2008 R2 | 2010-04-01 |
| SQL Server 2008 | 2008-08-06 |
| SQL Server 2005 | 2005-11-01 |
| SQL Server 2000 | 2000-11-01 |

## Section 1.1: INSERT / SELECT / UPDATE / DELETE: the basics of Data Manipulation Language

**D**ata **M**anipulation **L**anguage (DML for short) includes operations such as `INSERT`, `UPDATE` and `DELETE`:

```
-- Create a table HelloWorld

CREATE TABLE HelloWorld (
    Id INT IDENTITY,
    Description VARCHAR(1000)
)


-- DML Operation INSERT, inserting a row into the table
INSERT INTO HelloWorld (Description) VALUES ('Hello World')


-- DML Operation SELECT, displaying the table
SELECT * FROM HelloWorld


-- Select a specific column from table
SELECT Description FROM HelloWorld


-- Display number of records in the table
SELECT Count(*) FROM HelloWorld


-- DML Operation UPDATE, updating a specific row in the table
UPDATE HelloWorld SET Description = 'Hello, World!' WHERE Id = 1


-- Selecting rows from the table (see how the Description has changed after the update?)
SELECT * FROM HelloWorld


-- DML Operation - DELETE, deleting a row from the table
DELETE FROM HelloWorld WHERE Id = 1


-- Selecting the table. See table content after DELETE operation
SELECT * FROM HelloWorld
```

In this script we're **creating a table** to demonstrate some basic queries.

The following examples are showing how to **query tables:**

```
USE Northwind;
GO
SELECT TOP 10 * FROM Customers
ORDER BY CompanyName
```

will select the first 10 records of the `Customer` table, ordered by the column `CompanyName` from the database `Northwind` (which is one of Microsoft's sample databases, it can be downloaded from **here**):

| CustomerID | CompanyName | ContactName | ContactTitle | Address | City | Region | PostalCode | Country | Phone | Fax |
|---|---|---|---|---|---|---|---|---|---|---|
| ALFKI | Alfreds Futterkiste | Maria Anders | Sales Representative | Obere Str. 57 | Berlin | null | 12209 | Germany | 030-0074321 | 030-0076545 |
| ANATR | Ana Trujillo Emparedados y helados | Ana Trujillo | Owner | Avda. de la Constitución 2222 | México D.F. | null | 05021 | Mexico | (5) 555-4729 | (5) 555-3745 |
| ANTON | Antonio Moreno Taquería | Antonio Moreno | Owner | Mataderos 2312 | México D.F. | null | 05023 | Mexico | (5) 555-3932 | null |
| AROUT | Around the Horn | Thomas Hardy | Sales Representative | 120 Hanover Sq. | London | null | WA1 1DP | UK | (171) 555-7788 | (171) 555-6750 |
| BERGS | Berglunds snabbköp | Christina Berglund | Order Administrator | Berguvsvägen 8 | Luleå | null | S-958 22 | Sweden | 0921-12 34 65 | 0921-12 34 67 |
| BLAUS | Blauer See Delikatessen | Hanna Moos | Sales Representative | Forsterstr. 57 | Mannheim | null | 68306 | Germany | 0621-08460 | 0621-08924 |
| BLONP | Blondesddsl père et fils | Frédérique Citeaux | Marketing Manager | 24, place Kléber | Strasbourg | null | 67000 | France | 88.60.15.31 | 88.60.15.32 |
| BOLID | Bólido Comidas preparadas | Martín Sommer | Owner | C/ Araquil, 67 | Madrid | null | 28023 | Spain | (91) 555 22 82 | (91) 555 91 99 |
| BONAP | Bon app' | Laurence Lebihan | Owner | 12, rue des Bouchers | Marseille | null | 13008 | France | 91.24.45.40 | 91.24.45.41 |
| BOTTM | Bottom-Dollar Markets | Elizabeth Lincoln | Accounting Manager | 23 Tsawassen Blvd. | Tsawassen | BC | T2F 8M4 | Canada | (604) 555-4729 | (604) 555-3745 |

**Note** that `Use Northwind;` changes the default database for all subsequent queries. You can still reference the database by using the fully qualified syntax in the form of [Database].[Schema].[Table]:

```
SELECT TOP 10 * FROM Northwind.dbo.Customers
ORDER BY CompanyName

SELECT TOP 10 * FROM Pubs.dbo.Authors
ORDER BY City
```

This is useful if you're querying data from different databases. Note that `dbo`, specified "in between" is called a schema and needs to be specified while using the fully qualified syntax. You can think of it as a folder within your database. `dbo` is the default schema. The default schema may be omitted. All other user defined schemas need to be specified.

If the database table contains columns which are named like reserved words, e.g. `Date`, you need to enclose the column name in brackets, like this:

```
-- descending order
SELECT TOP 10 [Date] FROM dbo.MyLogTable
ORDER BY [Date] DESC
```

The same applies if the column name contains spaces in its name (which is not recommended). An alternative syntax is to use double quotes instead of square brackets, e.g.:

```
-- descending order
SELECT top 10 "Date" from dbo.MyLogTable
order by "Date" desc
```

is equivalent but not so commonly used. Notice the difference between double quotes and single quotes: Single quotes are used for strings, i.e.

```
-- descending order
SELECT top 10 "Date" from dbo.MyLogTable
where UserId='johndoe'
order by "Date" desc
```

is a valid syntax. Notice that T-SQL has a `N` prefix for NChar and NVarchar data types, e.g.

```
SELECT TOP 10 * FROM Northwind.dbo.Customers
WHERE CompanyName LIKE N'AL%'
ORDER BY CompanyName
```

returns all companies having a company name starting with `AL` (`%` is a wild card, use it as you would use the asterisk in a DOS command line, e.g. `DIR AL*`). For `LIKE`, there are a couple of wildcards available, look **here** to find out more details.

**Joins**

Joins are useful if you want to query fields which don't exist in one single table, but in multiple tables. For example: You want to query all columns from the `Region` table in the `Northwind` database. But you notice that you require also the `RegionDescription`, which is stored in a different table, `Region`. However, there is a common key, `RgionID` which you can use to combine this information in a single query as follows (`Top 5` just returns the first 5 rows, omit it to get all rows):

```
SELECT TOP 5 Territories.*,
    Regions.RegionDescription
FROM Territories
INNER JOIN Region
    ON Territories.RegionID=Region.RegionID
ORDER BY TerritoryDescription
```

will show all columns from `Territories` plus the `RegionDescription` column from `Region`. The result is ordered by `TerritoryDescription`.

**Table Aliases**

When your query requires a reference to two or more tables, you may find it useful to use a Table Alias. Table aliases are shorthand references to tables that can be used in place of a full table name, and can reduce typing and editing. The syntax for using an alias is:

```
<TableName> [as] <alias>
```

Where `as` is an optional keyword. For example, the previous query can be rewritten as:

```
SELECT TOP 5 t.*,
    r.RegionDescription
FROM Territories t
INNER JOIN Region r
    ON t.RegionID = r.RegionID
ORDER BY TerritoryDescription
```

Aliases must be unique for all tables in a query, even if you use the same table twice. For example, if your Employee table included a SupervisorId field, you can use this query to return an employee and his supervisor's name:

```
SELECT e.*,
    s.Name AS SupervisorName -- Rename the field for output
FROM Employee e
INNER JOIN Employee s
    ON e.SupervisorId = s.EmployeeId
WHERE e.EmployeeId = 111
```

**Unions**

As we have seen before, a Join adds columns from different table sources. But what if you want to combine rows

from different sources? In this case you can use a UNION. Suppose you're planning a party and want to invite not only employees but also the customers. Then you could run this query to do it:

```
SELECT FirstName+' '+LastName AS ContactName, Address, City FROM Employees
UNION
SELECT ContactName, Address, City FROM Customers
```

It will return names, addresses and cities from the employees and customers in one single table. Note that duplicate rows (if there should be any) are automatically eliminated (if you don't want this, use a `UNION ALL` instead). The column number, column names, order and data type must match across all the select statements that are part of the union - this is why the first SELECT combines `FirstName` and `LastName` from Employee into `ContactName`.

**Table Variables**

It can be useful, if you need to deal with temporary data (especially in a stored procedure), to use table variables: The difference between a "real" table and a table variable is that it just exists in memory for temporary processing.

**Example:**

```
DECLARE @Region TABLE
(
  RegionID int,
  RegionDescription NChar(50)
)
```

creates a table in memory. In this case the `@` prefix is mandatory because it is a variable. You can perform all DML operations mentioned above to insert, delete and select rows, e.g.

```
INSERT INTO @Region values(3,'Northern')
INSERT INTO @Region values(4,'Southern')
```

But normally, you would populate it based on a real table like

```
INSERT INTO @Region
SELECT * FROM dbo.Region WHERE RegionID>2;
```

which would read the filtered values from the real table `dbo.Region` and insert it into the memory table `@Region` - where it can be used for further processing. For example, you could use it in a join like

```
SELECT * FROM Territories t
JOIN @Region r ON t.RegionID=r.RegionID
```

which would in this case return all `Northern` and `Southern` territories. More detailed information can be found here. Temporary tables are discussed here, if you are interested to read more about that topic.

**NOTE:** Microsoft only recommends the use of table variables if the number of rows of data in the table variable are less than 100. If you will be working with larger amounts of data, use a **temporary table**, or temp table, instead.

## Section 1.2: SELECT all rows and columns from a table

Syntax:

```
SELECT *
```

```
FROM TABLE_NAME
```

Using the asterisk operator `*` serves as a shortcut for selecting all the columns in the table. All rows will also be selected because this `SELECT` statement does not have a `WHERE` clause, to specify any filtering criteria.

This would also work the same way if you added an alias to the table, for instance `e` in this case:

```
SELECT *
FROM Employees AS e
```

Or if you wanted to select all from a specific table you can use the alias + " .* ":

```
SELECT e.*, d.DepartmentName
FROM Employees AS e
    INNER JOIN Department AS d
        ON e.DepartmentID = d.DepartmentID
```

Database objects may also be accessed using fully qualified names:

```
SELECT * FROM [server_name].[database_name].[schema_name].[TABLE_NAME]
```

This is not necessarily recommended, as changing the server and/or database names would cause the queries using fully-qualified names to no longer execute due to invalid object names.

Note that the fields before `table_name` can be omitted in many cases if the queries are executed on a single server, database and schema, respectively. However, it is common for a database to have multiple schema, and in these cases the schema name should not be omitted when possible.

**Warning:** Using `SELECT *` in production code or stored procedures can lead to problems later on (as new columns are added to the table, or if columns are rearranged in the table), especially if your code makes simple assumptions about the order of columns, or number of columns returned. So it's safer to always explicitly specify column names in SELECT statements for production code.

```
SELECT col1, col2, col3
FROM TABLE_NAME
```

## Section 1.3: UPDATE Specific Row

```
UPDATE HelloWorlds
SET HelloWorld = 'HELLO WORLD!!!'
WHERE Id = 5
```

The above code updates the value of the field "HelloWorld" with "HELLO WORLD!!!" for the record where "Id = 5" in HelloWorlds table.

Note: In an update statement, It is advised to use a "where" clause to avoid updating the whole table unless and until your requirement is different.

## Section 1.4: DELETE All Rows

```
DELETE
FROM Helloworlds
```

This will delete all the data from the table. The table will contain no rows after you run this code. Unlike `DROP TABLE`,

this preserves the table itself and its structure and you can continue to insert new rows into that table.

Another way to delete all rows in table is truncate it, as follow:

```
TRUNCATE TABLE HelloWords
```

Difference with DELETE operation are several:

1. Truncate operation doesn't store in transaction log file
2. If exists `IDENTITY` field, this will be reset
3. TRUNCATE can be applied on whole table and no on part of it (instead with `DELETE` command you can associate a `WHERE` clause)

**Restrictions Of TRUNCATE**

1. Cannot TRUNCATE a table if there is a `FOREIGN KEY` reference
2. If the table is participated in an `INDEXED VIEW`
3. If the table is published by using `TRANSACTIONAL REPLICATION` or `MERGE REPLICATION`
4. It will not fire any TRIGGER defined in the table

[sic]

# Section 1.5: Comments in code

Transact-SQL supports two forms of comment writing. Comments are ignored by the database engine, and are meant for people to read.

**Comments** are preceded by `--` and are ignored until a new line is encountered:

```
-- This is a comment
SELECT *
FROM MyTable -- This is another comment
WHERE Id = 1;
```

**Slash star comments** begin with `/*` and end with `*/`. All text between those delimiters is considered as a comment block.

```
/* This is
a multi-line
comment block. */
SELECT Id = 1, [Message] = 'First row'
UNION ALL
SELECT 2, 'Second row'
/* This is a one liner */
SELECT 'More';
```

Slash star comments have the advantage of keeping the comment usable if the SQL Statement loses new line characters. This can happen when SQL is captured during troubleshooting.

Slash star comments can be nested and a starting `/*` inside a slash star comment needs to be ended with a `*/` to be valid. The following code will result in an error

```
/*
SELECT *
FROM CommentTable
WHERE Comment = '/*'
```

```
*/
```

The slash star even though inside the quote is considered as the start of a comment. Hence it needs to be ended with another closing star slash. The correct way would be

```
/*
SELECT *
FROM CommentTable
WHERE Comment = '/*'
*/  */
```

## Section 1.6: PRINT

Display a message to the output console. Using SQL Server Management Studio, this will be displayed in the messages tab, rather than the results tab:

```
PRINT 'Hello World!';
```

## Section 1.7: Select rows that match a condition

Generally, the syntax is:

```
SELECT <COLUMN names>
FROM <TABLE name>
WHERE <condition>
```

For example:

```
SELECT FirstName, Age
FROM Users
WHERE LastName = 'Smith'
```

Conditions can be complex:

```
SELECT FirstName, Age
FROM Users
WHERE LastName = 'Smith' AND (City = 'New York' OR City = 'Los Angeles')
```

## Section 1.8: UPDATE All Rows

A simple form of updating is incrementing all the values in a given field of the table. In order to do so, we need to define the field and the increment value

The following is an example that increments the `Score` field by 1 (in all rows):

```
UPDATE Scores
SET score = score + 1
```

This can be dangerous since you can corrupt your data if you accidentally make an UPDATE for a **specific Row** with an UPDATE for **All rows** in the table.

## Section 1.9: TRUNCATE TABLE

```
TRUNCATE TABLE Helloworlds
```

This code will delete all the data from the table Helloworlds. Truncate table is almost similar to `Delete from Table` code. The difference is that you can not use where clauses with Truncate. Truncate table is considered better than delete because it uses less transaction log spaces.

Note that if an identity column exists, it is reset to the initial seed value (for example, auto-incremented ID will restart from 1). This can lead to inconsistency if the identity columns is used as a foreign key in another table.

## Section 1.10: Retrieve Basic Server Information

```
SELECT @@VERSION
```

Returns the version of MS SQL Server running on the instance.

```
SELECT @@SERVERNAME
```

Returns the name of the MS SQL Server instance.

```
SELECT @@SERVICENAME
```

Returns the name of the Windows service MS SQL Server is running as.

```
SELECT serverproperty('ComputerNamePhysicalNetBIOS');
```

Returns the physical name of the machine where SQL Server is running. Useful to identify the node in a failover cluster.

```
SELECT * FROM fn_virtualservernodes();
```

In a failover cluster returns every node where SQL Server can run on. It returns nothing if not a cluster.

## Section 1.11: Create new table and insert records from old table

```
SELECT * INTO NewTable FROM OldTable
```

Creates a new table with structure of old table and inserts all rows into the new table.

**Some Restrictions**

> 1. You cannot specify a table variable or table-valued parameter as the new table.
> 2. You cannot use SELECT...INTO to create a partitioned table, even when the source table is partitioned. SELECT...INTO does not use the partition scheme of the source table; instead, the new table is created in the default filegroup. To insert rows into a partitioned table, you must first create the partitioned table and then use the INSERT INTO...SELECT FROM statement.
> 3. Indexes, constraints, and triggers defined in the source table are not transferred to the new table, nor can they be specified in the SELECT...INTO statement. If these objects are required, you can create them after executing the SELECT...INTO statement.
> 4. Specifying an ORDER BY clause does not guarantee the rows are inserted in the specified order. When a sparse column is included in the select list, the sparse column property does not transfer to the column in the new table. If this property is required in the new table, alter the column definition after executing the SELECT...INTO statement to include this property.
> 5. When a computed column is included in the select list, the corresponding column in the new table

> is not a computed column. The values in the new column are the values that were computed at the time SELECT...INTO was executed.

[*sic*]

## Section 1.12: Using Transactions to change data safely

Whenever you change data, in a Data Manipulation Language(DML) command, you can wrap your changes in a transaction. DML includes `UPDATE`, `TRUNCATE`, `INSERT` and `DELETE`. One of the ways that you can make sure that you're changing the right data would be to use a transaction.

DML changes will take a lock on the rows affected. When you begin a transaction, you must end the transaction or all objects being changed in the DML will remain locked by whoever began the transaction. You can end your transaction with either `ROLLBACK` or `COMMIT`. `ROLLBACK` returns everything within the transaction to its original state. `COMMIT` places the data into a final state where you cannot undo your changes without another DML statement.

**Example:**

```
--Create a test table

USE [your database]
GO
CREATE TABLE test_transaction (column_1 varchar(10))
GO

INSERT INTO
 dbo.test_transaction
        ( column_1 )
VALUES
        ( 'a' )

BEGIN TRANSACTION --This is the beginning of your transaction

UPDATE dbo.test_transaction
SET column_1 = 'B'
OUTPUT INSERTED.*
WHERE column_1 = 'A'


ROLLBACK TRANSACTION  --Rollback will undo your changes
           --Alternatively, use COMMIT to save your results

SELECT * FROM dbo.test_transaction   --View the table after your changes have been run

DROP TABLE dbo.test_transaction
```

**Notes:**

- This is a **simplified example** which does not include error handling. But any database operation can fail and hence throw an exception. **Here is an example** how such a required error handling might look like. You should **never** use transactions **without an error handler**, otherwise you might leave the transaction in an unknown state.
- Depending on the **isolation level**, transactions are putting locks on the data being queried or changed. You need to ensure that transactions are not running for a long time, because they will lock records in a database and can lead to **deadlocks** with other parallel running transactions. Keep the operations encapsulated in transactions as short as possible and minimize the impact with the amount of data you're locking.

# Section 1.13: Getting Table Row Count

The following example can be used to find the total row count for a specific table in a database if `table_name` is replaced by the the table you wish to query:

```
SELECT COUNT(*) AS [TotalRowCount] FROM TABLE_NAME;
```

It is also possible to get the row count for all tables by joining back to the table's partition based off the tables' HEAP (index_id = 0) or cluster clustered index (index_id = 1) using the following script:

```
SELECT  [TABLES].name                AS [TableName],
        SUM( [Partitions].[ROWS] )    AS [TotalRowCount]
FROM    sys.tables AS [TABLES]
JOIN    sys.partitions AS [Partitions]
    ON  [TABLES].[object_id]    =    [Partitions].[object_id]
    AND [Partitions].index_id IN ( 0, 1 )
--WHERE    [Tables].name = N'table name' /* uncomment to look for a specific table */
GROUP BY    [TABLES].name;
```

This is possible as every table is essentially a single partition table, unless extra partitions are added to it. This script also has the benefit of not interfering with read/write operations to the tables rows'.

# Chapter 2: Data Types

This section discusses the data types that SQL Server can use, including their data range, length, and limitations (if any.)

## Section 2.1: Exact Numerics

There are two basic classes of exact numeric data types - **Integer**, and **Fixed Precision and Scale**.

**Integer Data Types**

- bit
- tinyint
- smallint
- int
- bigint

*Integers* are numeric values that never contain a fractional portion, and always use a fixed amount of storage. The range and storage sizes of the integer data types are shown in this table:

| Data type | Range | Storage |
|---|---|---|
| bit | 0 or 1 | 1 bit ** |
| tinyint | 0 to 255 | 1 byte |
| smallint | -2^15 (-32,768) to 2^15-1 (32,767) | 2 bytes |
| int | -2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647) | 4 bytes |
| bigint | -2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807) | 8 bytes |

**Fixed Precision and Scale Data Types**

- numeric
- decimal
- smallmoney
- money

These data types are useful for representing numbers exactly. As long as the values can fit within the range of the values storable in the data type, the value will not have rounding issues. This is useful for any financial calculations, where rounding errors will result in clinical insanity for accountants.

Note that **decimal** and **numeric** are synonyms for the same data type.

| Data type | Range | Storage |
|---|---|---|
| Decimal [(p [, s])] or Numeric [(p [, s])] | -10^38 + 1 to 10^38 - 1 | See **Precision** table |

When defining a *decimal* or *numeric* data type, you may need to specify the Precision [p] and Scale [s].

Precision is the number of digits that can be stored. For example, if you needed to store values between 1 and 999, you would need a Precision of 3 (to hold the three digits in 100). If you do not specify a precision, the default precision is 18.

Scale is the number of digits after the decimal point. If you needed to store a number between 0.00 and 999.99, you would need to specify a Precision of 5 (five digits) and a Scale of 2 (two digits after the decimal point). You must specify a precision to specify a scale. The default scale is zero.

The Precision of a *decimal* or *numeric* data type defines the number of bytes required to store the value, as shown

below:

**Precision Table**

| Precision | Storage bytes |
|---|---|
| 1 - 9 | 5 |
| 10-19 | 9 |
| 20-28 | 13 |
| 29-38 | 17 |

**Monetary Fixed Data Types**

These data types are intended specifically for accounting and other monetary data. These type have a fixed Scale of 4 - you will always see four digits after the decimal place. For most systems working with most currencies, using a *numeric* value with a Scale of 2 will be sufficient. Note that no information about the type of currency represented is stored with the value.

| Data type | Range | Storage |
|---|---|---|
| money | -922,337,203,685,477.5808 to 922,337,203,685,477.5807 | 8 bytes |
| smallmoney | -214,748.3648 to 214,748.3647 | 4 bytes |

# Section 2.2: Approximate Numerics

- float [(**n**)]
- real

These data types are used to store floating point numbers. Since these types are intended to hold approximate numeric values only, these should not be used in cases where any rounding error is unacceptable. However, if you need to handle very large numbers, or numbers with an indeterminate number of digits after the decimal place, these may be your best option.

| Data type | Range | Size |
|---|---|---|
| float | -1.79E+308 to -2.23E-308, 0 and 2.23E-308 to 1.79E+308 | depends on **n** in table below |
| real | -3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38 | 4 Bytes |

**n** value table for *float* numbers. If no value is specified in the declaration of the float, the default value of 53 will be used. Note that *float(24)* is the equivalent of a *real* value.

| n value | Precision | Size |
|---|---|---|
| 1-24 | 7 digits | 4 bytes |
| 25-53 | 15 digits | 8 bytes |

# Section 2.3: Date and Time

These types are in all versions of SQL Server

- datetime
- smalldatetime

These types are in all versions of SQL Server after SQL Server 2012

- date
- datetimeoffset
- datetime2
- time

## Section 2.4: Character Strings

- char
- varchar
- text

## Section 2.5: Unicode Character Strings

- nchar
- nvarchar
- ntext

## Section 2.6: Binary Strings

- binary
- varbinary
- image

## Section 2.7: Other Data Types

- cursor
- timestamp
- hierarchyid
- uniqueidentifier
- sql_variant
- xml
- table
- Spatial Types

# Chapter 3: Converting data types

## Section 3.1: TRY PARSE

Version ≥ SQL Server 2012

It converts string data type to target data type(Date or Numeric).

For example, source data is string type and we need to covert to date type. If conversion attempt fails it returns NULL value.

Syntax: TRY_PARSE (string_value AS data_type [ USING culture ])

String_value – This is argument is source value which is NVARCHAR(4000) type.
Data_type – This argument is target data type either date or numeric.
Culture – It is an optional argument which helps to convert the value to in Culture format. Suppose you want to display the date in French, then you need to pass culture type as 'Fr-FR'. If you will not pass any valid culture name, then PARSE will raise an error.

```
DECLARE @fakeDate AS varchar(10);
DECLARE @realDate AS VARCHAR(10);
SET @fakeDate = 'iamnotadate';
SET @realDate = '13/09/2015';

SELECT TRY_PARSE(@fakeDate AS DATE); --NULL  as the parsing fails

SELECT TRY_PARSE(@realDate AS DATE); -- NULL due to type mismatch

SELECT TRY_PARSE(@realDate AS DATE USING 'Fr-FR'); -- 2015-09-13
```

## Section 3.2: TRY CONVERT

Version ≥ SQL Server 2012

It converts value to specified data type and if conversion fails it returns NULL. For example, source value in string format and we need date/integer format. Then this will help us to achieve the same.

Syntax: TRY_CONVERT ( data_type [ ( length ) ], expression [, style ] )

TRY_CONVERT() returns a value cast to the specified data type if the cast succeeds; otherwise, returns null.

Data_type - The datatype into which to convert. Here length is an optional parameter which helps to get result in specified length.
Expression - The value to be convert
Style - It is an optional parameter which determines formatting. Suppose you want date format like "May, 18 2013" then you need pass style as 111.

```
DECLARE @sampletext AS VARCHAR(10);
SET @sampletext = '123456';
DECLARE @ realDate AS VARCHAR(10);
SET @realDate = '13/09/2015';
SELECT TRY_CONVERT(INT, @sampletext); -- 123456
SELECT TRY_CONVERT(DATETIME, @sampletext); -- NULL
SELECT TRY_CONVERT(DATETIME, @realDate, 111); -- Sep, 13 2015
```

# Section 3.3: TRY CAST

Version ≥ SQL Server 2012

It converts value to specified data type and if conversion fails it returns NULL. For example, source value in string format and we need it in double/integer format. Then this will help us in achieving it.

Syntax: TRY_CAST ( expression AS data_type [ ( length ) ] )

TRY_CAST() returns a value cast to the specified data type if the cast succeeds; otherwise, returns null.

Expression - The source value which will go to cast.
Data_type - The target data type the source value will cast.
Length - It is an optional parameter that specifies the length of target data type.

```
DECLARE @sampletext AS VARCHAR(10);
SET @sampletext = '123456';

SELECT TRY_CAST(@sampletext AS INT); -- 123456
SELECT TRY_CAST(@sampletext AS DATE); -- NULL
```

# Section 3.4: Cast

The Cast() function is used to convert a data type variable or data from one data type to another data type.

Syntax

CAST ( [Expression] AS Datatype)

The data type to which you are casting an expression is the target type. The data type of the expression from which you are casting is the source type.

```
DECLARE @A varchar(2)
DECLARE @B varchar(2)

set @A='25a'
set @B='15'

Select CAST(@A as int) + CAST(@B as int)  as Result
--'25a' is casted to 25 (string to int)
--'15' is casted to 15 (string to int)

--Result
 --40

DECLARE @C varchar(2)  = 'a'

select CAST(@C as int) as Result
--Result
 --Conversion failed when converting the varchar value 'a' to data type int.
```

Throws error if failed

# Section 3.5: Convert

When you convert expressions from one type to another, in many cases there will be a need within a stored procedure or other routine to convert data from a datetime type to a varchar type. The Convert function is used for

such things. The CONVERT() function can be used to display date/time data in various formats. Syntax

CONVERT(data_type(length), expression, style)

Style - style values for datetime or smalldatetime conversion to character data. Add 100 to a style value to get a four-place year that includes the century (yyyy).

```
SELECT CONVERT(VARCHAR(20),GETDATE(),108)

13:27:16
```

# Chapter 4: User Defined Table Types

User defined table types (UDT for short) are data types that allows the user to define a table structure. User defined table types supports primary keys, unique constraints and default values.

## Section 4.1: creating a UDT with a single int column that is also a primary key

```
CREATE TYPE dbo.Ids as TABLE
(
    Id int PRIMARY KEY
)
```

## Section 4.2: Creating a UDT with multiple columns

```
CREATE TYPE MyComplexType as TABLE
(
    Id int,
    Name varchar(10)
)
```

## Section 4.3: Creating a UDT with a unique constraint:

```
CREATE TYPE MyUniqueNamesType as TABLE
(
    FirstName varchar(10),
    LastName varchar(10),
    UNIQUE (FirstName,LastName)
)
```

Note: constraints in user defined table types can not be named.

## Section 4.4: Creating a UDT with a primary key and a column with a default value:

```
CREATE TYPE MyUniqueNamesType as TABLE
(
    FirstName varchar(10),
    LastName varchar(10),
    CreateDate datetime default GETDATE()
    PRIMARY KEY (FirstName,LastName)
)
```

# Chapter 5: SELECT statement

In SQL, `SELECT` statements return sets of results from data collections like tables or views. `SELECT` statements can be used with various other clauses like `WHERE`, `GROUP BY`, or `ORDER BY` to further refine the desired results.

## Section 5.1: Basic SELECT from table

Select all columns from some table (system table in this case):

```
SELECT *
FROM sys.objects
```

Or, select just some specific columns:

```
SELECT object_id, name, TYPE, create_date
FROM sys.objects
```

## Section 5.2: Filter rows using WHERE clause

WHERE clause filters only those rows that satisfy some condition:

```
SELECT *
FROM sys.objects
WHERE TYPE = 'IT'
```

## Section 5.3: Sort results using ORDER BY

ORDER BY clause sorts rows in the returned result set by some column or expression:

```
SELECT *
FROM sys.objects
ORDER BY create_date
```

## Section 5.4: Group result using GROUP BY

GROUP BY clause groups rows by some value:

```
SELECT TYPE, COUNT(*) AS c
FROM sys.objects
GROUP BY TYPE
```

You can apply some function on each group (aggregate function) to calculate sum or count of the records in the group.

| type | c |
|---|---|
| SQ | 3 |
| S | 72 |
| IT | 16 |
| PK | 1 |
| U | 5 |

## Section 5.5: Filter groups using HAVING clause

HAVING clause removes groups that do not satisfy condition:

```
SELECT TYPE, COUNT(*) AS c
FROM sys.objects
GROUP BY TYPE
HAVING COUNT(*) < 10
```

| type | c |
|---|---|
| SQ | 3 |
| PK | 1 |
| U | 5 |

## Section 5.6: Returning only first N rows

TOP clause returns only first N rows in the result:

```
SELECT TOP 10 *
FROM sys.objects
```

## Section 5.7: Pagination using OFFSET FETCH

OFFSET FETCH clause is more advanced version of TOP. It enables you to skip N1 rows and take next N2 rows:

```
SELECT *
FROM sys.objects
ORDER BY object_id
OFFSET 50 ROWS FETCH NEXT 10 ROWS ONLY
```

You can use OFFSET without fetch to just skip first 50 rows:

```
SELECT *
FROM sys.objects
ORDER BY object_id
OFFSET 50 ROWS
```

## Section 5.8: SELECT without FROM (no data souce)

SELECT statement can be executed without FROM clause:

```
declare @var int = 17;

SELECT @var as c1, @var + 2 as c2, 'third' as c3
```

In this case, one row with values/results of expressions are returned.

# Chapter 6: Alias Names in SQL Server

Here is some of different ways to provide alias names to columns in Sql Server

## Section 6.1: Giving alias after Derived table name

This is a weird approach most of the people don't know this even exist.

```
CREATE TABLE AliasNameDemo(id INT,firstname VARCHAR(20),lastname VARCHAR(20))

INSERT INTO AliasNameDemo
VALUES      (1,'MyFirstName','MyLastName')

SELECT *
FROM   (SELECT firstname + ' ' + lastname
        FROM   AliasNameDemo) a (fullname)
```

- Demo

## Section 6.2: Using AS

This is ANSI SQL method works in all the RDBMS. Widely used approach.

```
CREATE TABLE AliasNameDemo (id INT,firstname VARCHAR(20),lastname VARCHAR(20))

INSERT INTO AliasNameDemo
VALUES      (1,'MyFirstName','MyLastName')

SELECT FirstName +' '+ LastName As FullName
FROM   AliasNameDemo
```

## Section 6.3: Using =

This is my preferred approach. Nothing related to performance just a personal choice. It makes the code to look clean. You can see the resulting column names easily instead of scrolling the code if you have a big expression.

```
CREATE TABLE AliasNameDemo (id INT,firstname VARCHAR(20),lastname VARCHAR(20))

INSERT INTO AliasNameDemo
VALUES      (1,'MyFirstName','MyLastName')

SELECT FullName = FirstName +' '+ LastName
FROM   AliasNameDemo
```

## Section 6.4: Without using AS

This syntax will be similar to using `AS` keyword. Just we don't have to use `AS` keyword

```
CREATE TABLE AliasNameDemo (id INT,firstname VARCHAR(20),lastname VARCHAR(20))

INSERT INTO AliasNameDemo
VALUES      (1,'MyFirstName','MyLastName')

SELECT FirstName +' '+ LastName FullName
FROM   AliasNameDemo
```

# Chapter 7: NULLs

In SQL Server, `NULL` represents data that is missing, or unknown. This means that `NULL` is not really a value; it's better described as a placeholder for a value. This is also the reason why you can't compare `NULL` with any value, and not even with another `NULL`.

## Section 7.1: COALESCE ()

`COALESCE ()` Evaluates the arguments in order and returns the current value of the first expression that initially does not evaluate to `NULL`.

```
DECLARE @MyInt int -- variable is null until it is set with value.
DECLARE @MyInt2 int -- variable is null until it is set with value.
DECLARE @MyInt3 int -- variable is null until it is set with value.

SET @MyInt3  = 3

SELECT COALESCE (@MyInt, @MyInt2 ,@MyInt3 ,5) -- Returns 3 : value of @MyInt3.
```

Although ISNULL() operates similarly to COALESCE(), the ISNULL() function only accepts two parameters - one to check, and one to use if the first parameter is NULL. See also `ISNULL`, below

## Section 7.2: ANSI NULLS

From MSDN

> In a future version of SQL Server, ANSI_NULLS will always be ON and any applications that explicitly set the option to OFF will generate an error. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

`ANSI NULLS` being set to off allows for a `=`/`<>` comparison of null values.

Given the following data:

```
id someVal
 ----
 0 NULL
 1 1
 2 2
```

And with ANSI NULLS on, this query:

```
SELECT id
FROM TABLE
WHERE someVal = NULL
```

would produce no results. However the same query, with ANSI NULLS off:

```
set ansi_nulls off

SELECT id
FROM table
```

```
WHERE someVal = NULL
```

Would return id `0`.

## Section 7.3: ISNULL()

The `IsNull()` function accepts two parameters, and returns the second parameter if the first one is `null`.

Parameters:

1. check expression. Any expression of any data type.
2. replacement value. This is the value that would be returned if the check expression is null. The replacement value must be of a data type that can be implicitly converted to the data type of the check expression.

The `IsNull()` function returns the same data type as the check expression.

```
DECLARE @MyInt int -- All variables are null until they are set with values.

SELECT ISNULL(@MyInt, 3) -- Returns 3.
```

See also `COALESCE`, above

## Section 7.4: Is null / Is not null

Since null is not a value, you can't use comparison operators with nulls.
To check if a column or variable holds null, you need to use `is null`:

```
DECLARE @Date date = '2016-08-03'
```

The following statement will select the value `6`, since all comparisons with null values evaluates to false or unknown:

```
SELECT CASE WHEN @DATE = NULL THEN 1
            WHEN @DATE <> NULL THEN 2
            WHEN @DATE > NULL THEN 3
            WHEN @DATE < NULL THEN 4
            WHEN @DATE IS NULL THEN 5
            WHEN @DATE IS NOT NULL THEN 6
```

Setting the content of the @Date variable to `null` and try again, the following statement will return `5`:

```
SET @Date = NULL -- Note that the '=' here is an assignment operator!

SELECT CASE WHEN @Date = NULL THEN 1
            WHEN @Date <> NULL THEN 2
            WHEN @Date > NULL THEN 3
            WHEN @Date < NULL THEN 4
            WHEN @Date IS NULL THEN 5
            WHEN @Date IS NOT NULL THEN 6
```

## Section 7.5: NULL comparison

`NULL` is a special case when it comes to comparisons.

Assume the following data.

```
```

```
id someVal
 ----
 0 NULL
 1 1
 2 2
```

With a query:

```
SELECT id
FROM TABLE
WHERE someVal = 1
```

would return id `1`

```
SELECT id
FROM TABLE
WHERE someVal <> 1
```

would return id `2`

```
SELECT id
FROM TABLE
WHERE someVal IS NULL
```

would return id `0`

```
SELECT id
FROM TABLE
WHERE someVal IS NOT NULL
```

would return both ids `1` and `2`.

If you wanted NULLs to be "counted" as values in a `=`, `<>` comparison, it must first be converted to a countable data type:

```
SELECT id
FROM TABLE
WHERE ISNULL(someVal, -1) <> 1
```

OR

```
SELECT id
FROM TABLE
WHERE someVal IS NULL OR someVal <> 1
```

returns `0` and `2`

Or you can change your ANSI Null setting.

## Section 7.6: NULL with NOT IN SubQuery

While handling not in sub-query with null in the sub-query we need to eliminate NULLS to get your expected results

```
create table #outertable (i int)
create table #innertable (i int)
```

```
insert into #outertable (i) values (1), (2),(3),(4), (5)
insert into #innertable (i) values (2), (3), (null)

select * from #outertable where i in (select i from #innertable)
--2
--3
--So far so good

select * from #outertable where i not in (select i from #innertable)
--Expectation here is to get 1,4,5 but it is not. It will get empty results because of the NULL it
executes as {select * from #outertable where i not in (null)}

--To fix this
select * from #outertable where i not in (select i from #innertable where i is not null)
--you will get expected results
--1
--4
--5
```

While handling not in sub-query with null be cautious with your expected output

# Chapter 8: Variables

## Section 8.1: Declare a Table Variable

```
DECLARE @Employees TABLE
(
    EmployeeID INT NOT NULL PRIMARY KEY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    ManagerID INT NULL
)
```

When you create a normal table, you use `CREATE TABLE Name (Columns)` syntax. When creating a table variable, you use `DECLARE @Name TABLE (Columns)` syntax.

To reference the table variable inside a `SELECT` statement, SQL Server requires that you give the table variable an alias, otherwise you'll get an error:

> Must declare the scalar variable "@TableVariableName".

i.e.

```
DECLARE @Table1 TABLE (Example INT)
DECLARE @Table2 TABLE (Example INT)

/*
-- the following two commented out statements would generate an error:
SELECT *
FROM @Table1
INNER JOIN @Table2 ON @Table1.Example = @Table2.Example

SELECT *
FROM @Table1
WHERE @Table1.Example = 1
*/

-- but these work fine:
SELECT *
FROM @Table1 T1
INNER JOIN @Table2 T2 ON T1.Example = T2.Example

SELECT *
FROM @Table1 Table1
WHERE Table1.Example = 1
```

## Section 8.2: Updating variables using SELECT

Using `SELECT`, you can update multiple variables at once.

```
DECLARE @Variable1 INT, @Variable2 VARCHAR(10)
SELECT @Variable1 = 1, @Variable2 = 'Hello'
PRINT @Variable1
PRINT @Variable2
```

> 1
>
> Hello

When using `SELECT` to update a variable from a table column, if there are multiple values, it will use the *last* value. (Normal order rules apply - if no sort is given, the order is not guaranteed.)

```
CREATE TABLE #Test (Example INT)
INSERT INTO #Test VALUES (1), (2)

DECLARE @Variable INT
SELECT @Variable = Example
FROM #Test
ORDER BY Example ASC

PRINT @Variable
```

> 2

```
SELECT TOP 1 @Variable = Example
FROM #Test
ORDER BY Example ASC

PRINT @Variable
```

> 1

If there are no rows returned by the query, the variable's value won't change:

```
SELECT TOP 0 @Variable = Example
FROM #Test
ORDER BY Example ASC

PRINT @Variable
```

> 1

# Section 8.3: Declare multiple variables at once, with initial values

```
DECLARE
  @Var1 INT = 5,
  @Var2 NVARCHAR(50) = N'Hello World',
  @Var3 DATETIME = GETDATE()
```

# Section 8.4: Updating a variable using SET

```
DECLARE @VariableName INT
SET @VariableName = 1
PRINT @VariableName
```

> 1

Using `SET`, you can only update one variable at a time.

## Section 8.5: Updating variables by selecting from a table

Depending on the structure of your data, you can create variables that update dynamically.

```
DECLARE @CurrentID int = (SELECT TOP 1 ID FROM Table ORDER BY CreateDate desc)


DECLARE @Year int = 2014
DECLARE @CurrentID int = (SELECT ID FROM Table WHERE Year = @Year)
```

In most cases, you will want to ensure that your query returns only one value when using this method.

## Section 8.6: Compound assignment operators

`Version ≥ SQL Server 2008 R2`

Supported compound operators:

> `+=` Add and assign
>
> `-=` Subtract and assign
>
> `*=` Multiply and assign
>
> `/=` Divide and assign
>
> `%=` Modulo and assign
>
> `&=` Bitwise AND and assign
>
> `^=` Bitwise XOR and assign
>
> `|=` Bitwise OR and assign

Example usage:

```
DECLARE @test INT = 42;
SET @test += 1;
PRINT @test;    --43
SET @test -= 1;
PRINT @test;    --42
SET @test *= 2
PRINT @test;    --84
SET @test /= 2;
PRINT @test;    --42
```

# Chapter 9: Dates

## Section 9.1: Date & Time Formatting using CONVERT

You can use the CONVERT function to cast a datetime datatype to a formatted string.

```
SELECT GETDATE() AS [RESULT] -- 2016-07-21 07:56:10.927
```

You can also use some built-in codes to convert into a specific format. Here are the options built into SQL Server:

```
DECLARE @convert_code INT = 100 -- See Table Below
SELECT CONVERT(VARCHAR(30), GETDATE(), @convert_code) AS [Result]
```

| @convert_code | Result |
|---|---|
| **100** | "Jul 21 2016 7:56AM" |
| **101** | "07/21/2016" |
| **102** | "2016.07.21" |
| **103** | "21/07/2016" |
| **104** | "21.07.2016" |
| **105** | "21-07-2016" |
| **106** | "21 Jul 2016" |
| **107** | "Jul 21, 2016" |
| **108** | "07:57:05" |
| **109** | "Jul 21 2016 7:57:45:707AM" |
| **110** | "07-21-2016" |
| **111** | "2016/07/21" |
| **112** | "20160721" |
| **113** | "21 Jul 2016 07:57:59:553" |
| **114** | "07:57:59:553" |
| **120** | "2016-07-21 07:57:59" |
| **121** | "2016-07-21 07:57:59.553" |
| **126** | "2016-07-21T07:58:34.340" |
| **127** | "2016-07-21T07:58:34.340" |
| **130** | "16 ???? 1437 7:58:34:340AM" |
| **131** | "16/10/1437 7:58:34:340AM" |

```
SELECT GETDATE() AS [RESULT]                                -- 2016-07-21 07:56:10.927
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),100) AS [RESULT] -- Jul 21 2016  7:56AM
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),101) AS [RESULT] -- 07/21/2016
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),102) AS [RESULT] -- 2016.07.21
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),103) AS [RESULT] -- 21/07/2016
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),104) AS [RESULT] -- 21.07.2016
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),105) AS [RESULT] -- 21-07-2016
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),106) AS [RESULT] -- 21 Jul 2016
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),107) AS [RESULT] -- Jul 21, 2016
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),108) AS [RESULT] -- 07:57:05
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),109) AS [RESULT] -- Jul 21 2016  7:57:45:707AM
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),110) AS [RESULT] -- 07-21-2016
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),111) AS [RESULT] -- 2016/07/21
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),112) AS [RESULT] -- 20160721
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),113) AS [RESULT] -- 21 Jul 2016 07:57:59:553
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),114) AS [RESULT] -- 07:57:59:553
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),120) AS [RESULT] -- 2016-07-21 07:57:59
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),121) AS [RESULT] -- 2016-07-21 07:57:59.553
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),126) AS [RESULT] -- 2016-07-21T07:58:34.340
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),127) AS [RESULT] -- 2016-07-21T07:58:34.340
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),130) AS [RESULT] -- 16 ???? 1437  7:58:34:340AM
UNION SELECT CONVERT(VARCHAR(30),GETDATE(),131) AS [RESULT] -- 16/10/1437  7:58:34:340AM
```

# Section 9.2: Date & Time Formatting using FORMAT

Version ≥ SQL Server 2012

You can utilize the new function: `FORMAT()`.

Using this you can transform your `DATETIME` fields to your own custom `VARCHAR` format.

**Example**

```
DECLARE @Date DATETIME = '2016-09-05 00:01:02.333'

SELECT FORMAT(@Date, N'dddd, MMMM dd, yyyy hh:mm:ss tt')
```

> Monday, September 05, 2016 12:01:02 AM

**Arguments**

Given the `DATETIME` being formatted is `2016-09-05 00:01:02.333`, the following chart shows what their output would be for the provided argument.

| Argument | Output |
|---|---|
| yyyy | 2016 |
| yy | 16 |
| MMMM | September |
| MM | 09 |
| M | 9 |
| dddd | Monday |
| ddd | Mon |
| dd | 05 |
| d | 5 |
| HH | 00 |
| H | 0 |
| hh | 12 |
| h | 12 |
| mm | 01 |
| m | 1 |
| ss | 02 |
| s | 2 |
| tt | AM |
| t | A |
| fff | 333 |
| ff | 33 |
| f | 3 |

You can also supply a single argument to the `FORMAT()` function to generate a pre-formatted output:

```
DECLARE @Date DATETIME = '2016-09-05 00:01:02.333'

SELECT FORMAT(@Date, N'U')
```

> Monday, September 05, 2016 4:01:02 AM

| Single Argument | Output |
|---|---|
| D | Monday, September 05, 2016 |
| d | 9/5/2016 |
| F | Monday, September 05, 2016 12:01:02 AM |
| f | Monday, September 05, 2016 12:01 AM |
| G | 9/5/2016 12:01:02 AM |
| g | 9/5/2016 12:01 AM |
| M | September 05 |
| O | 2016-09-05T00:01:02.3330000 |
| R | Mon, 05 Sep 2016 00:01:02 GMT |
| s | 2016-09-05T00:01:02 |
| T | 12:01:02 AM |
| t | 12:01 AM |
| U | Monday, September 05, 2016 4:01:02 AM |
| u | 2016-09-05 00:01:02Z |
| Y | September, 2016 |

*Note: The above list is using the `en-US` culture. A different culture can be specified for the `FORMAT()` via the third parameter:*

```
DECLARE @Date DATETIME = '2016-09-05 00:01:02.333'

SELECT FORMAT(@Date, N'U', 'zh-cn')
```

> 2016?9?5? 4:01:02

## Section 9.3: DATEADD for adding and subtracting time periods

General syntax:

```
DATEADD (datepart , number , datetime_expr)
```

To add a time measure, the `number` must be positive. To subtract a time measure, the `number` must be negative.

Examples

```
DECLARE @now DATETIME2 = GETDATE();
SELECT @now;                          --2016-07-21 14:39:46.4170000
SELECT DATEADD(YEAR, 1, @now)         --2017-07-21 14:39:46.4170000
SELECT DATEADD(QUARTER, 1, @now)      --2016-10-21 14:39:46.4170000
SELECT DATEADD(WEEK, 1, @now)         --2016-07-28 14:39:46.4170000
SELECT DATEADD(DAY, 1, @now)          --2016-07-22 14:39:46.4170000
SELECT DATEADD(HOUR, 1, @now)         --2016-07-21 15:39:46.4170000
SELECT DATEADD(MINUTE, 1, @now)       --2016-07-21 14:40:46.4170000
SELECT DATEADD(SECOND, 1, @now)       --2016-07-21 14:39:47.4170000
SELECT DATEADD(MILLISECOND, 1, @now)--2016-07-21 14:39:46.4180000
```

NOTE: `DATEADD` also accepts abbreviations in the `datepart` parameter. Use of these abbreviations is generally discouraged as they can be confusing (`m` vs `mi`, `ww` vs `w`, etc.).

## Section 9.4: Create function to calculate a person's age on a specific date

This function will take 2 datetime parameters, the DOB, and a date to check the age at

```
  CREATE FUNCTION [dbo].[Calc_Age]
    (
    @DOB datetime , @calcDate datetime
    )
    RETURNS int
    AS
   BEGIN
declare @age int

IF (@calcDate < @DOB  )
RETURN -1

-- If a DOB is supplied after the comparison date, then return -1
SELECT @age = YEAR(@calcDate) - YEAR(@DOB) +
  CASE WHEN DATEADD(year,YEAR(@calcDate) - YEAR(@DOB)
  ,@DOB) > @calcDate THEN -1 ELSE 0 END


RETURN @age


END
```

eg to check the age today of someone born on 1/1/2000

```
SELECT  dbo.Calc_Age('2000-01-01',Getdate())
```

## Section 9.5: Get the current DateTime

The built-in functions GETDATE and GETUTCDATE each return the current date and time without a time zone offset.

The return value of both functions is based on the operating system of the computer on which the instance of SQL Server is running.

The return value of GETDATE represents the current time in the same timezone as operating system. The return value of GETUTCDATE represents the current UTC time.

Either function can be included in the SELECT clause of a query or as part of boolean expression in the WHERE clause.

Examples:

```
-- example query that selects the current time in both the server time zone and UTC
SELECT GETDATE() as SystemDateTime, GETUTCDATE() as UTCDateTime

-- example query records with EventDate in the past.
SELECT * FROM MyEvents WHERE EventDate < GETDATE()
```

There are a few other built-in functions that return different variations of the current date-time:

```
SELECT
    GETDATE(),          --2016-07-21 14:27:37.447
    GETUTCDATE(),       --2016-07-21 18:27:37.447
```

```
    CURRENT_TIMESTAMP,   --2016-07-21 14:27:37.447
    SYSDATETIME(),       --2016-07-21 14:27:37.4485768
    SYSDATETIMEOFFSET(),--2016-07-21 14:27:37.4485768 -04:00
    SYSUTCDATETIME()     --2016-07-21 18:27:37.4485768
```

## Section 9.6: Getting the last day of a month

Using the `DATEADD` and `DATEDIFF` functions, it's possible to return the last date of a month.

```
SELECT DATEADD(d, -1, DATEADD(m, DATEDIFF(m, 0, '2016-09-23') + 1, 0))
-- 2016-09-30 00:00:00.000
```

Version ≥ SQL Server 2012

The `EOMONTH` function provides a more concise way to return the last date of a month, and has an optional parameter to offset the month.

```
SELECT EOMONTH('2016-07-21')         --2016-07-31
SELECT EOMONTH('2016-07-21', 4)      --2016-11-30
SELECT EOMONTH('2016-07-21', -5)     --2016-02-29
```

## Section 9.7: CROSS PLATFORM DATE OBJECT

Version ≥ SQL Server 2012

In Transact SQL , you may define an object as `Date` (or `DateTime`) using the `[DATEFROMPARTS][1]` (or `[DATETIMEFROMPARTS][1]`) function like following:

```
DECLARE @myDate DATE=DATEFROMPARTS(1988,11,28)
DECLARE @someMoment DATETIME=DATEFROMPARTS(1988,11,28,10,30,50,123)
```

The parameters you provide are Year, Month, Day for the `DATEFROMPARTS` function and, for the `DATETIMEFROMPARTS` function you will need to provide year, month, day, hour, minutes, seconds and milliseconds.

These methods are useful and worth being used because using the plain string to build a date(or datetime) may fail depending on the host machine region, location or date format settings.

## Section 9.8: Return just Date from a DateTime

There are many ways to return a Date from a DateTime object

1. `SELECT CONVERT(DATE, GETDATE())`
2. `SELECT DATEADD(dd, 0, DATEDIFF(dd, 0, GETDATE()))` returns 2016-07-21 00:00:00.000
3. `SELECT CAST(GETDATE() AS DATE)`
4. `SELECT CONVERT(CHAR(10),GETDATE(),111)`
5. `SELECT FORMAT(GETDATE(), 'yyyy-MM-dd')`

Note that options 4 and 5 returns a string, not a date.

## Section 9.9: DATEDIFF for calculating time period differences

General syntax:

```
DATEDIFF (datepart, datetime_expr1, datetime_expr2)
```

It will return a positive number if `datetime_expr` is in the past relative to `datetime_expr2`, and a negative number otherwise.

Examples

```
DECLARE @now DATETIME2 = GETDATE();
DECLARE @oneYearAgo DATETIME2 = DATEADD(YEAR, -1, @now);
SELECT @now                                 --2016-07-21 14:49:50.9800000
SELECT @oneYearAgo                          --2015-07-21 14:49:50.9800000
SELECT DATEDIFF(YEAR, @oneYearAgo, @now)       --1
SELECT DATEDIFF(QUARTER, @oneYearAgo, @now)    --4
SELECT DATEDIFF(WEEK, @oneYearAgo, @now)       --52
SELECT DATEDIFF(DAY, @oneYearAgo, @now)        --366
SELECT DATEDIFF(HOUR, @oneYearAgo, @now)       --8784
SELECT DATEDIFF(MINUTE, @oneYearAgo, @now)     --527040
SELECT DATEDIFF(SECOND, @oneYearAgo, @now)     --31622400
```

NOTE: `DATEDIFF` also accepts abbreviations in the `datepart` parameter. Use of these abbreviations is generally discouraged as they can be confusing (`m` vs `mi`, `ww` vs `w`, etc.).

`DATEDIFF` can also be used to determine the offset between UTC and the local time of the SQL Server. The following statement can be used to calculate the offset between UTC and local time (including timezone).

```
SELECT  DATEDIFF(hh, getutcdate(), getdate()) AS 'CentralTimeOffset'
```

# Section 9.10: DATEPART & DATENAME

`DATEPART` returns the specified `datepart` of the specified datetime expression as a numeric value.

`DATENAME` returns a character string that represents the specified `datepart` of the specified date. In practice `DATENAME` is mostly useful for getting the name of the month or the day of the week.

There are also some shorthand functions to get the year, month or day of a datetime expression, which behave like `DATEPART` with their respective `datepart` units.

Syntax:

```
DATEPART ( datepart , datetime_expr )
DATENAME ( datepart , datetime_expr )
DAY ( datetime_expr )
MONTH ( datetime_expr )
YEAR ( datetime_expr )
```

Examples:

```
DECLARE @now DATETIME2 = GETDATE();
SELECT @now                         --2016-07-21 15:05:33.8370000
SELECT DATEPART(YEAR, @now)         --2016
SELECT DATEPART(QUARTER, @now)      --3
SELECT DATEPART(WEEK, @now)         --30
SELECT DATEPART(HOUR, @now)         --15
SELECT DATEPART(MINUTE, @now)       --5
SELECT DATEPART(SECOND, @now)       --33
-- Differences between DATEPART and DATENAME:
SELECT DATEPART(MONTH, @now)        --7
SELECT DATENAME(MONTH, @now)        --July
SELECT DATEPART(WEEKDAY, @now)      --5
```

```
SELECT DATENAME(WEEKDAY, @now)    --Thursday
--shorthand functions
SELECT DAY(@now)    --21
SELECT MONTH(@now)  --7
SELECT YEAR(@now)   --2016
```

NOTE: `DATEPART` and `DATENAME` also accept abbreviations in the `datepart` parameter. Use of these abbreviations is generally discouraged as they can be confusing (`m` vs `mi`, `ww` vs `w`, etc.).

## Section 9.11: Date parts reference

These are the `datepart` values available to date & time functions:

| datepart | Abbreviations |
|---|---|
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| weekday | dw, w |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |

**NOTE**: Use of abbreviations is generally discouraged as they can be confusing (`m` vs `mi`, `ww` vs `w`, etc.). The long version of the `datepart` representation promotes clarity and readability, and should be used whenever possible (`month`, `minute`, week, `weekday`, etc.).

## Section 9.12: Date Format Extended

| Date Format | SQL Statement | Sample Output |
|---|---|---|
| YY-MM-DD | SELECT RIGHT(CONVERT(VARCHAR(10), SYSDATETIME(), 20), 8) AS [YY-MM-DD]<br>SELECT REPLACE(CONVERT(VARCHAR(8), SYSDATETIME(), 11), '/', '-') AS [YY-MM-DD] | 11-06-08 |
| YYYY-MM-DD | SELECT CONVERT(VARCHAR(10), SYSDATETIME(), 120) AS [YYYY-MM-DD]<br>SELECT REPLACE(CONVERT(VARCHAR(10), SYSDATETIME(), 111), '/', '-') AS [YYYY-MM-DD] | 2011-06-08 |
| YYYY-M-D | SELECT CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) + '-' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [YYYY-M-D] | 2011-6-8 |
| YY-M-D | SELECT RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) + '-' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [YY-M-D] | 11-6-8 |
| M-D-YYYY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [M-D-YYYY] | 6-8-2011 |
| M-D-YY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '-' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [M-D-YY] | 6-8-11 |
| D-M-YYYY | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [D-M-YYYY] | 8-6-2011 |

| | | |
|---|---|---|
| D-M-YY | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [D-M-YY] | 8-6-11 |
| YY-MM | SELECT RIGHT(CONVERT(VARCHAR(7), SYSDATETIME(), 20), 5) AS [YY-MM]<br>SELECT SUBSTRING(CONVERT(VARCHAR(10), SYSDATETIME(), 120), 3, 5) AS [YY-MM] | 11-06 |
| YYYY-MM | SELECT CONVERT(VARCHAR(7), SYSDATETIME(), 120) AS [YYYY-MM] | 2011-06 |
| YY-M | SELECT RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) + '-' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YY-M] | 11-6 |
| YYYY-M | SELECT CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) + '-' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YYYY-M] | 2011-6 |
| MM-YY | SELECT RIGHT(CONVERT(VARCHAR(8), SYSDATETIME(), 5), 5) AS [MM-YY]<br>SELECT SUBSTRING(CONVERT(VARCHAR(8), SYSDATETIME(), 5), 4, 5) AS [MM-YY] | 06-11 |
| MM-YYYY | SELECT RIGHT(CONVERT(VARCHAR(10), SYSDATETIME(), 105), 7) AS [MM-YYYY] | 06-2011 |
| M-YY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [M-YY] | 6-11 |
| M-YYYY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [M-YYYY] | 6-2011 |
| MM-DD | SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 10) AS [MM-DD] | 06-08 |
| DD-MM | SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 5) AS [DD-MM] | 08-06 |
| M-D | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [M-D] | 6-8 |
| D-M | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '-' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [D-M] | 8-6 |
| M/D/YYYY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [M/D/YYYY] | 6/8/2011 |
| M/D/YY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '/' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [M/D/YY] | 6/8/11 |
| D/M/YYYY | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [D/M/YYYY] | 8/6/2011 |
| D/M/YY | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [D/M/YY] | 8/6/11 |
| YYYY/M/D | SELECT CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [YYYY/M/D] | 2011/6/8 |
| YY/M/D | SELECT RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [YY/M/D] | 11/6/8 |
| MM/YY | SELECT RIGHT(CONVERT(VARCHAR(8), SYSDATETIME(), 3), 5) AS [MM/YY] | 06/11 |
| MM/YYYY | SELECT RIGHT(CONVERT(VARCHAR(10), SYSDATETIME(), 103), 7) AS [MM/YYYY] | 06/2011 |
| M/YY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [M/YY] | 6/11 |
| M/YYYY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [M/YYYY] | 6/2011 |
| YY/MM | SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 11) AS [YY/MM] | 11/06 |
| YYYY/MM | SELECT CONVERT(VARCHAR(7), SYSDATETIME(), 111) AS [YYYY/MM] | 2011/06 |
| YY/M | SELECT RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YY/M] | 11/6 |
| YYYY/M | SELECT CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YYYY/M] | 2011/6 |
| MM/DD | SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 1) AS [MM/DD] | 06/08 |
| DD/MM | SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 3) AS [DD/MM] | 08/06 |
| M/D | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [M/D] | 6/8 |
| D/M | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [D/M] | 8/6 |

| | | |
|---|---|---|
| MM.DD.YYYY | SELECT REPLACE(CONVERT(VARCHAR(10), SYSDATETIME(), 101), '/', '.') AS [MM.DD.YYYY] | 06.08.2011 |
| MM.DD.YY | SELECT REPLACE(CONVERT(VARCHAR(8), SYSDATETIME(), 1), '/', '.') AS [MM.DD.YY] | 06.08.11 |
| M.D.YYYY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [M.D.YYYY] | 6.8.2011 |
| M.D.YY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '.' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [M.D.YY] | 6.8.11 |
| DD.MM.YYYY | SELECT CONVERT(VARCHAR(10), SYSDATETIME(), 104) AS [DD.MM.YYYY] | 08.06.2011 |
| DD.MM.YY | SELECT CONVERT(VARCHAR(10), SYSDATETIME(), 4) AS [DD.MM.YY] | 08.06.11 |
| D.M.YYYY | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [D.M.YYYY] | 8.6.2011 |
| D.M.YY | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [D.M.YY] | 8.6.11 |
| YYYY.M.D | SELECT CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) + '.' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [YYYY.M.D] | 2011.6.8 |
| YY.M.D | SELECT RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) + '.' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [YY.M.D] | 11.6.8 |
| MM.YYYY | SELECT RIGHT(CONVERT(VARCHAR(10), SYSDATETIME(), 104), 7) AS [MM.YYYY] | 06.2011 |
| MM.YY | SELECT RIGHT(CONVERT(VARCHAR(8), SYSDATETIME(), 4), 5) AS [MM.YY] | 06.11 |
| M.YYYY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [M.YYYY] | 6.2011 |
| M.YY | SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [M.YY] | 6.11 |
| YYYY.MM | SELECT CONVERT(VARCHAR(7), SYSDATETIME(), 102) AS [YYYY.MM] | 2011.06 |
| YY.MM | SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 2) AS [YY.MM] | 11.06 |
| YYYY.M | SELECT CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) + '.' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YYYY.M] | 2011.6 |
| YY.M | SELECT RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) + '.' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YY.M] | 11.6 |
| MM.DD | SELECT RIGHT(CONVERT(VARCHAR(8), SYSDATETIME(), 2), 5) AS [MM.DD] | 06.08 |
| DD.MM | SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 4) AS [DD.MM] | 08.06 |
| MMDDYYYY | SELECT REPLACE(CONVERT(VARCHAR(10), SYSDATETIME(), 101), '/', '') AS [MMDDYYYY] | 06082011 |
| MMDDYY | SELECT REPLACE(CONVERT(VARCHAR(8), SYSDATETIME(), 1), '/', '') AS [MMDDYY] | 060811 |
| DDMMYYYY | SELECT REPLACE(CONVERT(VARCHAR(10), SYSDATETIME(), 103), '/', '') AS [DDMMYYYY] | 08062011 |
| DDMMYY | SELECT REPLACE(CONVERT(VARCHAR(8), SYSDATETIME(), 3), '/', '') AS [DDMMYY] | 080611 |
| MMYYYY | SELECT RIGHT(REPLACE(CONVERT(VARCHAR(10), SYSDATETIME(), 103), '/', ''), 6) AS [MMYYYY] | 062011 |
| MMYY | SELECT RIGHT(REPLACE(CONVERT(VARCHAR(8), SYSDATETIME(), 3), '/', ''), 4) AS [MMYY] | 0611 |
| YYYYMM | SELECT CONVERT(VARCHAR(6), SYSDATETIME(), 112) AS [YYYYMM] | 201106 |
| YYMM | SELECT CONVERT(VARCHAR(4), SYSDATETIME(), 12) AS [YYMM] | 1106 |
| Month DD, YYYY | SELECT DATENAME(MONTH, SYSDATETIME())+ ' ' + RIGHT('0' + DATENAME(DAY, SYSDATETIME()), 2) + ', ' + DATENAME(YEAR, SYSDATETIME()) AS [Month DD, YYYY] | June 08, 2011 |
| Mon YYYY | SELECT LEFT(DATENAME(MONTH, SYSDATETIME()), 3) + ' ' + DATENAME(YEAR, SYSDATETIME()) AS [Mon YYYY] | Jun 2011 |
| Month YYYY | SELECT DATENAME(MONTH, SYSDATETIME()) + ' ' + DATENAME(YEAR, SYSDATETIME()) AS [Month YYYY] | June 2011 |
| DD Month | SELECT RIGHT('0' + DATENAME(DAY, SYSDATETIME()), 2) + ' ' + DATENAME(MONTH, SYSDATETIME()) AS [DD Month] | 08 June |

| | | |
|---|---|---|
| Month DD | SELECT DATENAME(MONTH, SYSDATETIME()) + ' ' + RIGHT('0' + DATENAME(DAY, SYSDATETIME()), 2) AS [Month DD] | June 08 |
| DD Month YY | SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + ' ' + DATENAME(MM, SYSDATETIME()) + ' ' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [DD Month YY] | 08 June 11 |
| DD Month YYYY | SELECT RIGHT('0' + DATENAME(DAY, SYSDATETIME()), 2) + ' ' + DATENAME(MONTH, SYSDATETIME())+ ' ' + DATENAME(YEAR, SYSDATETIME()) AS [DD Month YYYY] | 08 June 2011 |
| Mon-YY | SELECT REPLACE(RIGHT(CONVERT(VARCHAR(9), SYSDATETIME(), 6), 6), ' ', '-') AS [Mon-YY] | Jun-08 |
| Mon-YYYY | SELECT REPLACE(RIGHT(CONVERT(VARCHAR(11), SYSDATETIME(), 106), 8), ' ', '-') AS [Mon-YYYY] | Jun-2011 |
| DD-Mon-YY | SELECT REPLACE(CONVERT(VARCHAR(9), SYSDATETIME(), 6), ' ', '-') AS [DD-Mon-YY] | 08-Jun-11 |
| DD-Mon-YYYY | SELECT REPLACE(CONVERT(VARCHAR(11), SYSDATETIME(), 106), ' ', '-') AS [DD-Mon-YYYY] | 08-Jun-2011 |

# Chapter 10: Generating a range of dates

| Parameter | Details |
|---|---|
| @FromDate | The inclusive lower boundary of the generated date range. |
| @ToDate | The inclusive upper boundary of the generated date range. |

## Section 10.1: Generating Date Range With Recursive CTE

Using a Recursive CTE, you can generate an inclusive range of dates:

```
Declare  @FromDate    Date = '2014-04-21',
         @ToDate      Date = '2014-05-02'

;With DateCte (Date) As
(
    Select  @FromDate Union All
    Select  DateAdd(Day, 1, Date)
    From    DateCte
    Where   Date < @ToDate
)
Select  Date
From    DateCte
Option  (MaxRecursion 0)
```

The default `MaxRecursion` setting is 100. Generating more than 100 dates using this method will require the `Option (MaxRecursion N)` segment of the query, where `N` is the desired `MaxRecursion` setting. Setting this to `0` will remove the `MaxRecursion` limitation altogether.

## Section 10.2: Generating a Date Range With a Tally Table

Another way you can generate a range of dates is by utilizing a Tally Table to create the dates between the range:

```
Declare   @FromDate   Date = '2014-04-21',
          @ToDate     Date = '2014-05-02'

;With
   E1(N) As (Select 1 From (Values (1), (1), (1), (1), (1), (1), (1), (1), (1), (1)) DT(N)),
   E2(N) As (Select 1 From E1 A Cross Join E1 B),
   E4(N) As (Select 1 From E2 A Cross Join E2 B),
   E6(N) As (Select 1 From E4 A Cross Join E2 B),
   Tally(N) As
   (
        Select    Row_Number() Over (Order By (Select Null))
        From    E6
   )
Select   DateAdd(Day, N - 1, @FromDate) Date
From     Tally
Where    N <= DateDiff(Day, @FromDate, @ToDate) + 1
```

# Chapter 11: Database Snapshots

## Section 11.1: Create a database snapshot

A database snapshot is a read-only, static view of a SQL Server database (the source database). It is similar to backup, but it is available as any other database so client can query snapshot database.

```
CREATE DATABASE MyDatabase_morning -- name of the snapshot
ON (
     NAME=MyDatabase_data, -- logical name of the data file of the source database
     FILENAME='C:\SnapShots\MySnapshot_Data.ss' -- snapshot file;
)
AS SNAPSHOT OF MyDatabase; -- name of source database
```

You can also create snapshot of database with multiple files:

```
CREATE DATABASE MyMultiFileDBSnapshot ON
    (NAME=MyMultiFileDb_ft, FILENAME='C:\SnapShots\MyMultiFileDb_ft.ss'),
    (NAME=MyMultiFileDb_sys, FILENAME='C:\SnapShots\MyMultiFileDb_sys.ss'),
    (NAME=MyMultiFileDb_data, FILENAME='C:\SnapShots\MyMultiFileDb_data.ss'),
    (NAME=MyMultiFileDb_indx, FILENAME='C:\SnapShots\MyMultiFileDb_indx.ss')
AS SNAPSHOT OF MultiFileDb;
```

## Section 11.2: Restore a database snapshot

If data in a source database becomes damaged or some wrong data is written into database, in some cases, reverting the database to a database snapshot that predates the damage might be an appropriate alternative to restoring the database from a backup.

```
RESTORE DATABASE MYDATABASE FROM DATABASE_SNAPSHOT='MyDatabase_morning';
```

> **Warning:** This will *delete all changes* made to the source database since the snapshot was taken!

## Section 11.3: DELETE Snapshot

You can delete existing snapshots of database using DELETE DATABASE statement:

```
DROP DATABASE Mydatabase_morning
```

In this statement you should reference name of the database snapshot.

# Chapter 12: COALESCE

## Section 12.1: Using COALESCE to Build Comma-Delimited String

We can get a comma delimited string from multiple rows using coalesce as shown below.

Since table variable is used, we need to execute whole query once. So to make easy to understand, I have added BEGIN and END block.

```
BEGIN

    --Table variable declaration to store sample records
    DECLARE @Table TABLE (FirstName varchar(256), LastName varchar(256))

    --Inserting sample records into table variable @Table
    INSERT INTO @Table (FirstName, LastName)
    VALUES
    ('John','Smith'),
    ('Jane','Doe')

    --Creating variable to store result
    DECLARE @Names varchar(4000)

    --Used COLESCE function, so it will concatenate comma seperated FirstName into @Names varible
    SELECT @Names = COALESCE(@Names + ',', '') + FirstName
    FROM @Table

    --Now selecting actual result
    SELECT @Names
    END
```

## Section 12.2: Getting the first not null from a list of column values

```
SELECT COALESCE(NULL, NULL, 'TechOnTheNet.com', NULL, 'CheckYourMath.com');
RESULT: 'TechOnTheNet.com'

SELECT COALESCE(NULL, 'TechOnTheNet.com', 'CheckYourMath.com');
RESULT: 'TechOnTheNet.com'

SELECT COALESCE(NULL, NULL, 1, 2, 3, NULL, 4);
RESULT: 1
```

## Section 12.3: Coalesce basic Example

`COALESCE()` returns the first `NON NULL` value in a list of arguments. Suppose we had a table containing phone numbers, and cell phone numbers and wanted to return only one for each user. In order to only obtain one, we can get the first `NON NULL` value.

```
DECLARE @Table TABLE (UserID int, PhoneNumber varchar(12), CellNumber varchar(12))
INSERT INTO @Table (UserID, PhoneNumber, CellNumber)
VALUES
(1,'555-869-1123',NULL),
(2,'555-123-7415','555-846-7786'),
(3,NULL,'555-456-8521')
```

```sql
SELECT
    UserID,
    COALESCE(PhoneNumber, CellNumber)
FROM
    @Table
```

# Chapter 13: IF...ELSE

## Section 13.1: Single IF statement

Like most of the other programming languages, T-SQL also supports IF..ELSE statements.

For example in the example below `1 = 1` is the expression, which evaluates to True and the control enters the `BEGIN..END` block and the Print statement prints the string `'One is equal to One'`

```
IF ( 1 = 1)  --<-- Some Expression
 BEGIN
   PRINT 'One is equal to One'
 END
```

## Section 13.2: Multiple IF Statements

We can use multiple IF statement to check multiple expressions totally independent from each other.

In the example below, each `IF` statement's expression is evaluated and if it is true the code inside the `BEGIN...END` block is executed. In this particular example, the First and Third expressions are true and only those print statements will be executed.

```
IF (1 = 1)  --<-- Some Expression      --<-- This is true
BEGIN
    PRINT 'First IF is True'           --<-- this will be executed
END

IF (1 = 2)  --<-- Some Expression
BEGIN
    PRINT 'Second IF is True'
END

IF (3 = 3)  --<-- Some Expression        --<-- This true
BEGIN
    PRINT 'Thrid IF is True'             --<-- this will be executed
END
```

## Section 13.3: Single IF..ELSE statement

In a single `IF..ELSE` statement, if the expression evaluates to True in the `IF` statement the control enters the first `BEGIN..END` block and only the code inside that block gets executed , Else block is simply ignored.

On the other hand if the expression evaluates to `False` the `ELSE BEGIN..END` block gets executed and the control never enters the first `BEGIN..END` Block.

In the Example below the expression will evaluate to false and the Else block will be executed printing the string `'First expression was not true'`

```
IF ( 1 <> 1)  --<-- Some Expression
 BEGIN
     PRINT 'One is equal to One'
 END
ELSE
 BEGIN
     PRINT 'First expression was not true'
```

```
END
```

## Section 13.4: Multiple IF... ELSE with final ELSE Statements

If we have Multiple `IF...ELSE IF` statements but we also want also want to execute some piece of code if none of expressions are evaluated to True , then we can simple add a final `ELSE` block which only gets executed if none of the `IF` or `ELSE IF` expressions are evaluated to true.

In the example below none of the `IF` or `ELSE IF` expression are True hence only `ELSE` block is executed and prints `'No other expression is true'`

```
IF ( 1 = 1 + 1 )
    BEGIN
       PRINT 'First If Condition'
    END
ELSE IF (1 = 2)
    BEGIN
        PRINT 'Second If Else Block'
    END
ELSE IF (1 = 3)
    BEGIN
        PRINT 'Third If Else Block'
    END
ELSE
    BEGIN
        PRINT 'No other expression is true'  --<-- Only this statement will be printed
    END
```

## Section 13.5: Multiple IF...ELSE Statements

More often than not we need to check multiple expressions and take specific actions based on those expressions. This situation is handled using multiple `IF...ELSE IF` statements.

In this example all the expressions are evaluated from top to bottom. As soon as an expression evaluates to true, the code inside that block is executed. If no expression is evaluated to true, nothing gets executed.

```
IF (1 = 1 + 1)
BEGIN
    PRINT 'First If Condition'
END
ELSE IF (1 = 2)
BEGIN
    PRINT 'Second If Else Block'
END
ELSE IF (1 = 3)
BEGIN
    PRINT 'Third If Else Block'
END
ELSE IF (1 = 1)      --<-- This is True
BEGIN
    PRINT 'Last Else Block'  --<-- Only this statement will be printed
END
```

# Chapter 14: CASE Statement

## Section 14.1: Simple CASE statement

In a simple case statement, one value or variable is checked against multiple possible answers. The code below is an example of a simple case statement:

```
SELECT CASE DATEPART(WEEKDAY, GETDATE())
    WHEN 1 THEN 'Sunday'
    WHEN 2 THEN 'Monday'
    WHEN 3 THEN 'Tuesday'
    WHEN 4 THEN 'Wednesday'
    WHEN 5 THEN 'Thursday'
    WHEN 6 THEN 'Friday'
    WHEN 7 THEN 'Saturday'
END
```

## Section 14.2: Searched CASE statement

In a Searched Case statement, each option can test one or more values independently. The code below is an example of a searched case statement:

```
DECLARE @FirstName varchar(30) = 'John'
DECLARE @LastName varchar(30) = 'Smith'

SELECT CASE
    WHEN LEFT(@FirstName, 1) IN ('a','e','i','o','u')
        THEN 'First name starts with a vowel'
    WHEN LEFT(@LastName, 1) IN ('a','e','i','o','u')
        THEN 'Last name starts with a vowel'
    ELSE
        'Neither name starts with a vowel'
END
```

# Chapter 15: MERGE

Starting with SQL Server 2008, it is possible to perform insert, update, or delete operations in a single statement using the MERGE statement.

The MERGE statement allows you to join a data source with a target table or view, and then perform multiple actions against the target based on the results of that join.

## Section 15.1: MERGE to Insert / Update / Delete

```
MERGE INTO targetTable

USING sourceTable
ON (targetTable.PKID = sourceTable.PKID)

WHEN MATCHED AND (targetTable.PKID > 100) THEN
    DELETE

WHEN MATCHED AND (targetTable.PKID <= 100) THEN
    UPDATE SET
        targetTable.ColumnA = sourceTable.ColumnA,
        targetTable.ColumnB = sourceTable.ColumnB

WHEN NOT MATCHED THEN
    INSERT (ColumnA, ColumnB) VALUES (sourceTable.ColumnA, sourceTable.ColumnB);

WHEN NOT MATCHED BY SOURCE THEN
    DELETE
; --< Required
```

Description:

- `MERGE INTO targetTable` - table to be modified
- `USING sourceTable` - source of data (can be table or view or table valued function)
- `ON ...` - join condition between `targetTable` and `sourceTable`.
- `WHEN MATCHED` - actions to take when a match is found
    - `AND (targetTable.PKID > 100)` - additional condition(s) that must be satisfied in order for the action to be taken
- `THEN DELETE` - delete matched record from the `targetTable`
- `THEN UPDATE` - update columns of matched record specified by `SET ...`
- `WHEN NOT MATCHED` - actions to take when match is not found in **`targetTable`**
- `WHEN NOT MATCHED BY SOURCE` - actions to take when match is not found in **`sourceTable`**

Comments:

If a specific action is not needed then omit the condition e.g. removing `WHEN NOT MATCHED THEN INSERT` will prevent records from being inserted

Merge statement requires a terminating semicolon.

Restrictions:

- `WHEN MATCHED` does not allow `INSERT` action
- `UPDATE` action can update a row only once. This implies that the join condition must produce unique matches.

## Section 15.2: Merge Using CTE Source

```
WITH SourceTableCTE AS
(
    SELECT * FROM SourceTable
)
MERGE
 TargetTable AS target
USING SourceTableCTE AS source
ON (target.PKID = source.PKID)
WHEN MATCHED THEN
    UPDATE SET target.ColumnA = source.ColumnA
WHEN NOT MATCHED THEN
    INSERT (ColumnA) VALUES (Source.ColumnA);
```

## Section 15.3: Merge Example - Synchronize Source And Target Table

To Illustrate the MERGE Statement, consider the following two tables -

1. **dbo.Product** : This table contains information about the product that company is currently selling
2. **dbo.ProductNew**: This table contains information about the product that the company will sell in the future.

The following T-SQL will create and populate these two tables

```
IF OBJECT_id(N'dbo.Product',N'U') IS NOT NULL
DROP TABLE dbo.Product
GO

CREATE TABLE dbo.Product (
ProductID INT PRIMARY KEY,
ProductName NVARCHAR(64),
PRICE MONEY
)

IF OBJECT_id(N'dbo.ProductNew',N'U') IS NOT NULL
DROP TABLE dbo.ProductNew
GO

CREATE TABLE dbo.ProductNew (
ProductID INT PRIMARY KEY,
ProductName NVARCHAR(64),
PRICE MONEY
)

INSERT INTO dbo.Product VALUES(1,'IPod',300)
,(2,'IPhone',400)
,(3,'ChromeCast',100)
,(4,'raspberry pi',50)

INSERT INTO dbo.ProductNew VALUES(1,'Asus Notebook',300)
,(2,'Hp Notebook',400)
,(3,'Dell Notebook',100)
,(4,'raspberry pi',50)
```

Now, Suppose we want to synchoronize the dbo.Product Target Table with the dbo.ProductNew table. Here is the criterion for this task:

1. Product that exist in both the dbo.ProductNew source table and the dbo.Product target table are updated in the dbo.Product target table with new new Products.
2. Any product in the dbo.ProductNew source table that do not exist in the dob.Product target table are inserted into the dbo.Product target table.
3. Any Product in the dbo.Product target table that do not exist in the dbo.ProductNew source table must be deleted from the dbo.Product target table. Here is the MERGE statement to perform this task.

```
MERGE dbo.Product AS SourceTbl
USING dbo.ProductNew AS TargetTbl ON (SourceTbl.ProductID = TargetTbl.ProductID)
WHEN MATCHED
            AND SourceTbl.ProductName <> TargetTbl.ProductName
            OR SourceTbl.Price <> TargetTbl.Price
    THEN UPDATE SET SourceTbl.ProductName = TargetTbl.ProductName,
                SourceTbl.Price = TargetTbl.Price
WHEN NOT MATCHED
    THEN INSERT (ProductID, ProductName, Price)
         VALUES (TargetTbl.ProductID, TargetTbl.ProductName, TargetTbl.Price)
WHEN NOT MATCHED BY SOURCE
    THEN DELETE
OUTPUT $action, INSERTED.*, DELETED.*;
```

Note:Semicolon must be present in the end of MERGE statement.

| | $action | ProductID | ProductName | PRICE | ProductID | ProductName | PRICE |
|---|---|---|---|---|---|---|---|
| 1 | UPDATE | 1 | Asus Notebook | 300.00 | 1 | IPod | 300.00 |
| 2 | UPDATE | 2 | Hp Notebook | 400.00 | 2 | IPhone | 400.00 |
| 3 | UPDATE | 3 | Dell Notebook | 100.00 | 3 | ChromeCast | 100.00 |

## Section 15.4: MERGE using Derived Source Table

```
MERGE INTO TargetTable  AS Target
USING (VALUES (1,'Value1'), (2, 'Value2'), (3,'Value3'))
       AS Source (PKID, ColumnA)
ON Target.PKID = Source.PKID
WHEN MATCHED THEN
    UPDATE SET target.ColumnA= source.ColumnA
WHEN NOT MATCHED THEN
    INSERT (PKID, ColumnA) VALUES (Source.PKID, Source.ColumnA);
```

## Section 15.5: Merge using EXCEPT

Use EXCEPT to prevent updates to unchanged records

```
MERGE TargetTable targ
USING SourceTable AS src
    ON src.id = targ.id
WHEN MATCHED
    AND EXISTS (
        SELECT src.field
        EXCEPT
        SELECT targ.field
        )
    THEN
        UPDATE
        SET field = src.field
WHEN NOT MATCHED BY TARGET
```

```
    THEN
        INSERT (
            id
            ,field
            )
        VALUES (
            src.id
            ,src.field
            )
WHEN NOT MATCHED BY SOURCE
    THEN
        DELETE;
```

# Chapter 16: INSERT INTO

The INSERT INTO statement is used to insert new records in a table.

## Section 16.1: INSERT multiple rows of data

To insert multiple rows of data in SQL Server 2008 or later:

```
INSERT INTO USERS VALUES
(2, 'Michael', 'Blythe'),
(3, 'Linda', 'Mitchell'),
(4, 'Jillian', 'Carson'),
(5, 'Garrett', 'Vargas');
```

To insert multiple rows of data in earlier versions of SQL Server, use "UNION ALL" like so:

```
INSERT INTO USERS (FIRST_NAME, LAST_NAME)
SELECT 'James', 'Bond' UNION ALL
SELECT 'Miss', 'Moneypenny' UNION ALL
SELECT 'Raoul', 'Silva'
```

Note, the "INTO" keyword is optional in INSERT queries. Another warning is that SQL server only supports 1000 rows in one INSERT so you have to split them in batches.

## Section 16.2: Use OUTPUT to get the new Id

When INSERTing, you can use `OUTPUT INSERTED.ColumnName` to get values from the newly inserted row, for example the newly generated Id - useful if you have an `IDENTITY` column or any sort of default or calculated value.

When programatically calling this (e.g., from ADO.net) you would treat it as a normal query and read the values as if you would've made a **`SELECT`**-statement.

```
-- CREATE TABLE OutputTest ([Id] INT NOT NULL PRIMARY KEY IDENTITY, [Name] NVARCHAR(50))

INSERT INTO OutputTest ([Name])
OUTPUT INSERTED.[Id]
VALUES ('Testing')
```

If the ID of the recently added row is required inside the same set of query or stored procedure.

```
-- CREATE a table variable having column with the same datatype of the ID

DECLARE @LastId TABLE ( id int);

INSERT INTO OutputTest ([Name])
OUTPUT INSERTED.[Id] INTO @LastId
VALUES ('Testing')

SELECT id FROM @LastId

-- We can set the value in a variable and use later in procedure

DECLARE @LatestId int = (SELECT id FROM @LastId)
```

## Section 16.3: INSERT from SELECT Query Results

To insert data retrieved from SQL query (single or multiple rows)

```
INSERT INTO Table_name (FirstName, LastName, Position)
SELECT FirstName, LastName, 'student' FROM Another_table_name
```

Note, 'student' in SELECT is a string constant that will be inserted in each row.

If required, you can select and insert data from/into the same table

## Section 16.4: INSERT a single row of data

A single row of data can be inserted in two ways:

```
INSERT INTO USERS(Id, FirstName, LastName)
VALUES (1, 'Mike', 'Jones');
```

Or

```
INSERT INTO USERS
VALUES (1, 'Mike', 'Jones');
```

Note that the second insert statement only allows the values in exactly the same order as the table columns whereas in the first insert, the order of the values can be changed like:

```
INSERT INTO USERS(FirstName, LastName, Id)
VALUES ('Mike', 'Jones', 1);
```

## Section 16.5: INSERT on specific columns

To do an insert on specific columns (as opposed to all of them) you must specify the columns you want to update.

```
INSERT INTO USERS (FIRST_NAME, LAST_NAME)
VALUES ('Stephen', 'Jiang');
```

This will only work if the columns that you did not list are nullable, identity, timestamp data type or computed columns; or columns that have a default value constraint. Therefore, if any of them are non-nullable, non-identity, non-timestamp, non-computed, non-default valued columns...then attempting this kind of insert will trigger an error message telling you that you have to provide a value for the applicable field(s).

## Section 16.6: INSERT Hello World INTO table

```
CREATE TABLE MyTableName
(
    Id INT,
    MyColumnName NVARCHAR(1000)
)
GO

INSERT INTO MyTableName (Id, MyColumnName)
VALUES (1, N'Hello World!')
GO
```

# Chapter 17: CREATE VIEW

## Section 17.1: CREATE Indexed VIEW

To create a view with an index, the view must be created using the `WITH SCHEMABINDING` keywords:

```
CREATE VIEW view_EmployeeInfo
WITH SCHEMABINDING
AS
    SELECT EmployeeID,
        FirstName,
        LastName,
        HireDate
    FROM [dbo].Employee
GO
```

Any clustered or non-clustered indexes can be now be created:

```
CREATE UNIQUE CLUSTERED INDEX IX_view_EmployeeInfo
ON view_EmployeeInfo
(
     EmployeeID ASC
)
```

**There Are some limitations to indexed Views:**

- The view definition can reference one or more tables in the same database.
- Once the unique clustered index is created, additional nonclustered indexes can be created against the view.
- You can update the data in the underlying tables – including inserts, updates, deletes, and even truncates.
- You can't modify the underlying tables and columns. The view is created with the WITH SCHEMABINDING option.
- It can't contain COUNT, MIN, MAX, TOP, outer joins, or a few other keywords or elements.

For more information about creating indexed Views you can read this MSDN article

## Section 17.2: CREATE VIEW

```
CREATE VIEW view_EmployeeInfo
AS
    SELECT EmployeeID,
        FirstName,
        LastName,
        HireDate
    FROM Employee
GO
```

Rows from views can be selected much like tables:

```
SELECT FirstName
FROM view_EmployeeInfo
```

You may also create a view with a calculated column. We can modify the view above as follows by adding a

calculated column:

```
CREATE VIEW view_EmployeeReport
AS
    SELECT EmployeeID,
        FirstName,
        LastName,
        Coalesce(FirstName,'') + ' ' + Coalesce(LastName,'') as FullName,
        HireDate
    FROM Employee
GO
```

This view adds an additional column that will appear when you `SELECT` rows from it. The values in this additional column will be dependent on the fields `FirstName` and `LastName` in the table `Employee` and will automatically update behind-the-scenes when those fields are updated.

## Section 17.3: CREATE VIEW With Encryption

```
CREATE VIEW view_EmployeeInfo
WITH ENCRYPTION
AS
SELECT EmployeeID, FirstName, LastName, HireDate
FROM Employee
GO
```

## Section 17.4: CREATE VIEW With INNER JOIN

```
CREATE VIEW view_PersonEmployee
AS
    SELECT P.LastName,
        P.FirstName,
        E.JobTitle
    FROM Employee AS E
    INNER JOIN Person AS P
        ON P.BusinessEntityID = E.BusinessEntityID
GO
```

Views can use joins to select data from numerous sources like tables, table functions, or even other views. This example uses the FirstName and LastName columns from the Person table and the JobTitle column from the Employee table.

This view can now be used to see all corresponding rows for Managers in the database:

```
SELECT *
FROM view_PersonEmployee
WHERE JobTitle LIKE '%Manager%'
```

## Section 17.5: Grouped VIEWs

A grouped VIEW is based on a query with a GROUP BY clause. Since each of the groups may have more than one row in the base from which it was built, these are necessarily read-only VIEWs. Such VIEWs usually have one or more aggregate functions and they are used for reporting purposes. They are also handy for working around weaknesses in SQL. Consider a VIEW that shows the largest sale in each state. The query is straightforward:

https://www.simple-talk.com/sql/t-sql-programming/sql-view-beyond-the-basics/

```
CREATE VIEW BigSales (state_code, sales_amt_total)
AS SELECT state_code, MAX(sales_amt)
     FROM Sales
    GROUP BY state_code;
```

## Section 17.6: UNION-ed VIEWs

VIEWs based on a UNION or UNION ALL operation are read-only because there is no single way to map a change onto just one row in one of the base tables. The UNION operator will remove duplicate rows from the results. Both the UNION and UNION ALL operators hide which table the rows came from. Such VIEWs must use a , because the columns in a UNION [ALL] have no names of their own. In theory, a UNION of two disjoint tables, neither of which has duplicate rows in itself should be updatable.

https://www.simple-talk.com/sql/t-sql-programming/sql-view-beyond-the-basics/

```
CREATE VIEW DepTally2 (emp_nbr, dependent_cnt)
AS (SELECT emp_nbr, COUNT(*)
      FROM Dependents
     GROUP BY emp_nbr)
   UNION
   (SELECT emp_nbr, 0
      FROM Personnel AS P2
     WHERE NOT EXISTS
          (SELECT *
             FROM Dependents AS D2
            WHERE D2.emp_nbr = P2.emp_nbr));
```

# Chapter 18: Views

## Section 18.1: Create a view with schema binding

If a view is created WITH SCHEMABINDING, the underlying table(s) can't be dropped or modified in such a way that they would break the view. For example, a table column referenced in a view can't be removed.

```
CREATE VIEW dbo.PersonsView
WITH SCHEMABINDING
AS
SELECT
    name,
    address
FROM dbo.PERSONS  -- database schema must be specified when WITH SCHEMABINDING is present
```

Views *without* schema binding can break if their underlying table(s) change or get dropped. Querying a broken view results in an error message. sp_refreshview can be used to ensure existing views without schema binding aren't broken.

## Section 18.2: Create a view

```
CREATE VIEW dbo.PersonsView
AS
SELECT
    name,
    address
FROM persons;
```

## Section 18.3: Create or replace view

This query will drop the view - if it already exists - and create a new one.

```
IF OBJECT_ID('dbo.PersonsView', 'V') IS NOT NULL
    DROP VIEW dbo.PersonsView
GO

CREATE VIEW dbo.PersonsView
AS
SELECT
    name,
    address
FROM persons;
```

# Chapter 19: UNION

## Section 19.1: Union and union all

**Union** operation combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union and will ignore any duplicates that exist. **Union all** also does the same thing but include even the duplicate values. The concept of union operation will be clear from the example below. Few things to consider while using union are:

1.The number and the order of the columns must be the same in all queries.

2.The data types must be compatible.

Example:

We have three tables : Marksheet1, Marksheet2 and Marksheet3. Marksheet3 is the duplicate table of Marksheet2 which contains same values as that of Marksheet2.

**Table1**: Marksheet1

| SubjectCode | SubjectName | MarksObtained |
|---|---|---|
| 101 | Physics | 87 |
| 102 | Chemistry | 75 |
| 103 | Maths | 85 |
| 104 | English | 89 |
| 105 | Computer | 95 |

**Table2**: Marksheet2

| CourseCode | CourseName | MarksObtained |
|---|---|---|
| 201 | PhysicsII | 82 |
| 202 | ChemistryII | 86 |
| 203 | MathsII | 95 |
| 204 | EnglishII | 70 |
| 205 | ComputerII | 86 |

**Table3**: Marksheet3

| SubjectCode | SubjectName | MarksObtained |
|---|---|---|
| 201 | PhysicsII | 82 |
| 202 | ChemistryII | 86 |
| 203 | MathsII | 95 |
| 204 | EnglishII | 70 |
| 205 | ComputerII | 86 |

**Union on tables Marksheet1 and Marksheet2**

```
SELECT SubjectCode, SubjectName, MarksObtained
FROM Marksheet1
UNION
SELECT CourseCode, CourseName, MarksObtained
FROM Marksheet2
```

**Note:** The output for union of the three tables will also be same as union on Marksheet1 and Marksheet2 because union operation does not take duplicate values.

```
SELECT SubjectCode, SubjectName, MarksObtained
FROM Marksheet1
UNION
SELECT CourseCode, CourseName, MarksObtained
FROM Marksheet2
UNION
SELECT SubjectCode, SubjectName, MarksObtained
FROM Marksheet3
```

**OUTPUT**

| | SubjectCode | SubjectName | MarksObtained |
|---|---|---|---|
| 1 | 101 | Physics | 87 |
| 2 | 102 | Chemistry | 75 |
| 3 | 103 | Maths | 85 |
| 4 | 104 | English | 89 |
| 5 | 105 | Computer | 95 |
| 6 | 201 | PhysicsII | 82 |
| 7 | 202 | ChemistryII | 86 |
| 8 | 203 | MathsII | 95 |
| 9 | 204 | EnglishII | 70 |
| 10 | 205 | ComputerII | 86 |

**Union All**

```
SELECT SubjectCode, SubjectName, MarksObtained
FROM Marksheet1
UNION ALL
SELECT CourseCode, CourseName, MarksObtained
FROM Marksheet2
UNION ALL
SELECT SubjectCode, SubjectName, MarksObtained
FROM Marksheet3
```

**OUTPUT**

| | SubjectCode | SubjectName | MarksObtained |
|---|---|---|---|
| 1 | 101 | Physics | 87 |
| 2 | 102 | Chemistry | 75 |
| 3 | 103 | Maths | 85 |
| 4 | 104 | English | 89 |
| 5 | 105 | Computer | 95 |
| 6 | 201 | PhysicsII | 82 |
| 7 | 202 | ChemistryII | 86 |
| 8 | 203 | MathsII | 95 |
| 9 | 204 | EnglishII | 70 |
| 10 | 205 | ComputerII | 86 |
| 11 | 201 | PhysicsII | 82 |
| 12 | 202 | ChemistryII | 86 |
| 13 | 203 | MathsII | 95 |
| 14 | 204 | EnglishII | 70 |
| 15 | 205 | ComputerII | 86 |

You will notice here that the duplicate values from Marksheet3 are also displayed using union all.

# Chapter 20: TRY/CATCH

## Section 20.1: Transaction in a TRY/CATCH

This will rollback both inserts due to an invalid datetime:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION -- First Rollback and then throw.
    THROW
END CATCH
```

This will commit both inserts:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

## Section 20.2: Raising errors in try-catch block

RAISERROR function will generate error in the TRY CATCH block:

```
DECLARE @msg nvarchar(50) = 'Here is a problem!'
BEGIN TRY
    print 'First statement';
    RAISERROR(@msg, 11, 1);
    print 'Second statement';
END TRY
BEGIN CATCH
    print 'Error: ' + ERROR_MESSAGE();
END CATCH
```

RAISERROR with second parameter greater than 10 (11 in this example) will stop execution in TRY BLOCK and raise an error that will be handled in CATCH block. You can access error message using ERROR_MESSAGE() function. Output of this sample is:

```
First statement
Error: Here is a problem!
```

## Section 20.3: Raising info messages in try catch block

RAISERROR with severity (second parameter) less or equal to 10 will not throw exception.

```
BEGIN TRY
    print 'First statement';
    RAISERROR( 'Here is a problem!', 10, 15);
    print 'Second statement';
END TRY
BEGIN CATCH
    print 'Error: ' + ERROR_MESSAGE();
END CATCH
```

After RAISERROR statement, third statement will be executed and CATCH block will not be invoked. Result of execution is:

```
First statement
Here is a problem!
Second statement
```

## Section 20.4: Re-throwing exception generated by RAISERROR

You can re-throw error that you catch in CATCH block using TRHOW statement:

```
DECLARE @msg nvarchar(50) = 'Here is a problem! Area: ''%s'' Line:''%i'''
BEGIN TRY
    print 'First statement';
    RAISERROR(@msg, 11, 1, 'TRY BLOCK', 2);
    print 'Second statement';
END TRY
BEGIN CATCH
    print 'Error: ' + ERROR_MESSAGE();
    THROW;
END CATCH
```

Note that in this case we are raising error with formatted arguments (fourth and fifth parameter). This might be useful if you want to add more info in message. Result of execution is:

```
First statement
Error: Here is a problem! Area: 'TRY BLOCK' Line:'2'
Msg 50000, Level 11, State 1, Line 26
Here is a problem! Area: 'TRY BLOCK' Line:'2'
```

## Section 20.5: Throwing exception in TRY/CATCH blocks

You can throw exception in try catch block:

```
DECLARE @msg nvarchar(50) = 'Here is a problem!'
BEGIN TRY
    print 'First statement';
    THROW 51000, @msg, 15;
    print 'Second statement';
END TRY
BEGIN CATCH
    print 'Error: ' + ERROR_MESSAGE();
    THROW;
```

```
END CATCH
```

Exception with be handled in CATCH block and then re-thrown using THROW without parameters.

```
First statement
Error: Here is a problem!
Msg 51000, Level 16, State 15, Line 39
Here is a problem!
```

THROW is similar to RAISERROR with following differences:

- Recommendation is that new applications should use THROW instead of RASIERROR.
- THROW can use any number as first argument (error number), RAISERROR can use only ids in sys.messages view
- THROW has severity 16 (cannot be changed)
- THROW cannot format arguments like RAISERROR. Use FORMATMESSAGE function as an argument of RAISERROR if you need this feature.

# Chapter 21: WHILE loop

## Section 21.1: Using While loop

The `WHILE` loop can be used as an alternative to `CURSORS`. The following example will print numbers from 0 to 99.

```
DECLARE @i int = 0;
WHILE(@i < 100)
BEGIN
   PRINT @i;
   SET @i = @i+1
END
```

## Section 21.2: While loop with min aggregate function usage

```
DECLARE @ID AS INT;

SET @ID = (SELECT MIN(ID) from TABLE);

WHILE @ID IS NOT NULL
BEGIN
    PRINT @ID;
    SET @ID = (SELECT MIN(ID) FROM TABLE WHERE ID > @ID);
END
```

# Chapter 22: OVER Clause

| Parameter | Details |
| --- | --- |
| PARTITION BY | The field(s) that follows PARTITION BY is the one that the 'grouping' will be based on |

## Section 22.1: Cumulative Sum

Using the Item Sales Table, we will try to find out how the sales of our items are increasing through dates. To do so we will calculate the *Cumulative Sum* of total sales per Item order by the sale date.

```
SELECT item_id, sale_Date
       SUM(quantity * price) OVER(PARTITION BY item_id ORDER BY sale_Date ROWS BETWEEN UNBOUNDED
PRECEDING) AS SalesTotal
  FROM SalesTable
```

## Section 22.2: Using Aggregation functions with OVER

Using the Cars Table, we will calculate the total, max, min and average amount of money each costumer spent and haw many times (COUNT) she brought a car for repairing.

Id CustomerId MechanicId Model Status Total Cost

```
SELECT CustomerId,
       SUM(TotalCost) OVER(PARTITION BY CustomerId) AS Total,
       AVG(TotalCost) OVER(PARTITION BY CustomerId) AS Avg,
       COUNT(TotalCost) OVER(PARTITION BY CustomerId) AS COUNT,
       MIN(TotalCost) OVER(PARTITION BY CustomerId) AS MIN,
       MAX(TotalCost) OVER(PARTITION BY CustomerId) AS MAX
  FROM CarsTable
 WHERE STATUS = 'READY'
```

Beware that using OVER in this fashion will not aggregate the rows returned. The above query will return the following:

| CustomerId | Total | Avg | Count | Min | Max |
| --- | --- | --- | --- | --- | --- |
| 1 | 430 | 215 | 2 | 200 | 230 |
| 1 | 430 | 215 | 2 | 200 | 230 |

The duplicated row(s) may not be that useful for reporting purposes.

If you wish to simply aggregate data, you will be better off using the GROUP BY clause along with the appropriate aggregate functions Eg:

```
SELECT CustomerId,
       SUM(TotalCost) AS Total,
       AVG(TotalCost) AS Avg,
       COUNT(TotalCost) AS COUNT,
       MIN(TotalCost) AS MIN,
       MAX(TotalCost)  AS MAX
  FROM CarsTable
 WHERE STATUS = 'READY'
GROUP BY CustomerId
```

# Section 22.3: Dividing Data into equally-partitioned buckets using NTILE

Let's say that you have exam scores for several exams and you want to divide them into quartiles per exam.

```
-- Setup data:
declare @values table(Id int identity(1,1) primary key, [Value] float, ExamId int)
insert into @values ([Value], ExamId) values
(65, 1), (40, 1), (99, 1), (100, 1), (90, 1), -- Exam 1 Scores
(91, 2), (88, 2), (83, 2), (91, 2), (78, 2), (67, 2), (77, 2) -- Exam 2 Scores

-- Separate into four buckets per exam:
select ExamId,
       ntile(4) over (partition by ExamId order by [Value] desc) as Quartile,
       Value, Id
from @values
order by ExamId, Quartile
```

| | ExamId | Quartile | Value | Id |
|---|---|---|---|---|
| 1 | 1 | 1 | 100 | 4 |
| 2 | 1 | 1 | 99 | 3 |
| 3 | 1 | 2 | 90 | 5 |
| 4 | 1 | 3 | 65 | 1 |
| 5 | 1 | 4 | 40 | 2 |
| 6 | 2 | 1 | 91 | 9 |
| 7 | 2 | 1 | 91 | 6 |
| 8 | 2 | 2 | 88 | 7 |
| 9 | 2 | 2 | 83 | 8 |
| 10 | 2 | 3 | 78 | 10 |
| 11 | 2 | 3 | 77 | 12 |
| 12 | 2 | 4 | 67 | 11 |

`ntile` works great when you really need a set number of buckets and each filled to approximately the same level. Notice that it would be trivial to separate these scores into percentiles by simply using `ntile(100)`.

# Section 22.4: Using Aggregation funtions to find the most recent records

Using the Library Database, we try to find the last book added to the database for each author. For this simple example we assume an always incrementing Id for each record added.

```
SELECT MostRecentBook.Name, MostRecentBook.Title
FROM ( SELECT Authors.Name,
              Books.Title,
              RANK() OVER (PARTITION BY Authors.Id ORDER BY Books.Id DESC) AS NewestRank
       FROM Authors
       JOIN Books ON Books.AuthorId = Authors.Id
     ) MostRecentBook
WHERE MostRecentBook.NewestRank = 1
```

Instead of RANK, two other functions can be used to order. In the previous example the result will be the same, but they give different results when the ordering gives multiple rows for each rank.

- `RANK()`: duplicates get the same rank, the next rank takes the number of duplicates in the previous rank into account

- `DENSE_RANK()`: duplicates get the same rank, the next rank is always one higher than the previous
- `ROW_NUMBER()`: will give each row a unique 'rank', 'ranking' the duplicates randomly

For example, if the table had a non-unique column CreationDate and the ordering was done based on that, the following query:

```
SELECT Authors.Name,
       Books.Title,
       Books.CreationDate,
       RANK() OVER (PARTITION BY Authors.Id ORDER BY Books.CreationDate DESC) AS RANK,
       DENSE_RANK() OVER (PARTITION BY Authors.Id ORDER BY Books.CreationDate DESC) AS DENSE_RANK,
       ROW_NUMBER() OVER (PARTITION BY Authors.Id ORDER BY Books.CreationDate DESC) AS ROW_NUMBER,
FROM Authors
JOIN Books ON Books.AuthorId = Authors.Id
```

Could result in:

| Author | Title | CreationDate | RANK | DENSE_RANK | ROW_NUMBER |
|---|---|---|---|---|---|
| Author 1 | Book 1 | 22/07/2016 | 1 | 1 | 1 |
| Author 1 | Book 2 | 22/07/2016 | 1 | 1 | 2 |
| Author 1 | Book 3 | 21/07/2016 | 3 | 2 | 3 |
| Author 1 | Book 4 | 21/07/2016 | 3 | 2 | 4 |
| Author 1 | Book 5 | 21/07/2016 | 3 | 2 | 5 |
| Author 1 | Book 6 | 04/07/2016 | 6 | 3 | 6 |
| Author 2 | Book 7 | 04/07/2016 | 1 | 1 | 1 |

# Chapter 23: GROUP BY

## Section 23.1: Simple Grouping

Orders Table

| CustomerId | ProductId | Quantity | Price |
|---|---|---|---|
| 1 | 2 | 5 | 100 |
| 1 | 3 | 2 | 200 |
| 1 | 4 | 1 | 500 |
| 2 | 1 | 4 | 50 |
| 3 | 5 | 6 | 700 |

When grouping by a specific column, only unique values of this column are returned.

```
SELECT customerId
FROM orders
GROUP BY customerId;
```

Return value:

| customerId |
|---|
| 1 |
| 2 |
| 3 |

Aggregate functions like `count()` apply to each group and not to the complete table:

```
SELECT customerId,
       COUNT(productId) AS numberOfProducts,
       SUM(price) AS totalPrice
FROM orders
GROUP BY customerId;
```

Return value:

| customerId | numberOfProducts | totalPrice |
|---|---|---|
| 1 | 3 | 800 |
| 2 | 1 | 50 |
| 3 | 1 | 700 |

## Section 23.2: GROUP BY multiple columns

One might want to GROUP BY more than one column

```
declare @temp table(age int, name varchar(15))

insert into @temp
select 18, 'matt' union all
select 21, 'matt' union all
select 21, 'matt' union all
select 18, 'luke' union all
select 18, 'luke' union all
select 21, 'luke' union all
select 18, 'luke' union all
select 21, 'luke'
```

```
SELECT Age, Name, count(1) count
FROM @temp
GROUP BY Age, Name
```

will group by both age and name and will produce:

| Age | Name | count |
|---|---|---|
| 18 | luke | 3 |
| 21 | luke | 2 |
| 18 | matt | 1 |
| 21 | matt | 2 |

## Section 23.3: GROUP BY with ROLLUP and CUBE

The ROLLUP operator is useful in generating reports that contain subtotals and totals.

- CUBE generates a result set that shows aggregates for all combinations of values in the selected columns.
- ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

| Item | Color | Quantity |
|---|---|---|
| Table | Blue | 124 |
| Table | Red | 223 |
| Chair | Blue | 101 |
| Chair | Red | 210 |

```
SELECT CASE WHEN (GROUPING(Item) = 1) THEN 'ALL'
            ELSE ISNULL(Item, 'UNKNOWN')
       END AS Item,
       CASE WHEN (GROUPING(Color) = 1) THEN 'ALL'
            ELSE ISNULL(Color, 'UNKNOWN')
       END AS Color,
       SUM(Quantity) AS QtySum
FROM Inventory
GROUP BY Item, Color WITH ROLLUP

Item                 Color                QtySum
-------------------- -------------------- --------------------------
Chair                Blue                 101.00
Chair                Red                  210.00
Chair                ALL                  311.00
TABLE                Blue                 124.00
TABLE                Red                  223.00
TABLE                ALL                  347.00
ALL                  ALL                  658.00
```

(7 row(s) affected)

If the ROLLUP keyword in the query is changed to CUBE, the CUBE result set is the same, except these two additional rows are returned at the end:

```
ALL                  Blue                 225.00
ALL                  Red                  433.00
```

https://technet.microsoft.com/en-us/library/ms189305(v=sql.90).aspx

# Section 23.4: Group by with multiple tables, multiple columns

Group by is often used with join statement. Let's assume we have two tables. The first one is the table of students:

| Id | Full Name | Age |
|---|---|---|
| 1 | Matt Jones | 20 |
| 2 | Frank Blue | 21 |
| 3 | Anthony Angel | 18 |

Second table is the table of subject each student can take:

| Subject_Id | Subject |
|---|---|
| 1 | Maths |
| 2 | P.E. |
| 3 | Physics |

And because one student can attend many subjects and one subject can be attended by many students (therefore N:N relationship) we need to have third "bounding" table. Let's call the table Students_subjects:

| Subject_Id | Student_Id |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 2 | 1 |
| 3 | 2 |
| 1 | 3 |
| 1 | 1 |

Now lets say we want to know the number of subjects each student is attending. Here the standalone `GROUP BY` statement is not sufficient as the information is not available through single table. Therefore we need to use `GROUP BY` with the `JOIN` statement:

```
SELECT Students.FullName, COUNT(Subject Id) AS SubjectNumber FROM Students_Subjects
LEFT JOIN Students
ON Students_Subjects.Student_id = Students.Id
GROUP BY Students.FullName
```

The result of the given query is as follows:

| FullName | SubjectNumber |
|---|---|
| Matt Jones | 3 |
| Frank Blue | 2 |
| Anthony Angel | 1 |

For an even more complex example of GROUP BY usage, let's say student might be able to assign the same subject to his name more than once (as shown in table Students_Subjects). In this scenario we might be able to count number of times each subject was assigned to a student by GROUPing by more than one column:

```
SELECT Students.FullName, Subjects.Subject,
COUNT(Students_subjects.Subject_id) AS NumberOfOrders
FROM ((Students_Subjects
INNER JOIN Students
ON Students_Subjcets.Student_id=Students.Id)
INNER JOIN Subjects
ON Students_Subjects.Subject_id=Subjects.Subject_id)
GROUP BY Fullname,Subject
```

This query gives the following result:

| FullName | Subject | SubjectNumber |
|---|---|---|
| Matt Jones | Maths | 2 |
| Matt Jones | P.E | 1 |
| Frank Blue | P.E | 1 |
| Frank Blue | Physics | 1 |
| Anthony Angel | Maths | 1 |

# Section 23.5: HAVING

Because the `WHERE` clause is evaluated before `GROUP BY`, you cannot use `WHERE` to pare down results of the grouping (typically an aggregate function, such as `COUNT(*)`). To meet this need, the `HAVING` clause can be used.

For example, using the following data:

```
DECLARE @orders TABLE(OrderID INT, Name NVARCHAR(100))

INSERT INTO @orders VALUES
( 1, 'Matt' ),
( 2, 'John' ),
( 3, 'Matt' ),
( 4, 'Luke' ),
( 5, 'John' ),
( 6, 'Luke' ),
( 7, 'John' ),
( 8, 'John' ),
( 9, 'Luke' ),
( 10, 'John' ),
( 11, 'Luke' )
```

If we want to get the number of orders each person has placed, we would use

```
SELECT Name, COUNT(*) AS 'Orders'
FROM @orders
GROUP BY Name
```

and get

| Name | Orders |
|---|---|
| Matt | 2 |
| John | 5 |
| Luke | 4 |

However, if we want to limit this to individuals who have placed more than two orders, we can add a `HAVING` clause.

```
SELECT Name, COUNT(*) AS 'Orders'
FROM @orders
GROUP BY Name
HAVING COUNT(*) > 2
```

will yield

| Name | Orders |
|---|---|
| John | 5 |
| Luke | 4 |

Note that, much like `GROUP BY`, the columns put in `HAVING` must exactly match their counterparts in the `SELECT` statement. If in the above example we had instead said

```
SELECT Name, COUNT(DISTINCT OrderID)
```

our `HAVING` clause would have to say

```
HAVING COUNT(DISTINCT OrderID) > 2
```

# Chapter 24: ORDER BY

## Section 24.1: Simple ORDER BY clause

Using the Employees Table, below is an example to return the Id, FName and LName columns in (ascending) LName order:

```
SELECT Id, FName, LName FROM Employees
ORDER BY LName
```

Returns:

| Id | FName | LName |
|---|---|---|
| 2 | John | Johnson |
| 1 | James | Smith |
| 4 | Johnathon | Smith |
| 3 | Michael | Williams |

To sort in descending order add the DESC keyword after the field parameter, e.g. the same query in LName descending order is:

```
SELECT Id, FName, LName FROM Employees
ORDER BY LName DESC
```

## Section 24.2: ORDER BY multiple fields

Multiple fields can be specified for the `ORDER BY` clause, in either ASCending or DESCending order.

For example, using the http://stackoverflow.com/documentation/sql/280/example-databases/1207/item-sales-table#t=201607211314066434211 table, we can return a query that sorts by SaleDate in ascending order, and Quantity in descending order.

```
SELECT ItemId, SaleDate, Quantity
FROM [Item Sales]
ORDER BY SaleDate ASC, Quantity DESC
```

Note that the `ASC` keyword is optional, and results are sorted in ascending order of a given field by default.

## Section 24.3: Custom Ordering

If you want to order by a column using something other than alphabetical/numeric ordering, you can use `case` to specify the order you want.

`order by Group` returns:

| Group | Count |
|---|---|
| Not Retired | 6 |
| Retired | 4 |
| Total | 10 |

`order by case group when 'Total' then 1 when 'Retired' then 2 else 3 end` returns:

| Group | Count |
|---|---|
| Total | 10 |

Retired 4
Not Retired 6

## Section 24.4: ORDER BY with complex logic

If we want to order the data differently for per group, we can add a `CASE` syntax to the `ORDER BY`. In this example, we want to order employees from Department 1 by last name and employees from Department 2 by salary.

| Id | FName | LName | PhoneNumber | ManagerId | DepartmentId | Salary | HireDate |
|---|---|---|---|---|---|---|---|
| 1 | James | Smith | 1234567890 | NULL | 1 | 1000 | 01-01-2002 |
| 2 | John | Johnson | 2468101214 | 1 | 1 | 400 | 23-03-2005 |
| 3 | Michael | Williams | 1357911131 | 1 | 2 | 600 | 12-05-2009 |
| 4 | Johnathon | Smith | 1212121212 | 2 | 1 | 500 | 24-07-2016 |
| 5 | Sam | Saxon | 1372141312 | 2 | 2 | 400 | 25-03-2015 |

The following query will provide the required results:

```
SELECT Id, FName, LName, Salary FROM Employees
ORDER BY CASE WHEN DepartmentId = 1 THEN LName ELSE Salary END
```

# Chapter 25: The STUFF Function

| Parameter | Details |
|---|---|
| character_expression | the existing string in your data |
| start_position | the position in `character_expression` to delete `length` and then insert the `replacement_string` |
| length | the number of characters to delete from `character_expression` |
| replacement_string | the sequence of characters to insert in `character_expression` |

## Section 25.1: Using FOR XML to Concatenate Values from Multiple Rows

One common use for the `FOR XML` function is to concatenate the values of multiple rows.

Here's an example using the Customers table:

```
SELECT
    STUFF( (SELECT ';' + Email
        FROM Customers
        WHERE (Email IS NOT NULL AND Email <> '')
        ORDER BY Email ASC
        FOR XML PATH('')),
    1, 1, '')
```

In the example above, `FOR XML PATH('')` is being used to concatenate email addresses, using `;` as the delimiter character. Also, the purpose of `STUFF` is to remove the leading `;` from the concatenated string. `STUFF` is also implicitly casting the concatenated string from XML to varchar.

Note: the result from the above example will be XML-encoded, meaning it will replace `<` characters with `<` etc. If you don't want this, change `FOR XML PATH('')` to `FOR XML PATH, TYPE).value('.[1]','varchar(MAX)')`, e.g.:

```
SELECT
    STUFF( (SELECT ';' + Email
        FROM Customers
        WHERE (Email IS NOT NULL AND Email <> '')
        ORDER BY Email ASC
        FOR XML PATH, TYPE).value('.[1]','varchar(900)'),
    1, 1, '')
```

This can be used to achieve a result similar to `GROUP_CONCAT` in MySQL or `string_agg` in PostgreSQL 9.0+, although we use subqueries instead of GROUP BY aggregates. (As an alternative, you can install a user-defined aggregate such as this one if you're looking for functionality closer to that of `GROUP_CONCAT`).

## Section 25.2: Basic Character Replacement with STUFF()

The `STUFF()` function inserts a string into another string by first deleting a specified number of characters. The following example, deletes "Svr" and replaces it with "Server". This happens by specifying the `start_position` and `length` of the replacement.

```
SELECT STUFF('SQL Svr Documentation', 5, 3, 'Server')
```

Executing this example will result in returning `SQL Server Documentation` instead of `SQL Svr Documentation`.

# Section 25.3: Basic Example of STUFF() function

STUFF(Original_Expression, Start, Length, Replacement_expression)

STUFF() function inserts Replacement_expression, at the start position specified, along with removing the characters specified using Length parameter.

```
SELECT FirstName, LastName,Email, STUFF(Email, 2, 3, '*****') AS StuffedEmail FROM Employee
```

**Executing this example will result in returning the given table**

| FirstName | LastName | Email | StuffedEmail |
|---|---|---|---|
| Jomes | Hunter | James@hotmail.com | J*****s@hotmail.com |
| Shyam | rathod | Shyam@hotmail.com | S*****m@hotmail.com |
| Ram | shinde | Ram@hotmail.com | R*****hotmail.com |

# Section 25.4: stuff for comma separated in sql server

`FOR XML PATH` and `STUFF` to concatenate the multiple rows into a single row:

```
  SELECT DISTINCT t1.id,
      STUFF(
             (SELECT ', ' + CONVERT(VARCHAR(10), t2.date, 120)
              FROM yourtable t2
              WHERE t1.id = t2.id
              FOR XML PATH (''))
              , 1, 1, '')  AS DATE
    FROM yourtable t1;
```

# Section 25.5: Obtain column names separated with comma (not a list)

```
/*
The result can be use for fast way to use columns on Insertion/Updates.
Works with tables and views.

Example: eTableColumns  'Customers'
ColumnNames
-----------------------------------------------------
Id, FName, LName, Email, PhoneNumber, PreferredContact

INSERT INTO Customers (Id, FName, LName, Email, PhoneNumber, PreferredContact)
    VALUES (5, 'Ringo', 'Star', 'two@beatles.now', NULL, 'EMAIL')
*/
CREATE PROCEDURE eTableColumns (@Table VARCHAR(100))
AS
SELECT ColumnNames =
   STUFF( (SELECT ', ' +  c.name
FROM
    sys.columns c
INNER JOIN
    sys.types t ON c.user_type_id = t.user_type_id
WHERE
    c.object_id = OBJECT_ID( @Table)
        FOR XML PATH, TYPE).value('.[1]','varchar(2000)'),
    1, 1, '')
GO
```

# Chapter 26: JSON in SQL Server

| Parameters | Details |
|---|---|
| expression | Typically the name of a variable or a column that contains JSON text. |
| path | A JSON path expression that specifies the property to update. path has the following syntax: [append] [ lax \| strict ] $.<json path> |
| jsonExpression | Is a Unicode character expression containing the JSON text. |

## Section 26.1: Index on JSON properties by using computed columns

When storing JSON documents in SQL Server, We need to be able to efficiently filter and sort query results on properties of the JSON documents.

```
CREATE TABLE JsonTable
(
    id int identity primary key,
    jsonInfo nvarchar(max),
    CONSTRAINT [Content should be formatted as JSON]
    CHECK (ISJSON(jsonInfo)>0)
)
```

```
INSERT INTO JsonTable
VALUES(N'{"Name":"John","Age":23}'),
(N'{"Name":"Jane","Age":31}'),
(N'{"Name":"Bob","Age":37}'),
(N'{"Name":"Adam","Age":65}')
GO
```

Given the above table If we want to find the row with the name = 'Adam', we would execute the following query.

```
SELECT *
FROM JsonTable WHERE
JSON_VALUE(jsonInfo, '$.Name') = 'Adam'
```

However this will require SQL server to perform a full table which on a large table is not efficent.

To speed this up we would like to add an index, however we cannot directly reference properties in the JSON document. The solution is to add a computed column on the JSON path `$.Name`, then add an index on the computed column.

```
ALTER TABLE JsonTable
ADD vName as JSON_VALUE(jsonInfo, '$.Name')

CREATE INDEX idx_name
ON JsonTable(vName)
```

Now when we execute the same query, instead of a full table scan SQL server uses an index to seek into the non-clustered index and find the rows that satisfy the specified conditions.

Note: For SQL server to use the index, you must create the computed column with the same expression that you plan to use in your queries - in this example `JSON_VALUE(jsonInfo, '$.Name')`, however you can also use the name of computed column `vName`

# Section 26.2: Join parent and child JSON entities using CROSS APPLY OPENJSON

Join parent objects with their child entities, for example we want a relational table of each person and their hobbies

```
DECLARE @json nvarchar(1000) =
N'[
    {
        "id":1,
        "user":{"name":"John"},
        "hobbies":[
            {"name": "Reading"},
            {"name": "Surfing"}
        ]
    },
    {
        "id":2,
        "user":{"name":"Jane"},
        "hobbies":[
            {"name": "Programming"},
            {"name": "Running"}
        ]
    }
 ]'
```

**Query**

```
SELECT
    JSON_VALUE(person.value, '$.id') AS Id,
    JSON_VALUE(person.value, '$.user.name') AS PersonName,
    JSON_VALUE(hobbies.value, '$.name') AS Hobby
FROM OPENJSON (@json) AS person
    CROSS APPLY OPENJSON(person.value, '$.hobbies') AS hobbies
```

Alternatively this query can be written using the WITH clause.

```
SELECT
    Id, person.PersonName, Hobby
FROM OPENJSON (@json)
WITH(
    Id INT '$.id',
    PersonName nvarchar(100) '$.user.name',
    Hobbies nvarchar(MAX) '$.hobbies' AS JSON
) AS person
CROSS APPLY OPENJSON(Hobbies)
WITH(
    Hobby nvarchar(100) '$.name'
)
```

**Result**

| Id | PersonName | Hobby |
|---|---|---|
| 1 | John | Reading |
| 1 | John | Surfing |
| 2 | Jane | Programming |
| 2 | Jane | Running |

# Section 26.3: Format Query Results as JSON with FOR JSON

Input table data (People table)

| Id | Name | Age |
|---|---|---|
| 1 | John | 23 |
| 2 | Jane | 31 |

Query

```
SELECT Id, Name, Age
FROM People
FOR JSON PATH
```

Result

```
[
    {"Id":1,"Name":"John","Age":23},
    {"Id":2,"Name":"Jane","Age":31}
]
```

# Section 26.4: Parse JSON text

**JSON_VALUE** and **JSON_QUERY** functions parse JSON text and return scalar values or objects/arrays on the path in JSON text.

```
DECLARE @json NVARCHAR(100) = '{"id": 1, "user":{"name":"John"}, "skills":["C#","SQL"]}'

SELECT
    JSON_VALUE(@json, '$.id') AS Id,
    JSON_VALUE(@json, '$.user.name') AS Name,
    JSON_QUERY(@json, '$.user') AS UserObject,
    JSON_QUERY(@json, '$.skills') AS Skills,
    JSON_VALUE(@json, '$.skills[0]') AS Skill0
```

**Result**

| Id | Name | UserObject | Skills | Skill0 |
|---|---|---|---|---|
| 1 | John | {"name":"John"} | ["C#","SQL"] | C# |

# Section 26.5: Format one table row as a single JSON object using FOR JSON

**WITHOUT_ARRAY_WRAPPER** option in *FOR JSON* clause will remove array brackets from the JSON output. This is useful if you are returning single row in the query.

> Note: this option will produce invalid JSON output if more than one row is returned.

Input table data (People table)

| Id | Name | Age |
|---|---|---|
| 1 | John | 23 |
| 2 | Jane | 31 |

Query

```
SELECT Id, Name, Age
FROM People
WHERE Id = 1
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
```

Result

```
{"Id":1,"Name":"John","Age":23}
```

## Section 26.6: Parse JSON text using OPENJSON function

**OPENJSON** function parses JSON text and returns multiple outputs. Values that should be returned are specified using the paths defined in the WITH clause. If a path is not specified for some column, the column name is used as a path. This function casts returned values to the SQL types defined in the WITH clause. AS JSON option must be specified in the column definition if some object/array should be returned.

```
DECLARE @json NVARCHAR(100) = '{"id": 1, "user":{"name":"John"}, "skills":["C#","SQL"]}'

SELECT *
FROM OPENJSON (@json)
    WITH(Id int '$.id',
        Name nvarchar(100) '$.user.name',
        UserObject nvarchar(max) '$.user' AS JSON,
        Skills nvarchar(max) '$.skills' AS JSON,
        Skill0 nvarchar(20) '$.skills[0]')
```

**Result**

| Id | Name | UserObject | Skills | Skill0 |
|---|---|---|---|---|
| 1 | John | {"name":"John"} | ["C#","SQL"] | C# |

# Chapter 27: OPENJSON

## Section 27.1: Transform JSON array into set of rows

OPENJSON function parses collection of JSON objects and returns values from JSON text as set of rows.

```
declare @json nvarchar(4000) = N'[
  {"Number":"SO43659","Date":"2011-05-31T00:00:00","Customer": "MSFT","Price":59.99,"Quantity":1},
  {"Number":"SO43661","Date":"2011-06-01T00:00:00","Customer":"Nokia","Price":24.99,"Quantity":3}
]'

SELECT    *
FROM OPENJSON (@json)
    WITH (
          Number   varchar(200),
          Date     datetime,
          Customer varchar(200),
          Quantity int
  )
```

In the WITH clause is specified return schema of OPENJSON function. Keys in the JSON objects are fetched by column names. If some key in JSON is not specified in the WITH clause (e.g. Price in this example) it will be ignored. Values are automatically converted into specified types.

| Number | Date | Customer | Quantity |
|---|---|---|---|
| SO43659 | 2011-05-31T00:00:00 | MSFT | 1 |
| SO43661 | 2011-06-01T00:00:00 | Nokia | 3 |

## Section 27.2: Get key:value pairs from JSON text

OPENJSON function parse JSON text and returns all key:value pairs at the first level of JSON:

```
declare @json NVARCHAR(4000) = N'{"Name":"Joe","age":27,"skills":["C#","SQL"]}';
SELECT * FROM OPENJSON(@json);
```

| key | value | type |
|---|---|---|
| Name | Joe | 1 |
| age | 27 | 2 |
| skills | ["C#","SQL"] | 4 |

Column type describe the type of value, i.e. null(0), string(1), number(2), boolean(3), array(4), and object(5).

## Section 27.3: Transform nested JSON fields into set of rows

OPENJSON function parses collection of JSON objects and returns values from JSON text as set of rows. If the values in input object are nested, additional mapping parameter can be specified in each column in WITH clause:

```
declare @json nvarchar(4000) = N'[

{"data":{"num":"SO43659","date":"2011-05-31T00:00:00"},"info":{"customer":"MSFT","Price":59.99,"qty
":1}},

{"data":{"number":"SO43661","date":"2011-06-01T00:00:00"},"info":{"customer":"Nokia","Price":24.99,
"qty":3}}
]'

SELECT    *
```

```
FROM OPENJSON (@json)
    WITH (
          Number   varchar(200) '$.data.num',
          Date     datetime '$.data.date',
          Customer varchar(200) '$.info.customer',
          Quantity int '$.info.qty',
  )
```

In the WITH clause is specified return schema of OPENJSON function. After the type is specified path to the JSON nodes where returned value should be found. Keys in the JSON objects are fetched by these paths. Values are automatically converted into specified types.

| Number | Date | Customer | Quantity |
|---|---|---|---|
| SO43659 | 2011-05-31T00:00:00 | MSFT | 1 |
| SO43661 | 2011-06-01T00:00:00 | Nokia | 3 |

# Section 27.4: Extracting inner JSON sub-objects

OPENJSON can extract fragments of JSON objects inside the JSON text. In the column definition that references JSON sub-object set the type nvarchar(max) and AS JSON option:

```
declare @json nvarchar(4000) = N'[

{"Number":"SO43659","Date":"2011-05-31T00:00:00","info":{"customer":"MSFT","Price":59.99,"qty":1}},

{"Number":"SO43661","Date":"2011-06-01T00:00:00","info":{"customer":"Nokia","Price":24.99,"qty":3}}
]'

SELECT    *
FROM OPENJSON (@json)
    WITH (
          Number   varchar(200),
          Date     datetime,
          Info nvarchar(max) '$.info' AS JSON
  )
```

Info column will be mapped to "Info" object. Results will be:

| Number | Date | Info |
|---|---|---|
| SO43659 | 2011-05-31T00:00:00 | {"customer":"MSFT","Price":59.99,"qty":1} |
| SO43661 | 2011-06-01T00:00:00 | {"customer":"Nokia","Price":24.99,"qty":3} |

# Section 27.5: Working with nested JSON sub-arrays

JSON may have complex structure with inner arrays. In this example, we have array of orders with nested sub array of OrderItems.

```
declare @json nvarchar(4000) = N'[
  {"Number":"SO43659","Date":"2011-05-31T00:00:00",
    "Items":[{"Price":11.99,"Quantity":1},{"Price":12.99,"Quantity":5}]},
  {"Number":"SO43661","Date":"2011-06-01T00:00:00",

"Items":[{"Price":21.99,"Quantity":3},{"Price":22.99,"Quantity":2},{"Price":23.99,"Quantity":2}]}
]'
```

We can parse root level properties using OPENJSON that will return Items array AS JSON fragment. Then we can apply OPENJSON again on Items array and open inner JSON table. First level table and inner table will be "joined"

like in the JOIN between standard tables:

```
SELECT    *
FROM
    OPENJSON (@json)
    WITH (  NUMBER VARCHAR(200), DATE datetime,
            Items nvarchar(MAX) AS JSON )
        CROSS APPLY
            OPENJSON (Items)
            WITH ( Price FLOAT, Quantity INT)
```

Results:

| Number | Date | Items | Price | Quantity |
|---|---|---|---|---|
| SO43659 | 2011-05-31 00:00:00.000 | [{"Price":11.99,"Quantity":1},{"Price":12.99,"Quantity":5}] | 11.99 | 1 |
| SO43659 | 2011-05-31 00:00:00.000 | [{"Price":11.99,"Quantity":1},{"Price":12.99,"Quantity":5}] | 12.99 | 5 |
| SO43661 | 2011-06-01 00:00:00.000 | [{"Price":21.99,"Quantity":3},{"Price":22.99,"Quantity":2},{"Price":23.99,"Quantity":2}] | 21.99 | 3 |
| SO43661 | 2011-06-01 00:00:00.000 | [{"Price":21.99,"Quantity":3},{"Price":22.99,"Quantity":2},{"Price":23.99,"Quantity":2}] | 22.99 | 2 |
| SO43661 | 2011-06-01 00:00:00.000 | [{"Price":21.99,"Quantity":3},{"Price":22.99,"Quantity":2},{"Price":23.99,"Quantity":2}] | 23.99 | 2 |

# Chapter 28: FOR JSON

## Section 28.1: FOR JSON PATH

Formats results of SELECT query as JSON text. FOR JSON PATH clause is added after query:

```
SELECT top 3 object_id, name, TYPE, principal_id FROM sys.objects
FOR JSON PATH
```

Column names will be used as keys in JSON, and cell values will be generated as JSON values. Result of the query would be an array of JSON objects:

```
[
  {"object_id":3,"name":"sysrscols","type":"S "},
  {"object_id":5,"name":"sysrowsets","type":"S "},
  {"object_id":6,"name":"sysclones","type":"S "}
]
```

NULL values in principal_id column will be ignored (they will not be generated).

## Section 28.2: FOR JSON PATH with column aliases

FOR JSON PATH enables you to control format of the output JSON using column aliases:

```
SELECT top 3 object_id AS id, name AS [DATA.name], TYPE AS [DATA.type]
FROM sys.objects
FOR JSON PATH
```

Column alias will be used as a key name. Dot-separated column aliases (data.name and data.type) will be generated as nested objects. If two column have the same prefix in dot notation, they will be grouped together in single object (data in this example):

```
[
  {"id":3,"data":{"name":"sysrscols","type":"S "}},
  {"id":5,"data":{"name":"sysrowsets","type":"S "}},
  {"id":6,"data":{"name":"sysclones","type":"S "}}
]
```

## Section 28.3: FOR JSON clause without array wrapper (single object in output)

WITHOUT_ARRAY_WRAPPER option enables you to generate a single object instead of the array. Use this option if you know that you will return single row/object:

```
SELECT top 3 object_id, name, TYPE, principal_id
FROM sys.objects
WHERE object_id = 3
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
```

Single object will be returned in this case:

```
{"object_id":3,"name":"sysrscols","type":"S "}
```

## Section 28.4: INCLUDE_NULL_VALUES

FOR JSON clause ignores NULL values in cells. If you want to generate "key": null pairs for cells that contain NULL values, add INCLUDE_NULL_VALUES option in the query:

```
SELECT top 3 object_id, name, TYPE, principal_id
FROM sys.objects
FOR JSON PATH, INCLUDE_NULL_VALUES
```

NULL values in principal_id column will be generated:

```
[
  {"object_id":3,"name":"sysrscols","type":"S ","principal_id":null},
  {"object_id":5,"name":"sysrowsets","type":"S ","principal_id":null},
  {"object_id":6,"name":"sysclones","type":"S ","principal_id":null}
]
```

## Section 28.5: Wrapping results with ROOT object

Wraps returned JSON array in additional root object with specified key:

```
SELECT top 3 object_id, name, TYPE FROM sys.objects
FOR JSON PATH, ROOT('data')
```

Result of the query would be array of JSON objects inside the wrapper object:

```
{
  "data":[
            {"object_id":3,"name":"sysrscols","type":"S "},
            {"object_id":5,"name":"sysrowsets","type":"S "},
            {"object_id":6,"name":"sysclones","type":"S "}
          ]
}
```

## Section 28.6: FOR JSON AUTO

Automatically nests values from the second table as a nested sub-array of JSON objects:

```
SELECT top 5 o.object_id, o.name, c.column_id, c.name
FROM sys.objects o
    JOIN sys.columns c ON o.object_id = c.object_id
FOR JSON AUTO
```

Result of the query would be array of JSON objects:

```
[
  {
   "object_id":3,
   "name":"sysrscols",
   "c":[
        {"column_id":12,"name":"bitpos"},
        {"column_id":6,"name":"cid"}
       ]
  },
  {
    "object_id":5,
    "name":"sysrowsets",
```

```
    "c":[
         {"column_id":13,"name":"colguid"},
         {"column_id":3,"name":"hbcolid"},
         {"column_id":8,"name":"maxinrowlen"}
     ]
  }
]
```

## Section 28.7: Creating custom nested JSON structure

If you need some complex JSON structure that cannot be created using FOR JSON PATH or FOR JSON AUTO, you can customize your JSON output by putting FOR JSON sub-queries as column expressions:

```
SELECT top 5 o.object_id, o.name,
        (SELECT column_id, c.name
            FROM sys.columns c WHERE o.object_id = c.object_id
            FOR JSON PATH) AS COLUMNS,
        (SELECT parameter_id, name
            FROM sys.parameters p WHERE o.object_id = p.object_id
            FOR JSON PATH) AS parameters
FROM sys.objects o
FOR JSON PATH
```

Each sub-query will produce JSON result that will be included in the main JSON content.

# Chapter 29: Queries with JSON data

## Section 29.1: Using values from JSON in query

JSON_VALUE function enables you to take a data from JSON text on the path specified as the second argument, and use this value in any part of the select query:

```
SELECT ProductID, Name, Color, SIZE, Price, JSON_VALUE(DATA, '$.Type') AS TYPE
FROM Product
WHERE JSON_VALUE(DATA, '$.Type') = 'part'
```

## Section 29.2: Using JSON values in reports

Once JSON values are extracted from JSON text, you can use them ina any part of the query. You can create some kind of report on JSON data with grouping aggregations, etc:

```
SELECT JSON_VALUE(DATA, '$.Type') AS TYPE,
        AVG( CAST(JSON_VALUE(DATA, '$.ManufacturingCost') AS FLOAT) ) AS cost
FROM Product
GROUP BY JSON_VALUE(DATA, '$.Type')
HAVING JSON_VALUE(DATA, '$.Type') IS NOT NULL
```

## Section 29.3: Filter-out bad JSON text from query results

If some JSON text might not be properly formatted, you can remove those entries from query using ISJSON function.

```
SELECT ProductID, Name, Color, SIZE, Price, JSON_VALUE(DATA, '$.Type') AS TYPE
FROM Product
WHERE JSON_VALUE(DATA, '$.Type') = 'part'
AND ISJSON(DATA) > 0
```

## Section 29.4: Update value in JSON column

JSON_MODIFY function can be used to update value on some path. You can use this function to modify original value of JSON cell in UPDATE statement:

```
update Product
set Data = JSON_MODIFY(Data, '$.Price', 24.99)
where ProductID = 17;
```

JSON_MODIFY function will update or create Price key (if it does not exists). If new value is NULL, the key will be removed. JSON_MODIFY function will treat new value as string (escape special characters, wrap it with double quotes to create proper JSON string). If your new value is JSON fragment, you should wrap it with JSON_QUERY function:

```
update Product
set Data = JSON_MODIFY(Data, '$.tags', JSON_QUERY('["promo","new"]'))
where ProductID = 17;
```

JSON_QUERY function without second parameter behaves like a "cast to JSON". Since the result of JSON_QUERY is valid JSON fragment (object or array), JSON_MODIFY will no escape this value when modifies input JSON.

## Section 29.5: Append new value into JSON array

JSON_MODIFY function can be used to append new value to some array inside JSON:

```
update Product
set Data = JSON_MODIFY(Data, 'append $.tags', "sales")
where ProductID = 17;
```

New value will be appended at the end of the array, or a new array with value ["sales"] will be created. JSON_MODIFY function will treat new value as string (escape special characters, wrap it with double quotes to create proper JSON string). If your new value is JSON fragment, you should wrap it with JSON_QUERY function:

```
update Product
set Data = JSON_MODIFY(Data, 'append $.tags', JSON_QUERY('{"type":"new"}'))
where ProductID = 17;
```

JSON_QUERY function without second parameter behaves like a "cast to JSON". Since the result of JSON_QUERY is valid JSON fragment (object or array), JSON_MODIFY will no escape this value when modifies input JSON.

## Section 29.6: JOIN table with inner JSON collection

If you have a "child table" formatted as JSON collection and stored in-row as JSON column, you can unpack this collection, transform it to table and join it with parent row. Instead of the standard JOIN operator, you should use CROSS APPLY. In this example, product parts are formatted as collection of JSON objects in and stored in Data column:

```
SELECT ProductID, Name, SIZE, Price, Quantity, PartName, Code
FROM Product
    CROSS APPLY OPENJSON(DATA, '$.Parts') WITH (PartName VARCHAR(20), Code VARCHAR(5))
```

Result of the query is equivalent to the join between Product and Part tables.

## Section 29.7: Finding rows that contain value in the JSON array

In this example, Tags array may contain various keywords like ["promo", "sales"], so we can open this array and filter values:

```
SELECT ProductID, Name, Color, SIZE, Price, Quantity
FROM Product
    CROSS APPLY OPENJSON(DATA, '$.Tags')
WHERE VALUE = 'sales'
```

OPENJSON will open inner collection of tags and return it as table. Then we can filter results by some value in the table.

# Chapter 30: Storing JSON in SQL tables

## Section 30.1: JSON stored as text column

JSON is textual format, so it is stored in standard NVARCHAR columns. NoSQL collection is equivalent to two column key value table:

```
CREATE TABLE ProductCollection (
  Id int identity primary key,
  Data nvarchar(max)
)
```

Use `nvarchar(max)` as you are not sure what would be the size of your JSON documents. `nvarchar(4000)` and `varchar(8000)` have better performance but with size limit to 8KB.

## Section 30.2: Ensure that JSON is properly formatted using ISJSON

Since JSON is stored textual column, you might want to ensure that it is properly formatted. You can add CHECK constraint on JSON column that checks is text properly formatted JSON:

```
CREATE TABLE ProductCollection (
  Id int identity primary key,
  Data nvarchar(max)
       CONSTRAINT [Data should be formatted as JSON]
       CHECK (ISJSON(Data) > 0)
)
```

If you already have a table, you can add check constraint using the ALTER TABLE statement:

```
ALTER TABLE ProductCollection
    ADD CONSTRAINT [Data should be formatted as JSON]
        CHECK (ISJSON(Data) > 0)
```

## Section 30.3: Expose values from JSON text as computed columns

You can expose values from JSON column as computed columns:

```
CREATE TABLE ProductCollection (
  Id int identity primary key,
  Data nvarchar(max),
  Price AS JSON_VALUE(Data, '$.Price'),
  Color JSON_VALUE(Data, '$.Color') PERSISTED
)
```

If you add PERSISTED computed column, value from JSON text will be materialized in this column. This way your queries can faster read value from JSON text because no parsing is needed. Each time JSON in this row changes, value will be re-calculated.

## Section 30.4: Adding index on JSON path

Queries that filter or sort data by some value in JSON column usually use full table scan.

```
SELECT * FROM ProductCollection
WHERE JSON_VALUE(DATA, '$.Color') = 'Black'
```

To optimize these kind of queries, you can add non-persisted computed column that exposes JSON expression used in filter or sort (in this example JSON_VALUE(Data, '$.Color')), and create index on this column:

```
ALTER TABLE ProductCollection
ADD vColor as JSON_VALUE(Data, '$.Color')

CREATE INDEX idx_JsonColor
ON ProductCollection(vColor)
```

Queries will use the index instead of plain table scan.

## Section 30.5: JSON stored in in-memory tables

If you can use memory-optimized tables, you can store JSON as text:

```
CREATE TABLE ProductCollection (
  Id int identity primary key nonclustered,
  Data nvarchar(max)
) WITH (MEMORY_OPTIMIZED=ON)
```

Advantages of JSON in in-memory:

- JSON data is always in memory so there is no disk access
- There are no locks and latches while working with JSON

# Chapter 31: Modify JSON text

## Section 31.1: Modify value in JSON text on the specified path

JSON_MODIFY function uses JSON text as input parameter, and modifies a value on the specified path using third argument:

```
declare @json nvarchar(4000) = N'{"Id":1,"Name":"Toy Car","Price":34.99}'
set @json = JSON_MODIFY(@json, '$.Price', 39.99)
print @json -- Output: {"Id":1,"Name":"Toy Car","Price":39.99}
```

As a result, we will have new JSON text with "Price":39.99 and other value will not be changed. If object on the specified path does not exists, JSON_MODIFY will insert key:value pair.

In order to delete key:value pair, put NULL as new value:

```
declare @json nvarchar(4000) = N'{"Id":1,"Name":"Toy Car","Price":34.99}'
set @json = JSON_MODIFY(@json, '$.Price', NULL)
print @json -- Output: {"Id":1,"Name":"Toy Car"}
```

JSON_MODIFY will by default delete key if it does not have value so you can use it to delete a key.

## Section 31.2: Append a scalar value into a JSON array

JSON_MODIFY has 'append' mode that appends value into array.

```
declare @json nvarchar(4000) = N'{"Id":1,"Name":"Toy Car","Tags":["toy","game"]}'
set @json = JSON_MODIFY(@json, 'append $.Tags', 'sales')
print @json -- Output: {"Id":1,"Name":"Toy Car","Tags":["toy","game","sales"]}
```

If array on the specified path does not exists, JSON_MODIFY(append) will create new array with a single element:

```
declare @json nvarchar(4000) = N'{"Id":1,"Name":"Toy Car","Price":34.99}'
set @json = JSON_MODIFY(@json, 'append $.Tags', 'sales')
print @json -- Output {"Id":1,"Name":"Toy Car","Tags":["sales"]}
```

## Section 31.3: Insert new JSON Object in JSON text

JSON_MODIFY function enables you to insert JSON objects into JSON text:

```
declare @json nvarchar(4000) = N'{"Id":1,"Name":"Toy Car"}'
set @json = JSON_MODIFY(@json, '$.Price',
                        JSON_QUERY('{"Min":34.99,"Recommended":45.49}'))
print @json
-- Output: {"Id":1,"Name":"Toy Car","Price":{"Min":34.99,"Recommended":45.49}}
```

Since third parameter is text you need to wrap it with JSON_QUERY function to "cast" text to JSON. Without this "cast", JSON_MODIFY will treat third parameter as plain text and escape characters before inserting it as string value. Without JSON_QUERY results will be:

```
{"Id":1,"Name":"Toy Car","Price":'{\"Min\":34.99,\"Recommended\":45.49}'}
```

JSON_MODIFY will insert this object if it does not exist, or delete it if value of third parameter is NULL.

## Section 31.4: Insert new JSON array generated with FOR JSON query

You can generate JSON object using standard SELECT query with FOR JSON clause and insert it into JSON text as third parameter:

```
declare @json nvarchar(4000) = N'{"Id":17,"Name":"WWI"}'
set @json = JSON_MODIFY(@json, '$.tables',
                        (select name from sys.tables FOR JSON PATH) )
print @json

(1 row(s) affected)
{"Id":1,"Name":"master","tables":[{"name":"Colors"},{"name":"Colors_Archive"},{"name":"OrderLines"}
,{"name":"PackageTypes"},{"name":"PackageTypes_Archive"},{"name":"StockGroups"},{"name":"StockItemS
tockGroups"},{"name":"StockGroups_Archive"},{"name":"StateProvinces"},{"name":"CustomerTransactions
"},{"name":"StateProvinces_Archive"},{"name":"Cities"},{"name":"Cities_Archive"},{"name":"SystemPar
ameters"},{"name":"InvoiceLines"},{"name":"Suppliers"},{"name":"StockItemTransactions"},{"name":"Su
ppliers_Archive"},{"name":"Customers"},{"name":"Customers_Archive"},{"name":"PurchaseOrders"},{"nam
e":"Orders"},{"name":"People"},{"name":"StockItems"},{"name":"People_Archive"},{"name":"ColdRoomTem
peratures"},{"name":"ColdRoomTemperatures_Archive"},{"name":"VehicleTemperatures"},{"name":"StockIt
ems_Archive"},{"name":"Countries"},{"name":"StockItemHoldings"},{"name":"sysdiagrams"},{"name":"Pur
chaseOrderLines"},{"name":"Countries_Archive"},{"name":"DeliveryMethods"},{"name":"DeliveryMethods_
Archive"},{"name":"PaymentMethods"},{"name":"SupplierTransactions"},{"name":"PaymentMethods_Archive
"},{"name":"TransactionTypes"},{"name":"SpecialDeals"},{"name":"TransactionTypes_Archive"},{"name":
"SupplierCategories"},{"name":"SupplierCategories_Archive"},{"name":"BuyingGroups"},{"name":"Invoic
es"},{"name":"BuyingGroups_Archive"},{"name":"CustomerCategories"},{"name":"CustomerCategories_Arch
ive"}]}
```

JSON_MODIFY will know that select query with FOR JSON clause generates valid JSON array and it will just insert it into JSON text.

> You can use all FOR JSON options in SELECT query, **except WITHOUT_ARRAY_WRAPPER**, which will generate single object instead of JSON array. See other example in this topic to see how insert single JSON object.

## Section 31.5: Insert single JSON object generated with FOR JSON clause

You can generate JSON object using standard SELECT query with FOR JSON clause and WITHOUT_ARRAY_WRAPPER option, and insert it into JSON text as a third parameter:

```
declare @json nvarchar(4000) = N'{"Id":17,"Name":"WWI"}'
set @json = JSON_MODIFY(@json, '$.table',
                        JSON_QUERY(
                         (select name, create_date, schema_id
                           from sys.tables
                           where name = 'Colors'
                           FOR JSON PATH, WITHOUT_ARRAY_WRAPPER)))
print @json

(1 row(s) affected)
{"Id":17,"Name":"WWI","table":{"name":"Colors","create_date":"2016-06-02T10:04:03.280","schema_id":
13}}
```

FOR JSON with WITHOUT_ARRAY_WRAPPER option may generate invalid JSON text if SELECT query returns more than one result (you should use TOP 1 or filter by primary key in this case). Therefore, JSON_MODIFY will assume

that returned result is just a plain text and escape it like any other text if you don't wrap it with JSON_QUERY function.

> You should wrap **FOR JSON, WITHOUT_ARRAY_WRAPPER** query with **JSON_QUERY** function in order to cast result to JSON.

# Chapter 32: FOR XML PATH

## Section 32.1: Using FOR XML PATH to concatenate values

The `FOR XML PATH` can be used for concatenating values into string. The example below concatenates values into a `CSV` string:

```
DECLARE @DataSource TABLE
(
    [rowID] TINYINT
   ,[FirstName] NVARCHAR(32)
);

INSERT INTO @DataSource ([rowID], [FirstName])
VALUES (1, 'Alex')
      ,(2, 'Peter')
      ,(3, 'Alexsandyr')
      ,(4, 'George');

SELECT STUFF
(
    (
        SELECT ',' + [FirstName]
        FROM @DataSource
        ORDER BY [rowID] DESC
        FOR XML PATH(''), TYPE
    ).value('.', 'NVARCHAR(MAX)')
    ,1
    ,1
    ,''
);
```

Few important notes:

- the `ORDER BY` clause can be used to order the values in a preferred way
- if a longer value is used as the concatenation separator, the `STUFF` function parameter must be changed too;

```
SELECT STUFF
(
    (
        SELECT '---' + [FirstName]
        FROM @DataSource
        ORDER BY [rowID] DESC
        FOR XML PATH(''), TYPE
    ).value('.', 'NVARCHAR(MAX)')
    ,1
    ,3 -- the "3" could also be represented as: LEN('---') for clarity
    ,''
);
```

- as the `TYPE` option and `.value` function are used, the concatenation works with `NVARCHAR(MAX)` string

## Section 32.2: Specifying namespaces

`Version ≥ SQL Server 2008`

```
WITH XMLNAMESPACES (
    DEFAULT 'http://www.w3.org/2000/svg',
    'http://www.w3.org/1999/xlink' AS xlink
```

```
)
SELECT
    'example.jpg' AS 'image/@xlink:href',
    '50px' AS 'image/@width',
    '50px' AS 'image/@height'
FOR XML PATH('svg')
```

```
<svg xmlns:xlink="http://www.w3.org/1999/xlink" xmlns="http://www.w3.org/2000/svg">
    <image xlink:href="firefox.jpg" width="50px" height="50px"/>
</svg>
```

## Section 32.3: Specifying structure using XPath expressions

```
SELECT
    'XPath example' AS 'head/title',
    'This example demonstrates ' AS 'body/p',
    'https://www.w3.org/TR/xpath/' AS 'body/p/a/@href',
    'XPath expressions' AS 'body/p/a'
FOR XML PATH('html')
```

```
<html>
    <head>
        <title>XPath example</title>
    </head>
    <body>
        <p>This example demonstrates <a href="https://www.w3.org/TR/xpath/">XPath
expressions</a></p>
    </body>
</html>
```

In `FOR XML PATH`, columns without a name become text nodes. `NULL` or `''` therefore become empty text nodes. Note: you can convert a named column to an unnamed one by using `AS *`

```
DECLARE @tempTable TABLE (Ref INT, Des NVARCHAR(100), Qty INT)
INSERT INTO @tempTable VALUES (100001, 'Normal', 1), (100002, 'Foobar', 1), (100003, 'Hello World',
2)

SELECT ROW_NUMBER() OVER (ORDER BY Ref) AS '@NUM',
     'REF' AS 'FLD/@NAME', REF AS 'FLD', '',
     'DES' AS 'FLD/@NAME', DES AS 'FLD', '',
     'QTY' AS 'FLD/@NAME', QTY AS 'FLD'
FROM @tempTable
FOR XML PATH('LIN'), ROOT('row')
```

```
<row>
  <LIN NUM="1">
    <FLD NAME="REF">100001</FLD>
    <FLD NAME="DES">Normal</FLD>
    <FLD NAME="QTY">1</FLD>
  </LIN>
  <LIN NUM="2">
    <FLD NAME="REF">100002</FLD>
    <FLD NAME="DES">Foobar</FLD>
    <FLD NAME="QTY">1</FLD>
  </LIN>
  <LIN NUM="3">
    <FLD NAME="REF">100003</FLD>
    <FLD NAME="DES">Hello World</FLD>
    <FLD NAME="QTY">2</FLD>
  </LIN>
</row>
```

Using (empty) text nodes helps to separate the previously output node from the next one, so that SQL Server knows to start a new element for the next column. Otherwise, it gets confused when the attribute already exists on what it thinks is the "current" element.

For example, without the the empty strings between the element and the attribute in the `SELECT` statement, SQL Server gives an error:

> Attribute-centric column 'FLD/@NAME' must not come after a non-attribute-centric sibling in XML hierarchy in FOR XML PATH.

Also note that this example also wrapped the XML in a root element named `row`, specified by `ROOT('row')`

## Section 32.4: Hello World XML

```
SELECT 'Hello World' FOR XML PATH('example')
```

```
<example>Hello World</example>
```

# Chapter 33: Join

In Structured Query Language (SQL), a JOIN is a method of linking two data tables in a single query, allowing the database to return a set that contains data from both tables at once, or using data from one table to be used as a Filter on the second table. There are several types of JOINs defined within the ANSI SQL standard.

## Section 33.1: Inner Join

`Inner join` returns only those records/rows that match/exists in both the tables based on one or more conditions (specified using `ON` keyword). It is the most common type of join. The general syntax for `inner join` is:

```
SELECT *
FROM table_1
INNER JOIN table_2
  ON table_1.column_name = table_2.column_name
```

It can also be simplified as just `JOIN`:

```
SELECT *
FROM table_1
JOIN table_2
  ON table_1.column_name = table_2.column_name
```

Example

```
/* Sample data. */
DECLARE @Animal table (
    AnimalId Int IDENTITY,
    Animal Varchar(20)
);

DECLARE @AnimalSound table (
    AnimalSoundId Int IDENTITY,
    AnimalId Int,
    Sound Varchar(20)
);

INSERT INTO @Animal (Animal) VALUES ('Dog');
INSERT INTO @Animal (Animal) VALUES ('Cat');
INSERT INTO @Animal (Animal) VALUES ('Elephant');

INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (1, 'Barks');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (2, 'Meows');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (3, 'Trumpets');
/* Sample data prepared. */

SELECT
    *
FROM
    @Animal
    JOIN @AnimalSound
        ON @Animal.AnimalId = @AnimalSound.AnimalId;
```

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           Dog                 1             1           Barks
2           Cat                 2             2           Meows
3           Elephant            3             3           Trumpets
```

### Using inner join with left outer join (Substitute for Not exists)

This query will return data from table 1 where fields matching with table2 with a key and data not in Table 1 when comparing with Table2 with a condition and key

```
SELECT *
  FROM Table1 t1
    INNER JOIN Table2 t2 ON t1.ID_Column = t2.ID_Column
    LEFT  JOIN Table3 t3 ON t1.ID_Column = t3.ID_Column
  WHERE t2.column_name = column_value
    AND t3.ID_Column IS NULL
  ORDER BY t1.column_name;
```

## Section 33.2: Outer Join

### Left Outer Join

`LEFT JOIN` returns all rows from the left table, matched to rows from the right table where the `ON` clause conditions are met. Rows in which the `ON` clause is not met have `NULL` in all of the right table's columns. The syntax of a `LEFT JOIN` is:

```
SELECT * FROM table_1 AS t1
LEFT JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

### Right Outer Join

`RIGHT JOIN` returns all rows from the right table, matched to rows from the left table where the `ON` clause conditions are met. Rows in which the `ON` clause is not met have `NULL` in all of the left table's columns. The syntax of a `RIGHT JOIN` is:

```
SELECT * FROM table_1 AS t1
RIGHT JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

### Full Outer Join

`FULL JOIN` combines `LEFT JOIN` and `RIGHT JOIN`. All rows are returned from both tables, regardless of whether the conditions in the `ON` clause are met. Rows that do not satisfy the `ON` clause are returned with `NULL` in all of the opposite table's columns (that is, for a row in the left table, all columns in the right table will contain `NULL`, and vice versa). The syntax of a `FULL JOIN` is:

```
SELECT * FROM table_1 AS t1
FULL JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

### Examples

```
/* Sample test data. */
DECLARE @Animal table (
    AnimalId Int IDENTITY,
    Animal Varchar(20)
);

DECLARE @AnimalSound table (
    AnimalSoundId Int IDENTITY,
    AnimalId Int,
    Sound Varchar(20)
```

```
);

INSERT INTO @Animal (Animal) VALUES ('Dog');
INSERT INTO @Animal (Animal) VALUES ('Cat');
INSERT INTO @Animal (Animal) VALUES ('Elephant');
INSERT INTO @Animal (Animal) VALUES ('Frog');

INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (1, 'Barks');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (2, 'Meows');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (3, 'Trumpet');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (5, 'Roars');
/* Sample data prepared. */
```

**LEFT OUTER JOIN**

```
SELECT *
FROM @Animal AS t1
LEFT JOIN @AnimalSound AS t2 ON t1.AnimalId = t2.AnimalId;
```

Results for `LEFT JOIN`

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           Dog                 1             1           Barks
2           Cat                 2             2           Meows
3           Elephant            3             3           Trumpet
4           Frog                NULL          NULL        NULL
```

**RIGHT OUTER JOIN**

```
SELECT *
FROM @Animal AS t1
RIGHT JOIN @AnimalSound AS t2 ON t1.AnimalId = t2.AnimalId;
```

Results for `RIGHT JOIN`

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           Dog                 1             1           Barks
2           Cat                 2             2           Meows
3           Elephant            3             3           Trumpet
NULL        NULL                4             5           Roars
```

**FULL OUTER JOIN**

```
SELECT *
FROM @Animal AS t1
FULL JOIN @AnimalSound AS t2 ON t1.AnimalId = t2.AnimalId;
```

Results for `FULL JOIN`

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           Dog                 1             1           Barks
2           Cat                 2             2           Meows
3           Elephant            3             3           Trumpet
```

```
4           Frog                NULL          NULL        NULL
NULL        NULL                4             5           Roars
```

# Section 33.3: Using Join in an Update

Joins can also be used in an `UPDATE` statement:

```
CREATE TABLE Users (
    UserId int NOT NULL,
    AccountId int NOT NULL,
    RealName nvarchar(200) NOT NULL
)

CREATE TABLE Preferences (
    UserId int NOT NULL,
    SomeSetting bit NOT NULL
)
```

Update the `SomeSetting` column of the `Preferences` table filtering by a predicate on the `Users` table as follows:

```
UPDATE p
SET p.SomeSetting = 1
FROM Users u
JOIN Preferences p ON u.UserId = p.UserId
WHERE u.AccountId = 1234
```

`p` is an alias for `Preferences` defined in the `FROM` clause of the statement. Only rows with a matching `AccountId` from the `Users` table will be updated.

**Update with left outer join statements**

```
Update t
SET  t.Column1=100
FROM Table1 t LEFT JOIN Table12 t2
ON t2.ID=t.ID
```

**Update tables with inner join and aggregate function**

```
UPDATE t1
SET t1.field1 = t2.field2Sum
FROM table1 t1
INNER JOIN (select field3, sum(field2) as field2Sum
from table2
group by field3) as t2
on t2.field3 = t1.field3
```

# Section 33.4: Join on a Subquery

Joining on a subquery is often used when you want to get aggregate data (such as Count, Avg, Max, or Min) from a child/details table and display that along with records from the parent/header table. For example, you may want to retrieve the top/first child row based on Date or Id or maybe you want a Count of all Child Rows or an Average.

This example uses aliases which makes queries easier to read when you have multiple tables involved. In this case we are retrieving all rows from the parent table Purchase Orders and retrieving only the last (or most recent) child row from the child table PurchaseOrderLineItems. This example assumes the child table uses incremental numeric Id's.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
  item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
     (
       SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, MAX(l.id) AS Id
       FROM PurchaseOrderLineItems l
       GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
     ) AS item ON item.PurchaseOrderId = po.Id
```

## Section 33.5: Cross Join

`A cross join` is a Cartesian join, meaning a Cartesian product of both the tables. This join does not need any condition to join two tables. Each row in the left table will join to each row of the right table. Syntax for a cross join:

```
SELECT * FROM table_1
CROSS JOIN table_2
```

Example:

```
/* Sample data. */
DECLARE @Animal table (
    AnimalId Int IDENTITY,
    Animal Varchar(20)
);

DECLARE @AnimalSound table (
    AnimalSoundId Int IDENTITY,
    AnimalId Int,
    Sound Varchar(20)
);

INSERT INTO @Animal (Animal) VALUES ('Dog');
INSERT INTO @Animal (Animal) VALUES ('Cat');
INSERT INTO @Animal (Animal) VALUES ('Elephant');

INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (1, 'Barks');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (2, 'Meows');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (3, 'Trumpet');
/* Sample data prepared. */

SELECT
    *
FROM
    @Animal
    CROSS JOIN @AnimalSound;
```

Results:

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           Dog                 1             1           Barks
2           Cat                 1             1           Barks
3           Elephant            1             1           Barks
1           Dog                 2             2           Meows
2           Cat                 2             2           Meows
3           Elephant            2             2           Meows
1           Dog                 3             3           Trumpet
2           Cat                 3             3           Trumpet
```

```
3           Elephant            3           3           Trumpet
```

Note that there are other ways that a CROSS JOIN can be applied. This is a an *"old style"* join (deprecated since ANSI SQL-92) with no condition, which results in a cross/Cartesian join:

```
SELECT *
FROM @Animal, @AnimalSound;
```

This syntax also works due to an "always true" join condition, but is not recommended and should be avoided, in favor of explicit `CROSS JOIN` syntax, for the sake of readability.

```
SELECT *
FROM
    @Animal
    JOIN @AnimalSound
        ON 1=1
```

# Section 33.6: Self Join

A table can be joined onto itself in what is known as a self join, combining records in the table with other records in the same table. Self joins are typically used in queries where a hierarchy in the table's columns is defined.

Consider the sample data in a table called `Employees`:

| ID | Name | Boss_ID |
|---|---|---|
| 1 | Bob | 3 |
| 2 | Jim | 1 |
| 3 | Sam | 2 |

Each employee's `Boss_ID` maps to another employee's `ID`. To retrieve a list of employees with their respective boss' name, the table can be joined on itself using this mapping. Note that joining a table in this manner requires the use of an alias (`Bosses` in this case) on the second reference to the table to distinguish itself from the original table.

```
SELECT Employees.Name,
    Bosses.Name AS Boss
FROM Employees
INNER JOIN Employees AS Bosses
    ON Employees.Boss_ID = Bosses.ID
```

Executing this query will output the following results:

| Name | Boss |
|---|---|
| Bob | Sam |
| Jim | Bob |
| Sam | Jim |

# Section 33.7: Accidentally turning an outer join into an inner join

Outer joins return all the rows from one or both tables, plus matching rows.

```
Table People
PersonID FirstName
       1 Alice
       2 Bob
```

```
       3 Eve

Table Scores
PersonID Subject Score
       1 Math    100
       2 Math     54
       2 Science  98
```

Left joining the tables:

```
SELECT * FROM People a
LEFT JOIN Scores b
ON a.PersonID = b.PersonID
```

Returns:

```
PersonID FirstName PersonID Subject Score
       1 Alice            1 Math    100
       2 Bob              2 Math     54
       2 Bob              2 Science  98
       3 Eve           NULL NULL    NULL
```

If you wanted to return all the people, with any applicable math scores, a common mistake is to write:

```
SELECT * FROM People a
LEFT JOIN Scores b
ON a.PersonID = b.PersonID
WHERE Subject = 'Math'
```

This would remove Eve from your results, in addition to removing Bob's science score, as `Subject` is `NULL` for her.

The correct syntax to remove non-Math records while retaining all individuals in the `People` table would be:

```
SELECT * FROM People a
LEFT JOIN Scores b
ON a.PersonID = b.PersonID
AND b.Subject = 'Math'
```

## Section 33.8: Delete using Join

Joins can also be used in a `DELETE` statement. Given a schema as follows:

```
CREATE TABLE Users (
    UserId int NOT NULL,
    AccountId int NOT NULL,
    RealName nvarchar(200) NOT NULL
)

CREATE TABLE Preferences (
    UserId int NOT NULL,
    SomeSetting bit NOT NULL
)
```

We can delete rows from the `Preferences` table, filtering by a predicate on the `Users` table as follows:

```
DELETE p
FROM Users u
```

```
INNER JOIN Preferences p ON u.UserId = p.UserId
WHERE u.AccountId = 1234
```

Here `p` is an alias for `Preferences` defined in the `FROM` clause of the statement and we only delete rows that have a matching `AccountId` from the `Users` table.

# Chapter 34: cross apply

## Section 34.1: Join table rows with dynamically generated rows from a cell

CROSS APPLY enables you to "join" rows from a table with dynamically generated rows returned by some table-value function.

Imagine that you have a Company table with a column that contains an array of products (ProductList column), and a function that parse these values and returns a set of products. You can select all rows from a Company table, apply this function on a ProductList column and "join" generated results with parent Company row:

```
SELECT *
FROM Companies c
     CROSS APPLY dbo.GetProductList( c.ProductList ) p
```

For each row, value of *ProductList* cell will be provided to the function, and the function will return those products as a set of rows that can be joined with the parent row.

## Section 34.2: Join table rows with JSON array stored in cell

CROSS APPLY enables you to "join" rows from a table with collection of JSON objects stored in a column.

Imagine that you have a Company table with a column that contains an array of products (ProductList column) formatted as JSON array. OPENJSON table value function can parse these values and return the set of products. You can select all rows from a Company table, parse JSON products with OPENJSON and "join" generated results with parent Company row:

```
SELECT *
FROM Companies c
     CROSS APPLY OPENJSON( c.ProductList )
                 WITH ( Id INT, Title nvarchar(30), Price money)
```

For each row, value of *ProductList* cell will be provided to OPENJSON function that will transform JSON objects to rows with the schema defined in WITH clause.

## Section 34.3: Filter rows by array values

If you store a list of tags in a row as coma separated values, *STRING_SPLIT* function enables you to transform list of tags into a table of values. **CROSS APPLY** enables you to "join" values parsed by *STRING_SPLIT* function with a parent row.

Imagine that you have a Product table with a column that contains an array of comma separated tags (e.g. promo,sales,new). STRING_SPLIT and CROSS APPLY enable you to join product rows with their tags so you can filter products by tags:

```
SELECT *
FROM Products p
     CROSS APPLY STRING_SPLIT( p.Tags, ',' ) tags
WHERE tags.value = 'promo'
```

For each row, value of *Tags* cell will be provided to STRING_SPLIT function that will return tag values. Then you can filter rows by these values.

**Note:** *STRING_SPLIT* function is not available before **SQL Server 2016**

# Chapter 35: Computed Columns

## Section 35.1: A column is computed from an expression

A computed column is computed from an expression that can use other columns in the same table. The expression can be a noncomputed column name, constant, function, and any combination of these connected by one or more operators.

Create table with a computed column

```
Create table NetProfit
(
    SalaryToEmployee            int,
    BonusDistributed            int,
    BusinessRunningCost         int,
    BusinessMaintenanceCost     int,
    BusinessEarnings            int,
    BusinessNetIncome
                As BusinessEarnings - (SalaryToEmployee          +
                                       BonusDistributed          +
                                       BusinessRunningCost       +
                                       BusinessMaintenanceCost    )

)
```

Value is computed and stored in the computed column automatically on inserting other values.

```
Insert Into NetProfit
    (SalaryToEmployee,
     BonusDistributed,
     BusinessRunningCost,
     BusinessMaintenanceCost,
     BusinessEarnings)
Values
    (1000000,
     10000,
     1000000,
     50000,
     2500000)
```

## Section 35.2: Simple example we normally use in log tables

```
CREATE TABLE [dbo].[ProcessLog](
[LogId] [int] IDENTITY(1,1) NOT NULL,
[LogType] [varchar](20) NULL,
[StartTime] [datetime] NULL,
[EndTime] [datetime] NULL,
[RunMinutes]  AS (datediff(minute,coalesce([StartTime],getdate()),coalesce([EndTime],getdate())))
```

This gives run difference in minutes for runtime which will be very handy..

# Chapter 36: Common Table Expressions

## Section 36.1: Generate a table of dates using CTE

```
DECLARE @startdate CHAR(8), @numberDays TINYINT

SET @startdate = '20160101'
SET @numberDays = 10;

WITH CTE_DatesTable
AS
(
  SELECT CAST(@startdate as date) AS [date]
  UNION ALL
  SELECT DATEADD(dd, 1, [date])
  FROM CTE_DatesTable
  WHERE DATEADD(dd, 1, [date]) <= DateAdd(DAY, @numberDays-1, @startdate)
)

SELECT [date] FROM CTE_DatesTable

OPTION (MAXRECURSION 0)
```

This example returns a single-column table of dates, starting with the date specified in the @startdate variable, and returning the next @numberDays worth of dates.

## Section 36.2: Employee Hierarchy

**Table Setup**

```
CREATE TABLE dbo.Employees
(
    EmployeeID INT NOT NULL PRIMARY KEY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    ManagerID INT NULL
)

GO

INSERT INTO Employees VALUES (101, 'Ken', 'Sánchez', NULL)
INSERT INTO Employees VALUES (102, 'Keith', 'Hall', 101)
INSERT INTO Employees VALUES (103, 'Fred', 'Bloggs', 101)
INSERT INTO Employees VALUES (104, 'Joseph', 'Walker', 102)
INSERT INTO Employees VALUES (105, 'Žydr?', 'Klyb?', 101)
INSERT INTO Employees VALUES (106, 'Sam', 'Jackson', 105)
INSERT INTO Employees VALUES (107, 'Peter', 'Miller', 103)
INSERT INTO Employees VALUES (108, 'Chloe', 'Samuels', 105)
INSERT INTO Employees VALUES (109, 'George', 'Weasley', 105)
INSERT INTO Employees VALUES (110, 'Michael', 'Kensington', 106)
```

**Common Table Expression**

```
;WITH cteReports (EmpID, FirstName, LastName, SupervisorID, EmpLevel) AS
(
    SELECT EmployeeID, FirstName, LastName, ManagerID, 1
    FROM Employees
    WHERE ManagerID IS NULL

    UNION ALL
```

```
    SELECT e.EmployeeID, e.FirstName, e.LastName, e.ManagerID, r.EmpLevel + 1
    FROM Employees        AS e
    INNER JOIN cteReports AS r ON e.ManagerID = r.EmpID
)

SELECT
    FirstName + ' ' + LastName AS FullName,
    EmpLevel,
    (SELECT FirstName + ' ' + LastName FROM Employees WHERE EmployeeID = cteReports.SupervisorID)
AS ManagerName
FROM cteReports
ORDER BY EmpLevel, SupervisorID
```

**Output:**

| FullName | EmpLevel | ManagerName |
|---|---|---|
| Ken Sánchez | 1 | *null* |
| Keith Hall | 2 | Ken Sánchez |
| Fred Bloggs | 2 | Ken Sánchez |
| Žydre Klybe | 2 | Ken Sánchez |
| Joseph Walker | 3 | Keith Hall |
| Peter Miller | 3 | Fred Bloggs |
| Sam Jackson | 3 | Žydre Klybe |
| Chloe Samuels | 3 | Žydre Klybe |
| George Weasley | 3 | Žydre Klybe |
| Michael Kensington | 4 | Sam Jackson |

## Section 36.3: Recursive CTE

This example shows how to get every year from this year to 2011 (2012 - 1).

```
WITH yearsAgo
(
    myYear
)
AS
(
     -- Base Case: This is where the recursion starts
     SELECT DATEPART(year, GETDATE()) AS myYear

     UNION ALL  -- This MUST be UNION ALL (cannot be UNION)

     -- Recursive Section: This is what we're doing with the recursive call
     SELECT yearsAgo.myYear - 1
     FROM yearsAgo
     WHERE yearsAgo.myYear >= 2012
)
     SELECT myYear FROM yearsAgo;  -- A single SELECT, INSERT, UPDATE, or DELETE
```

**myYear**
2016
2015
2014
2013
2012
2011

You can control the recursion (think stack overflow in code) with MAXRECURSION as a query option that will limit the number of recursive calls.

```
WITH yearsAgo
(
    myYear
)
AS
(
     -- Base Case
     SELECT DATEPART(year , GETDATE()) AS myYear
     UNION ALL
     -- Recursive Section
     SELECT yearsAgo.myYear - 1
     FROM yearsAgo
     WHERE yearsAgo.myYear >= 2002
)
     SELECT * FROM yearsAgo
     OPTION (MAXRECURSION 10);
```

> Msg 530, Level 16, State 1, Line 2The statement terminated. The maximum recursion 10 has been exhausted before statement completion.

## Section 36.4: Delete duplicate rows using CTE

Employees table :

```
|  ID  | FirstName | LastName | Gender | Salary |
+------+-----------+----------+--------+--------+
|  1   | Mark      | Hastings | Male   | 60000  |
|  1   | Mark      | Hastings | Male   | 60000  |
|  2   | Mary      | Lambeth  | Female | 30000  |
|  2   | Mary      | Lambeth  | Female | 30000  |
|  3   | Ben       | Hoskins  | Male   | 70000  |
|  3   | Ben       | Hoskins  | Male   | 70000  |
|  3   | Ben       | Hoskins  | Male   | 70000  |
+------+-----------+----------+--------+--------+
```

CTE (Common Table Expression) :

```
WITH EmployeesCTE AS
(
   SELECT *, ROW_NUMBER()OVER(PARTITION BY ID ORDER BY ID) AS RowNumber
   FROM Employees
)
DELETE FROM EmployeesCTE WHERE RowNumber > 1
```

Execution result :

```
|  ID  | FirstName | LastName | Gender | Salary |
+------+-----------+----------+--------+--------+
|  1   | Mark      | Hastings | Male   | 60000  |
|  2   | Mary      | Lambeth  | Female | 30000  |
|  3   | Ben       | Hoskins  | Male   | 70000  |
+------+-----------+----------+--------+--------+
```

## Section 36.5: CTE with multiple AS statements

```
;WITH cte_query_1
AS
(
    SELECT *
    FROM database.table1
),
cte_query_2
AS
(
    SELECT *
    FROM database.table2
)
SELECT *
FROM cte_query_1
WHERE cte_query_one.fk IN
(
    SELECT PK
    FROM cte_query_2
)
```

With common table expressions, it is possible to create multiple queries using comma-separated AS statements. A query can then reference any or all of those queries in many different ways, even joining them.

## Section 36.6: Find nth highest salary using CTE

Employees table :

```
|  ID  | FirstName | LastName | Gender | Salary |
+------+-----------+----------+--------+--------+
| 1    | Jahangir  | Alam     | Male   | 70000  |
| 2    | Arifur    | Rahman   | Male   | 60000  |
| 3    | Oli       | Ahammed  | Male   | 45000  |
| 4    | Sima      | Sultana  | Female | 70000  |
| 5    | Sudeepta  | Roy      | Male   | 80000  |
+------+-----------+----------+--------+--------+
```

CTE (Common Table Expression) :

```
 WITH RESULT AS
(
    SELECT SALARY,
           DENSE_RANK() OVER (ORDER BY SALARY DESC) AS DENSERANK
    FROM EMPLOYEES
)
SELECT TOP 1 SALARY
FROM RESULT
WHERE DENSERANK = 1
```

To find 2nd highest salary simply replace N with 2. Similarly, to find 3rd highest salary, simply replace N with 3.

# Chapter 37: Move and copy data around tables

## Section 37.1: Copy data from one table to another

This code selects data out of a table and displays it in the query tool (usually SSMS)

```
SELECT Column1, Column2, Column3 FROM MySourceTable;
```

This code inserts that data into a table:

```
INSERT INTO MyTargetTable (Column1, Column2, Column3)
SELECT Column1, Column2, Column3 FROM MySourceTable;
```

## Section 37.2: Copy data into a table, creating that table on the fly

This code selects data out of a table:

```
SELECT Column1, Column2, Column3 FROM MySourceTable;
```

This code creates a new table called `MyNewTable` and puts that data into it

```
SELECT Column1, Column2, Column3
INTO MyNewTable
FROM MySourceTable;
```

## Section 37.3: Move data into a table (assuming unique keys method)

To *move* data you first insert it into the target, then delete whatever you inserted from the source table. This is not a normal SQL operation but it may be enlightening

What did you insert? Normally in databases you need to have one or more columns that you can use to uniquely identify rows so we will assume that and make use of it.

This statement selects some rows

```
SELECT Key1, Key2, Column3, Column4 FROM MyTable;
```

First we insert these into our target table:

```
INSERT INTO TargetTable (Key1, Key2, Column3, Column4)
SELECT Key1, Key2, Column3, Column4 FROM MyTable;
```

Now *assuming records in both tables are unique* on `Key1`,`Key2`, we can use that to find and delete data out of the source table

```
DELETE MyTable
WHERE EXISTS (
    SELECT * FROM TargetTable
    WHERE TargetTable.Key1 = SourceTable.Key1
```

```
    AND TargetTable.Key2 = SourceTable.Key2
);
```

This will only work correctly if `Key1`, `Key2` are unique in both tables

Lastly, we don't want the job half done. If we wrap this up in a transaction then either all data will be moved, or nothing will happen. This ensures we don't insert the data in then find ourselves unable to delete the data out of the source.

```
BEGIN TRAN;

INSERT INTO TargetTable (Key1, Key2, Column3, Column4)
SELECT Key1, Key2, Column3, Column4 FROM MyTable;


DELETE MyTable
WHERE EXISTS (
    SELECT * FROM TargetTable
    WHERE TargetTable.Key1 = SourceTable.Key1
    AND TargetTable.Key2 = SourceTable.Key2
);

COMMIT TRAN;
```

# Chapter 38: Limit Result Set

| Parameter | Details |
|---|---|
| `TOP` | Limiting keyword. Use with a number. |
| `PERCENT` | Percentage keyword. Comes after `TOP` and limiting number. |

As database tables grow, it's often useful to limit the results of queries to a fixed number or percentage. This can be achieved using SQL Server's `TOP` keyword or `OFFSET FETCH` clause.

## Section 38.1: Limiting With PERCENT

This example limits `SELECT` result to 15 percentage of total row count.

```
SELECT TOP 15 PERCENT *
FROM TABLE_NAME
```

## Section 38.2: Limiting with FETCH

`Version ≥ SQL Server 2012`

`FETCH` is generally more useful for pagination, but can be used as an alternative to `TOP`:

```
SELECT *
FROM TABLE_NAME
ORDER BY 1
OFFSET 0 ROWS
FETCH NEXT 50 ROWS ONLY
```

## Section 38.3: Limiting With TOP

This example limits `SELECT` result to 100 rows.

```
SELECT TOP 100 *
FROM TABLE_NAME;
```

It is also possible to use a variable to specify the number of rows:

```
DECLARE @CountDesiredRows int = 100;
SELECT TOP (@CountDesiredRows) *
FROM table_name;
```

# Chapter 39: Retrieve Information about your Instance

## Section 39.1: General Information about Databases, Tables, Stored procedures and how to search them

**Query to search last executed sp's in db**

```
SELECT execquery.last_execution_time AS [DATE TIME], execsql.text AS [Script]
FROM sys.dm_exec_query_stats AS execquery
CROSS APPLY sys.dm_exec_sql_text(execquery.sql_handle) AS execsql
ORDER BY execquery.last_execution_time DESC
```

**Query to search through Stored procedures**

```
SELECT o.type_desc AS ROUTINE_TYPE,o.[name] AS ROUTINE_NAME,
m.definition AS ROUTINE_DEFINITION
FROM sys.sql_modules AS m INNER JOIN sys.objects AS o
ON m.object_id = o.object_id WHERE m.definition LIKE '%Keyword%'
ORDER BY ROUTINE_NAME
```

**Query to Find Column From All Tables of Database**

```
SELECT t.name AS TABLE_NAME,
SCHEMA_NAME(schema_id) AS schema_name,
c.name AS column_name
FROM sys.tables AS t
INNER JOIN sys.columns c ON t.OBJECT_ID = c.OBJECT_ID
WHERE c.name LIKE 'Keyword%'
ORDER BY schema_name, TABLE_NAME;
```

**Query to to check restore details**

```
WITH LastRestores AS
(
SELECT
    DatabaseName = [d].[name] ,
    [d].[create_date] ,
    [d].[compatibility_level] ,
    [d].[collation_name] ,
    r.*,
    RowNum = ROW_NUMBER() OVER (PARTITION BY d.Name ORDER BY r.[restore_date] DESC)
FROM master.sys.databases d
LEFT OUTER JOIN msdb.dbo.[restorehistory] r ON r.[destination_database_name] = d.Name
)
SELECT *
FROM [LastRestores]
WHERE [RowNum] = 1
```

**Query to to find the log**

```
SELECT top 100 * FROM databaselog
ORDER BY Posttime DESC
```

**Query to to check the Sps details**

```
SELECT name, create_date, modify_date
FROM sys.objects
WHERE TYPE = 'P'
ORDER BY modify_date DESC
```

## Section 39.2: Get information on current sessions and query executions

```
sp_who2
```

This procedure can be used to find information on current SQL server sessions. Since it is a procedure, it's often helpful to store the results into a temporary table or table variable so one can order, filter, and transform the results as needed.

The below can be used for a queryable version of `sp_who2`:

```
-- Create a variable table to hold the results of sp_who2 for querying purposes

DECLARE @who2 TABLE (
      SPID INT NULL,
      Status VARCHAR(1000) NULL,
      Login SYSNAME NULL,
      HostName SYSNAME NULL,
      BlkBy SYSNAME NULL,
      DBName SYSNAME NULL,
      Command VARCHAR(8000) NULL,
      CPUTime INT NULL,
      DiskIO INT NULL,
      LastBatch VARCHAR(250) NULL,
      ProgramName VARCHAR(250) NULL,
      SPID2 INT NULL, -- a second SPID for some reason...?
      REQUESTID INT NULL
)

INSERT INTO @who2
EXEC sp_who2

SELECT    *
FROM    @who2 w
WHERE    1=1
```

Examples:

```
-- Find specific user sessions:
SELECT  *
FROM    @who2 w
WHERE   1=1
    and  login = 'userName'

-- Find longest CPUTime queries:
SELECT  top 5 *
FROM    @who2 w
WHERE   1=1
order   by CPUTime desc
```

## Section 39.3: Information about SQL Server version

To discover SQL Server's edition, product level and version number as well as the host machine name and the server type:

```
SELECT    SERVERPROPERTY('MachineName') AS Host,
          SERVERPROPERTY('InstanceName') AS Instance,
          DB_NAME() AS DatabaseContext,
          SERVERPROPERTY('Edition') AS Edition,
          SERVERPROPERTY('ProductLevel') AS ProductLevel,
          CASE SERVERPROPERTY('IsClustered')
            WHEN 1 THEN 'CLUSTERED'
            ELSE 'STANDALONE' END AS ServerType,
          @@VERSION AS VersionNumber;
```

## Section 39.4: Retrieve Edition and Version of Instance

```
SELECT    SERVERPROPERTY('ProductVersion') AS ProductVersion,
          SERVERPROPERTY('ProductLevel') AS ProductLevel,
          SERVERPROPERTY('Edition') AS Edition,
          SERVERPROPERTY('EngineEdition') AS EngineEdition;
```

## Section 39.5: Retrieve Instance Uptime in Days

```
SELECT  DATEDIFF(DAY, login_time, getdate()) UpDays
FROM    master..sysprocesses
WHERE   spid = 1
```

## Section 39.6: Retrieve Local and Remote Servers

To retrieve a list of all servers registered on the instance:

```
EXEC sp_helpserver;
```

# Chapter 40: With Ties Option

## Section 40.1: Test Data

```
CREATE TABLE #TEST
(
Id INT,
Name VARCHAR(10)
)

Insert Into #Test
select 1,'A'
Union All
Select 1,'B'
union all
Select 1,'C'
union all
Select 2,'D'
```

Below is the output of above table,As you can see Id Column is repeated three times..

```
Id   Name
1    A
1    B
1    C
2    D
```

Now Lets check the output using simple order by..

```
SELECT Top (1)  Id,Name FROM
#test
ORDER BY Id ;
```

***Output :(Output of above query is not guaranteed to be same every time )***

```
Id   Name
1    B
```

Lets run the Same query With Ties Option..

```
SELECT Top (1) WITH Ties Id,Name
 FROM
#test
ORDER BY Id
```

***Output :***

```
Id   Name
1    A
1    B
1    C
```

As you can see SQL Server outputs all the Rows **which are tied with** Order by Column. Lets see one more Example to understand this better..

```
SELECT Top (1) WITH Ties Id,Name
```

```
 FROM
#test
ORDER BY Id ,Name
```

**Output:**

```
Id   Name
1    A
```

In Summary ,when we use with Ties Option,SQL Server Outputs all the Tied rows irrespective of limit we impose

# Chapter 41: String Functions

## Section 41.1: Quotename

Returns a Unicode string surrounded by delimiters to make it a valid SQL Server delimited identifier.

Parameters:

1. character string. A string of Unicode data, up to 128 characters (`sysname`). If an input string is longer than 128 characters function returns `null`.
2. quote character. **Optional**. A single character to use as a delimiter. Can be a single quotation mark (`'` or ``` ` ```), a left or right bracket ({,[,(,< or >,),],}) or a double quotation mark (`"`). Any other value will return null. Default value is square brackets.

```
SELECT QUOTENAME('what''s my name?')      -- Returns [what's my name?]

SELECT QUOTENAME('what''s my name?', '[') -- Returns [what's my name?]
SELECT QUOTENAME('what''s my name?', ']') -- Returns [what's my name?]

SELECT QUOTENAME('what''s my name?', '''') -- Returns 'what''s my name?'

SELECT QUOTENAME('what''s my name?', '"') -- Returns "what's my name?"

SELECT QUOTENAME('what''s my name?', ')') -- Returns (what's my name?)
SELECT QUOTENAME('what''s my name?', '(') -- Returns (what's my name?)

SELECT QUOTENAME('what''s my name?', '<') -- Returns <what's my name?>
SELECT QUOTENAME('what''s my name?', '>') -- Returns <what's my name?>

SELECT QUOTENAME('what''s my name?', '{') -- Returns {what's my name?}
SELECT QUOTENAME('what''s my name?', '}') -- Returns {what's my name?}

SELECT QUOTENAME('what''s my name?', '`') -- Returns `what's my name?`
```

## Section 41.2: Replace

Returns a string (`varchar` or `nvarchar`) where all occurrences of a specified sub string is replaced with another sub string.

Parameters:

1. string expression. This is the string that would be searched. It can be a character or binary data type.
2. pattern. This is the sub string that would be replaced. It can be a character or binary data type. The pattern argument cannot be an empty string.
3. replacement. This is the sub string that would replace the pattern sub string. It can be a character or binary data.

```
SELECT REPLACE('This is my string', 'is', 'XX') -- Returns 'ThXX XX my string'.
```

**Notes:**

- If string expression is not of type `varchar(max)` or `nvarchar(max)`, the `replace` function truncates the return value at 8,000 chars.
- Return data type depends on input data types - returns `nvarchar` if one of the input values is `nvarchar`, or `varchar` otherwise.

- Return `NULL` if any of the input parameters is `NULL`

## Section 41.3: Substring

Returns a substring that starts with the char that's in the specified start index and the specified max length.

Parameters:

1. Character expression. The character expression can be of any data type that can be implicitly converted to `varchar` or `nvarchar`, except for `text` or `ntext`.
2. Start index. A number (`int` or `bigint`) that specifies the start index of the requested substring. (**Note:** strings in sql server are base 1 index, meaning that the first character of the string is index 1). This number can be less then 1. In this case, If the sum of start index and max length is greater then 0, the return string would be a string starting from the first char of the character expression and with the length of (start index + max length - 1). If it's less then 0, an empty string would be returned.
3. Max length. An integer number between 0 and `bigint` max value (9,223,372,036,854,775,807). If the max length parameter is negative, an error will be raised.

```
SELECT SUBSTRING('This is my string', 6, 5) -- returns 'is my'
```

If the max length + start index is more then the number of characters in the string, the entier string is returned.

```
SELECT SUBSTRING('Hello World',1,100) -- returns 'Hello World'
```

If the start index is bigger then the number of characters in the string, an empty string is returned.

```
SELECT SUBSTRING('Hello World',15,10) -- returns ''
```

## Section 41.4: String_Split

`Version ≥ SQL Server 2016`

Splits a string expression using a character separator. Note that `STRING_SPLIT()` is a table-valued function and therefore must be used within `FROM` clause.

Parameters:

1. string. Any character type expression (`char`, `nchar`, `varchar` or `nvarchar`)
2. seperator. A single character expression of any type (`char(1)`, `nchar(1)`, `varchar(1)` or `nvarchar(1)`).

Returns a single column table where each row contains a fragment of the string. The name of the columns is `value`, and the datatype is `nvarchar` if any of the parameters is either `nchar` or `nvarchar`, otherwise `varchar`.

The following example splits a string using space as a separator:

```
SELECT VALUE FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

Result:

```
value
-----
Lorem
ipsum
dolor
sit
```

```
amet.
```

Remarks:

> The `STRING_SPLIT` function is available only under compatibility level **130**. If your database compatibility level is lower than 130, SQL Server will not be able to find and execute `STRING_SPLIT` function. You can change the compatibility level of a database using the following command:

```
ALTER DATABASE [database_name] SET COMPATIBILITY_LEVEL = 130
```

```
Version < SQL Server 2016
```

Older versions of sql server does not have a built in split string function. There are many user defined functions that handles the problem of splitting a string. You can read Aaron Bertrand's article Split strings the right way – or the next best way for a comprehensive comparison of some of them.

## Section 41.5: Left

Returns a sub string starting with the left most char of a string and up to the maximum length specified.

Parameters:

1. character expression. The character expression can be of any data type that can be implicitly converted to `varchar` or `nvarchar`, except for `text` or `ntext`
2. max length. An integer number between 0 and `bigint` max value (9,223,372,036,854,775,807).
   If the max length parameter is negative, an error will be raised.

```
SELECT LEFT('This is my string', 4) -- result: 'This'
```

If the max length is more then the number of characters in the string, the entier string is returned.

```
SELECT LEFT('This is my string', 50) -- result: 'This is my string'
```

## Section 41.6: Right

Returns a sub string that is the right most part of the string, with the specified max length.

Parameters:

1. character expression. The character expression can be of any data type that can be implicitly converted to `varchar` or `nvarchar`, except for `text` or `ntext`
2. max length. An integer number between 0 and `bigint` max value (9,223,372,036,854,775,807). If the max length parameter is negative, an error will be raised.

```
SELECT RIGHT('This is my string', 6) -- returns 'string'
```

If the max length is more then the number of characters in the string, the entier string is returned.

```
SELECT RIGHT('This is my string', 50) -- returns 'This is my string'
```

# Section 41.7: Soundex

Returns a four-character code (`varchar`) to evaluate the phonetic similarity of two strings.

Parameters:

1. character expression. An alphanumeric expression of character data.

The soundex function creates a four-character code that is based on how the character expression would sound when spoken. the first char is the the upper case version of the first character of the parameter, the rest 3 characters are numbers representing the letters in the expression (except a, e, i, o, u, h, w and y that are ignored).

```
SELECT SOUNDEX ('Smith') -- Returns 'S530'

SELECT SOUNDEX ('Smythe') -- Returns 'S530'
```

# Section 41.8: Format

Version ≥ SQL Server 2012

Returns a `NVARCHAR` value formatted with the specified format and culture (if specified). This is primarily used for converting date-time types to strings.

Parameters:

1. `value`. An expression of a supported data type to format. valid types are listed below.
2. `format`. An `NVARCHAR` format pattern. See Microsoft official documentation for standard and custom format strings.
3. `culture`. **Optional**. `nvarchar` argument specifying a culture. The default value is the culture of the current session.

**DATE**

Using standard format strings:

```
DECLARE @d DATETIME = '2016-07-31';

SELECT
    FORMAT ( @d, 'd', 'en-US' ) AS 'US English Result' -- Returns '7/31/2016'
   ,FORMAT ( @d, 'd', 'en-gb' ) AS 'Great Britain English Result' -- Returns '31/07/2016'
   ,FORMAT ( @d, 'd', 'de-de' ) AS 'German Result' -- Returns '31.07.2016'
   ,FORMAT ( @d, 'd', 'zh-cn' ) AS 'Simplified Chinese (PRC) Result' -- Returns '2016/7/31'
   ,FORMAT ( @d, 'D', 'en-US' ) AS 'US English Result' -- Returns 'Sunday, July 31, 2016'
   ,FORMAT ( @d, 'D', 'en-gb' ) AS 'Great Britain English Result' -- Returns '31 July 2016'
   ,FORMAT ( @d, 'D', 'de-de' ) AS 'German Result' -- Returns 'Sonntag, 31. Juli 2016'
```

Using custom format strings:

```
SELECT FORMAT( @d, 'dd/MM/yyyy', 'en-US' ) AS 'DateTime Result' -- Returns '31/07/2016'
      ,FORMAT(123456789,'###-##-####') AS 'Custom Number Result' -- Returns '123-45-6789',
      ,FORMAT( @d,'dddd, MMMM dd, yyyy hh:mm:ss tt','en-US') AS 'US' -- Returns 'Sunday, July 31,
2016 12:00:00 AM'
      ,FORMAT( @d,'dddd, MMMM dd, yyyy hh:mm:ss tt','hi-IN') AS 'Hindi' -- Returns ??????, ?????
31, 2016 12:00:00 ?????????
      ,FORMAT ( @d, 'dddd', 'en-US' )  AS 'US' -- Returns 'Sunday'
      ,FORMAT ( @d, 'dddd', 'hi-IN' )  AS 'Hindi' -- Returns '??????'
```

`FORMAT` can also be used for formatting `CURRENCY`,`PERCENTAGE` and `NUMBERS`.

**CURRENCY**

```
DECLARE @Price1 INT = 40
SELECT FORMAT(@Price1,'c','en-US') AS 'CURRENCY IN US Culture' -- Returns '$40.00'
       ,FORMAT(@Price1,'c','de-DE') AS 'CURRENCY IN GERMAN Culture' -- Returns '40,00 €'
```

We can specify the number of digits after the decimal.

```
DECLARE @Price DECIMAL(5,3) = 40.356
SELECT FORMAT( @Price, 'C') AS 'Default', -- Returns '$40.36'
       FORMAT( @Price, 'C0') AS 'With 0 Decimal', -- Returns '$40'
       FORMAT( @Price, 'C1') AS 'With 1 Decimal', -- Returns '$40.4'
       FORMAT( @Price, 'C2') AS 'With 2 Decimal', -- Returns '$40.36'
```

**PERCENTAGE**

```
   DECLARE @Percentage float = 0.35674
   SELECT FORMAT( @Percentage, 'P') AS '% Default', -- Returns '35.67 %'
   FORMAT( @Percentage, 'P0') AS '% With 0 Decimal', -- Returns '36 %'
   FORMAT( @Percentage, 'P1') AS '% with 1 Decimal'  -- Returns '35.7 %'
```

**NUMBER**

```
DECLARE @Number AS DECIMAL(10,2) = 454545.389
SELECT FORMAT( @Number, 'N','en-US') AS 'Number Format in US', -- Returns '454,545.39'
FORMAT( @Number, 'N','en-IN')  AS 'Number Format in INDIA', -- Returns '4,54,545.39'
FORMAT( @Number, '#.0')     AS 'With 1 Decimal', -- Returns '454545.4'
FORMAT( @Number, '#.00')    AS 'With 2 Decimal', -- Returns '454545.39'
FORMAT( @Number, '#,##.00') AS 'With Comma and 2 Decimal', -- Returns '454,545.39'
FORMAT( @Number, '##.00')   AS 'Without Comma and 2 Decimal', -- Returns '454545.39'
FORMAT( @Number, '000000000') AS 'Left-padded to nine digits' -- Returns '000454545'
```

Valid value types list: (source)

```
Category         Type             .Net type
-----------------------------------------
Numeric          bigint           Int64
Numeric          int              Int32
Numeric          smallint         Int16
Numeric          tinyint          Byte
Numeric          decimal          SqlDecimal
Numeric          numeric          SqlDecimal
Numeric          float            Double
Numeric          real             Single
Numeric          smallmoney       Decimal
Numeric          money            Decimal
Date and Time    date             DateTime
Date and Time    time             TimeSpan
Date and Time    datetime         DateTime
Date and Time    smalldatetime    DateTime
Date and Time    datetime2        DateTime
Date and Time    datetimeoffset   DateTimeOffset
```

**Important Notes:**

- `FORMAT` returns `NULL` for errors other than a culture that is not valid. For example, `NULL` is returned if the value

specified in format is not valid.
- `FORMAT` relies on the presence of the .NET Framework Common Language Runtime (CLR).
- `FORMAT` relies upon CLR formatting rules which dictate that colons and periods must be escaped. Therefore, when the format string (second parameter) contains a colon or period, the colon or period must be escaped with backslash when an input value (first parameter) is of the time data type.

See also Date & Time Formatting using FORMAT documentation example.

## Section 41.9: String_escape

`Version ≥ SQL Server 2016`

Escapes special characters in texts and returns text (`nvarchar(max)`) with escaped characters.

Parameters:

1. text. is a `nvarchar` expression representing the string that should be escaped.
2. type. Escaping rules that will be applied. Currently the only supported value is `'json'`.

```
SELECT STRING_ESCAPE('\  /
\\    "     ', 'json') -- returns '\\\t\/\n\\\\\t\"\t'
```

List of characters that will be escaped:

```
Special character    Encoded sequence
-------------------------------------
Quotation mark (")   \"
Reverse solidus (\)  \\
Solidus (/)          \/
Backspace            \b
Form feed            \f
New line             \n
Carriage return      \r
Horizontal tab       \t


Control character    Encoded sequence
-------------------------------------
CHAR(0)              \u0000
CHAR(1)              \u0001
...                  ...
CHAR(31)             \u001f
```

## Section 41.10: ASCII

Returns an int value representing the ASCII code of the leftmost character of a string.

```
SELECT ASCII('t') -- Returns 116
SELECT ASCII('T') -- Returns 84
SELECT ASCII('This') -- Returns 84
```

If the string is Unicode and the leftmost character is not ASCII but representable in the current collation, a value greater than 127 can be returned:

```
SELECT ASCII(N'ï') -- returns 239 when `SERVERPROPERTY('COLLATION') =
'SQL_Latin1_General_CP1_CI_AS'`
```

If the string is Unicode and the leftmost character cannot be represented in the current collation, the int value of 63 is returned: (which represents question mark in ASCII):

```
SELECT ASCII(N'?') -- returns 63
SELECT ASCII(NCHAR(2039)) -- returns 63
```

## Section 41.11: Char

Returns a char represented by an int ASCII code.

```
SELECT CHAR(116) -- Returns 't'
SELECT CHAR(84)  -- Returns 'T'
```

This can be used to introduce new line/line feed `CHAR(10)`, carriage returns `CHAR(13)`, etc. See AsciiTable.com for reference.

If the argument value is not between 0 and 255, the CHAR function returns `NULL`.
The return data type of the `CHAR` function is `char(1)`

## Section 41.12: Concat

`Version ≥ SQL Server 2012`

Returns a string that is the result of two or more strings joined together. `CONCAT` accepts two or more arguments.

```
SELECT CONCAT('This', ' is', ' my', ' string') -- returns 'This is my string'
```

Note: Unlike concatenating strings using the string concatenation operator (+), when passing a null value to the `concat` function it will implicitly convert it to an empty string:

```
SELECT CONCAT('This', NULL, ' is', ' my', ' string'), -- returns 'This is my string'
       'This' + NULL + ' is' + ' my' + ' string' -- returns NULL.
```

Also arguments of a non-string type will be implicitly converted to a string:

```
SELECT CONCAT('This', ' is my ', 3, 'rd string') -- returns 'This is my 3rd string'
```

Non-string type variables will also be converted to string format, no need to manually covert or cast it to string:

```
DECLARE @Age INT=23;
SELECT CONCAT('Ram is ', @Age,' years old');  -- returns 'Ram is 23 years old'
```

`Version < SQL Server 2012`

Older versions do not support `CONCAT` function and must use the string concatenation operator (+) instead. Non-string types must be cast or converted to string types in order to concatenate them this way.

```
SELECT 'This is the number ' + CAST(42 AS VARCHAR(5)) --returns 'This is the number 42'
```

## Section 41.13: LTrim

Returns a character expression (`varchar` or `nvarchar`) after removing all leading white spaces, i.e., white spaces

from the left through to the first non-white space character.

Parameters:

1. character expression. Any expression of character or binary data that can be implicitly converted to `varcher`, except `text`, `ntext` and `image`.

```
SELECT LTRIM('    This is my string') -- Returns 'This is my string'
```

## Section 41.14: RTrim

Returns a character expression (`varchar` or `nvarchar`) after removing all trailing white spaces, i.e., spaces from the right end of the string up until the first non-white space character to the left.

Parameters:

1. character expression. Any expression of character or binary data that can be implicitly converted to `varcher`, except `text`, `ntext` and `image`.

```
SELECT RTRIM('This is my string     ') -- Returns 'This is my string'
```

## Section 41.15: PatIndex

Returns the starting position of the first occurrence of a the specified pattern in the specified expression.

Parameters:

1. pattern. A character expression the contains the sequence to be found. Limited to A maximum length of 8000 chars. Wildcards (`%`, `_`) can be used in the pattern. If the pattern does not start with a wildcard, it may only match whatever is in the beginning of the expression. If it doesn't end with a wildcard, it may only match whatever is in the end of the expression.
2. expression. Any string data type.

```
SELECT PATINDEX('%ter%', 'interesting') -- Returns 3.

SELECT PATINDEX('%t_r%t%', 'interesting') -- Returns 3.

SELECT PATINDEX('ter%', 'interesting') -- Returns 0, since 'ter' is not at the start.

SELECT PATINDEX('inter%', 'interesting') -- Returns 1.

SELECT PATINDEX('%ing', 'interesting') -- Returns 9.
```

## Section 41.16: Space

Returns a string (`varchar`) of repeated spaces.

Parameters:

1. integer expression. Any integer expression, up to 8000. If negative, `null` is returned. if 0, an empty string is returned. (To return a string longer then 8000 spaces, use Replicate.

```
SELECT SPACE(-1) -- Returns NULL
SELECT SPACE(0)  -- Returns an empty string
```

```
SELECT SPACE(3)  -- Returns '   ' (a string containing 3 spaces)
```

# Section 41.17: Difference

Returns an integer (`int`) value that indicates the difference between the soundex values of two character expressions.

Parameters:

1. character expression 1.
2. character expression 2.

Both parameters are alphanumeric expressions of character data.

The integer returned is the number of chars in the soundex values of the parameters that are the same, so 4 means that the expressions are very similar and 0 means that they are very different.

```
SELECT  SOUNDEX('Green'),  -- G650
        SOUNDEX('Greene'),  -- G650
        DIFFERENCE('Green','Greene') -- Returns 4

SELECT  SOUNDEX('Blotchet-Halls'),  -- B432
        SOUNDEX('Greene'),  -- G650
        DIFFERENCE('Blotchet-Halls', 'Greene') -- Returns 0
```

# Section 41.18: Len

Returns the number of characters of a string.
Note: the `LEN` function ignores trailing spaces:

```
SELECT LEN('My string'), -- returns 9
       LEN('My string   '), -- returns 9
       LEN('   My string') -- returns 12
```

If the length including trailing spaces is desired there are several techniques to achieve this, although each has its drawbacks. One technique is to append a single character to the string, and then use the `LEN` minus one:

```
DECLARE @str varchar(100) = 'My string   '
SELECT LEN(@str + 'x') - 1 -- returns 12
```

The drawback to this is if the type of the string variable or column is of the maximum length, the append of the extra character is discarded, and the resulting length will still not count trailing spaces. To address that, the following modified version solves the problem, and gives the correct results in all cases at the expense of a small amount of additional execution time, and because of this (correct results, including with surrogate pairs, and reasonable execution speed) appears to be the best technique to use:

```
SELECT LEN(CONVERT(NVARCHAR(MAX), @str) + 'x') - 1
```

Another technique is to use the`DATALENGTH` function.

```
DECLARE @str varchar(100) = 'My string   '
SELECT DATALENGTH(@str) -- returns 12
```

It's important to note though that `DATALENGTH` returns the length in bytes of the string in memory. This will be different for `varchar` vs. `nvarchar`.

```
DECLARE @str nvarchar(100) = 'My string   '
SELECT DATALENGTH(@str) -- returns 24
```

You can adjust for this by dividing the datalength of the string by the datalength of a single character (which must be of the same type). The example below does this, and also handles the case where the target string happens to be empty, thus avoiding a divide by zero.

```
DECLARE @str nvarchar(100) = 'My string   '
SELECT DATALENGTH(@str) / DATALENGTH(LEFT(LEFT(@str, 1) + 'x', 1)) -- returns 12
```

Even this, though, has a problem in SQL Server 2012 and above. It will produce incorrect results when the string contains surrogate pairs (some characters can occupy more bytes than other characters in the same string).

Another technique is to use `REPLACE` to convert spaces to a non-space character, and take the `LEN` of the result. This gives correct results in all cases, but has very poor execution speed with long strings.

# Section 41.19: Lower

Returns a character expression (`varchar` or `nvarchar`) after converting all uppercase characters to lowercase.

Parameters:

1. Character expression. Any expression of character or binary data that can be implicitly converted to `varchar`.

```
SELECT LOWER('This IS my STRING') -- Returns 'this is my string'

DECLARE @String NCHAR(17) = N'This IS my STRING';
SELECT LOWER(@String) -- Returns 'this is my string'
```

# Section 41.20: Upper

Returns a character expression (`varchar` or `nvarchar`) after converting all lowercase characters to uppercase.

Parameters:

1. Character expression. Any expression of character or binary data that can be implicitly converted to `varchar`.

```
SELECT UPPER('This IS my STRING') -- Returns 'THIS IS MY STRING'

DECLARE @String NCHAR(17) = N'This IS my STRING';
SELECT UPPER(@String) -- Returns 'THIS IS MY STRING'
```

# Section 41.21: Unicode

Returns the integer value representing the Unicode value of the first character of the input expression.

Parameters:

1. Unicode character expression. Any valid `nchar` or `nvarchar` expression.

```
SELECT UNICODE(N'?') -- Returns 400

DECLARE @Unicode nvarchar(11) = N'? is a char'
SELECT UNICODE(@Unicode) -- Returns 400
```

# Section 41.22: NChar

Returns the Unicode character(s) (`nchar(1)` or `nvarchar(2)`) corresponding to the integer argument it receives, as defined by the Unicode standard.

Parameters:

1. integer expression. Any integer expression that is a positive number between 0 and 65535, or if the collation of the database supports supplementary character (CS) flag, the supported range is between 0 to 1114111. If the integer expression does not fall inside this range, `null` is returned.

```
SELECT NCHAR(257) -- Returns '?'
SELECT NCHAR(400) -- Returns '?'
```

# Section 41.23: Str

Returns character data (`varchar`) converted from numeric data.

Parameters:

1. float expression. An approximate numeric data type with a decimal point.
2. length. **optional.** The total length of the string expression that would return, including digits, decimal point and leading spaces (if needed). The default value is 10.
3. decimal. **optional.** The number of digits to the right of the decimal point. If higher then 16, the result would be truncated to sixteen places to the right of the decimal point.

```
SELECT STR(1.2) -- Returns '         1'

SELECT STR(1.2, 3) -- Returns '  1'

SELECT STR(1.2, 3, 2) -- Returns '1.2'

SELECT STR(1.2, 5, 2) -- Returns ' 1.20'

SELECT STR(1.2, 5, 5) -- Returns '1.200'

SELECT STR(1, 5, 2) -- Returns ' 1.00'

SELECT STR(1) -- Returns '         1'
```

# Section 41.24: Reverse

Returns a string value in reversed order.

Parameters:

1. string expression. Any string or binary data that can be implicitly converted to `varchar`.

```
SELECT REVERSE('Sql Server') -- Returns 'revreS lqS'
```

# Section 41.25: Replicate

Repeats a string value a specified number of times.

Parameters:

1. string expression. String expression can be a character string or binary data.
2. integer expression. Any integer type, including `bigint`. If negative, `null` is returned. If 0, an empty string is returned.

```
SELECT REPLICATE('a', -1)  -- Returns NULL

SELECT REPLICATE('a', 0)  -- Returns ''

SELECT REPLICATE('a', 5)  -- Returns 'aaaaa'

SELECT REPLICATE('Abc', 3) -- Returns 'AbcAbcAbc'
```

**Note:** If string expression is not of type `varchar(max)` or `nvarchar(max)`, the return value will not exceed 8000 chars. Replicate will stop before adding the string that will cause the return value to exceed that limit:

```
SELECT LEN(REPLICATE('a b c d e f g h i j k l', 350)) -- Returns 7981

SELECT LEN(REPLICATE(CAST('a b c d e f g h i j k l' AS VARCHAR(MAX)), 350)) -- Returns 8050
```

# Section 41.26: CharIndex

Returns the start index of a the first occurrence of string expression inside another string expression.

Parameters list:

1. String to find (up to 8000 chars)
2. String to search (any valid character data type and length, including binary)
3. (Optional) index to start. A number of type int or big int. If omitted or less then 1, the search starts at the beginning of the string.

If the string to search is `varchar(max)`, `nvarchar(max)` or `varbinary(max)`, the `CHARINDEX` function will return a `bigint` value. Otherwise, it will return an `int`.

```
SELECT CHARINDEX('is', 'this is my string') -- returns 3
SELECT CHARINDEX('is', 'this is my string', 4) -- returns 6
SELECT CHARINDEX(' is', 'this is my string') -- returns 5
```

# Chapter 42: Logical Functions

## Section 42.1: CHOOSE

Version ≥ SQL Server 2012

Returns the item at the specified index from a list of values. If `index` exceeds the bounds of `values` then `NULL` is returned.

Parameters:

1. `index`: integer, index to item in `values`. 1-based.
2. `values`: any type, comma separated list

```
SELECT CHOOSE (1, 'apples', 'pears', 'oranges', 'bananas') AS chosen_result

chosen_result
-------------
apples
```

## Section 42.2: IIF

Version ≥ SQL Server 2012

Returns one of two values, depending on whether a given Boolean expression evaluates to true or false.

Parameters:

1. `boolean_expression` evaluated to dtermine what value to return
2. `true_value` returned if `boolean_expression` evaluates to true
3. `false_value` returned if `boolean_expression` evaluates to false

```
SELECT IIF (42 > 23, 'I knew that!', 'That is not true.') AS iif_result

iif_result
------------
I knew that!
```

Version < SQL Server 2012

`IIF` may be replaced by a `CASE` statement. The above example my be written as

```
SELECT CASE WHEN 42 > 23 THEN 'I knew that!' ELSE 'That is not true.' END AS iif_result

iif_result
------------
I knew that!
```

# Chapter 43: Aggregate Functions

Aggregate functions in SQL Server run calculations on sets of values, returning a single value.

## Section 43.1: SUM()

Returns sum of numeric values in a given column.

We have table as shown in figure that will be used to perform different aggregate functions. The table name is *Marksheet*.

| SubjectCode | SubjectName | MarksObtained |
|---|---|---|
| 101 | Physics | 87 |
| 102 | Chemistry | 75 |
| 103 | Maths | 85 |
| 104 | English | 89 |
| 105 | Computer | 95 |

```
SELECT SUM(MarksObtained) FROM Marksheet
```

The `sum` function doesn't consider rows with NULL value in the field used as parameter

In the above example if we have another row like this:

```
106    Italian    NULL
```

This row will not be consider in sum calculation

## Section 43.2: AVG()

Returns average of numeric values in a given column.

We have table as shown in figure that will be used to perform different aggregate functions. The table name is *Marksheet*.

| SubjectCode | SubjectName | MarksObtained |
|---|---|---|
| 101 | Physics | 87 |
| 102 | Chemistry | 75 |
| 103 | Maths | 85 |
| 104 | English | 89 |
| 105 | Computer | 95 |

```
SELECT AVG(MarksObtained) FROM Marksheet
```

The `average` function doesn't consider rows with NULL value in the field used as parameter

In the above example if we have another row like this:

```
106    Italian    NULL
```

This row will not be consider in average calculation

## Section 43.3: MAX()

Returns the largest value in a given column.

We have table as shown in figure that will be used to perform different aggregate functions. The table name is *Marksheet*.

| SubjectCode | SubjectName | MarksObtained |
|---|---|---|
| 101 | Physics | 87 |
| 102 | Chemistry | 75 |
| 103 | Maths | 85 |
| 104 | English | 89 |
| 105 | Computer | 95 |

```
SELECT MAX(MarksObtained) FROM Marksheet
```

## Section 43.4: MIN()

Returns the smallest value in a given column.

We have table as shown in figure that will be used to perform different aggregate functions. The table name is *Marksheet*.

| SubjectCode | SubjectName | MarksObtained |
|---|---|---|
| 101 | Physics | 87 |
| 102 | Chemistry | 75 |
| 103 | Maths | 85 |
| 104 | English | 89 |
| 105 | Computer | 95 |

```
SELECT MIN(MarksObtained) FROM Marksheet
```

## Section 43.5: COUNT()

Returns the total number of values in a given column.

We have table as shown in figure that will be used to perform different aggregate functions. The table name is *Marksheet*.

| SubjectCode | SubjectName | MarksObtained |
|---|---|---|
| 101 | Physics | 87 |
| 102 | Chemistry | 75 |
| 103 | Maths | 85 |
| 104 | English | 89 |
| 105 | Computer | 95 |

```
SELECT COUNT(MarksObtained) FROM Marksheet
```

The `count` function doesn't consider rows with NULL value in the field used as parameter. Usually the count parameter is * (all fields) so only if all fields of row are NULLs this row will not be considered

In the above example if we have another row like this:

```
106    Italian    NULL
```

This row will not be consider in count calculation

**NOTE**

The function `COUNT(*)` returns the number of rows in a table. This value can also be obtained by using a constant non-null expression that contains no column references, such as `COUNT(1)`.

Example

```
SELECT COUNT(1) FROM Marksheet
```

## Section 43.6: COUNT(Column_Name) with GROUP BY Column_Name

Most of the time we like to get the total number of occurrence of a column value in a table for example:

TABLE NAME : REPORTS

| ReportName | ReportPrice |
|---|---|
| Test | 10.00 $ |
| Test | 10.00 $ |
| Test | 10.00 $ |
| Test 2 | 11.00 $ |
| Test | 10.00 $ |
| Test 3 | 14.00 $ |
| Test 3 | 14.00 $ |
| Test 4 | 100.00 $ |

```
SELECT
    ReportName AS REPORT NAME,
    COUNT(ReportName) AS COUNT
FROM
    REPORTS
GROUP BY
    ReportName
```

| REPORT NAME | COUNT |
|---|---|
| Test | 4 |
| Test 2 | 1 |
| Test 3 | 2 |
| Test 4 | 1 |

# Chapter 44: String Aggregate functions in SQL Server

## Section 44.1: Using STUFF for string aggregation

We have a Student table with SubjectId. Here the requirement is to concatenate based on subjectId.

All SQL Server versions

```
create table #yourstudent (subjectid int, studentname varchar(10))

insert into #yourstudent (subjectid, studentname) values
 ( 1       ,'Mary'    )
,( 1       ,'John'    )
,( 1       ,'Sam'     )
,( 2       ,'Alaina')
,( 2       ,'Edward')

select subjectid,  stuff(( select concat( ',', studentname) from #yourstudent y where y.subjectid =
u.subjectid for xml path('')),1,1, '')
    from #yourstudent u
    group by subjectid
```

## Section 44.2: String_Agg for String Aggregation

In case of SQL Server 2017 or vnext we can use in-built STRING_AGG for this aggregation. For same student table,

```
create table #yourstudent (subjectid int, studentname varchar(10))

insert into #yourstudent (subjectid, studentname) values
 ( 1       ,'Mary'    )
,( 1       ,'John'    )
,( 1       ,'Sam'     )
,( 2       ,'Alaina')
,( 2       ,'Edward')

select subjectid, string_agg(studentname, ',') from #yourstudent
    group by subjectid
```

# Chapter 45: Ranking Functions

| Arguments | Details |
|---|---|
| `<partition_by_clause>` | Divides the result set produced by the FROM clause into partitions to which the DENSE_RANK function is applied. For the `PARTITION BY` syntax, see OVER Clause (Transact-SQL). |
| `<order_by_clause>` | Determines the order in which the DENSE_RANK function is applied to the rows in a partition. |
| `OVER ( [ partition_by_clause ] order_by_clause)` | `partition_by_clause` divides the result set produced by the `FROM` clause into partitions to which the function is applied. If not specified, the function treats all rows of the query result set as a single group. order_by_clause determines the order of the data before the function is applied. The order_by_clause is required. The `<rows or range clause>` of the `OVER` clause cannot be specified for the `RANK` function. For more information, see OVER Clause (Transact-SQL). |

## Section 45.1: DENSE_RANK ()

Same as that of RANK(). It returns rank without any gaps:

```
SELECT  Studentid, Name,Subject,Marks,
DENSE_RANK() OVER(partition BY name ORDER BY Marks DESC)Rank
FROM Exam
ORDER BY name

Studentid    Name    Subject    Marks    Rank
101          Ivan    Science    80       1
101          Ivan    Maths      70       2
101          Ivan    Social     60       3
102          Ryan    Social     70       1
102          Ryan    Maths      60       2
102          Ryan    Science    50       3
103          Tanvi   Maths      90       1
103          Tanvi   Science    90       1
103          Tanvi   Social     80       2
```

## Section 45.2: RANK()

A RANK() Returns the rank of each row in the result set of partitioned column.

Eg :

```
SELECT Studentid,Name,Subject,Marks,
RANK() OVER(partition BY name ORDER BY Marks DESC)Rank
FROM Exam
ORDER BY name,subject

   Studentid    Name    Subject    Marks    Rank

    101         Ivan    Maths       70       2
    101         Ivan    Science     80       1
    101         Ivan    Social      60       3
    102         Ryan    Maths       60       2
    102         Ryan    Science     50       3
    102         Ryan    Social      70       1
    103         Tanvi   Maths       90       1
    103         Tanvi   Science     90       1
    103         Tanvi   Social      80       3
```

# Chapter 46: Window functions

## Section 46.1: Centered Moving Average

Calculate a 6-month (126-business-day) centered moving average of a price:

```
SELECT TradeDate, AVG(Px) OVER (ORDER BY TradeDate ROWS BETWEEN 63 PRECEDING AND 63 FOLLOWING) AS
PxMovingAverage
FROM HistoricalPrices
```

Note that, because it will take *up to* 63 rows before and after each returned row, at the beginning and end of the TradeDate range it will not be centered: When it reaches the largest TradeDate it will only be able to find 63 preceding values to include in the average.

## Section 46.2: Find the single most recent item in a list of timestamped events

In tables recording events there is often a datetime field recording the time an event happened. Finding the single most recent event can be difficult because it's always possible that two events were recorded with exactly identical timestamps. You can use row_number() over (order by ...) to make sure all records are uniquely ranked, and select the top one (where my_ranking=1)

```
SELECT *
FROM (
    SELECT
        *,
        ROW_NUMBER() OVER (ORDER BY crdate DESC) AS my_ranking
    FROM sys.sysobjects
) g
WHERE my_ranking=1
```

This same technique can be used to return a single row from any dataset with potentially duplicate values.

## Section 46.3: Moving Average of last 30 Items

Moving Average of last 30 Items sold

```
SELECT
    value_column1,
    (   SELECT
            AVG(value_column1) AS moving_average
        FROM Table1 T2
        WHERE ( SELECT
                    COUNT(*)
                FROM Table1 T3
                WHERE date_column1 BETWEEN T2.date_column1 AND T1.date_column1
                ) BETWEEN 1 AND 30
    ) AS MovingAvg
FROM Table1 T1
```

# Chapter 47: PIVOT / UNPIVOT

## Section 47.1: Dynamic PIVOT

One problem with the `PIVOT` query is that you have to specify all values inside the `IN` selection if you want to see them as columns. A quick way to circumvent this problem is to create a dynamic IN selection making your `PIVOT` dynamic.

For demonstration we will use a table `Books` in a `Bookstore`'s database. We assume that the table is quite de-normalised and has following columns

```
Table: Books
----------------------------
BookId (Primary Key Column)
Name
Language
NumberOfPages
EditionNumber
YearOfPrint
YearBoughtIntoStore
ISBN
AuthorName
Price
NumberOfUnitsSold
```

Creation script for the table will be like:

```
CREATE TABLE [dbo].[BookList](
      [BookId] [int] NOT NULL,
      [Name] [nvarchar](100) NULL,
      [Language] [nvarchar](100) NULL,
      [NumberOfPages] [int] NULL,
      [EditionNumber] [nvarchar](10) NULL,
      [YearOfPrint] [int] NULL,
      [YearBoughtIntoStore] [int] NULL,
[NumberOfBooks] [int] NULL,
[ISBN] [nvarchar](30) NULL,
      [AuthorName] [nvarchar](200) NULL,
      [Price] [money] NULL,
      [NumberOfUnitsSold] [int] NULL,
 CONSTRAINT [PK_BookList] PRIMARY KEY CLUSTERED
(
      [BookId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]

GO
```

Now if we need to query on the database and figure out number of books in English, Russian, German, Hindi, Latin languages bought into the bookstore every year and present our output in a small report format, we can use PIVOT query like this

```
SELECT * FROM
  (
   SELECT YearBoughtIntoStore AS [YEAR Bought],[LANGUAGE], NumberOfBooks
   FROM BookList
```

```
  ) sourceData
 PIVOT
  (
  SUM(NumberOfBooks)
  FOR [LANGUAGE] IN (English, Russian, German, Hindi, Latin)
  ) pivotrReport
```

Special case is when we do not have a full list of the languages, so we'll use dynamic SQL like below

```
DECLARE @query VARCHAR(4000)
DECLARE @languages VARCHAR(2000)
SELECT @languages =
        STUFF((SELECT DISTINCT '],['+LTRIM([Language])FROM [dbo].[BookList]
        ORDER BY '],['+LTRIM([Language]) FOR XML PATH('') ),1,2,'') + ']'
SET @query=
'SELECT * FROM
  (SELECT YearBoughtIntoStore AS [Year Bought],[Language],NumberOfBooks
   FROM BookList) sourceData
PIVOT(SUM(NumberOfBooks)FOR [Language] IN ('+ @languages +')) pivotrReport' EXECUTE(@query)
```

# Section 47.2: Simple PIVOT & UNPIVOT (T-SQL)

Below is a simple example which shows average item's price of each item per weekday.

First, suppose we have a table which keeps daily records of all items' prices.

```
CREATE TABLE tbl_stock(item NVARCHAR(10), weekday NVARCHAR(10), price INT);

INSERT INTO tbl_stock VALUES
('Item1', 'Mon', 110), ('Item2', 'Mon', 230), ('Item3', 'Mon', 150),
('Item1', 'Tue', 115), ('Item2', 'Tue', 231), ('Item3', 'Tue', 162),
('Item1', 'Wed', 110), ('Item2', 'Wed', 240), ('Item3', 'Wed', 162),
('Item1', 'Thu', 109), ('Item2', 'Thu', 228), ('Item3', 'Thu', 145),
('Item1', 'Fri', 120), ('Item2', 'Fri', 210), ('Item3', 'Fri', 125),
('Item1', 'Mon', 122), ('Item2', 'Mon', 225), ('Item3', 'Mon', 140),
('Item1', 'Tue', 110), ('Item2', 'Tue', 235), ('Item3', 'Tue', 154),
('Item1', 'Wed', 125), ('Item2', 'Wed', 220), ('Item3', 'Wed', 142);
```

The table should look like below:

```
+========+=========+=======+
|   item | weekday | price |
+========+=========+=======+
|  Item1 |    Mon  |   110 |
+--------+---------+-------+
|  Item2 |    Mon  |   230 |
+--------+---------+-------+
|  Item3 |    Mon  |   150 |
+--------+---------+-------+
|  Item1 |    Tue  |   115 |
+--------+---------+-------+
|  Item2 |    Tue  |   231 |
+--------+---------+-------+
|  Item3 |    Tue  |   162 |
+--------+---------+-------+
|          . . .           |
+--------+---------+-------+
|  Item2 |    Wed  |   220 |
+--------+---------+-------+
```

```
|  Item3 |    Wed  |   142 |
+--------+---------+-------+
```

In order to perform aggregation which is to find the average price per item for each week day, we are going to use the relational operator `PIVOT` to rotate the column `weekday` of table-valued expression into aggregated row values as below:

```
SELECT * FROM tbl_stock
PIVOT (
    AVG(price) FOR weekday IN ([Mon], [Tue], [Wed], [Thu], [Fri])
) pvt;
```

Result:

```
+--------+------+------+------+------+------+
|  item  |  Mon |  Tue |  Wed |  Thu |  Fri |
+--------+------+------+------+------+------+
|  Item1 |  116 |  112 |  117 |  109 |  120 |
|  Item2 |  227 |  233 |  230 |  228 |  210 |
|  Item3 |  145 |  158 |  152 |  145 |  125 |
+--------+------+------+------+------+------+
```

Lastly, in order to perform the reverse operation of `PIVOT`, we can use the relational operator `UNPIVOT` to rotate columns into rows as below:

```
SELECT * FROM tbl_stock
PIVOT (
    AVG(price) FOR weekday IN ([Mon], [Tue], [Wed], [Thu], [Fri])
) pvt
UNPIVOT (
    price FOR weekday IN ([Mon], [Tue], [Wed], [Thu], [Fri])
) unpvt;
```

Result:

```
+=======+========+=========+
|  item |  price | weekday |
+=======+========+=========+
| Item1 |    116 |     Mon |
+-------+--------+---------+
| Item1 |    112 |     Tue |
+-------+--------+---------+
| Item1 |    117 |     Wed |
+-------+--------+---------+
| Item1 |    109 |     Thu |
+-------+--------+---------+
| Item1 |    120 |     Fri |
+-------+--------+---------+
| Item2 |    227 |     Mon |
+-------+--------+---------+
| Item2 |    233 |     Tue |
+-------+--------+---------+
| Item2 |    230 |     Wed |
+-------+--------+---------+
| Item2 |    228 |     Thu |
+-------+--------+---------+
| Item2 |    210 |     Fri |
```

```
+-------+--------+---------+
| Item3 |    145 |     Mon |
+-------+--------+---------+
| Item3 |    158 |     Tue |
+-------+--------+---------+
| Item3 |    152 |     Wed |
+-------+--------+---------+
| Item3 |    145 |     Thu |
+-------+--------+---------+
| Item3 |    125 |     Fri |
+-------+--------+---------+
```

## Section 47.3: Simple Pivot - Static Columns

Using Item Sales Table from Example Database, let us calculate and show the total Quantity we sold of each Product.

This can be easily done with a group by, but lets assume we to 'rotate' our result table in a way that for each Product Id we have a column.

```
SELECT [100], [145]
  FROM (SELECT ItemId , Quantity
          FROM #ItemSalesTable
       ) AS pivotIntermediate
 PIVOT (   SUM(Quantity)
           FOR ItemId IN ([100], [145])
       ) AS pivotTable
```

Since our 'new' columns are numbers (in the source table), we need to square brackets [ ]

This will give us an output like

**100 145**
45  18

# Chapter 48: Dynamic SQL Pivot

This topic covers how to do a dynamic pivot in SQL Server.

## Section 48.1: Basic Dynamic SQL Pivot

```
if object_id('tempdb.dbo.#temp') is not null drop table #temp
create table #temp
(
    dateValue datetime,
    category varchar(3),
    amount decimal(36,2)
)

insert into #temp values ('1/1/2012', 'ABC', 1000.00)
insert into #temp values ('2/1/2012', 'DEF', 500.00)
insert into #temp values ('2/1/2012', 'GHI', 800.00)
insert into #temp values ('2/10/2012', 'DEF', 700.00)
insert into #temp values ('3/1/2012', 'ABC', 1100.00)


DECLARE
    @cols AS NVARCHAR(MAX),
    @query  AS NVARCHAR(MAX);

SET @cols = STUFF((SELECT distinct ',' + QUOTENAME(c.category)
            FROM #temp c
            FOR XML PATH(''), TYPE
            ).value('.', 'NVARCHAR(MAX)')
        ,1,1,'')

set @query = '
            SELECT
                dateValue,
                ' + @cols + '
            from
            (
                select
                     dateValue,
                     amount,
                     category
                from #temp
           ) x
            pivot
            (
                 sum(amount)
                for category in (' + @cols + ')
            ) p '


exec sp_executeSql @query
```

# Chapter 49: Partitioning

## Section 49.1: Retrieve Partition Boundary Values

```
SELECT        ps.name AS PartitionScheme
            , fg.name AS [FileGroup]
            , prv.*
            , LAG(prv.Value) OVER (PARTITION BY ps.name ORDER BY ps.name, boundary_id) AS
PreviousBoundaryValue

FROM        sys.partition_schemes ps
INNER JOIN    sys.destination_data_spaces dds
            ON dds.partition_scheme_id = ps.data_space_id
INNER JOIN    sys.filegroups fg
            ON dds.data_space_id = fg.data_space_id
INNER JOIN    sys.partition_functions f
            ON f.function_id = ps.function_id
INNER JOIN    sys.partition_range_values prv
            ON f.function_id = prv.function_id
            AND dds.destination_id = prv.boundary_id
```

## Section 49.2: Switching Partitions

According to this [TechNet Microsoft page][1],

> **Partitioning data** enables you to manage and access subsets of your data quickly and efficiently while maintaining the integrity of the entire data collection.

When you call the following query the data is not physically moved; only the metadata about the location of the data changes.

```
ALTER TABLE [SourceTable] SWITCH TO [TargetTable]
```

The tables must have the same columns with the same data types and NULL settings, they need to be in the same file group and the new target table must be empty. See the page link above for more info on switching partitions.

[1]: https://technet.microsoft.com/en-us/library/ms191160(v=sql.105).aspx The column `IDENTITY` property may differ.

## Section 49.3: Retrieve partition table,column, scheme, function, total and min-max boundry values using single query

```
SELECT DISTINCT
    object_name(i.object_id) AS [Object Name],
    c.name AS [Partition COLUMN],
    s.name AS [Partition Scheme],
    pf.name AS [Partition FUNCTION],
    prv.tot AS [Partition COUNT],
    prv.miVal AS [MIN Boundry VALUE],
    prv.maVal AS [MAX Boundry VALUE]
FROM sys.objects o
INNER JOIN sys.indexes i ON i.object_id = o.object_id
INNER JOIN sys.columns c ON c.object_id = o.object_id
```

```
INNER JOIN sys.index_columns ic ON ic.object_id = o.object_id
    AND ic.column_id = c.column_id
    AND ic.partition_ordinal = 1
INNER JOIN sys.partition_schemes s ON i.data_space_id = s.data_space_id
INNER JOIN sys.partition_functions pf ON pf.function_id = s.function_id
OUTER APPLY(SELECT
                COUNT(*) tot, MIN(VALUE) miVal, MAX(VALUE) maVal
            FROM sys.partition_range_values prv
            WHERE prv.function_id = pf.function_id) prv
--WHERE object_name(i.object_id) = 'table_name'
ORDER BY OBJECT_NAME(i.object_id)
```

Just un-comment `where` clause and replace `table_name` with `actual table name` to view the detail of desired object.

# Chapter 50: Stored Procedures

In SQL Server, a procedure is a stored program that you can pass parameters into. It does not return a value like a function does. However, it can return a success/failure status to the procedure that called it.

## Section 50.1: Creating and executing a basic stored procedure

Using the `Authors` table in the Library Database

```
CREATE PROCEDURE GetName
(
    @input_id INT = NULL,       --Input parameter,  id of the person, NULL default
    @name VARCHAR(128) = NULL   --Input parameter, name of the person, NULL default
)
AS
BEGIN
    SELECT Name + ' is from ' + Country
    FROM Authors
    WHERE Id = @input_id OR Name = @name
END
GO
```

You can execute a procedure with a few different syntaxes. First, you can use `EXECUTE` or `EXEC`

```
EXECUTE GetName @id = 1
EXEC Getname @name = 'Ernest Hemingway'
```

Additionally, you can omit the EXEC command. Also, you don't have to specify what parameter you are passing in, as you pass in all parameters.

```
GetName NULL, 'Ernest Hemingway'
```

When you want to specify the input parameters in a different order than how they are declared in the procedure you can specify the parameter name and assign values. For example

```
 CREATE PROCEDURE dbo.sProcTemp
 (
    @Param1 INT,
    @Param2 INT
)
AS
BEGIN

    SELECT
        Param1 = @Param1,
        Param2 = @Param2

END
```

the normal order to execute this procedure is to specify the value for @Param1 first and then @Param2 second. So it will look something like this

```
  EXEC dbo.sProcTemp @Param1 = 0,@Param2=1
```

But it's also possible that you can use the following

```
EXEC dbo.sProcTemp @Param2 = 0,@Param1=1
```

in this, you are specifying the value for @param2 first and @Param1 second. Which means you do not have to keep the same order as it is declared in the procedure but you can have any order as you wish. but you will need to specify to which parameter you are setting the value

**Access stored procedure from any database**

And also you can create a procedure with a prefix `sp_` these procuedres, like all system stored procedures, can be executed without specifying the database because of the default behavior of SQL Server. When you execute a stored procedure that starts with "sp_", SQL Server looks for the procedure in the master database first. If the procedure is not found in master, it looks in the active database. If you have a stored procedure that you want to access from all your databases, create it in master and use a name that includes the "sp_" prefix.

```
Use Master

CREATE PROCEDURE sp_GetName
(
    @input_id INT = NULL,       --Input parameter,  id of the person, NULL default
    @name VARCHAR(128) = NULL   --Input parameter, name of the person, NULL default
)
AS
BEGIN
    SELECT Name + ' is from ' + Country
    FROM Authors
    WHERE Id = @input_id OR Name = @name
END
GO
```

# Section 50.2: Stored Procedure with If...Else and Insert Into operation

Create example table `Employee`:

```
CREATE TABLE Employee
(
    Id INT,
    EmpName VARCHAR(25),
    EmpGender VARCHAR(6),
    EmpDeptId INT
)
```

Creates stored procedure that checks whether the values passed in stored procedure are not null or non empty and perform insert operation in Employee table.

```
CREATE PROCEDURE spSetEmployeeDetails
(
    @ID int,
    @Name VARCHAR(25),
    @Gender VARCHAR(6),
    @DeptId INT
)
AS
BEGIN
    IF (
        (@ID IS NOT NULL AND LEN(@ID) !=0)
        AND (@Name IS NOT NULL AND LEN(@Name) !=0)
```

```
        AND (@Gender IS NOT NULL AND LEN(@Gender) !=0)
        AND (@DeptId IS NOT NULL AND LEN(@DeptId) !=0)
    )
    BEGIN
        INSERT INTO Employee
        (
            Id,
            EmpName,
            EmpGender,
            EmpDeptId
        )
        VALUES
        (
            @ID,
            @Name,
            @Gender,
            @DeptId
        )
    END
ELSE
    PRINT 'Incorrect Parameters'
END
GO
```

Execute the stored procedure

```
DECLARE @ID INT,
    @Name VARCHAR(25),
    @Gender VARCHAR(6),
    @DeptId INT

EXECUTE spSetEmployeeDetails
    @ID = 1,
    @Name = 'Subin Nepal',
    @Gender = 'Male',
    @DeptId = 182666
```

## Section 50.3: Dynamic SQL in stored procedure

Dynamic SQL enables us to generate and run SQL statements at run time. Dynamic SQL is needed when our SQL statements contains identifier that may change at different compile times.

Simple Example of dynamic SQL:

```
CREATE PROC sp_dynamicSQL
@table_name      NVARCHAR(20),
@col_name        NVARCHAR(20),
@col_value       NVARCHAR(20)
AS
BEGIN
DECLARE  @Query  NVARCHAR(max)
SET      @Query = 'SELECT * FROM ' + @table_name
SET      @Query = @Query + ' WHERE ' + @col_name + ' = ' + ''''+@col_value+''''
EXEC     (@Query)
END
```

In the above sql query, we can see that we can use above query by defining values in `@table_name`, `@col_name`, and `@col_value` at run time. The query is generated at runtime and executed. This is technique in which we can create whole scripts as string in a variable and execute it. We can create more complex queries using dynamic SQL

and concatenation concept. This concept is very powerful when you want to create a script that can be used under several conditions.

Executing stored procedure

```
DECLARE @table_name     NVARCHAR(20) = 'ITCompanyInNepal',
        @col_name       NVARCHAR(20) = 'Headquarter',
        @col_value      NVARCHAR(20) = 'USA'

EXEC    sp_dynamicSQL   @table_name,
                        @col_name,
                        @col_value
```

Table I have used

| ID | CompanyName | CompanyAddress | Headquarter | NumberOfEmployee |
|---|---|---|---|---|
| 1 | CompanyOne | Kathmandu | USA | 300 |
| 2 | CompanyTwo | Kathmandu | USA | 260 |
| 3 | CompanyThree | Kathmandu | Nepal | 300 |
| 4 | CompanyFour | Kathmandu | Nepal | 180 |
| 6 | CompanySix | Janakpur | USA | 50 |
| 7 | CompanySeven | Janakpur | Australia | 100 |
| 8 | CompanyEight | Birganj | Australia | 150 |
| 9 | CompanyNine | Biratnagar | Canada | 200 |
| 10 | CompanyTen | Pokhara | India | 85 |

Output

| ID | CompanyName | CompanyAddress | Headquarter | NumberOfEmployee |
|---|---|---|---|---|
| 1 | CompanyOne | Kathmandu | USA | 300 |
| 2 | CompanyTwo | Kathmandu | USA | 260 |
| 6 | CompanySix | Janakpur | USA | 50 |
| 1 | CompanyA | Banglore | USA | 400 |
| 2 | CompanyB | Banglore | USA | 450 |

# Section 50.4: STORED PROCEDURE with OUT parameters

Stored procedures can return values using the `OUTPUT` keyword in its parameter list.

**Creating a stored procedure with a single out parameter**

```
CREATE PROCEDURE SprocWithOutParams
(
    @InParam VARCHAR(30),
    @OutParam VARCHAR(30) OUTPUT
)
AS
BEGIN
    SELECT @OutParam = @InParam + ' must come out'
    RETURN
END
GO
```

**Executing the stored procedure**

```
DECLARE @OutParam VARCHAR(30)
EXECUTE SprocWithOutParams 'what goes in', @OutParam OUTPUT
PRINT @OutParam
```

**Creating a stored procedure with multiple out parameters**

```
CREATE PROCEDURE SprocWithOutParams2
(
    @InParam VARCHAR(30),
    @OutParam VARCHAR(30) OUTPUT,
    @OutParam2 VARCHAR(30) OUTPUT
)
AS
BEGIN
    SELECT @OutParam = @InParam +' must come out'
    SELECT @OutParam2 = @InParam +' must come out'
    RETURN
END
GO
```

**Executing the stored procedure**

```
DECLARE @OutParam VARCHAR(30)
DECLARE @OutParam2 VARCHAR(30)
EXECUTE SprocWithOutParams2 'what goes in', @OutParam OUTPUT, @OutParam2 OUTPUT
PRINT @OutParam
PRINT @OutParam2
```

## Section 50.5: Simple Looping

First lets get some data into a temp table named `#systables` and ad a incrementing row number so we can query one record at a time

```
SELECT
    o.name,
    ROW_NUMBER() OVER (ORDER BY o.name) AS rn
INTO
    #systables
FROM
    sys.objects AS o
WHERE
    o.type = 'S'
```

Next we declare some variables to control the looping and store the table name in this example

```
declare
    @rn int = 1,
    @maxRn int = (
                    select
                        max(rn)
                    from
                        #systables as s
                    )
declare    @tablename sys name
```

Now we can loop using a simple while. We increment `@rn` in the `SELECT` statement but this could also have been a separate statement for ex `set @rn = @rn + 1` it will depend on your requirements. We also use the value of `@rn` before it's incremented to select a single record from `#systables`. Lastly we print the table name.

```
while @rn <= @maxRn
    begin

        select
            @tablename = name,
            @rn = @rn + 1
        from
```

```
            #systables as s
        where
            s.rn = @rn

        print @tablename
    end
```

## Section 50.6: Simple Looping

```
CREATE PROCEDURE SprocWithSimpleLoop
(
    @SayThis VARCHAR(30),
    @ThisManyTimes INT
)
AS
BEGIN
    WHILE @ThisManyTimes > 0
    BEGIN
        PRINT @SayThis;
        SET @ThisManyTimes = @ThisManyTimes - 1;
    END

    RETURN;
END
GO
```

# Chapter 51: Retrieve information about the database

## Section 51.1: Retrieve a List of all Stored Procedures

The following queries will return a list of all Stored Procedures in the database, with basic information about each Stored Procedure:

Version ≥ SQL Server 2005

```
SELECT *
FROM INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_TYPE = 'PROCEDURE'
```

The `ROUTINE_NAME`, `ROUTINE_SCHEMA` and `ROUTINE_DEFINITION` columns are generally the most useful.

Version ≥ SQL Server 2005

```
SELECT *
FROM sys.objects
WHERE TYPE = 'P'
```

Version ≥ SQL Server 2005

```
SELECT *
FROM sys.procedures
```

Note that this version has an advantage over selecting from sys.objects since it includes the additional columns `is_auto_executed`, `is_execution_replicated`, `is_repl_serializable`, and `skips_repl_constraints`.

Version < SQL Server 2005

```
SELECT *
FROM sysobjects
WHERE TYPE = 'P'
```

Note that the output contains many columns that will never relate to a stored procedure.

The next set of queries will return all Stored Procedures in the database that include the string 'SearchTerm':

Version < SQL Server 2005

```
SELECT o.name
FROM syscomments c
INNER JOIN sysobjects o
    ON c.id=o.id
WHERE o.xtype = 'P'
    AND c.TEXT LIKE '%SearchTerm%'
```

Version ≥ SQL Server 2005

```
SELECT p.name
FROM sys.sql_modules AS m
INNER JOIN sys.procedures AS p
    ON m.object_id = p.object_id
WHERE definition LIKE '%SearchTerm%'
```

## Section 51.2: Get the list of all databases on a server

**Method 1:** Below query will be applicable for SQL Server 2000+ version (Contains 12 columns)

```
SELECT * FROM dbo.sysdatabases
```

**Method 2:** Below query extract information about databases with more informations (ex: State, Isolation, recovery model etc.)

Note: This is a catalog view and will be available SQL SERVER 2005+ versions

```
SELECT * FROM sys.databases
```

**Method 3:** To see just database names you can use undocumented sp_MSForEachDB

```
EXEC sp_MSForEachDB 'SELECT ''?'' AS DatabaseName'
```

**Method 4:** Below SP will help you to provide database size along with databases name , owner, status etc. on the server

```
EXEC sp_helpdb
```

**Method 5** Similarly, below stored procedure will give database name, database size and Remarks

```
EXEC sp_databases
```

## Section 51.3: Count the Number of Tables in a Database

This query will return the number of tables in the specified database.

```
USE YourDatabaseName
SELECT COUNT(*) from INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE'
```

Following is another way this can be done for all user tables with SQL Server 2008+. The reference is here.

```
SELECT COUNT(*) FROM sys.tables
```

## Section 51.4: Database Files

Display all data files for all databases with size and growth info

```
SELECT  d.name AS 'Database',
        d.database_id,
        SF.fileid,
        SF.name AS 'LogicalFileName',
        CASE SF.status & 0x100000
            WHEN 1048576 THEN 'Percentage'
            WHEN 0 THEN 'MB'
        END AS 'FileGrowthOption',
        Growth AS GrowthUnit,
        ROUND(((CAST(SIZE AS FLOAT)*8)/1024)/1024,2) [SizeGB], -- Convert 8k pages to GB
        Maxsize,
        filename AS PhysicalFileName

FROM    Master.SYS.SYSALTFILES SF
JOIN    Master.SYS.Databases d ON sf.fileid = d.database_id

ORDER BY d.name
```

## Section 51.5: See if Enterprise-specific features are being used

It is sometimes useful to verify that your work on Developer edition hasn't introduced a dependency on any features restricted to Enterprise edition.

You can do this using the `sys.dm_db_persisted_sku_features` system view, like so:

```
SELECT * FROM sys.dm_db_persisted_sku_features
```

Against the database itself.

This will list the features being used, if any.

## Section 51.6: Determine a Windows Login's Permission Path

This will show the user type and permission path (which windows group the user is getting its permissions from).

```
xp_logininfo 'DOMAIN\user'
```

## Section 51.7: Search and Return All Tables and Columns Containing a Specified Column Value

This script, from here and here, will return all Tables and Columns where a specified value exists. This is powerful in finding out where a certain value is in a database. It can be taxing, so it is suggested that it be executed in a backup / test enviroment first.

```
DECLARE @SearchStr nvarchar(100)
SET @SearchStr = '## YOUR STRING HERE ##'


    -- Copyright © 2002 Narayana Vyas Kondreddi. All rights reserved.
    -- Purpose: To search all columns of all tables for a given search string
    -- Written by: Narayana Vyas Kondreddi
    -- Site: http://vyaskn.tripod.com
    -- Updated and tested by Tim Gaunt
    -- http://www.thesitedoctor.co.uk
    --
http://blogs.thesitedoctor.co.uk/tim/2010/02/19/Search+Every+Table+And+Field+In+A+SQL+Server+Database+Updated.aspx
    -- Tested on: SQL Server 7.0, SQL Server 2000, SQL Server 2005 and SQL Server 2010
    -- Date modified: 03rd March 2011 19:00 GMT
    CREATE TABLE #Results (ColumnName nvarchar(370), ColumnValue nvarchar(3630))

    SET NOCOUNT ON

    DECLARE @TableName nvarchar(256), @ColumnName nvarchar(128), @SearchStr2 nvarchar(110)
    SET  @TableName = ''
    SET @SearchStr2 = QUOTENAME('%' + @SearchStr + '%','''')

    WHILE @TableName IS NOT NULL

    BEGIN
        SET @ColumnName = ''
        SET @TableName =
        (
            SELECT MIN(QUOTENAME(TABLE_SCHEMA) + '.' + QUOTENAME(TABLE_NAME))
            FROM     INFORMATION_SCHEMA.TABLES
```

```
            WHERE         TABLE_TYPE = 'BASE TABLE'
                AND    QUOTENAME(TABLE_SCHEMA) + '.' + QUOTENAME(TABLE_NAME) > @TableName
                AND    OBJECTPROPERTY(
                        OBJECT_ID(
                            QUOTENAME(TABLE_SCHEMA) + '.' + QUOTENAME(TABLE_NAME)
                             ), 'IsMSShipped'
                               ) = 0
        )

        WHILE (@TableName IS NOT NULL) AND (@ColumnName IS NOT NULL)

        BEGIN
            SET @ColumnName =
            (
                SELECT MIN(QUOTENAME(COLUMN_NAME))
                FROM     INFORMATION_SCHEMA.COLUMNS
                WHERE         TABLE_SCHEMA    = PARSENAME(@TableName, 2)
                    AND    TABLE_NAME    = PARSENAME(@TableName, 1)
                    AND    DATA_TYPE IN ('char', 'varchar', 'nchar', 'nvarchar', 'int', 'decimal')
                    AND    QUOTENAME(COLUMN_NAME) > @ColumnName
            )

            IF @ColumnName IS NOT NULL

            BEGIN
                INSERT INTO #Results
                EXEC
                (
                    'SELECT ''' + @TableName + '.' + @ColumnName + ''', LEFT(' + @ColumnName + ',
3630) FROM ' + @TableName + ' (NOLOCK) ' +
                    ' WHERE ' + @ColumnName + ' LIKE ' + @SearchStr2
                )
            END
        END
    END

    SELECT ColumnName, ColumnValue FROM #Results

DROP TABLE #Results
- See more at: http://thesitedoctor.co.uk/blog/search-every-table-and-field-in-a-sql-server-
database-updated#sthash.bBEqfJVZ.dpuf
```

## Section 51.8: Get all schemas, tables, columns and indexes

```
SELECT
    s.name AS [schema],
    t.object_id AS [table_object_id],
    t.name AS [TABLE_NAME],
    c.column_id,
    c.name AS [column_name],
    i.name AS [index_name],
    i.type_desc AS [index_type]
FROM sys.schemas AS s
INNER JOIN sys.tables AS t
    ON s.schema_id = t.schema_id
INNER JOIN sys.columns AS c
    ON t.object_id = c.object_id
LEFT JOIN sys.index_columns AS ic
    ON c.object_id = ic.object_id AND c.column_id = ic.column_id
LEFT JOIN sys.indexes AS i
    ON ic.object_id = i.object_id AND ic.index_id = i.index_id
```

```
ORDER BY [schema], [TABLE_NAME], c.column_id;
```

## Section 51.9: Return a list of SQL Agent jobs, with schedule information

```
USE msdb
Go


SELECT dbo.sysjobs.Name AS 'Job Name',
      'Job Enabled' = CASE dbo.sysjobs.Enabled
            WHEN 1 THEN 'Yes'
            WHEN 0 THEN 'No'
      END,
      'Frequency' = CASE dbo.sysschedules.freq_type
            WHEN 1 THEN 'Once'
            WHEN 4 THEN 'Daily'
            WHEN 8 THEN 'Weekly'
            WHEN 16 THEN 'Monthly'
            WHEN 32 THEN 'Monthly relative'
            WHEN 64 THEN 'When SQLServer Agent starts'
      END,
      'Start Date' = CASE active_start_date
            WHEN 0 THEN null
            ELSE
            substring(convert(varchar(15),active_start_date),1,4) + '/' +
            substring(convert(varchar(15),active_start_date),5,2) + '/' +
            substring(convert(varchar(15),active_start_date),7,2)
      END,
      'Start Time' = CASE len(active_start_time)
            WHEN 1 THEN cast('00:00:0' + right(active_start_time,2) as char(8))
            WHEN 2 THEN cast('00:00:' + right(active_start_time,2) as char(8))
            WHEN 3 THEN cast('00:0'
                        + Left(right(active_start_time,3),1)
                        +':' + right(active_start_time,2) as char (8))
            WHEN 4 THEN cast('00:'
                        + Left(right(active_start_time,4),2)
                        +':' + right(active_start_time,2) as char (8))
            WHEN 5 THEN cast('0'
                        + Left(right(active_start_time,5),1)
                        +':' + Left(right(active_start_time,4),2)
                        +':' + right(active_start_time,2) as char (8))
            WHEN 6 THEN cast(Left(right(active_start_time,6),2)
                        +':' + Left(right(active_start_time,4),2)
                        +':' + right(active_start_time,2) as char (8))
      END,

      CASE len(run_duration)
            WHEN 1 THEN cast('00:00:0'
                        + cast(run_duration as char) as char (8))
            WHEN 2 THEN cast('00:00:'
                        + cast(run_duration as char) as char (8))
            WHEN 3 THEN cast('00:0'
                        + Left(right(run_duration,3),1)
                        +':' + right(run_duration,2) as char (8))
            WHEN 4 THEN cast('00:'
                        + Left(right(run_duration,4),2)
                        +':' + right(run_duration,2) as char (8))
            WHEN 5 THEN cast('0'
                        + Left(right(run_duration,5),1)
                        +':' + Left(right(run_duration,4),2)
```

```
                        +':' + right(run_duration,2) as char (8))
            WHEN 6 THEN cast(Left(right(run_duration,6),2)
                        +':' + Left(right(run_duration,4),2)
                        +':' + right(run_duration,2) as char (8))
      END as 'Max Duration',
    CASE(dbo.sysschedules.freq_subday_interval)
            WHEN 0 THEN 'Once'
            ELSE cast('Every '
                        + right(dbo.sysschedules.freq_subday_interval,2)
                        + ' '
                        +     CASE(dbo.sysschedules.freq_subday_type)
                                    WHEN 1 THEN 'Once'
                                    WHEN 4 THEN 'Minutes'
                                    WHEN 8 THEN 'Hours'
                                END as char(16))
    END as 'Subday Frequency'
FROM dbo.sysjobs
LEFT OUTER JOIN dbo.sysjobschedules
ON dbo.sysjobs.job_id = dbo.sysjobschedules.job_id
INNER JOIN dbo.sysschedules ON dbo.sysjobschedules.schedule_id = dbo.sysschedules.schedule_id
LEFT OUTER JOIN (SELECT job_id, max(run_duration) AS run_duration
            FROM dbo.sysjobhistory
            GROUP BY job_id) Q1
ON dbo.sysjobs.job_id = Q1.job_id
WHERE Next_run_time = 0

UNION

SELECT dbo.sysjobs.Name AS 'Job Name',
      'Job Enabled' = CASE dbo.sysjobs.Enabled
            WHEN 1 THEN 'Yes'
            WHEN 0 THEN 'No'
      END,
      'Frequency' = CASE dbo.sysschedules.freq_type
            WHEN 1 THEN 'Once'
            WHEN 4 THEN 'Daily'
            WHEN 8 THEN 'Weekly'
            WHEN 16 THEN 'Monthly'
            WHEN 32 THEN 'Monthly relative'
            WHEN 64 THEN 'When SQLServer Agent starts'
      END,
      'Start Date' = CASE next_run_date
            WHEN 0 THEN null
            ELSE
            substring(convert(varchar(15),next_run_date),1,4) + '/' +
            substring(convert(varchar(15),next_run_date),5,2) + '/' +
            substring(convert(varchar(15),next_run_date),7,2)
      END,
      'Start Time' = CASE len(next_run_time)
            WHEN 1 THEN cast('00:00:0' + right(next_run_time,2) as char(8))
            WHEN 2 THEN cast('00:00:' + right(next_run_time,2) as char(8))
            WHEN 3 THEN cast('00:0'
                        + Left(right(next_run_time,3),1)
                        +':' + right(next_run_time,2) as char (8))
            WHEN 4 THEN cast('00:'
                        + Left(right(next_run_time,4),2)
                        +':' + right(next_run_time,2) as char (8))
            WHEN 5 THEN cast('0' + Left(right(next_run_time,5),1)
                        +':' + Left(right(next_run_time,4),2)
                        +':' + right(next_run_time,2) as char (8))
            WHEN 6 THEN cast(Left(right(next_run_time,6),2)
                        +':' + Left(right(next_run_time,4),2)
```

```
                        +':' + right(next_run_time,2) as char (8))
      END,

      CASE len(run_duration)
            WHEN 1 THEN cast('00:00:0'
                        + cast(run_duration as char) as char (8))
            WHEN 2 THEN cast('00:00:'
                        + cast(run_duration as char) as char (8))
            WHEN 3 THEN cast('00:0'
                        + Left(right(run_duration,3),1)
                        +':' + right(run_duration,2) as char (8))
            WHEN 4 THEN cast('00:'
                        + Left(right(run_duration,4),2)
                        +':' + right(run_duration,2) as char (8))
            WHEN 5 THEN cast('0'
                        + Left(right(run_duration,5),1)
                        +':' + Left(right(run_duration,4),2)
                        +':' + right(run_duration,2) as char (8))
            WHEN 6 THEN cast(Left(right(run_duration,6),2)
                        +':' + Left(right(run_duration,4),2)
                        +':' + right(run_duration,2) as char (8))
      END as 'Max Duration',
    CASE(dbo.sysschedules.freq_subday_interval)
            WHEN 0 THEN 'Once'
            ELSE cast('Every '
                        + right(dbo.sysschedules.freq_subday_interval,2)
                        + ' '
                        +     CASE(dbo.sysschedules.freq_subday_type)
                                    WHEN 1 THEN 'Once'
                                    WHEN 4 THEN 'Minutes'
                                    WHEN 8 THEN 'Hours'
                              END as char(16))
    END as 'Subday Frequency'
FROM dbo.sysjobs
LEFT OUTER JOIN dbo.sysjobschedules ON dbo.sysjobs.job_id = dbo.sysjobschedules.job_id
INNER JOIN dbo.sysschedules ON dbo.sysjobschedules.schedule_id = dbo.sysschedules.schedule_id
LEFT OUTER JOIN (SELECT job_id, max(run_duration) AS run_duration
            FROM dbo.sysjobhistory
            GROUP BY job_id) Q1
ON dbo.sysjobs.job_id = Q1.job_id
WHERE Next_run_time <> 0

ORDER BY [Start Date],[Start Time]
```

# Section 51.10: Retrieve Tables Containing Known Column

This query will return all `COLUMNS` and their associated `TABLES` for a given column name. It is designed to show you what tables (unknown) contain a specified column (known)

```
SELECT
    c.name AS ColName,
    t.name AS TableName
FROM
    sys.columns c
    JOIN sys.tables t ON c.object_id = t.object_id
WHERE
    c.name LIKE '%MyName%'
```

## Section 51.11: Show Size of All Tables in Current Database

```
SELECT
    s.name + '.' + t.NAME AS TableName,
    SUM(a.used_pages)*8 AS 'TableSizeKB'  --a page in SQL Server is 8kb
FROM sys.tables t
    JOIN sys.schemas s ON t.schema_id = s.schema_id
    LEFT JOIN sys.indexes i ON t.OBJECT_ID = i.object_id
    LEFT JOIN sys.partitions p ON i.object_id = p.OBJECT_ID AND i.index_id = p.index_id
    LEFT JOIN sys.allocation_units a ON p.partition_id = a.container_id
GROUP BY
    s.name, t.name
ORDER BY
    --Either sort by name:
    s.name + '.' + t.NAME
    --Or sort largest to smallest:
    --SUM(a.used_pages) desc
```

## Section 51.12: Retrieve Database Options

The following query returns the database options and metadata:

```
SELECT * FROM sys.databases WHERE name = 'MyDatabaseName';
```

## Section 51.13: Find every mention of a field in the database

```
SELECT DISTINCT
 o.name AS Object_Name,o.type_desc
 FROM sys.sql_modules m
    INNER JOIN sys.objects  o ON m.object_id=o.object_id
 WHERE m.definition LIKE '%myField%'
 ORDER BY 2,1
```

Will find mentions of `myField` in SProcs, Views, etc.

## Section 51.14: Retrieve information on backup and restore operations

To get the list of all backup operations performed on the current database instance:

```
SELECT sdb.Name AS DatabaseName,
    COALESCE(CONVERT(VARCHAR(50), bus.backup_finish_date, 120),'-') AS LastBackUpDateTime
FROM sys.sysdatabases sdb
    LEFT OUTER JOIN msdb.dbo.backupset bus ON bus.database_name = sdb.name
ORDER BY sdb.name, bus.backup_finish_date DESC
```

To get the list of all restore operations performed on the current database instance:

```
SELECT
    [d].[name] AS database_name,
    [r].restore_date AS last_restore_date,
    [r].[user_name],
    [bs].[backup_finish_date] AS backup_creation_date,
    [bmf].[physical_device_name] AS [backup_file_used_for_restore]
FROM master.sys.databases [d]
    LEFT OUTER JOIN msdb.dbo.[restorehistory] r ON r.[destination_database_name] = d.Name
    INNER JOIN msdb.dbo.backupset [bs] ON [r].[backup_set_id] = [bs].[backup_set_id]
```

```
    INNER JOIN msdb.dbo.backupmediafamily bmf ON [bs].[media_set_id] = [bmf].[media_set_id]
ORDER BY [d].[name], [r].restore_date DESC
```

# Chapter 52: Split String function in SQL Server

## Section 52.1: Split string in Sql Server 2008/2012/2014 using XML

Since there is no `STRING_SPLIT` function we need to use XML hack to split the string into rows:

**Example:**

```
SELECT split.a.value('.', 'VARCHAR(100)') AS VALUE
FROM   (SELECT CAST ('<M>' + REPLACE('A|B|C', '|', '</M><M>')+ '</M>' AS XML) AS DATA) AS A
       CROSS apply DATA.nodes ('/M') AS Split(a);
```

**Result:**

```
+-----+
|Value|
+-----+
|A    |
+-----+
|B    |
+-----+
|C    |
+-----+
```

## Section 52.2: Split a String in Sql Server 2016

In **SQL Server 2016** finally they have introduced Split string function : **STRING_SPLIT**

**Parameters:** It accepts two parameters

***String***:

> Is an expression of any character type (i.e. nvarchar, varchar, nchar or char).

***separator*** :

> Is a single character expression of any character type (e.g. nvarchar(1), varchar(1), nchar(1) or char(1)) that is used as separator for concatenated strings.

**Note:** You should always check if the expression is a non-empty string.

**Example:**

```
SELECT VALUE
FROM STRING_SPLIT('a|b|c','|')
```

In above example

```
String    : 'a|b|c'
separator : '|'
```

**Result :**

```
+-----+
|Value|
+-----+
|a    |
+-----+
|b    |
+-----+
|c    |
+-----+
```

**If it's an empty string:**

```
SELECT VALUE
FROM STRING_SPLIT('',',')
```

**Result :**

```
  +-----+
  |Value|
  +-----+
1 |     |
  +-----+
```

You can avoid the above situation by adding a `WHERE` clause

```
SELECT VALUE
FROM STRING_SPLIT('',',')
WHERE LTRIM(RTRIM(VALUE))<>''
```

# Section 52.3: T-SQL Table variable and XML

```
Declare @userList Table(UserKey VARCHAR(60))
Insert into @userList values ('bill'),('jcom'),('others')
--Declared a table variable and insert 3 records

Declare @text XML
Select  @text = (
        select UserKey  from @userList for XML Path('user'), root('group')
)
--Set the XML value from Table

Select @text

--View the variable value
XML:
\<group>\<user>\<UserKey>bill\</UserKey>\</user>\<user>\<UserKey>jcom\</UserKey>\</user>\<user>\<Us
erKey>others\</UserKey>\</user>\</group>
```

# Chapter 53: Insert

## Section 53.1: Add a row to a table named Invoices

```
INSERT INTO Invoices [ /* column names may go here */ ]
VALUES (123, '1234abc', '2016-08-05 20:18:25.770', 321, 5, '2016-08-04');
```

- Column names are required if the table you are inserting into contains a column with the IDENTITY attribute.

```
INSERT INTO Invoices ([ID], [Num], [DateTime], [Total], [Term], [DueDate])
VALUES (123, '1234abc', '2016-08-05 20:18:25.770', 321, 5, '2016-08-25');
```

# Chapter 54: Primary Keys

## Section 54.1: Create table w/ identity column as primary key

```
-- Identity primary key - unique arbitrary increment number
create table person (
id int identity(1,1) primary key not null,
firstName varchar(100) not null,
lastName varchar(100) not null,
dob DateTime not null,
ssn varchar(9) not null
)
```

## Section 54.2: Create table w/ GUID primary key

```
-- GUID primary key - arbitrary unique value for table
create table person (
id uniqueIdentifier default (newId()) primary key,
firstName varchar(100) not null,
lastName varchar(100) not null,
dob DateTime not null,
ssn varchar(9) not null
)
```

## Section 54.3: Create table w/ natural key

```
-- natural primary key - using an existing piece of data within the table that uniquely identifies
the record
create table person (
firstName varchar(100) not null,
lastName varchar(100) not null,
dob DateTime not null,
ssn varchar(9) primary key not null
)
```

## Section 54.4: Create table w/ composite key

```
-- composite key - using two or more existing columns within a table to create a primary key
create table person (
firstName varchar(100) not null,
lastName varchar(100) not null,
dob DateTime not null,
ssn varchar(9) not null,
primary key (firstName, lastName, dob)
)
```

## Section 54.5: Add primary key to existing table

```
ALTER TABLE person
 ADD CONSTRAINT pk_PersonSSN PRIMARY KEY (ssn)
```

Note, if the primary key column (in this case `ssn`) has more than one row with the same candidate key, the above statement will fail, as primary key values must be unique.

## Section 54.6: Delete primary key

```
ALTER TABLE Person
 DROP CONSTRAINT pk_PersonSSN
```

# Chapter 55: Foreign Keys

## Section 55.1: Foreign key relationship/constraint

Foreign keys enables you to define relationship between two tables. One (parent) table need to have primary key that uniquely identifies rows in the table. Other (child) table can have value of the primary key from the parent in one of the columns. FOREIGN KEY REFERENCES constraint ensures that values in child table must exist as a primary key value in the parent table.

In this example we have parent Company table with CompanyId primary key, and child Employee table that has id of the company where this employee works.

```
create table Company (
   CompanyId int primary key,
   Name nvarchar(200)
)
create table Employee (
    EmployeeId int,
    Name nvarchar(200),
    CompanyId int
        foreign key references Company(companyId)
)
```

**foreign key references** ensures that values inserted in Employee.CompanyId column must also exist in Company.CompanyId column. Also, nobody can delete company in company table if there is ate least one employee with a matching companyId in child table.

FOREIGN KEY relationship ensures that rows in two tables cannot be "unlinked".

## Section 55.2: Maintaining relationship between parent/child rows

Let's assume that we have one row in Company table with companyId 1. We can insert row in employee table that has companyId 1:

```
insert into Employee values (17, 'John', 1)
```

However, we cannot insert employee that has non-existing CompanyId:

```
insert into Employee values (17, 'John', 111111)
```

Msg 547, Level 16, State 0, Line 12 The INSERT statement conflicted with the FOREIGN KEY constraint "FK__Employee__Compan__1EE485AA". The conflict occurred in database "MyDb", table "dbo.Company", column 'CompanyId'. The statement has been terminated.

Also, we cannot delete parent row in company table as long as there is at least one child row in employee table that references it.

```
delete from company where CompanyId = 1
```

Msg 547, Level 16, State 0, Line 14 The DELETE statement conflicted with the REFERENCE constraint "FK__Employee__Compan__1EE485AA". The conflict occurred in database "MyDb", table "dbo.Employee", column 'CompanyId'. The statement has been terminated.

Foreign key relationship ensures that Company and employee rows will not be "unlinked".

## Section 55.3: Adding foreign key relationship on existing table

**FOREIGN KEY** constraint can be added on existing tables that are still not in relationship. Imagine that we have Company and Employee tables where Employee table CompanyId column but don't have foreign key relationship. ALTER TABLE statement enables you to add **foreign key** constraint on an existing column that references some other table and primary key in that table:

```
alter table Employee
    add  foreign key (CompanyId) references Company(CompanyId)
```

## Section 55.4: Add foreign key on existing table

**FOREIGN KEY** columns with constraint can be added on existing tables that are still not in relationship. Imagine that we have Company and Employee tables where Employee table don't have CompanyId column. ALTER TABLE statement enables you to add new column with **foreign key** constraint that references some other table and primary key in that table:

```
alter table Employee
    add CompanyId int foreign key references Company(CompanyId)
```

## Section 55.5: Getting information about foreign key constraints

sys.foreignkeys system view returns information about all foreign key relationships in database:

```
SELECT name,
 OBJECT_NAME(referenced_object_id) AS [parent TABLE],
 OBJECT_NAME(parent_object_id) AS [child TABLE],
 delete_referential_action_desc,
 update_referential_action_desc
FROM sys.foreign_keys
```

# Chapter 56: Last Inserted Identity

## Section 56.1: @@IDENTITY and MAX(ID)

```
SELECT MAX(Id) FROM Employees -- Display the value of Id in the last row in Employees table.
GO
INSERT INTO Employees (FName, LName, PhoneNumber) -- Insert a new row
VALUES ('John', 'Smith', '25558696525')
GO
SELECT @@IDENTITY
GO
SELECT MAX(Id) FROM Employees -- Display the value of Id of the newly inserted row.
GO
```

The last two SELECT statements values are the same.

## Section 56.2: SCOPE_IDENTITY()

```
CREATE TABLE dbo.logging_table(log_id INT IDENTITY(1,1) PRIMARY KEY,
                               log_message VARCHAR(255))

CREATE TABLE dbo.person(person_id INT IDENTITY(1,1) PRIMARY KEY,
                        person_name VARCHAR(100) NOT NULL)
GO;

CREATE TRIGGER dbo.InsertToADifferentTable ON dbo.person
AFTER INSERT
AS
    INSERT INTO dbo.logging_table(log_message)
    VALUES('Someone added something to the person table')
GO;

INSERT INTO dbo.person(person_name)
VALUES('John Doe')

SELECT SCOPE_IDENTITY();
```

This will return the most recently added identity value produced on the same connection, within the current scope. In this case, 1, for the first row in the dbo.person table.

## Section 56.3: @@IDENTITY

```
CREATE TABLE dbo.logging_table(log_id INT IDENTITY(1,1) PRIMARY KEY,
                               log_message VARCHAR(255))

CREATE TABLE dbo.person(person_id INT IDENTITY(1,1) PRIMARY KEY,
                        person_name VARCHAR(100) NOT NULL)
GO;

CREATE TRIGGER dbo.InsertToADifferentTable ON dbo.person
AFTER INSERT
AS
    INSERT INTO dbo.logging_table(log_message)
    VALUES('Someone added something to the person table')
GO;

INSERT INTO dbo.person(person_name)
```

```
VALUES('John Doe')

SELECT @@IDENTITY;
```

This will return the most recently-added identity on the same connection, regardless of scope. In this case, whatever the current value of the identity column on logging_table is, assuming no other activity is occurring on the instance of SQL Server and no other triggers fire from this insert.

## Section 56.4: IDENT_CURRENT('tablename')

```
SELECT IDENT_CURRENT('dbo.person');
```

This will select the most recently-added identity value on the selected table, regardless of connection or scope.

# Chapter 57: SCOPE_IDENTITY()

## Section 57.1: Introduction with Simple Example

SCOPE_IDENTITY() returns the last identity value inserted into an identity column in the same scope. A scope is a module: a stored procedure, trigger, function, or batch. Therefore, two statements are in the same scope if they are in the same stored procedure, function, or batch.

```
INSERT INTO  ([column1],[column2]) VALUES (8,9);
GO
SELECT SCOPE_IDENTITY() AS [SCOPE_IDENTITY];
GO
```

# Chapter 58: Sequences

## Section 58.1: Create Sequence

```
CREATE SEQUENCE [dbo].[CustomersSeq]
AS INT
START WITH 10001
INCREMENT BY 1
MINVALUE -1;
```

## Section 58.2: Use Sequence in Table

```
CREATE TABLE [dbo].[Customers]
(
    CustomerID INT DEFAULT (NEXT VALUE FOR [dbo].[CustomersSeq]) NOT NULL,
    CustomerName VARCHAR(100),
);
```

## Section 58.3: Insert Into Table with Sequence

```
INSERT INTO [dbo].[Customers]
       ([CustomerName])
 VALUES
       ('Jerry'),
       ('Gorge')

SELECT * FROM [dbo].[Customers]
```

**Results**

| CustomerID | CustomerName |
|---|---|
| 10001 | Jerry |
| 10002 | Gorge |

## Section 58.4: Delete From & Insert New

```
DELETE FROM [dbo].[Customers]
WHERE CustomerName = 'Gorge';

INSERT INTO [dbo].[Customers]
       ([CustomerName])
 VALUES ('George')

SELECT * FROM [dbo].[Customers]
```

**Results**

| CustomerID | CustomerName |
|---|---|
| 10001 | Jerry |
| 10003 | George |

# Chapter 59: Index

## Section 59.1: Create Clustered index

With a clustered index the leaf pages contain the actual table rows. Therefore, there can be only one clustered index.

```
CREATE TABLE Employees
(
    ID CHAR(900),
    FirstName NVARCHAR(3000),
    LastName NVARCHAR(3000),
    StartYear CHAR(900)
)
GO

CREATE CLUSTERED INDEX IX_Clustered
ON Employees(ID)
GO
```

## Section 59.2: Drop index

```
DROP INDEX IX_NonClustered ON Employees
```

## Section 59.3: Create Non-Clustered index

Non-clustered indexes have a structure separate from the data rows. A non-clustered index contains the non-clustered index key values and each key value entry has a pointer to the data row that contains the key value. There can be maximum 999 non-clustered index on SQL Server 2008/ 2012.

Link for reference: https://msdn.microsoft.com/en-us/library/ms143432.aspx

```
CREATE TABLE Employees
(
    ID CHAR(900),
    FirstName NVARCHAR(3000),
    LastName NVARCHAR(3000),
    StartYear CHAR(900)
)
GO

CREATE NONCLUSTERED INDEX IX_NonClustered
ON Employees(StartYear)
GO
```

## Section 59.4: Show index info

```
SP_HELPINDEX tableName
```

## Section 59.5: Returns size and fragmentation indexes

```
sys.dm_db_index_physical_stats (
    { database_id | NULL | 0 | DEFAULT }
  , { object_id | NULL | 0 | DEFAULT }
  , { index_id | NULL | 0 | -1 | DEFAULT }
```

```
  , { partition_number | NULL | 0 | DEFAULT }
  , { mode | NULL | DEFAULT }
)


Sample :

SELECT * FROM sys.dm_db_index_physical_stats
    (DB_ID(N'DBName'), OBJECT_ID(N'IX_NonClustered '), NULL, NULL , 'DETAILED');
```

# Section 59.6: Reorganize and rebuild index

| avg_fragmentation_in_percent value | Corrective statement |
|---|---|
| *>5% and < = 30%* | REORGANIZE |
| *>30%* | REBUILD |

```
ALTER INDEX IX_NonClustered ON tableName REORGANIZE;


ALTER INDEX ALL ON Production.Product
 REBUILD WITH (FILLFACTOR = 80, SORT_IN_TEMPDB = ON,
          STATISTICS_NORECOMPUTE = ON);
```

# Section 59.7: Rebuild or reorganize all indexes on a table

Rebuilding indexes is done using the following statement

```
ALTER INDEX All ON tableName REBUILD;
```

This drops the index and recreates it, removing fragementation, reclaims disk space and reorders index pages.

One can also reorganize an index using

```
ALTER INDEX All ON tableName REORGANIZE;
```

which will use minimal system resources and defragments the leaf level of clustered and nonclustered indexes on tables and views by physically reordering the leaf-level pages to match the logical, left to right, order of the leaf nodes

# Section 59.8: Rebuild all index database

```
EXEC sp_MSForEachTable 'ALTER INDEX ALL ON ? REBUILD'
```

# Section 59.9: Index on view

```
CREATE VIEW  View_Index02
WITH SCHEMABINDING
AS
SELECT c.CompanyName, o.OrderDate, o.OrderID, od.ProductID
     FROM dbo.Customers C
        INNER JOIN dbo.orders O ON c.CustomerID=o.CustomerID
            INNER JOIN dbo.[Order Details] od ON o.OrderID=od.OrderID
GO

CREATE UNIQUE CLUSTERED INDEX IX1 ON
    View_Index02(OrderID, ProductID)
```

# Section 59.10: Index investigations

You could use "SP_HELPINDEX Table_Name", but Kimberly Tripp has a stored procedure (that can be found here), which is better example, as it shows more about the indexes, including columns and filter definition, for example: Usage:

```
USE Adventureworks
EXEC sp_SQLskills_SQL2012_helpindex 'dbo.Product'
```

Alternatively, Tibor Karaszi has a stored procedure (found here). The later will show information on index usage too, and optionally provide a list of index suggestions. Usage:

```
USE Adventureworks
EXEC sp_indexinfo 'dbo.Product'
```

# Chapter 60: Full-Text Indexing

## Section 60.1: A. Creating a unique index, a full-text catalog, and a full-text index

The following example creates a unique index on the JobCandidateID column of the HumanResources.JobCandidate table of the AdventureWorks2012 sample database. The example then creates a default full-text catalog, ft. Finally, the example creates a full-text index on the Resume column, using the ft catalog and the system stoplist.

```
USE AdventureWorks2012;
GO
CREATE UNIQUE INDEX ui_ukJobCand ON HumanResources.JobCandidate(JobCandidateID);
CREATE FULLTEXT CATALOG ft AS DEFAULT;
CREATE FULLTEXT INDEX ON HumanResources.JobCandidate(Resume)
   KEY INDEX ui_ukJobCand
   WITH STOPLIST = SYSTEM;
GO
```

https://www.simple-talk.com/sql/learn-sql-server/understanding-full-text-indexing-in-sql-server/

https://msdn.microsoft.com/en-us/library/cc879306.aspx

https://msdn.microsoft.com/en-us/library/ms142571.aspx

## Section 60.2: Creating a full-text index on several table columns

```
USE AdventureWorks2012;
GO
CREATE FULLTEXT CATALOG production_catalog;
GO
CREATE FULLTEXT INDEX ON Production.ProductReview
 (
  ReviewerName
     Language 1033,
  EmailAddress
     Language 1033,
  Comments
     Language 1033
 )
  KEY INDEX PK_ProductReview_ProductReviewID
      ON production_catalog;
GO
```

## Section 60.3: Creating a full-text index with a search property list without populating it

```
USE AdventureWorks2012;
GO
CREATE FULLTEXT INDEX ON Production.Document
  (
  Title
      Language 1033,
  DocumentSummary
      Language 1033,
  Document
```

```
      TYPE COLUMN FileExtension
      Language 1033
  )
  KEY INDEX PK_Document_DocumentID
         WITH STOPLIST = SYSTEM, SEARCH PROPERTY LIST = DocumentPropertyList, CHANGE_TRACKING OFF,
NO POPULATION;
   GO
```

And populating it later with

```
ALTER FULLTEXT INDEX ON Production.Document SET CHANGE_TRACKING AUTO;
GO
```

## Section 60.4: Full-Text Search

```
SELECT product_id
FROM products
WHERE CONTAINS(product_description, "Snap Happy 100EZ" OR FORMSOF(THESAURUS,'Snap Happy') OR
'100EZ')
AND product_cost < 200 ;


SELECT candidate_name,SSN
FROM candidates
WHERE CONTAINS(candidate_resume,"SQL Server") AND candidate_division =DBA;
```

For more and detailed info https://msdn.microsoft.com/en-us/library/ms142571.aspx

# Chapter 61: Trigger

A trigger is a special type of stored procedure, which is executed automatically after an event occurs. There are two types of triggers: Data Definition Language Triggers and Data Manipulation Language Triggers.

It is usually bound to a table and fires automatically. You cannot explicitly call any trigger.

## Section 61.1: DML Triggers

DML Triggers are fired as a response to dml statements (`insert`, `update` or `delete`).
A dml trigger can be created to address one or more dml events for a single table or view. This means that a single dml trigger can handle inserting, updating and deleting records from a specific table or view, but in can only handle data being changed on that single table or view.

DML Triggers provides access to `inserted` and `deleted` tables that holds information about the data that was / will be affected by the insert, update or delete statement that fired the trigger.

**Note** that DML triggers are statement based, not row based. This means that if the statement effected more then one row, the inserted or deleted tables will contain more then one row.

Examples:

```
CREATE TRIGGER tblSomething_InsertOrUpdate ON tblSomething
FOR INSERT
AS

    INSERT INTO tblAudit (TableName, RecordId, Action)
    SELECT 'tblSomething', Id, 'Inserted'
    FROM Inserted

GO

CREATE TRIGGER tblSomething_InsertOrUpdate ON tblSomething
FOR UPDATE
AS

    INSERT INTO tblAudit (TableName, RecordId, Action)
    SELECT 'tblSomething', Id, 'Updated'
    FROM Inserted

GO

CREATE TRIGGER tblSomething_InsertOrUpdate ON tblSomething
FOR DELETE
AS

    INSERT INTO tblAudit (TableName, RecordId, Action)
    SELECT 'tblSomething', Id, 'Deleted'
    FROM Deleted

GO
```

All the examples above will add records to tblAudit whenever a record is added, deleted or updated in tblSomething.

# Section 61.2: Types and classifications of Trigger

In SQL Server, there are two categories of triggers: DDL Triggers and DML Triggers.

DDL Triggers are fired in response to Data Definition Language (DDL) events. These events primarily correspond to Transact-SQL statements that start with the keywords `CREATE`, `ALTER` and `DROP`.

DML Triggers are fired in response to Data Manipulation Language (DML) events. These events corresponds to Transact-SQL statements that start with the keywords `INSERT`, `UPDATE` and `DELETE`.

DML triggers are classified into two main types:

1. After Triggers (for triggers)
    - AFTER INSERT Trigger.
    - AFTER UPDATE Trigger.
    - AFTER DELETE Trigger.
2. Instead of triggers
    - INSTEAD OF INSERT Trigger.
    - INSTEAD OF UPDATE Trigger.
    - INSTEAD OF DELETE Trigger.

# Chapter 62: Cursors

## Section 62.1: Basic Forward Only Cursor

Normally you would want to avoid using cursors as they can have negative impacts on performance. However in some special cases you may need to loop through your data record by record and perform some action.

```
DECLARE @orderId AS INT

-- here we are creating our cursor, as a local cursor and only allowing
-- forward operations
DECLARE rowCursor CURSOR LOCAL FAST_FORWARD FOR
    -- this is the query that we want to loop through record by record
    SELECT [OrderId]
    FROM [dbo].[Orders]

-- first we need to open the cursor
OPEN rowCursor

-- now we will initialize the cursor by pulling the first row of data, in this example the
[OrderId] column,
-- and storing the value into a variable called @orderId
FETCH NEXT FROM rowCursor INTO @orderId

-- start our loop and keep going until we have no more records to loop through
WHILE @@FETCH_STATUS = 0
BEGIN

    PRINT @orderId

    -- this is important, as it tells SQL Server to get the next record and store the [OrderId]
column value into the @orderId variable
    FETCH NEXT FROM rowCursor INTO @orderId

END

-- this will release any memory used by the cursor
CLOSE rowCursor
DEALLOCATE rowCursor
```

## Section 62.2: Rudimentary cursor syntax

A simple cursor syntax, operating on a few example test rows:

```
/* Prepare test data */
DECLARE @test_table TABLE
(
    Id INT,
    Val VARCHAR(100)
);
INSERT INTO @test_table(Id, Val)
VALUES
    (1, 'Foo'),
    (2, 'Bar'),
    (3, 'Baz');
/* Test data prepared */

/* Iterator variable @myId, for example sake */
```

```
DECLARE @myId INT;

/* Cursor to iterate rows and assign values to variables */
DECLARE myCursor CURSOR FOR
    SELECT Id
    FROM @test_table;

/* Start iterating rows */
OPEN myCursor;
FETCH NEXT FROM myCursor INTO @myId;

/* @@FETCH_STATUS global variable will be 1 / true until there are no more rows to fetch */
WHILE @@FETCH_STATUS = 0
BEGIN

    /* Write operations to perform in a loop here. Simple SELECT used for example */
    SELECT Id, Val
    FROM @test_table
    WHERE Id = @myId;

    /* Set variable(s) to the next value returned from iterator; this is needed otherwise the
cursor will loop infinitely. */
    FETCH NEXT FROM myCursor INTO @myId;
END
/* After all is done, clean up */
CLOSE myCursor;
DEALLOCATE myCursor;
```

Results from SSMS. Note that these are all separate queries, they are in no way unified. Notice how the query engine processes each iteration one by one instead of as a set.

```
        Id          Val
1                   Foo
(1 row(s) affected)
        Id          Val
2                   Bar
(1 row(s) affected)
        Id          Val
3                   Baz
(1 row(s) affected)
```

# Chapter 63: Transaction isolation levels

## Section 63.1: Read Committed

Version ≥ SQL Server 2008 R2

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

This isolation level is the 2nd most permissive. It prevents dirty reads. The behavior of `READ COMMITTED` depends on the setting of the `READ_COMMITTED_SNAPSHOT`:

- If set to OFF (the default setting) the transaction uses shared locks to prevent other transactions from modifying rows used by the current transaction, as well as block the current transaction from reading rows modified by other transactions.
- If set to ON, the `READCOMMITTEDLOCK` table hint can be used to request shared locking instead of row versioning for transactions running in `READ COMMITTED` mode.

Note: `READ COMMITTED` is the default SQL Server behavior.

## Section 63.2: What are "dirty reads"?

Dirty reads (or uncommitted reads) are reads of rows which are being modified by an open transaction.

This behavior can be replicated by using 2 separate queries: one to open a transaction and write some data to a table without committing, the other to select the data to be written (but not yet committed) with this isolation level.

**Query 1** - Prepare a transaction but do not finish it:

```
CREATE TABLE dbo.demo (
    col1 INT,
    col2 VARCHAR(255)
);
GO
--This row will get committed normally:
BEGIN TRANSACTION;
    INSERT INTO dbo.demo(col1, col2)
    VALUES (99, 'Normal transaction');
COMMIT TRANSACTION;
--This row will be "stuck" in an open transaction, causing a dirty read
BEGIN TRANSACTION;
    INSERT INTO dbo.demo(col1, col2)
    VALUES (42, 'Dirty read');
--Do not COMMIT TRANSACTION or ROLLBACK TRANSACTION here
```

**Query 2** - Read the rows including the open transaction:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM dbo.demo;
```

Returns:

```
col1        col2
----------- ---------------------------------------
99          Normal transaction
```

```
42          Dirty read
```

P.S.: Don't forget to clean up this demo data:

```
COMMIT TRANSACTION;
DROP TABLE dbo.demo;
GO
```

## Section 63.3: Read Uncommitted

Version ≥ SQL Server 2008 R2

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

This is the most permissive isolation level, in that it does not cause any locks at all. It specifies that statements can read all rows, including rows that have been written in transactions but not yet committed (i.e., they are still in transaction). This isolation level can be subject to "dirty reads".

## Section 63.4: Repeatable Read

Version ≥ SQL Server 2008 R2

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

This transaction isolation level is slightly less permissive than `READ COMMITTED`, in that shared locks are placed on all data read by each statement in the transaction and are held **until the transaction completes**, as opposed to being released after each statement.

Note: Use this option only when necessary, as it is more likely to cause database performance degradation as well as deadlocks than `READ COMMITTED`.

## Section 63.5: Snapshot

Version ≥ SQL Server 2008 R2

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

Specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction, i.e., it will only read data that has been committed prior to the transaction starting.

`SNAPSHOT` transactions do not request or cause any locks on the data that is being read, as it is only reading the version (or snapshot) of the data that existed at the time the transaction began.

A transaction running in `SNAPSHOT` isolation level read only its own data changes while it is running. For example, a transaction could update some rows and then read the updated rows, but that change will only be visible to the current transaction until it is committed.

Note: The `ALLOW_SNAPSHOT_ISOLATION` database option must be set to ON before the `SNAPSHOT` isolation level can be used.

## Section 63.6: Serializable

Version ≥ SQL Server 2008 R2

```
SET TRANSACTION ISOLATION LEVEL SERIALIZEABLE
```

This isolation level is the most restrictive. It requests **range locks** the range of key values that are read by each statement in the transaction. This also means that `INSERT` statements from other transactions will be blocked if the rows to be inserted are in the range locked by the current transaction.

This option has the same effect as setting `HOLDLOCK` on all tables in all `SELECT` statements in a transaction.

Note: This transaction isolation has the lowest concurrency and should only be used when necessary.

# Chapter 64: Advanced options

## Section 64.1: Enable and show advanced options

```
Exec sp_configure 'show advanced options' ,1
RECONFIGURE
GO
-- Show all configure
sp_configure
```

## Section 64.2: Enable backup compression default

```
Exec sp_configure 'backup compression default',1
GO
RECONFIGURE;
```

## Section 64.3: Enable cmd permission

```
EXEC sp_configure 'xp_cmdshell', 1
GO
RECONFIGURE
```

## Section 64.4: Set default fill factor percent

```
sp_configure 'fill factor', 100;
GO
RECONFIGURE;
```

The server must be restarted before the change can take effect.

## Section 64.5: Set system recovery interval

```
USE master;
GO
-- Set recovery every 3 min
EXEC sp_configure 'recovery interval', '3';
RECONFIGURE WITH OVERRIDE;
```

## Section 64.6: Set max server memory size

```
USE master
EXEC sp_configure 'max server memory (MB)', 64
RECONFIGURE WITH OVERRIDE
```

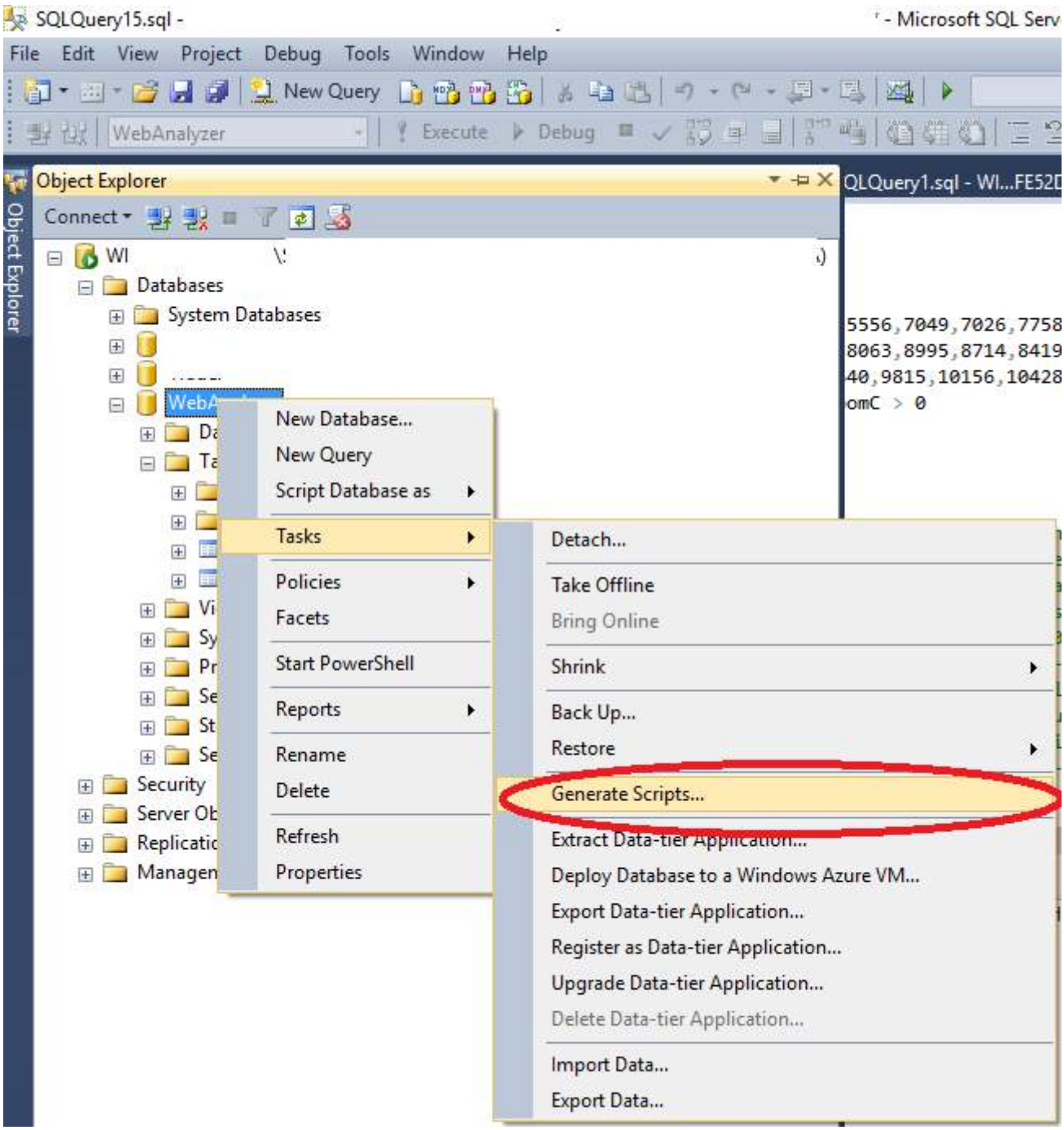
## Section 64.7: Set number of checkpoint tasks

```
EXEC sp_configure "number of checkpoint tasks", 4
```

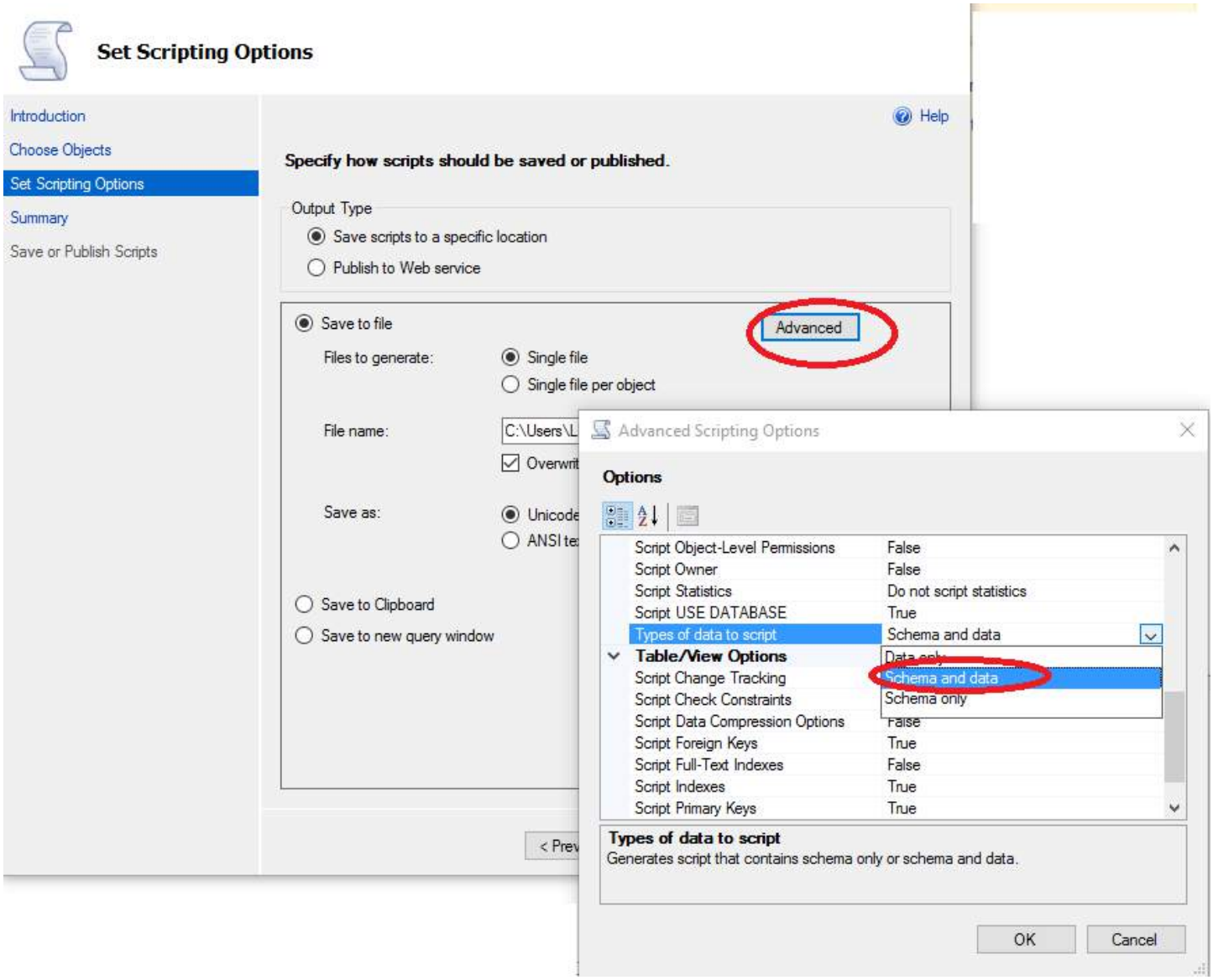
# Chapter 65: Migration

## Section 65.1: How to generate migration scripts

1. Click `Right Mouse` on Database you want to migrate then -> `Tasks` -> `Generate Scripts...`




2. Wizard will open click `Next` then chose objects you want to migrate and click `Next` again, then click `Advanced` scroll a bit down and in `Types of data to script` choose `Schema and data` (unless you want only structures)




3. Click couple more times `Next` and `Finish` and you should have your database scripted in `.sql` file.
4. run `.sql` file on your new server, and you should be done.

# Chapter 66: Table Valued Parameters

## Section 66.1: Using a table valued parameter to insert multiple rows to a table

First, define a used defined table type to use:

```
CREATE TYPE names as TABLE
(
    FirstName varchar(10),
    LastName varchar(10)
)
GO
```

Create the stored procedure:

```
CREATE PROCEDURE prInsertNames
(
    @Names dbo.Names READONLY -- Note: You must specify the READONLY
)
AS

INSERT INTO dbo.TblNames (FirstName, LastName)
SELECT FirstName, LastName
FROM @Names
GO
```

Executing the stored procedure:

```
DECLARE @names dbo.Names
INSERT INTO @Names VALUES
('Zohar', 'Peled'),
('First', 'Last')

EXEC dbo.prInsertNames @Names
```

# Chapter 67: DBMAIL

## Section 67.1: Send simple email

This code sends a simple text-only email to recipient@someaddress.com

```
EXEC msdb.dbo.sp_send_dbmail
    @profile_name = 'The Profile Name',
    @recipients = 'recipient@someaddress.com',
    @body = 'This is a simple email sent from SQL Server.',
    @subject = 'Simple email'
```

## Section 67.2: Send results of a query

This attaches the results of the query `SELECT * FROM Users` and sends it to `recipient@someaddress.com`

```
EXEC msdb.dbo.sp_send_dbmail
    @profile_name = 'The Profile Name',
    @recipients = 'recipient@someaddress.com',
    @query = 'SELECT * FROM Users',
    @subject = 'List of users',
    @attach_query_result_as_file = 1;
```

## Section 67.3: Send HTML email

HTML content must be passed to `sp_send_dbmail`

Version ≥ SQL Server 2012

```
DECLARE @html VARCHAR(MAX);
SET @html = CONCAT
(
    '<html><body>',
    '<h1>Some Header Text</h1>',
    '<p>Some paragraph text</p>',
    '</body></html>'
)
```

Version < SQL Server 2012

```
DECLARE @html VARCHAR(MAX);
SET @html =
    '<html><body>' +
    '<h1>Some Header Text</h1>' +
    '<p>Some paragraph text</p>' +
    '</body></html>';
```

Then use the `@html` variable with the `@body argument`. The HTML string can also be passed directly to `@body`, although it may make the code harder to read.

```
EXEC msdb.dbo.sp_send_dbmail
    @recipients='recipient@someaddress.com',
    @subject = 'Some HTML content',
    @body = @html,
    @body_format = 'HTML';
```

# Chapter 68: In-Memory OLTP (Hekaton)

## Section 68.1: Declare Memory-Optimized Table Variables

For faster performance you can memory-optimize your table variable. Here is the T-SQL for a traditional table variable:

```
DECLARE @tvp TABLE
(
    col1   INT NOT NULL ,
    Col2   CHAR(10)
);
```

To define memory-optimized variables, you must first create a memory-optimized table type and then declare a variable from it:

```
CREATE TYPE dbo.memTypeTable
AS TABLE
(
    Col1  INT NOT NULL INDEX ix1,
    Col2  CHAR(10)
)
WITH
    (MEMORY_OPTIMIZED = ON);
```

Then we can use the table type like this:

```
DECLARE @tvp memTypeTable
insert INTO @tvp
values (1,'1'),(2,'2'),(3,'3'),(4,'4'),(5,'5'),(6,'6')

SELECT * FROM @tvp
```

Result:

```
Col1    Col2
1       1
2       2
3       3
4       4
5       5
6       6
```

## Section 68.2: Create Memory Optimized Table

```
-- Create demo database
CREATE DATABASE SQL2016_Demo
 ON  PRIMARY
(
    NAME = N'SQL2016_Demo',
    FILENAME = N'C:\Dump\SQL2016_Demo.mdf',
    SIZE = 5120KB,
    FILEGROWTH = 1024KB
 )
 LOG ON
 (
```

```
    NAME = N'SQL2016_Demo_log',
    FILENAME = N'C:\Dump\SQL2016_Demo_log.ldf',
    SIZE = 1024KB,
    FILEGROWTH = 10%
 )
GO

use SQL2016_Demo
go

-- Add Filegroup by MEMORY_OPTIMIZED_DATA type
ALTER DATABASE SQL2016_Demo
    ADD FILEGROUP MemFG CONTAINS MEMORY_OPTIMIZED_DATA
GO


--Add a file to defined filegroup
ALTER DATABASE SQL2016_Demo ADD FILE
    (
        NAME = MemFG_File1,
        FILENAME = N'C:\Dump\MemFG_File1' -- your file path, check directory exist before executing
this code
    )
TO FILEGROUP MemFG
GO

--Object Explorer -- check database created
GO

-- create memory optimized table 1
CREATE TABLE dbo.MemOptTable1
(
    Column1     INT         NOT NULL,
    Column2     NVARCHAR(4000)  NULL,
    SpidFilter  SMALLINT    NOT NULL   DEFAULT (@@spid),

    INDEX ix_SpidFiler NONCLUSTERED (SpidFilter),
    INDEX ix_SpidFilter HASH (SpidFilter) WITH (BUCKET_COUNT = 64),

    CONSTRAINT CHK_soSessionC_SpidFilter
        CHECK ( SpidFilter = @@spid ),
)
    WITH
        (MEMORY_OPTIMIZED = ON,
         DURABILITY = SCHEMA_AND_DATA);  --or DURABILITY = SCHEMA_ONLY
go

-- create memory optimized table 2
CREATE TABLE MemOptTable2
(
    ID INT NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT = 10000),
    FullName NVARCHAR(200) NOT NULL,
    DateAdded DATETIME NOT NULL
) WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
GO
```

## Section 68.3: Show created .dll files and tables for Memory Optimized Tables

```
SELECT
    OBJECT_ID('MemOptTable1') AS MemOptTable1_ObjectID,
```

```
    OBJECT_ID('MemOptTable2') AS MemOptTable2_ObjectID
GO

SELECT
    name,description
FROM sys.dm_os_loaded_modules
WHERE name LIKE '%XTP%'
GO
```

Show all Memory Optimized Tables:

```
SELECT
    name,type_desc,durability_desc,Is_memory_Optimized
FROM sys.tables
    WHERE Is_memory_Optimized = 1
GO
```

## Section 68.4: Create Memory Optimized System-Versioned Temporal Table

```
CREATE TABLE [dbo].[MemOptimizedTemporalTable]
(
    [BusinessDocNo] [bigint] NOT NULL,
    [ProductCode] [int] NOT NULL,
    [UnitID] [tinyint] NOT NULL,
    [PriceID] [tinyint] NOT NULL,
    [SysStartTime] [datetime2](7) GENERATED ALWAYS AS ROW START NOT NULL,
    [SysEndTime] [datetime2](7) GENERATED ALWAYS AS ROW END NOT NULL,
    PERIOD FOR SYSTEM_TIME ([SysStartTime], [SysEndTime]),

    CONSTRAINT [PK_MemOptimizedTemporalTable]  PRIMARY KEY NONCLUSTERED
    (
        [BusinessDocNo] ASC,
        [ProductCode] ASC
    )
)
WITH (
    MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA, -- Memory Optimized Option ON
    SYSTEM_VERSIONING = ON (HISTORY_TABLE = [dbo].[MemOptimizedTemporalTable_History] ,
DATA_CONSISTENCY_CHECK = ON )
)
```

more informations

## Section 68.5: Memory-Optimized Table Types and Temp tables

For example, this is traditional tempdb-based table type:

```
CREATE TYPE dbo.testTableType AS TABLE
(
   col1 INT NOT NULL,
   col2 CHAR(10)
);
```

To memory-optimize this table type simply add the option `memory_optimized=on`, and add an index if there is none on the original type:

```
CREATE TYPE dbo.testTableType AS TABLE
(
   col1 INT NOT NULL,
   col2 CHAR(10)
)WITH (MEMORY_OPTIMIZED=ON);
```

Global temporary table is like this:

```
CREATE TABLE ##tempGlobalTabel
(
    Col1   INT   NOT NULL ,
    Col2   NVARCHAR(4000)
);
```

Memory-optimized global temporary table:

```
CREATE TABLE dbo.tempGlobalTabel
(
    Col1   INT   NOT NULL   INDEX ix NONCLUSTERED,
    Col2   NVARCHAR(4000)
)
    WITH
        (MEMORY_OPTIMIZED = ON,
         DURABILITY = SCHEMA_ONLY);
```

To memory-optimize global temp tables (##temp):

1. Create a new `SCHEMA_ONLY` memory-optimized table with the same schema as the global `##temp` table
    - Ensure the new table has at least one index
2. Change all references to `##temp` in your Transact-SQL statements to the new memory-optimized table temp
3. Replace the `DROP TABLE ##temp` statements in your code with `DELETE FROM temp`, to clean up the contents
4. Remove the `CREATE TABLE ##temp` statements from your code – these are now redundant

more informations

# Chapter 69: Temporal Tables

## Section 69.1: CREATE Temporal Tables

```
CREATE TABLE dbo.Employee
(
  [EmployeeID] int NOT NULL PRIMARY KEY CLUSTERED
  , [Name] nvarchar(100) NOT NULL
  , [Position] varchar(100) NOT NULL
  , [Department] varchar(100) NOT NULL
  , [Address] nvarchar(1024) NOT NULL
  , [AnnualSalary] decimal (10,2) NOT NULL
  , [ValidFrom] datetime2 (2) GENERATED ALWAYS AS ROW START
  , [ValidTo] datetime2 (2) GENERATED ALWAYS AS ROW END
  , PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo)
 )
 WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.EmployeeHistory));
```

**INSERTS:** On an **INSERT**, the system sets the value for the **ValidFrom** column to the begin time of the current transaction (in the UTC time zone) based on the system clock and assigns the value for the **ValidTo** column to the maximum value of 9999-12-31. This marks the row as open.

**UPDATES:** On an **UPDATE**, the system stores the previous value of the row in the history table and sets the value for the **ValidTo** column to the begin time of the current transaction (in the UTC time zone) based on the system clock. This marks the row as closed, with a period recorded for which the row was valid. In the current table, the row is updated with its new value and the system sets the value for the **ValidFrom** column to the begin time for the transaction (in the UTC time zone) based on the system clock. The value for the updated row in the current table for the **ValidTo** column remains the maximum value of 9999-12-31.

**DELETES**: On a **DELETE**, the system stores the previous value of the row in the history table and sets the value for the **ValidTo** column to the begin time of the current transaction (in the UTC time zone) based on the system clock. This marks the row as closed, with a period recorded for which the previous row was valid. In the current table, the row is removed. Queries of the current table will not return this row. Only queries that deal with history data return data for which a row is closed.

**MERGE**: On a **MERGE**, the operation behaves exactly as if up to three statements (an **INSERT**, an **UPDATE**, and/or a **DELETE**) executed, depending on what is specified as actions in the **MERGE** statement.

**Tip :** The times recorded in the system datetime2 columns are based on the begin time of the transaction itself. For example, all rows inserted within a single transaction will have the same UTC time recorded in the column corresponding to the start of the **SYSTEM_TIME** period.

## Section 69.2: FOR SYSTEM_TIME ALL

Returns the union of rows that belong to the current and the history table.

```
SELECT * FROM Employee
    FOR SYSTEM_TIME ALL
```

## Section 69.3: Creating a Memory-Optimized System-Versioned Temporal Table and cleaning up the SQL Server history table

Creating a temporal table with a default history table is a convenient option when you want to control naming and

still rely on system to create history table with default configuration. In the example below, a new system-versioned memory-optimized temporal table linked to a new disk-based history table.

```
CREATE SCHEMA History
GO
CREATE TABLE dbo.Department
(
    DepartmentNumber char(10) NOT NULL PRIMARY KEY NONCLUSTERED,
    DepartmentName varchar(50) NOT NULL,
    ManagerID int  NULL,
    ParentDepartmentNumber char(10) NULL,
    SysStartTime datetime2 GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,
    SysEndTime datetime2 GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,
    PERIOD FOR SYSTEM_TIME (SysStartTime,SysEndTime)
)
WITH
    (
        MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA,
        SYSTEM_VERSIONING = ON ( HISTORY_TABLE = History.DepartmentHistory )
    );
```

**Cleaning up the SQL Server history table** Over time the history table can grow significantly. Since inserting, updating or deleting data from the history table are not allowed, the only way to clean up the history table is first to disable system versioning:

```
ALTER TABLE dbo.Employee
```

SET (SYSTEM_VERSIONING = OFF); GO

Delete unnecessary data from the history table:

```
    DELETE FROM dbo.EmployeeHistory
```

WHERE EndTime <= '2017-01-26 14:00:29';

and then re-enable system versioning:

```
ALTER TABLE dbo.Employee
```

SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = [dbo].[EmployeeHistory], DATA_CONSISTENCY_CHECK = ON));

Cleaning the history table in Azure SQL Databases is a little different, since Azure SQL databases have built-in support for cleaning of the history table. First, temporal history retention cleanup need to be enable on a database level:

```
ALTER DATABASE CURRENT
```

SET TEMPORAL_HISTORY_RETENTION ON GO

Then set the retention period per table:

```
ALTER TABLE dbo.Employee
```

SET (SYSTEM_VERSIONING = ON (HISTORY_RETENTION_PERIOD = 90 DAYS));

This will delete all data in the history table older than 90 days. SQL Server 2016 on-premise databases do not

support TEMPORAL_HISTORY_RETENTION and HISTORY_RETENTION_PERIOD and either of the above two queries are executed on the SQL Server 2016 on-premise databases the following errors will occur.

For TEMPORAL_HISTORY_RETENTION error will be:

```
Msg 102, Level 15, State 6, Line 34
```

Incorrect syntax near 'TEMPORAL_HISTORY_RETENTION'.

For HISTORY_RETENTION_PERIOD error will be:

```
Msg 102, Level 15, State 1, Line 39
```

Incorrect syntax near 'HISTORY_RETENTION_PERIOD'.

## Section 69.4: FOR SYSTEM_TIME BETWEEN <start_date_time> AND <end_date_time>

Same as above in the FOR SYSTEM_TIME FROM <start_date_time>TO <end_date_time> description, except the table of rows returned includes rows that became active on the upper boundary defined by the <end_date_time> endpoint.

```
SELECT * FROM Employee
    FOR SYSTEM_TIME BETWEEN  '2015-01-01' AND '2015-12-31'
```

## Section 69.5: FOR SYSTEM_TIME FROM <start_date_time> TO <end_date_time>

Returns a table with the values for all row versions that were active within the specified time range, regardless of whether they started being active before the <start_date_time> parameter value for the FROM argument or ceased being active after the <end_date_time> parameter value for the TO argument. Internally, a union is performed between the temporal table and its history table and the results are filtered to return the values for all row versions that were active at any time during the time range specified. Rows that became active exactly on the lower boundary defined by the FROM endpoint are included and records that became active exactly on the upper boundary defined by the TO endpoint are not included.

```
SELECT * FROM Employee
    FOR SYSTEM_TIME FROM '2015-01-01' TO '2015-12-31'
```

## Section 69.6: FOR SYSTEM_TIME CONTAINED IN (<start_date_time> , <end_date_time>)

Returns a table with the values for all row versions that were opened and closed within the specified time range defined by the two datetime values for the CONTAINED IN argument. Rows that became active exactly on the lower boundary or ceased being active exactly on the upper boundary are included.

```
SELECT * FROM Employee
    FOR SYSTEM_TIME CONTAINED IN ('2015-04-01', '2015-09-25')
```

## Section 69.7: How do I query temporal data?

```
SELECT * FROM Employee
```

```
    FOR SYSTEM_TIME
        BETWEEN '2014-01-01 00:00:00.0000000' AND '2015-01-01 00:00:00.0000000'
            WHERE EmployeeID = 1000 ORDER BY ValidFrom;
```

## Section 69.8: Return actual value specified point in time(FOR SYSTEM_TIME AS OF <date_time>)

Returns a table with a rows containing the values that were actual (current) at the specified point in time in the past.

```
SELECT * FROM Employee
    FOR SYSTEM_TIME AS  OF '2016-08-06 08:32:37.91'
```

# Chapter 70: Use of TEMP Table

## Section 70.1: Dropping temp tables

Temp tables must have unique IDs (within the session, for local temp tables, or within the server, for global temp tables). Trying to create a table using a name that already exists will return the following error:

```
There is already an object named '#tempTable' in the database.
```

If your query produces temp tables, and you want to run it more than once, you will need to drop the tables before trying to generate them again. The basic syntax for this is:

```
drop table #tempTable
```

Trying to execute this syntax before the table exists (e.g. on the first run of your syntax) will cause another error:

```
Cannot drop the table '#tempTable', because it does not exist or you do not have permission.
```

To avoid this, you can check to see if the table already exists before dropping it, like so:

```
IF OBJECT_ID ('tempdb..#tempTable', 'U') is not null DROP TABLE #tempTable
```

## Section 70.2: Local Temp Table

- Will be available till the current connection persists for the user.

  Automatically deleted when the user disconnects.

  The name should start with # (#temp)

  ```
  CREATE TABLE #LocalTempTable(
                  StudentID      int,
                  StudentName    varchar(50),
                  StudentAddress varchar(150))
  ```

```
insert into #LocalTempTable values ( 1, 'Ram','India');

select * from #LocalTempTable
```

After executing all these statements if we close the query window and open it again and try inserting and select it will show an error message

```
“Invalid object name #LocalTempTable”
```

## Section 70.3: Global Temp Table

- Will start with ## (##temp).

  Will be deleted only if user disconnects all connections.

  It behaves like a permanent table.

```
CREATE TABLE ##NewGlobalTempTable(
               StudentID      int,
               StudentName    varchar(50),
               StudentAddress varchar(150))

Insert Into ##NewGlobalTempTable values ( 1,'Ram','India');
Select * from ##NewGlobalTempTable
```

Note: These are viewable by all users of the database, irrespective of permissions level.

# Chapter 71: Scheduled Task or Job

SQL Server Agent uses SQL Server to store job information. Jobs contain one or more job steps. Each step contains its own task,i.e: backing up a database. SQL Server Agent can run a job on a schedule, in response to a specific event, or on demand.

## Section 71.1: Create a scheduled Job

**Create a Job**

- To add a job first we have to use a stored procedure named sp_add_job

```
USE msdb ;
GO
EXEC dbo.sp_add_job
@job_name = N'Weekly Job' ;  -- the job name
```

- Then we have to add a job step using a stored procedure named sp_add_jobStep

```
EXEC sp_add_jobstep
@job_name = N'Weekly Job',  -- Job name to add a step
@step_name = N'Set database to read only',  -- step name
@subsystem = N'TSQL',  -- Step type
@command = N'ALTER DATABASE SALES SET READ_ONLY',   -- Command
@retry_attempts = 5,  --Number of attempts
@retry_interval = 5 ; -- in minutes
```

- Target the job to a server

```
EXEC dbo.sp_add_jobserver
@job_name = N'Weekly Sales Data Backup',
@server_name = 'MyPC\data;   -- Default is LOCAL
GO
```

**Create a schedule using SQL**

To Create a schedule we have to use a system stored procedure called sp_add_schedule

```
USE msdb
GO

EXEC sp_add_schedule
    @schedule_name = N'NightlyJobs' ,  -- specify the schedule name
    @freq_type = 4,   -- A value indicating when a job is to be executed (4) means Daily
    @freq_interval = 1,  -- The days that a job is executed and depends on the value of
`freq_type`.
    @active_start_time = 010000 ;   -- The time on which execution of a job can begin
GO
```

There are more parameters that can be used with `sp_add_schedule` you can read more about in the the link provided above.

**Attaching schedule to a JOB**

To attach a schedule to an SQL agent job you have to use a stored procedure called sp_attach_schedule

```
-- attaches the schedule to the job BackupDatabase
EXEC sp_attach_schedule
   @job_name = N'BackupDatabase',  -- The job name to attach with
   @schedule_name = N'NightlyJobs' ;  -- The schedule name
GO
```

# Chapter 72: Isolation levels and locking

## Section 72.1: Examples of setting the isolation level

Example of setting the isolation level:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM Products WHERE ProductId=1;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; --return to the default one
```

1. `READ UNCOMMITTED` - means that a query in the current transaction can't access the modified data from another transaction that is not yet committed - no dirty reads! BUT, nonrepeatable reads and phantom reads are possible, because data can still be modified by other transactions.
2. `REPEATABLE READ` - means that a query in the the current transaction can't access the modified data from another transaction that is not yet committed - no dirty reads! No other transactions can modify data being read by the current transaction until it is completed, which eliminates NONREPEATABLE reads. BUT, if another transaction inserts NEW ROWS and the query is executed more then once, phantom rows can appear starting the second read (if it matches the where statement of the query).
3. `SNAPSHOT` - only able to return data that exists at the beginning of the query. Ensures consistency of the data. It prevents dirty reads, nonrepeatable reads and phantom reads. To use that - DB configurationis required:

```
ALTER DATABASE DBTestName SET ALLOW_SNAPSHOT_ISOLATION ON;GO;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

4. `READ COMMITTED` - default isolation of the SQL server. It prevents reading the data that is changed by another transaction until committed. It uses shared locking and row versioning on the tables which prevents dirty reads. It depends on DB configuration READ_COMMITTED_SNAPSHOT - if enabled - row versioning is used. to enable - use this:

```
ALTER DATABASE DBTestName SET ALLOW_SNAPSHOT_ISOLATION ON;GO;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; --return to the default one
```

5. `SERIALIZABLE` - uses physical locks that are acquired and held until end of the transaction, which prevents dirty reads, phantom reads, nonrepeatable reads. BUT, it impacts on the performance of the DataBase, because the concurrent transactions are serialized and are being executed one by one.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
```

# Chapter 73: Sorting/ordering rows

## Section 73.1: Basics

First, let's setup the example table.

```
-- Create a table as an example
CREATE TABLE SortOrder
(
    ID INT IDENTITY PRIMARY KEY,
    [Text] VARCHAR(256)
)
GO

-- Insert rows into the table
INSERT INTO SortOrder ([Text])
SELECT ('Lorem ipsum dolor sit amet, consectetur adipiscing elit')
UNION ALL SELECT ('Pellentesque eu dapibus libero')
UNION ALL SELECT ('Vestibulum et consequat est, ut hendrerit ligula')
UNION ALL SELECT ('Suspendisse sodales est congue lorem euismod, vel facilisis libero pulvinar')
UNION ALL SELECT ('Suspendisse lacus est, aliquam at varius a, fermentum nec mi')
UNION ALL SELECT ('Praesent tincidunt tortor est, nec consequat dolor malesuada quis')
UNION ALL SELECT ('Quisque at tempus arcu')
GO
```

Remember that when retrieving data, if you don't specify a row ordering clause (ORDER BY) SQL server does not guarantee the sorting (order of the columns) **at any time**. Really, at any time. And there's no point arguing about that, it has been shown literaly thousands of times and all over the internet.

No ORDER BY == no sorting. End of story.

```
-- It may seem the rows are sorted by identifiers,
-- but there is really no way of knowing if it will always work.
-- And if you leave it like this in production, Murphy gives you a 100% that it wont.
SELECT * FROM SortOrder
GO
```

There are two directions data can be ordered by:

- ascending (moving upwards), using ASC
- descending (moving downwards), using DESC

```
-- Ascending - upwards
SELECT * FROM SortOrder ORDER BY ID ASC
GO

-- Ascending is default
SELECT * FROM SortOrder ORDER BY ID
GO

-- Descending - downwards
SELECT * FROM SortOrder ORDER BY ID DESC
GO
```

When ordering by the textual column ((n)char or (n)varchar), pay attention that the order respects the collation. For more information on collation look up for the topic.

Ordering and sorting of data can consume resources. This is where properly created indexes come handy. For more information on indexes look up for the topic.

There is a possibility to pseudo-randomize the order of rows in your resultset. Just force the ordering to appear nondeterministic.

```
SELECT * FROM SortOrder ORDER BY CHECKSUM(NEWID())
GO
```

Ordering can be remembered in a stored procedure, and that's the way you should do it if it is the last step of manipulating the rowset before showing it to the end user.

```
CREATE PROCEDURE GetSortOrder
AS
    SELECT *
    FROM SortOrder
    ORDER BY ID DESC
GO

EXEC GetSortOrder
GO
```

There is a limited (and hacky) support for ordering in the SQL Server views as well, but be encouraged NOT to use it.

```
/* This may or may not work, and it depends on the way
   your SQL Server and updates are installed */
CREATE VIEW VwSortOrder1
AS
    SELECT TOP 100 PERCENT *
    FROM SortOrder
    ORDER BY ID DESC
GO

SELECT * FROM VwSortOrder1
GO

-- This will work, but hey... should you really use it?
CREATE VIEW VwSortOrder2
AS
    SELECT TOP 99999999 *
    FROM SortOrder
    ORDER BY ID DESC
GO

SELECT * FROM VwSortOrder2
GO
```

For ordering you can either use column names, aliases or column numbers in your ORDER BY.

```
SELECT *
FROM SortOrder
ORDER BY [Text]

-- New resultset column aliased as 'Msg', feel free to use it for ordering
SELECT ID, [Text] + ' (' + CAST(ID AS nvarchar(10)) + ')' AS Msg
FROM SortOrder
ORDER BY Msg

-- Can be handy if you know your tables, but really NOT GOOD for production
```

```
SELECT *
FROM SortOrder
ORDER BY 2
```

I advise against using the numbers in your code, except if you want to forget about it the moment after you execute it.

## Section 73.2: Order by Case

If you want to sort your data numerically or alphabetically, you can simply use `order by [column]`. If you want to sort using a custom hierarchy, use a case statement.

```
Group
-----
Total
Young
MiddleAge
Old
Male
Female
```

Using a basic `order by`:

```
SELECT * FROM MyTable
ORDER BY GROUP
```

returns an alphabetical sort, which isn't always desirable:

```
Group
-----
Female
Male
MiddleAge
Old
Total
Young
```

Adding a 'case' statement, assigning ascending numerical values in the order you want your data sorted:

```
SELECT * FROM MyTable
ORDER BY CASE GROUP
    WHEN 'Total' THEN 10
    WHEN 'Male' THEN 20
    WHEN 'Female' THEN 30
    WHEN 'Young' THEN 40
    WHEN 'MiddleAge' THEN 50
    WHEN 'Old' THEN 60
    END
```

returns data in the order specified:

```
Group
-----
Total
Male
```

```
Female
Young
MiddleAge
Old
```

# Chapter 74: Privileges or Permissions

## Section 74.1: Simple rules

**Granting permission to create tables**

```
USE AdventureWorks;
GRANT CREATE TABLE TO MelanieK;
GO
```

**Granting SHOWPLAN permission to an application role**

```
USE AdventureWorks2012;
GRANT SHOWPLAN TO AuditMonitor;
GO
```

**Granting CREATE VIEW with GRANT OPTION**

```
USE AdventureWorks2012;
GRANT CREATE VIEW TO CarmineEs WITH GRANT OPTION;
GO
```

**Granting all rights to a user on a specific database**

```
use YourDatabase
go
exec sp_addrolemember 'db_owner', 'UserName'
go
```

# Chapter 75: SQLCMD

## Section 75.1: SQLCMD.exe called from a batch file or command line

```
echo off

cls

sqlcmd.exe -S "your server name" -U "sql user name" -P "sql password" -d "name of databse" -Q "here
you may write your query/stored procedure"
```

Batch files like these can be used to automate tasks, for example to make backups of databases at a specified time (can be scheduled with Task Scheduler) for a SQL Server Express version where Agent Jobs can't be used.

# Chapter 76: Resource Governor

## Section 76.1: Reading the Statistics

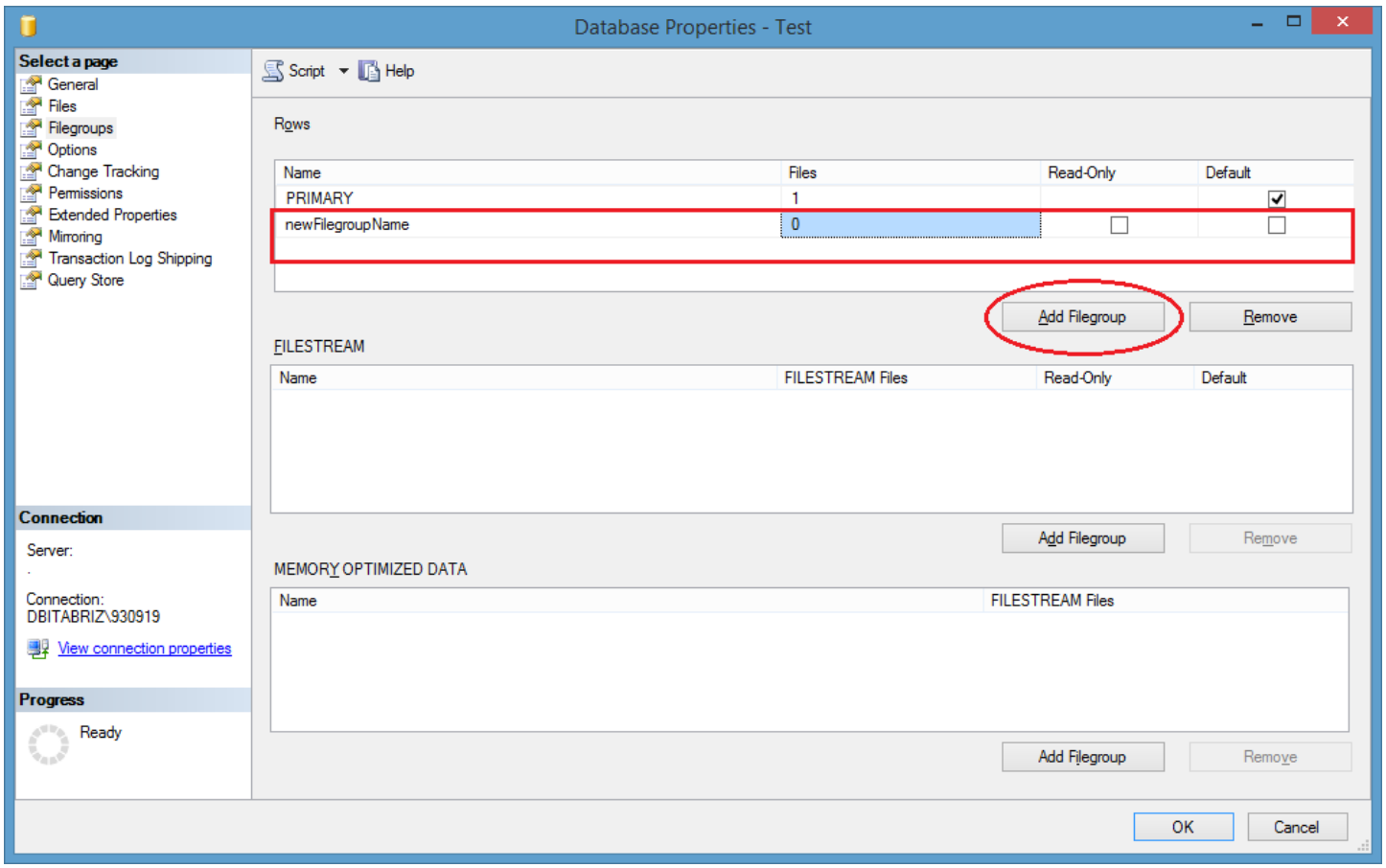
```
SELECT *
FROM sys.dm_resource_governor_workload_groups

SELECT *
FROM sys.dm_resource_governor_resource_pools
```

# Chapter 77: File Group

## Section 77.1: Create filegroup in database

We can create it by two way. First from database properties designer mode:




And by sql scripts:

```
USE master;
GO
-- Create the database with the default data
-- filegroup and a log file. Specify the
-- growth increment and the max size for the
-- primary data file.

CREATE DATABASE TestDB ON PRIMARY
(
    NAME = 'TestDB_Primary',
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\TestDB_Prm.mdf',
    SIZE = 1 GB,
    MAXSIZE = 10 GB,
    FILEGROWTH = 1 GB
), FILEGROUP TestDB_FG1
(
    NAME = 'TestDB_FG1_1',
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\TestDB_FG1_1.ndf',
    SIZE = 10 MB,
    MAXSIZE = 10 GB,
    FILEGROWTH = 1 GB
),
```

```
(
    NAME = 'TestDB_FG1_2',
    FILENAME = 'C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\TestDB_FG1_2.ndf',
    SIZE = 10 MB,
    MAXSIZE = 10 GB,
    FILEGROWTH = 1 GB
) LOG ON
(
    NAME = 'TestDB_log',
    FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\TestDB.ldf',
    SIZE = 10 MB,
    MAXSIZE = 10 GB,
    FILEGROWTH = 1 GB
);

go
ALTER DATABASE TestDB MODIFY FILEGROUP TestDB_FG1 DEFAULT;
go

-- Create a table in the user-defined filegroup.
USE TestDB;
Go

CREATE TABLE MyTable
(
    col1 INT PRIMARY KEY,
    col2 CHAR(8)
)
ON TestDB_FG1;
GO
```

# Chapter 78: Basic DDL Operations in MS SQL Server

## Section 78.1: Getting started

This section describes some basic **DDL** (="**D**ata **D**efinition **L**anguage") commands to create a database, a table within a database, a view and finally a stored procedure.

**Create Database**

The following SQL command creates a new database `Northwind` on the current server, using path`C:\Program Files\Microsoft SQL Server\MSSQL11.INSTSQL2012\MSSQL\DATA\`:

```
USE [master]
GO

CREATE DATABASE [Northwind]
 CONTAINMENT = NONE
 ON  PRIMARY
 (
  NAME = N'Northwind',
  FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL11.INSTSQL2012\MSSQL\DATA\Northwind.mdf'
, SIZE = 5120KB , MAXSIZE = UNLIMITED, FILEGROWTH = 1024KB
 )
 LOG ON
 (
  NAME = N'Northwind_log',
  FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL11.INSTSQL2012\MSSQL\DATA\Northwind_log.ldf' , SIZE = 1536KB , MAXSIZE = 2048GB ,
FILEGROWTH = 10%
 )
GO

ALTER DATABASE [Northwind] SET COMPATIBILITY_LEVEL = 110
GO
```

**Note:** A T-SQL database consists of two files, the database file `*.mdf`, and its transaction log `*.ldf`. Both need to be specified when a new database is created.

**Create Table**

The following SQL command creates a new table `Categories` in the current database, using schema `dbo` (you can switch database context with `Use <DatabaseName>`):

```
CREATE TABLE dbo.Categories(
    CategoryID int IDENTITY NOT NULL,
    CategoryName nvarchar(15) NOT NULL,
    Description ntext NULL,
    Picture image NULL,
      CONSTRAINT PK_Categories PRIMARY KEY CLUSTERED
      (
         CategoryID ASC
      )
      WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
            ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON PRIMARY
) ON PRIMARY TEXTIMAGE_ON PRIMARY
```

### Create View

The following SQL command creates a new view `Summary_of_Sales_by_Year` in the current database, using schema `dbo` (you can switch database context with `Use <DatabaseName>`):

```
CREATE VIEW dbo.Summary_of_Sales_by_Year AS
  SELECT ord.ShippedDate, ord.OrderID, ordSub.Subtotal
  FROM Orders ord
  INNER JOIN [Order Subtotals] ordSub ON ord.OrderID = ordSub.OrderID
```

This will join tables `Orders` and `[Order Subtotals]` to display the columns `ShippedDate`, `OrderID` and `Subtotal`. Because table `[Order Subtotals]` has a blank in its name in the Northwind database, it needs to be enclosed in square brackets.

### Create Procedure

The following SQL command creates a new stored procedure `CustOrdersDetail` in the current database, using schema `dbo` (you can switch database context with `Use <DatabaseName>`):

```
CREATE PROCEDURE dbo.MyCustOrdersDetail @OrderID int, @MinQuantity int=0
AS BEGIN
  SELECT ProductName,
    UnitPrice=ROUND(Od.UnitPrice, 2),
    Quantity,
    Discount=CONVERT(int, Discount * 100),
    ExtendedPrice=ROUND(CONVERT(money, Quantity * (1 - Discount) * Od.UnitPrice), 2)
  FROM Products P, [Order Details] Od
  WHERE Od.ProductID = P.ProductID and Od.OrderID = @OrderID
  and Od.Quantity>=@MinQuantity
END
```

This stored procedure, after it has been created, can be invoked as follows:

```
exec dbo.MyCustOrdersDetail 10248
```

which will return all order details with @OrderId=10248 (and quantity >=0 as default). Or you can specify the optional parameter

```
exec dbo.MyCustOrdersDetail 10248, 10
```

which will return only orders with a minimum quantity of 10 (or more).

# Chapter 79: Subqueries

## Section 79.1: Subqueries

A subquery is a query within another SQL query. A subquery is also called inner query or inner select and the statement containing a subquery is called an outer query or outer select.

**Note**

1. Subqueries must be enclosed within parenthesis,
2. An ORDER BY cannot be used in a subquery.
3. The image type such as BLOB, array, text datatypes are not allowed in subqueries.

Subqueries can be used with select, insert, update and delete statement within where, from, select clause along with IN, comparison operators, etc.

We have a table named ITCompanyInNepal on which we will perform queries to show subqueries examples:

| ID | CompanyName | CompanyAddress | Headquarter | NumberOfEmployee |
|---|---|---|---|---|
| 1 | CompanyOne | Kathmandu | USA | 350 |
| 2 | CompanyTwo | Kathmandu | USA | 310 |
| 3 | CompanyThree | Kathmandu | Nepal | 300 |
| 4 | CompanyFour | Kathmandu | Nepal | 180 |
| 5 | CompanyFive | Birgunj | Denmark | 150 |
| 6 | CompanySix | Janakpur | USA | 100 |
| 7 | CompanySeven | Janakpur | Australia | 100 |
| 8 | CompanyEight | Birganj | Australia | 150 |
| 9 | CompanyNine | Biratnagar | Canada | 200 |
| 10 | CompanyTen | Pokhara | India | 85 |

Examples: **SubQueries With Select Statement**

with **In** operator and **where** clause:

```
SELECT *
FROM ITCompanyInNepal
WHERE Headquarter IN (SELECT Headquarter
                      FROM ITCompanyInNepal
                      WHERE Headquarter = 'USA');
```

with **comparison operator** and **where** clause

```
SELECT *
FROM ITCompanyInNepal
WHERE NumberOfEmployee < (SELECT AVG(NumberOfEmployee)
                          FROM ITCompanyInNepal
                         )
```

with **select** clause

```
SELECT   CompanyName,
         CompanyAddress,
         Headquarter,
         (SELECT SUM(NumberOfEmployee)
         FROM ITCompanyInNepal
```

```
        WHERE Headquarter = 'USA') AS TotalEmployeeHiredByUSAInKathmandu
FROM    ITCompanyInNepal
WHERE   CompanyAddress = 'Kathmandu' AND Headquarter = 'USA'
```

**Subqueries with insert statement**

We have to insert data from IndianCompany table to ITCompanyInNepal. The table for IndianCompany is shown below:

| ID | CompanyName | CompanyAddress | Headquarter | NumberOfEmployee |
|---|---|---|---|---|
| 1 | CompanyA | Banglore | USA | 450 |
| 2 | CompanyB | Banglore | USA | 500 |
| 3 | CompanyC | Hyderabad | Denmark | 480 |
| 4 | CompanyD | Hyderabad | Australia | 780 |
| 5 | CompanyE | Delhi | Canada | 790 |

```
INSERT INTO ITCompanyInNepal
SELECT *
FROM IndianCompany
```

**Subqueries with update statement**

Suppose all the companies whose headquarter is USA decided to fire 50 employees from all US based companies of Nepal due to some change in policy of USA companies.

```
UPDATE ITCompanyInNepal
SET NumberOfEmployee = NumberOfEmployee - 50
WHERE Headquarter IN (SELECT Headquarter
                      FROM ITCompanyInNepal
                      WHERE Headquarter = 'USA')
```

**Subqueries with Delete Statement**

Suppose all the companies whose headquarter is Denmark decided to shutdown their companies from Nepal.

```
DELETE FROM ITCompanyInNepal
WHERE Headquarter IN (SELECT Headquarter
                      FROM ITCompanyInNepal
                      WHERE Headquarter = 'Denmark')
```

# Chapter 80: Pagination

Row Offset and Paging in Various Versions of SQL Server

## Section 80.1: Pagination with OFFSET FETCH

Version ≥ SQL Server 2012

The `OFFSET FETCH` clause implements pagination in a more concise manner. With it, it's possible to skip N1 rows (specified in `OFFSET`) and return the next N2 rows (specified in `FETCH`):

```
SELECT *
FROM sys.objects
ORDER BY object_id
OFFSET 40 ROWS FETCH NEXT 10 ROWS ONLY
```

The `ORDER BY` clause is required in order to provide deterministic results.

## Section 80.2: Paginaton with inner query

In earlier versions of SQL Server, developers had to use double sorting combined with the `TOP` keyword to return rows in a page:

```
 SELECT TOP 10 *
 FROM
 (
    SELECT
    TOP 50 object_id,
        name,
        TYPE,
        create_date
    FROM sys.objects
    ORDER BY name ASC
) AS DATA
ORDER BY name DESC
```

The inner query will return the first 50 rows ordered by `name`. Then the outer query will reverse the order of these 50 rows and select the top 10 rows (these will be last 10 rows in the group before the reversal).

## Section 80.3: Paging in Various Versions of SQL Server

**SQL Server 2012 / 2014**

```
DECLARE @RowsPerPage INT = 10, @PageNumber INT = 4
SELECT OrderId, ProductId
FROM OrderDetail
ORDER BY OrderId
OFFSET (@PageNumber - 1) * @RowsPerPage ROWS
FETCH NEXT @RowsPerPage ROWS ONLY
```

**SQL Server 2005/2008/R2**

```
DECLARE @RowsPerPage INT = 10, @PageNumber INT = 4
SELECT OrderId, ProductId
FROM (
    SELECT OrderId, ProductId, ROW_NUMBER() OVER (ORDER BY OrderId) AS RowNum
    FROM OrderDetail) AS OD
WHERE OD.RowNum BETWEEN ((@PageNumber - 1 ) * @RowsPerPage) + 1
```

```
AND @RowsPerPage * @PageNumber
```

**SQL Server 2000**

```
DECLARE @RowsPerPage INT = 10, @PageNumber INT = 4
SELECT OrderId, ProductId
FROM (SELECT TOP (@RowsPerPage) OrderId, ProductId
        FROM (SELECT TOP ((@PageNumber)*@RowsPerPage) OrderId, ProductId
                FROM OrderDetail
                ORDER BY OrderId) AS OD
    ORDER BY OrderId DESC) AS OD2
ORDER BY OrderId ASC
```

# Section 80.4: SQL Server 2012/2014 using ORDER BY OFFSET and FETCH NEXT

For getting the next 10 rows just run this query:

```
SELECT * FROM TableName ORDER BY id OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

**Key points to consider when using it:**

- `ORDER BY` is mandatory to use `OFFSET` and `FETCH` clause.
- `OFFSET` clause is mandatory with `FETCH`. You can never use, `ORDER BY` ... `FETCH`.
- `TOP` cannot be combined with `OFFSET` and `FETCH` in the same query expression.

# Section 80.5: Pagination using ROW_NUMBER with a Common Table Expression

Version ≥ SQL Server 2008

The `ROW_NUMBER` function can assign an incrementing number to each row in a result set. Combined with a Common Table Expression that uses a `BETWEEN` operator, it is possible to create 'pages' of result sets. For example: page one containing results 1-10, page two containing results 11-20, page three containing results 21-30, and so on.

```
WITH data
AS
(
    SELECT ROW_NUMBER() OVER (ORDER BY name) AS row_id,
        object_id,
        name,
        type,
        create_date
    FROM sys.objects
)
SELECT *
FROM data
WHERE row_id BETWEEN 41 AND 50
```

Note: It is not possible to use `ROW_NUMBER` in a `WHERE` clause like:

```
SELECT object_id,
    name,
    TYPE,
    create_date
FROM sys.objects
WHERE ROW_NUMBER() OVER (ORDER BY name) BETWEEN 41 AND 50
```

Although this would be more convenient, SQL server will return the following error in this case:

> Msg 4108, Level 15, State 1, Line 6
>
> Windowed functions can only appear in the SELECT or ORDER BY clauses.

# Chapter 81: CLUSTERED COLUMNSTORE

## Section 81.1: Adding clustered columnstore index on existing table

CREATE CLUSTERED COLUMNSTORE INDEX enables you to organize a table in column format:

```
DROP TABLE IF EXISTS Product
GO
CREATE TABLE Product (
    Name nvarchar(50) NOT NULL,
    Color nvarchar(15),
    Size nvarchar(5) NULL,
    Price money NOT NULL,
    Quantity int
)
GO
CREATE CLUSTERED COLUMNSTORE INDEX cci ON Product
```

## Section 81.2: Rebuild CLUSTERED COLUMNSTORE index

Clustered column store index can be rebuilt if you have a lot of deleted rows:

```
ALTER INDEX cci ON Products
REBUILD PARTITION = ALL
```

Rebuilding CLUSTERED COLUMNSTORE will "reload" data from the current table into new one and apply compression again, remove deleted rows, etc.

You can rebuild one or more partitions.

## Section 81.3: Table with CLUSTERED COLUMNSTORE index

If you want to have a table organized in column-store format instead of row store, add INDEX cci CLUSTERED COLUMNSTORE in definition of table:

```
DROP TABLE IF EXISTS Product
GO
CREATE TABLE Product (
    ProductID int,
    Name nvarchar(50) NOT NULL,
    Color nvarchar(15),
    Size nvarchar(5) NULL,
    Price money NOT NULL,
    Quantity int,
    INDEX cci CLUSTERED COLUMNSTORE
)
```

COLUMSTORE tables are better for tables where you expect full scans and reports, while row store tables are better for tables where you will read or update smaller sets of rows.

# Chapter 82: Parsename

| 'object_name' | object_piece |
| --- | --- |
| Is the name of the object for which to retrieve the specified object part. object_name is sysname. This parameter is an optionally-qualified object name. If all parts of the object name are qualified, this name can have four parts: the server name, the database name, the owner name, and the object name. | Is the object part to return. object_piece is of type int, and can have these values:1 = Object name 2 = Schema name 3 = Database name 4 = Server name |

## Section 82.1: PARSENAME

```
Declare @ObjectName nVarChar(1000)
Set @ObjectName = 'HeadOfficeSQL1.Northwind.dbo.Authors'

SELECT
 PARSENAME(@ObjectName, 4) as Server
,PARSENAME(@ObjectName, 3) as DB
,PARSENAME(@ObjectName, 2) as Owner
,PARSENAME(@ObjectName, 1) as Object
```

Returns:

| Server | DB |
| --- | --- |
| HeadofficeSQL1 | Northwind |

| Owner | Object |
| --- | --- |
| dbo | Authors |

# Chapter 83: Installing SQL Server on Windows

## Section 83.1: Introduction

These are the available editions of SQL Server, as told by the Editions Matrix:

- Express: Entry-level free database. Includes core-RDBMS functionality. Limited to 10G of disk size. Ideal for development and testing.
- Standard Edition: Standard Licensed edition. Includes core functionality and Business Intelligence capabilities.
- Enterprise Edition: Full-featured SQL Server edition. Includes advanced security and data warehousing capabilities.
- Developer Edition: Includes all of the features from Enterprise Edition and no limitations, and it is free to download and use for development purposes only.

After downloading/acquiring SQL Server, the installation gets executed with SQLSetup.exe, which is available as a GUI or a command-line program.

Installing via either of these will require you to specify a product key and run some initial configuration that includes enabling features, separate services and setting the initial parameters for each of them. Additional services and features can be enabled at any time by running the SQLSetup.exe program in either the command-line or the GUI version.

# Chapter 84: Analyzing a Query

## Section 84.1: Scan vs Seek

When viewing an execution plan, you may see that SQL Server decided to do a Seek or a Scan.

A Seek occurs when SQL Server knows where it needs to go and only grab specific items. This typically occurs when good filters on put in a query, such as `where name = 'Foo'`.

A Scan is when SQL Server doesn't know exactly where all of the data it needs is, or decided that the Scan would be more efficient than a Seek if enough of the data is selected.

Seeks are typically faster since they are only grabbing a sub-section of the data, whereas Scans are selecting a majority of the data.

# Chapter 85: Query Hints

## Section 85.1: JOIN Hints

When you join two tables, SQL Server query optimizer (QO) can choose different types of joins that will be used in query:

- HASH join
- LOOP join
- MERGE join

QO will explore plans and choose the optimal operator for joining tables. However, if you are sure that you know what would be the optimal join operator, you can specify what kind of JOIN should be used. Inner LOOP join will force QO to choose Nested loop join while joining two tables:

```
SELECT top 100 *
FROM Sales.Orders o
    INNER loop JOIN Sales.OrderLines ol
    ON o.OrderID = ol.OrderID
```

inner merge join will force MERGE join operator:

```
SELECT top 100 *
FROM Sales.Orders o
    INNER MERGE JOIN Sales.OrderLines ol
    ON o.OrderID = ol.OrderID
```

inner hash join will force HASH join operator:

```
SELECT top 100 *
FROM Sales.Orders o
    INNER hash JOIN Sales.OrderLines ol
    ON o.OrderID = ol.OrderID
```

## Section 85.2: GROUP BY Hints

When you use GROUP BY clause, SQL Server query optimizer (QO) can choose different types of grouping operators:

- HASH Aggregate that creates hash-map for grouping entries
- Stream Aggregate that works well with pre-ordered inputs

You can explicitly require that QO picks one or another aggregate operator if you know what would be the optimal. With OPTION (ORDER GROUP), QO will always choose Stream aggregate and add Sort operator in front of Stream aggregate if input is not sorted:

```
SELECT OrderID, AVG(Quantity)
FROM Sales.OrderLines
GROUP BY OrderID
OPTION (ORDER GROUP)
```

With OPTION (HASH GROUP), QO will always choose Hash aggregate :

```
SELECT OrderID, AVG(Quantity)
```

```
FROM Sales.OrderLines
GROUP BY OrderID
OPTION (HASH GROUP)
```

## Section 85.3: FAST rows hint

Specifies that the query is optimized for fast retrieval of the first number_rows. This is a nonnegative integer. After the first number_rows are returned, the query continues execution and produces its full result set.

```
SELECT OrderID, AVG(Quantity)
FROM Sales.OrderLines
GROUP BY OrderID
OPTION (FAST 20)
```

## Section 85.4: UNION hints

When you use UNION operator on two query results, Query optimizer (QO) can use following operators to create a union of two result sets:

- Merge (Union)
- Concat (Union)
- Hash Match (Union)

You can explicitly specify what operator should be used using OPTION() hint:

```
SELECT OrderID, OrderDate, ExpectedDeliveryDate, Comments
FROM Sales.Orders
WHERE OrderDate > DATEADD(DAY, -1, getdate())
UNION
SELECT PurchaseOrderID AS OrderID, OrderDate, ExpectedDeliveryDate, Comments
FROM Purchasing.PurchaseOrders
WHERE OrderDate > DATEADD(DAY, -1, getdate())
OPTION(HASH UNION)
-- or OPTION(CONCAT UNION)
-- or OPTION(MERGE UNION)
```

## Section 85.5: MAXDOP Option

Specifies the max degree of parallelism for the query specifying this option.

```
SELECT OrderID,
    AVG(Quantity)
FROM Sales.OrderLines
GROUP BY OrderID
OPTION (MAXDOP 2);
```

This option overrides the MAXDOP configuration option of sp_configure and Resource Governor. If MAXDOP is set to zero then the server chooses the max degree of parallelism.

## Section 85.6: INDEX Hints

Index hints are used to force a query to use a specific index, instead of allowing SQL Server's Query Optimizer to choose what it deems the best index. In some cases you may gain benefits by specifying the index a query must use. Usually SQL Server's Query Optimizer chooses the best index suited for the query, but due to missing/outdated statistics or specific needs you can force it.

```sql
SELECT *
FROM mytable WITH (INDEX (ix_date))
WHERE field1 > 0
    AND CreationDate > '20170101'
```

# Chapter 86: Query Store

## Section 86.1: Enable query store on database

Query store can be enabled on database by using the following command:

```
ALTER DATABASE tpch SET QUERY_STORE = ON
```

SQL Server/Azure SQL Database will collect information about executed queries and provide information in sys.query_store views:

- sys.query_store_query
- sys.query_store_query_text
- sys.query_store_plan
- sys.query_store_runtime_stats
- sys.query_store_runtime_stats_interval
- sys.database_query_store_options
- sys.query_context_settings

## Section 86.2: Get execution statistics for SQL queries/plans

The following query will return informationa about qeries, their plans and average statistics regarding their duration, CPU time, physical and logical io reads.

```
SELECT Txt.query_text_id, Txt.query_sql_text, Pl.plan_id,
       avg_duration, avg_cpu_time,
       avg_physical_io_reads, avg_logical_io_reads
FROM sys.query_store_plan AS Pl
JOIN sys.query_store_query AS Qry
    ON Pl.query_id = Qry.query_id
JOIN sys.query_store_query_text AS Txt
    ON Qry.query_text_id = Txt.query_text_id
JOIN sys.query_store_runtime_stats Stats
    ON Pl.plan_id = Stats.plan_id
```

## Section 86.3: Remove data from query store

If you want to remove some query or query plan from query store, you can use the following commands:

```
EXEC sp_query_store_remove_query 4;
EXEC sp_query_store_remove_plan 3;
```

Parameters for these stored procedures are query/plan id retrieved from system views.

You can also just remove execution statistics for particular plan without removing the plan from the store:

```
EXEC sp_query_store_reset_exec_stats 3;
```

Parameter provided to this procedure plan id.

## Section 86.4: Forcing plan for query

SQL Query optimizer will choose the baes possible plan that he can find for some query. If you can find some plan

that works optimally for some query, you can force QO to always use that plan using the following stored procedure:

```
EXEC sp_query_store_unforce_plan @query_id, @plan_id
```

From this point, QO will always use plan provided for the query.

If you want to remove this binding, you can use the following stored procedure:

```
EXEC sp_query_store_force_plan @query_id, @plan_id
```

From this point, QO will again try to find the best plan.

# Chapter 87: Querying results by page

## Section 87.1: Row_Number()

```
SELECT ROW_NUMBER() OVER(ORDER BY UserName) AS RowID, UserFirstName, UserLastName
FROM Users
```

From which it will yield a result set with a RowID field which you can use to page between.

```
SELECT *
FROM
    ( SELECT ROW_NUMBER() OVER(ORDER BY UserName) AS RowID, UserFirstName, UserLastName
      FROM Users
    ) AS RowResults
WHERE RowID BETWEEN 5 AND 10
```

# Chapter 88: Schemas

## Section 88.1: Purpose

Schema refers to a specific database tables and how they are related to each other. It provides an organisational blueprint of how the database is constructed. Additional benefits of implementing database schemas is that schemas can be used as a method restricting / granting access to specific tables within a database.

## Section 88.2: Creating a Schema

```
CREATE SCHEMA dvr AUTHORIZATION Owner
    CREATE TABLE sat_Sales (source int, cost int, partid int)
    GRANT SELECT ON SCHEMA :: dvr TO  User1
    DENY SELECT ON SCHEMA :: dvr to User 2
GO
```

## Section 88.3: Alter Schema

```
ALTER SCHEMA dvr
    TRANSFER dbo.tbl_Staging;
GO
```

This would transfer the tbl_Staging table from the dbo schema to the dvr schema

## Section 88.4: Dropping Schemas

```
DROP SCHEMA dvr
```

# Chapter 89: Backup and Restore Database

| Parameter | Details |
|---|---|
| *database* | The name of the database to backup or restore |
| *backup_device* | The device to backup or restore the database from, Like {DISK or TAPE}. Can be separated by commas ( , ) |
| *with_options* | Various options which can be used while performing the operation. Like formatting the disk where the backup is to be placed or restoring the database with replace option. |

## Section 89.1: Basic Backup to disk with no options

The following command backs up the *'Users'* database to *'D:\DB_Backup'* file. Its better to not give an extension.

```
BACKUP DATABASE Users TO DISK = 'D:\DB_Backup'
```

## Section 89.2: Basic Restore from disk with no options

The following command restores the *'Users'* database from *'D:\DB_Backup'* file.

```
RESTORE DATABASE Users FROM DISK = 'D:\DB_Backup'
```

## Section 89.3: RESTORE Database with REPLACE

When you try to restore database from another server you might get the following error:

> Error 3154: The backup set holds a backup of a database other than the existing database.

In that case you should use WITH REPLACE option to replace database with the database from backup:

```
RESTORE DATABASE WWIDW
FROM DISK = 'C:\Backup\WideWorldImportersDW-Full.bak'
WITH REPLACE
```

Even in this case you might get the errors saying that files cannot be located on some path:

> Msg 3156, Level 16, State 3, Line 1 File 'WWI_Primary' cannot be restored to 'D:\Data\WideWorldImportersDW.mdf'. Use WITH MOVE to identify a valid location for the file.

This error happens probably because your files were not placed on the same folder path that exist on new server. In that case you should move individual database files to new location:

```
RESTORE DATABASE WWIDW
FROM DISK = 'C:\Backup\WideWorldImportersDW-Full.bak'
WITH REPLACE,
MOVE 'WWI_Primary' to 'C:\Data\WideWorldImportersDW.mdf',
MOVE 'WWI_UserData' to 'C:\Data\WideWorldImportersDW_UserData.ndf',
MOVE 'WWI_Log' to 'C:\Data\WideWorldImportersDW.ldf',
MOVE 'WWIDW_InMemory_Data_1' to 'C:\Data\WideWorldImportersDW_InMemory_Data_1'
```

With this statement you can replace database with all database files moved to new location.

# Chapter 90: Transaction handling

| Parameter | Details |
| --- | --- |
| transaction_name | for naming your transaction - useful with the parameter [*with mark*] which will allow a meaningfull logging -- case-sensitive (!) |

with mark ['description'] can be added to [*transaction_name*] and will store a mark in the log

## Section 90.1: basic transaction skeleton with error handling

```
BEGIN TRY -- start error handling
    BEGIN TRANSACTION; -- from here on transactions (modifictions) are not final
        -- start your statement(s)
        select 42/0 as ANSWER  -- simple SQL Query with an error
        -- end your statement(s)
    COMMIT TRANSACTION; -- finalize all transactions (modifications)
END TRY   -- end error handling -- jump to end
BEGIN CATCH -- execute this IF an error occurred
        ROLLBACK TRANSACTION; -- undo any transactions (modifications)
-- put together some information as a query
    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;  -- final line of error handling
GO -- execute previous code
```

# Chapter 91: Natively compiled modules (Hekaton)

## Section 91.1: Natively compiled stored procedure

In a procedure with native compilation, T-SQL code is compiled to dll and executed as native C code. To create a Native Compiled stored Procedure you need to:

- Use standard CREATE PROCEDURE syntax
- Set NATIVE_COMPILATION option in stored procedure definition
- Use SCHEMABINDING option in stored procedure definition
- Define EXECUTE AS OWNER option in stored procedure definition

Instead of standard BEGIN END block, you need to use BEGIN ATOMIC block:

```
BEGIN ATOMIC
   WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT, LANGUAGE='us_english')
   -- T-Sql code goes here
END
```

Example:

```
CREATE PROCEDURE usp_LoadMemOptTable (@maxRows INT, @FullName NVARCHAR(200))
WITH
    NATIVE_COMPILATION,
    SCHEMABINDING,
    EXECUTE AS OWNER
AS
BEGIN ATOMIC
WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT, LANGUAGE='us_english')
    DECLARE @i INT = 1
    WHILE @i <= @maxRows
    BEGIN
        INSERT INTO dbo.MemOptTable3 VALUES(@i, @FullName, GETDATE())
        SET @i = @i+1
    END
END
GO
```

## Section 91.2: Natively compiled scalar function

Code in natively compiled function will be transformed into C code and compiled as dll. To create a Native Compiled scalar function you need to:

- Use standard CREATE FUNCTION syntax
- Set NATIVE_COMPILATION option in function definition
- Use SCHEMABINDING option in function definition

Instead of standard BEGIN END block, you need to use BEGIN ATOMIC block:

```
BEGIN ATOMIC
   WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT, LANGUAGE='us_english')
   -- T-Sql code goes here
END
```

Example:

```
CREATE FUNCTION [dbo].[udfMultiply]( @v1 int, @v2 int )
RETURNS bigint
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'English')

    DECLARE @ReturnValue bigint;
    SET @ReturnValue = @v1 * @v2;

    RETURN (@ReturnValue);
END

-- usage sample:
SELECT dbo.udfMultiply(10, 12)
```

## Section 91.3: Native inline table value function

Native compiled table value function returns table as result. Code in natively compiled function will be transformed into C code and compiled as dll. Only inline table valued functions are supported in version 2016. To create a native table value function you need to:

- Use standard CREATE FUNCTION syntax
- Set NATIVE_COMPILATION option in function definition
- Use SCHEMABINDING option in function definition

Instead of standard BEGIN END block, you need to use BEGIN ATOMIC block:

```
BEGIN ATOMIC
   WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT, LANGUAGE='us_english')
   -- T-Sql code goes here
END
```

Example:

```
CREATE FUNCTION [dbo].[udft_NativeGetBusinessDoc]
(
   @RunDate VARCHAR(25)
)
RETURNS TABLE
WITH SCHEMABINDING,
     NATIVE_COMPILATION
AS
     RETURN
(
    SELECT BusinessDocNo,
           ProductCode,
           UnitID,
           ReasonID,
           PriceID,
           RunDate,
           ReturnPercent,
           Qty,
           RewardAmount,
           ModifyDate,
           UserID
    FROM dbo.[BusinessDocDetail_11]
    WHERE RunDate >= @RunDate
```

```
);
```

# Chapter 92: Spatial Data

There are 2 spatial data types

**Geometry** X/Y coordinate system for a flat surface

**Geography** Latitude/Longitude coordinate system for a curved surface (the earth). There are multiple projections of curved surfaces so each geography spatial must let SQL Server know which projection to use. The usual Spatial Reference ID (SRID) is 4326, which is measuring distances in Kilometers. This is the default SRID used in most web maps

## Section 92.1: POINT

Creates a single Point. This will be a geometry or geography point depending on the class used.

| Parameter | Detail |
|---|---|
| Lat or X | Is a float expression representing the x-coordinate of the Point being generated |
| Long or Y | Is a float expression representing the y-coordinate of the Point being generated |
| String | Well Known Text (WKB) of a geometry/geography shape |
| Binary | Well Known Binary (WKB) of a geometry/geography shape |
| SRID | Is an int expression representing the spatial reference ID (SRID) of the geometry/geography instance you wish to return |

```
--Explicit constructor
DECLARE @gm1 GEOMETRY = GEOMETRY::Point(10,5,0)

DECLARE @gg1 GEOGRAPHY = GEOGRAPHY::Point(51.511601,-0.096600,4326)

--Implicit constructor (using WKT - Well Known Text)
DECLARE @gm1 GEOMETRY = GEOMETRY::STGeomFromText('POINT(5 10)', 0)

DECLARE @gg1 GEOGRAPHY= GEOGRAPHY::STGeomFromText('POINT(-0.096600 51.511601)', 4326)

--Implicit constructor (using WKB - Well Known Binary)
DECLARE @gm1 GEOMETRY = GEOMETRY::STGeomFromWKB(0x0101000000000000000000144000000000000002440, 0)

DECLARE @gg1 GEOGRAPHY= GEOGRAPHY::STGeomFromWKB(0x01010000005F29CB10C7BAB8BFEACC3D247CC14940,
4326)
```

# Chapter 93: Dynamic SQL

## Section 93.1: Execute SQL statement provided as string

In some cases, you would need to execute SQL query placed in string. EXEC, EXECUTE, or system procedure sp_executesql can execute any SQL query provided as string:

```
sp_executesql N'SELECT * FROM sys.objects'
-- or
sp_executesql @stmt = N'SELECT * FROM sys.objects'
-- or
EXEC sp_executesql N'SELECT * FROM sys.objects'
-- or
EXEC('SELECT * FROM sys.columns')
-- or
EXECUTE('SELECT * FROM sys.tables')
```

This procedure will return the same result-set as SQL query provided as statement text. sp_executesql can execute SQL query provided as string literal, variable/parameter, or even expression:

```
declare @table nvarchar(40) = N'product items'
EXEC(N'SELECT * FROM ' + @table)
declare @sql nvarchar(40) = N'SELECT * FROM ' + QUOTENAME(@table);
EXEC sp_executesql @sql
```

You need QUOTENAME function to escape special characters in @table variable. Without this function you would get syntax error if @table variable contains something like spaces, brackets, or any other special character.

## Section 93.2: Dynamic SQL executed as different user

You can execute SQL query as different user using AS USER = 'name of database user'

```
EXEC(N'SELECT * FROM product') AS USER = 'dbo'
```

SQL query will be executed under dbo database user. All permission checks applicable to dbo user will be checked on SQL query.

## Section 93.3: SQL Injection with dynamic SQL

Dynamic queries are

```
SET @sql = N'SELECT COUNT(*) FROM AppUsers WHERE Username = ''' + @user + ''' AND Password = ''' +
@pass + ''''
EXEC(@sql)
```

If value of user variable is ***myusername'' OR 1=1 --*** the following query will be executed:

```
SELECT COUNT(*)
FROM AppUsers
WHERE Username = 'myusername' OR 1=1 --' AND Password = ''
```

Comment at the end of value of variable @username will comment-out trailing part of the query and condition 1=1 will be evaluated. Application that checks it there at least one user returned by this query will return count greater than 0 and login will succeed.

Using this approach attacker can login into application even if he don't know valid username and password.

## Section 93.4: Dynamic SQL with parameters

In order to avoid injection and escaping problems, dynamic SQL queries should be executed with parameters, e.g.:

```
SET @sql = N'SELECT COUNT(*) FROM AppUsers WHERE Username = @user AND Password = @pass
EXEC sp_executesql @sql, '@user nvarchar(50), @pass nvarchar(50)', @username, @password
```

Second parameter is a list of parameters used in query with their types, after this list are provided variables that will be used as parameter values.

sp_executesql will escape special characters and execute sql query.

# Chapter 94: Dynamic data masking

## Section 94.1: Adding default mask on the column

If you add default mask on the column, instead of actual value in SELECT statement will be shown mask:

```
ALTER TABLE  Company
ALTER COLUMN Postcode ADD MASKED WITH (FUNCTION = 'default()')
```

## Section 94.2: Mask email address using Dynamic data masking

If you have email column you can mask it with email() mask:

```
ALTER TABLE  Company
ALTER COLUMN Email ADD MASKED WITH (FUNCTION = 'email()')
```

When user tries to select emails from Company table, he will get something like the following values:

mXXX@XXXX.com

zXXX@XXXX.com

rXXX@XXXX.com

## Section 94.3: Add partial mask on column

You can add partial mask on the column that will show few characters from te beginning and the end of the string and show mask instead of the characters in the middle:

```
ALTER TABLE  Company
ALTER COLUMN Phone ADD MASKED WITH (FUNCTION = 'partial(5,"XXXXXXX",2)')
```

In the parameters of the partial function you can specify how many values from the beginning will be shown, how many values from the end will be shown, and what woudl be the pattern that is shown in the middle.

When user tries to select emails from Company table, he will get something like the following values:

(381)XXXXXXX39

(360)XXXXXXX01

(415)XXXXXXX05

## Section 94.4: Showing random value from the range using random() mask

Random mask will show a rundom number from the specified range instead of the actual value:

```
ALTER TABLE  Product
ALTER COLUMN Price ADD MASKED WITH (FUNCTION = 'random(100,200)')
```

Note that is some cases displayed value might match actual value in column (if randomly selected number matches

value in the cell).

## Section 94.5: Controlling who can see unmasked data

You can grant in-privileged users right to see unmasked values using the following statement:

```
GRANT UNMASK TO MyUser
```

If some user already has unmask permission, you can revoke this permission:

```
REVOKE UNMASK TO MyUser
```

# Chapter 95: Export data in txt file by using SQLCMD

## Section 95.1: By using SQLCMD on Command Prompt

**Command Structure is**

sqlcmd -S yourservername\instancename -d database_name -o outputfilename_withpath -Q "your select query"

Switches are as follows

-S for servername and instance name

-d for source database

-o for target outputfile (it will create output file)

-Q for query to fetch data

# Chapter 96: Common Language Runtime Integration

## Section 96.1: Enable CLR on database

CLR procedures are not enabled by default. You need to run the following queries to enable CLR:

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

In addition, if some CLR module need external access, you should set TRUSTWORTHY property to ON in your database:

```
ALTER DATABASE MyDbWithClr SET TRUSTWORTHY ON
```

## Section 96.2: Adding .dll that contains Sql CLR modules

Procedures, functions, triggers, and types written in .Net languages are stored in .dll files. Once you create .dll file containing CLR procedures you should import it into SQL Server:

```
CREATE ASSEMBLY MyLibrary
FROM 'C:\lib\MyStoredProcedures.dll'
    WITH PERMISSION_SET = EXTERNAL_ACCESS
```

PERMISSION_SET is Safe by default meaning that code in .dll don't need permission to access external resources (e.g. files, web sites, other servers), and that it will not use native code that can access memory.

PERMISSION_SET = EXTERNAL_ACCESS is used to mark assemblies that contain code that will access external resources.

you can find information about current CLR assembly files in sys.assemblies view:

```
SELECT *
FROM sys.assemblies asms
WHERE is_user_defined = 1
```

## Section 96.3: Create CLR Function in SQL Server

If you have created .Net function, compiled it into .dll, and imported it into SQL server as an assembly, you can create user-defined function that references function in that assembly:

```
CREATE FUNCTION dbo.TextCompress(@input nvarchar(max))
RETURNS varbinary(max)
AS EXTERNAL NAME MyLibrary.[Name.Space.ClassName].TextCompress
```

You need to specify name of the function and signature with input parameters and return values that match .Net function. In AS EXTERNAL NAME clause you need to specify assembly name, namespace/class name where this

function is placed and name of the method in the class that contains the code that will be exposed as function.

You can find information about the CLR functions using the following query:

```
SELECT * FROM dbo.sysobjects WHERE TYPE ='FS'
```

## Section 96.4: Create CLR User-defined type in SQL Server

If you have create .Net class that represents some user-defined type, compiled it into .dll, and imported it into SQL server as an assembly, you can create user-defined function that references this class:

```
CREATE TYPE dbo.Point
EXTERNAL NAME MyLibrary.[Name.Space.Point]
```

You need to specify name of the type that will be used in T-SQL queries. In EXTERNAL NAME clause you need to specify assembly name, namespace, and class name.

## Section 96.5: Create CLR procedure in SQL Server

If you have created .Net method in some class, compiled it into .dll, and imported it into SQL server as an assembly, you can create user-defined stored procedure that references method in that assembly:

```
CREATE PROCEDURE dbo.DoSomethng(@input nvarchar(max))
AS EXTERNAL NAME MyLibrary.[Name.Space.ClassName].DoSomething
```

You need to specify name of the procedure and signature with input parameters that match .Net method. In AS EXTERNAL NAME clause you need to specify assembly name, namespace/class name where this procedure is placed and name of the method in the class that contains the code that will be exposed as procedure.

# Chapter 97: Delimiting special characters and reserved words

## Section 97.1: Basic Method

The basic method to escape reserved words for SQL Server is the use of the square brackets ([ and ]). For example, *Description* and *Name* are reserved words; however, if there is an object using both as names, the syntax used is:

```
SELECT [Description]
FROM   dbo.TableName
WHERE  [Name] = 'foo'
```

The only special character for SQL Server is the single quote `'` and it is escaped by doubling its usage. For example, to find the name *O'Shea* in the same table, the following syntax would be used:

```
SELECT [Description]
FROM   dbo.TableName
WHERE  [Name] = 'O''Shea'
```

# Chapter 98: DBCC

## Section 98.1: DBCC statement

DBCC statements act as Database Console Commands for SQL Server. To get the syntax information for the specified DBCC command use DBCC HELP (...) statement.

The following example returns all DBCC statements for which Help is available:

```
DBCC HELP ('?');
```

The following example returns options for DBCC CHECKDB statement:

```
DBCC HELP ('CHECKDB');
```

## Section 98.2: DBCC maintenance commands

DBCC commands enable user to maintain space in database, clean caches, shrink databases and tables.

Examples are:

```
DBCC DROPCLEANBUFFERS
```

Removes all clean buffers from the buffer pool, and columnstore objects from the columnstore object pool.

```
DBCC FREEPROCCACHE
-- or
DBCC FREEPROCCACHE (0x060006001ECA270EC0215D05000000000000000000000000);
```

Removes all SQL query in plan cache. Every new plan will be recompiled: You can specify plan handle, query handle to clean plans for the specific query plan or SQL statement.

```
DBCC FREESYSTEMCACHE ('ALL', myresourcepool);
-- or
DBCC FREESYSTEMCACHE;
```

Cleans all cached entries created by system. It can clean entries o=in all or some specified resource pool (**myresourcepool** in the example above)

```
DBCC FLUSHAUTHCACHE
```

Empties the database authentication cache containing information about logins and firewall rules.

```
DBCC SHRINKDATABASE (MyDB [, 10]);
```

Shrinks database MyDB to 10%. Second parameter is optional. You can use database id instead of name.

```
DBCC SHRINKFILE (DataFile1, 7);
```

Shrinks data file named DataFile1 in the current database. Target size is 7 MB (tis parameter is optional).

```
DBCC CLEANTABLE (AdventureWorks2012,'Production.Document', 0)
```

Reclaims a space from specified table

## Section 98.3: DBCC validation statements

DBCC commands enable user to validate state of database.

```
ALTER TABLE Table1 WITH NOCHECK ADD CONSTRAINT chkTab1 CHECK (Col1 > 100);
GO
DBCC CHECKCONSTRAINTS(Table1);
--OR
DBCC CHECKCONSTRAINTS ('Table1.chkTable1');
```

Check constraint is added with nocheck options, so it will not be checked on existing data. DBCC will trigger constraint check.

Following DBCC commands check integrity of database, table or catalog:

```
DBCC CHECKTABLE tablename1 | tableid
DBCC CHECKDB databasename1 | dbid
DBCC CHECKFILEGROUP filegroup_name | filegroup_id | 0
DBCC CHECKCATALOG databasename1 | database_id1 | 0
```

## Section 98.4: DBCC informational statements

DBCC commands can show information about database objects.

```
DBCC PROCCACHE
```

Displays information in a table format about the procedure cache.

```
DBCC OUTPUTBUFFER ( session_id [ , request_id ])
```

Returns the current output buffer in hexadecimal and ASCII format for the specified session_id (and optional request_id).

```
DBCC INPUTBUFFER ( session_id [ , request_id ])
```

Displays the last statement sent from a client to an instance of Microsoft SQL Server.

```
DBCC SHOW_STATISTICS ( table_or_indexed_view_name , column_statistic_or_index_name)
```

## Section 98.5: DBCC Trace commands

Trace flags in SQL Server are used to modify behavior of SQL server, turn on/off some features. DBCC commands can control trace flags:

The following example switches on trace flag 3205 globally and 3206 for the current session:

```
DBCC TRACEON (3205, -1);
DBCC TRACEON (3206);
```

The following example switches off trace flag 3205 globally and 3206 for the current session:

```
DBCC TRACEON (3205, -1);
```

```
DBCC TRACEON (3206);
```

The following example displays the status of trace flags 2528 and 3205:

```
DBCC TRACESTATUS (2528, 3205);
```

# Chapter 99: BULK Import

## Section 99.1: BULK INSERT

BULK INSERT command can be used to import file into SQL Server:

```
BULK INSERT People
FROM 'f:\orders\people.csv'
```

BULK INSERT command will map columns in files with columns in target table.

## Section 99.2: BULK INSERT with options

You can customize parsing rules using different options in WITH clause:

```
BULK INSERT People
FROM 'f:\orders\people.csv'
WITH  (  CODEPAGE = '65001',
         FIELDTERMINATOR =',',
         ROWTERMINATOR ='\n'
      );
```

In this example, CODEPAGE specifies that a source file in UTF-8 file, and TERMINATORS are coma and new line.

## Section 99.3: Reading entire content of file using OPENROWSET(BULK)

You can read content of file using OPENROWSET(BULK) function and store content in some table:

```
INSERT INTO myTable(content)
   SELECT BulkColumn
          FROM OPENROWSET(BULK N'C:\Text1.txt', SINGLE_BLOB) AS Document;
```

SINGLE_BLOB option will read entire content from a file as single cell.

## Section 99.4: Read file using OPENROWSET(BULK) and format file

Yu can define format of the file that will be imported using FORMATFILE option:

```
INSERT INTO mytable
SELECT a.*
FROM OPENROWSET(BULK 'c:\test\values.txt',
   FORMATFILE = 'c:\test\values.fmt') AS a;
```

The format file, format_file.fmt, describes the columns in values.txt:

```
9.0
2
1  SQLCHAR  0  10 "\t"        1  ID               SQL_Latin1_General_Cp437_BIN
2  SQLCHAR  0  40 "\r\n"      2  Description      SQL_Latin1_General_Cp437_BIN
```

## Section 99.5: Read json file using OPENROWSET(BULK)

You can use OPENROWSET to read content of file and pass it to some other function that will parse results.

The following example shows hot to read entire content of JSON file using OPENROWSET(BULK) and then provide BulkColumn to OPENJSON function that will parse JSON and return columns:

```
SELECT book.*
 FROM OPENROWSET (BULK 'C:\JSON\Books\books.json', SINGLE_CLOB) AS j
 CROSS APPLY OPENJSON(BulkColumn)
       WITH( id nvarchar(100), name nvarchar(100), price FLOAT,
             pages INT, author nvarchar(100)) AS book
```

# Chapter 100: Service broker

## Section 100.1: Basics

Service broker is technology based on asyncronous communication between two(or more) entities. Service broker consists of: message types, contracts, queues, services, routes, and at least instance endpoints

More: https://msdn.microsoft.com/en-us/library/bb522893.aspx

## Section 100.2: Enable service broker on database

```
ALTER DATABASE [MyDatabase] SET ENABLE_BROKER WITH ROLLBACK IMMEDIATE;
```

## Section 100.3: Create basic service broker construction on database (single database communication)

```
USE [MyDatabase]

CREATE MESSAGE TYPE [//initiator] VALIDATION = WELL_FORMED_XML;
    GO

CREATE CONTRACT [//call/contract]
(
    [//initiator] SENT BY INITIATOR
)
GO

CREATE QUEUE  InitiatorQueue;
GO

CREATE QUEUE  TargetQueue;
GO

CREATE SERVICE InitiatorService
      ON QUEUE InitiatorQueue
(
     [//call/contract]

)

CREATE SERVICE TargetService
ON QUEUE TargetQueue
(
     [//call/contract]

)

GRANT SEND ON SERVICE::[InitiatorService] TO PUBLIC
GO

GRANT SEND ON SERVICE::[TargetService] TO PUBLIC
GO
```

We don't need route for one database communication.

## Section 100.4: How to send basic communication through service broker

For this demonstration we will use service broker construction created in another part of this documentation. Mentioned part is named **3. Create basic service broker construction on database (single database communication)**.

```
USE [MyDatabase]

DECLARE @ch uniqueidentifier = NEWID()
DECLARE @msg XML

BEGIN DIALOG CONVERSATION @ch
    FROM SERVICE [InitiatorService]
    TO SERVICE 'TargetService'
    ON CONTRACT [//call/contract]
    WITH ENCRYPTION = OFF; -- more possible options

        SET @msg = (
                    SELECT 'HelloThere' "elementNum1"
                    FOR XML PATH(''), ROOT('ExampleRoot'), ELEMENTS XSINIL, TYPE
                    );

SEND ON CONVERSATION @ch MESSAGE TYPE [//initiator] (@msg);
END CONVERSATION @ch;
```

After this conversation will be your msg in TargetQueue

## Section 100.5: How to receive conversation from TargetQueue automatically

For this demonstration we will use service broker construction created in another part of this documentation. Mentioned part is called **3. Create basic service broker construction on database (single database communication)**.

First we need to create a procedure that is able to read and process data from the Queue

```
USE [MyDatabase]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

CREATE PROCEDURE [dbo].[p_RecieveMessageFromTargetQueue]

    AS
    BEGIN

    declare
    @message_body xml,
    @message_type_name nvarchar(256),
    @conversation_handle uniqueidentifier,
    @messagetypename nvarchar(256);
```

```
   WHILE 1=1
   BEGIN

    BEGIN TRANSACTION
        WAITFOR(
        RECEIVE TOP(1)
        @message_body = CAST(message_body as xml),
        @message_type_name = message_type_name,
        @conversation_handle = conversation_handle,
        @messagetypename = message_type_name
        FROM DwhInsertSmsQueue
        ), TIMEOUT 1000;

         IF (@@ROWCOUNT = 0)
            BEGIN
                ROLLBACK TRANSACTION
                BREAK
            END

         IF (@messagetypename = '//initiator')
             BEGIN

                IF OBJECT_ID('MyDatabase..MyExampleTableHelloThere') IS NOT NULL
                    DROP TABLE dbo.MyExampleTableHelloThere

                SELECT @message_body.value('(/ExampleRoot/"elementNum1")[1]', 'VARCHAR(50)') AS
MyExampleMessage
                INTO dbo.MyExampleTableHelloThere

             END



         IF (@messagetypename = 'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog')
            BEGIN
                END CONVERSATION @conversation_handle;
            END

    COMMIT TRANSACTION
   END


END
```

Second step: Allow your TargetQueue to automatically run your procedure:

```
USE [MyDatabase]

ALTER QUEUE [dbo].[TargetQueue] WITH STATUS = ON , RETENTION = OFF ,
ACTIVATION
 (  STATUS = ON , --activation status
    PROCEDURE_NAME = dbo.p_RecieveMessageFromTargetQueue , --procedure name
    MAX_QUEUE_READERS = 1 , --number of readers
    EXECUTE AS SELF  )
```

# Chapter 101: Permissions and Security

## Section 101.1: Assign Object Permissions to a user

In Production its good practice to secure your data and only allow operations on it to be undertaken via Stored Procedures. This means your application can't directly run CRUD operations on your data and potentially cause problems. Assigning permissions is a time-consuming, fiddly and generally onerous task. For this reason its often easier to harness some of the (considerable) power contained in the INFORMATION_SCHEMA er schema which is contained in every SQL Server database.

Instead individually assigning permissions to a user on a piece-meal basis, just run the script below, copy the output and then run it in a Query window.

```
SELECT 'GRANT EXEC ON core.' + r.ROUTINE_NAME + ' TO ' + <MyDatabaseUsername>
FROM INFORMATION_SCHEMA.ROUTINES r
WHERE r.ROUTINE_CATALOG = '<MyDataBaseName>'
```

# Chapter 102: Database permissions

## Section 102.1: Changing permissions

```
GRANT SELECT ON [dbo].[someTable] TO [aUser];

REVOKE SELECT ON [dbo].[someTable] TO [aUser];
--REVOKE SELECT [dbo].[someTable] FROM [aUser]; is equivalent

DENY SELECT ON [dbo].[someTable] TO [aUser];
```

## Section 102.2: CREATE USER

```
--implicitly map this user to a login of the same name as the user
CREATE USER [aUser];

--explicitly mapping what login the user should be associated with
CREATE USER [aUser] FOR LOGIN [aUser];
```

## Section 102.3: CREATE ROLE

```
CREATE ROLE [myRole];
```

## Section 102.4: Changing role membership

```
-- SQL 2005+
exec sp_addrolemember @rolename = 'myRole', @membername = 'aUser';
exec sp_droprolemember @rolename = 'myRole', @membername = 'aUser';

-- SQL 2008+
ALTER ROLE [myRole] ADD MEMBER [aUser];
ALTER ROLE [myRole] DROP MEMBER [aUser];
```

Note: role members can be any database-level principal. That is, you can add a role as a member in another role. Also, adding/dropping role members is idempotent. That is, attempting to add/drop will result in their presence/absence (respectively) in the role regardless of the current state of their role membership.

# Chapter 103: Row-level security

## Section 103.1: RLS filter predicate

Sql Server 2016+ and Azure Sql database enables you to automatically filter rows that are returned in select statement using some predicate. This feature is called **Row-level security**.

First, you need a table-valued function that contains some predicate that describes what it the condition that will allow users to read data from some table:

```
DROP FUNCTION IF EXISTS dbo.pUserCanAccessCompany
GO
CREATE FUNCTION

dbo.pUserCanAccessCompany(@CompanyID int)

    RETURNS TABLE
    WITH SCHEMABINDING
AS RETURN (
    SELECT 1 as canAccess WHERE

    CAST(SESSION_CONTEXT(N'CompanyID') as int) = @CompanyID

)
```

In this example, the predicate says that only users that have a value in SESSION_CONTEXT that is matching input argument can access the company. You can put any other condition e.g. that checks database role or database_id of the current user, etc.

> Most of the code above is a template that you will copy-paste. The only thing that will change here is the name and arguments of predicate and condition in WHERE clause. Now you create security policy that will apply this predicate on some table.

Now you can create security policy that will apply predicate on some table:

```
CREATE SECURITY POLICY dbo.CompanyAccessPolicy
    ADD FILTER PREDICATE dbo.pUserCanAccessCompany(CompanyID) ON dbo.Company
    WITH (State=ON)
```

This security policy assigns predicate to company table. Whenever someone tries to read data from Company table , security policy will apply predicate on each row, pass CompanyID column as a parameter of the predicate, and predicate will evaluate should this row be returned in the result of SELECT query.

## Section 103.2: Altering RLS security policy

Security policy is a group of predicates associated to tables that can be managed together. You can add, or remove predicates or turn on/off entire policy.

You can add more predicates on tables in the existing security policy.

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy
    ADD FILTER PREDICATE dbo.pUserCanAccessCompany(CompanyID) ON dbo.Company
```

You can drop some predicates from security policy:

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy
    DROP FILTER PREDICATE ON dbo.Company
```

You can disable security policy

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy WITH ( STATE = OFF );
```

You can enable security policy that was disabled:

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy WITH ( STATE = ON );
```

## Section 103.3: Preventing updated using RLS block predicate

Row-level security enables you to define some predicates that will control who could update rows in the table. First you need to define some table-value function that represents predicate that wll control access policy.

CREATE FUNCTION

dbo.pUserCanAccessProduct(@CompanyID int)

```
RETURNS TABLE
WITH SCHEMABINDING
```

AS RETURN ( SELECT 1 as canAccess WHERE

CAST(SESSION_CONTEXT(N'CompanyID') as int) = @CompanyID

) In this example, the predicate says that only users that have a value in SESSION_CONTEXT that is matching input argument can access the company. You can put any other condition e.g. that checks database role or database_id of the current user, etc.

> Most of the code above is a template that you will copy-paste. The only thing that will change here is the name and arguments of predicate and condition in WHERE clause. Now you create security policy that will apply this predicate on some table.

Now we can create security policy with the predicate that will block updates on product table if CompanyID column in table do not satisfies predicate.

CREATE SECURITY POLICY dbo.ProductAccessPolicy ADD BLOCK PREDICATE
dbo.pUserCanAccessProduct(CompanyID) ON dbo.Product

This predicate will be applied on all operations. If you want to apply predicate on some operation you can write something like:

CREATE SECURITY POLICY dbo.ProductAccessPolicy ADD BLOCK PREDICATE
dbo.pUserCanAccessProduct(CompanyID) ON dbo.Product AFTER INSERT

Possible options that you can add after block predicate definition are:

[ { AFTER { INSERT | UPDATE } }
| { BEFORE { UPDATE | DELETE } } ]

# Chapter 104: Encryption

| Optional Parameters | Details |
| --- | --- |
| `WITH PRIVATE KEY` | For CREATE CERTIFICATE, a private key can be specified: `(FILE='D:\Temp\CertTest\private.pvk', DECRYPTION BY PASSWORD = 'password');` |

## Section 104.1: Encryption by certificate

```
CREATE CERTIFICATE My_New_Cert
FROM FILE = 'D:\Temp\CertTest\certificateDER.cer'
GO
```

Create the certificate

```
SELECT EncryptByCert(Cert_ID('My_New_Cert'),
'This text will get encrypted') encryption_test
```

Usually, you would encrypt with a symmetric key, that key would get encrypted by the asymmetric key (public key) from your certificate.

Also, note that encryption is limited to certain lengths depending on key length and returns NULL otherwise. Microsoft writes: "The limits are: a 512 bit RSA key can encrypt up to 53 bytes, a 1024 bit key can encrypt up to 117 bytes, and a 2048 bit key can encrypt up to 245 bytes."

EncryptByAsymKey has the same limits. For UNICODE this would be divided by 2 (16 bits per character), so 58 characters for a 1024 bit key.

## Section 104.2: Encryption of database

```
USE TDE
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE My_New_Cert
GO

ALTER DATABASE TDE
SET ENCRYPTION ON
GO
```

This uses 'Transparent Data Encryption' (TDE)

## Section 104.3: Encryption by symmetric key

```
-- Create the key and protect it with the cert
CREATE SYMMETRIC KEY My_Sym_Key
WITH ALGORITHM = AES_256
ENCRYPTION BY CERTIFICATE My_New_Cert;
GO

-- open the key
OPEN SYMMETRIC KEY My_Sym_Key
DECRYPTION BY CERTIFICATE My_New_Cert;

-- Encrypt
SELECT EncryptByKey(Key_GUID('SSN_Key_01'), 'This text will get encrypted');
```

## Section 104.4: Encryption by passphrase

```
SELECT EncryptByPassphrase('MyPassPhrase', 'This text will get encrypted')
```

This will also encrypt but then by passphrase instead of asymmetric(certificate) key or by an explicit symmetric key.

# Chapter 105: PHANTOM read

In database systems, isolation determines how transaction integrity is visible to other users and systems, so it defines how/when the changes made by one operation become visible to other. The phantom read may occurs when you getting data not yet commited to database.

## Section 105.1: Isolation level READ UNCOMMITTED

Create a sample table on a sample database

```
CREATE TABLE [dbo].[Table_1](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [title] [varchar](50) NULL,
 CONSTRAINT [PK_Table_1] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Now open a First query editor (on the database) insert the code below, and execute (**do not touch the --rollback**) in this case you insert a row on DB but do **not** commit changes.

```
begin tran

INSERT INTO Table_1 values('Title 1')

SELECT * FROM [Test].[dbo].[Table_1]

--rollback
```

Now open a Second Query Editor (on the database), insert the code below and execute

```
begin tran

set transaction isolation level READ UNCOMMITTED

SELECT * FROM [Test].[dbo].[Table_1]
```

You may notice that on second editor you can see the newly created row (but not committed) from first transaction. On first editor execute the rollback (select the rollback word and execute).

```
-- Rollback the first transaction
rollback
```

Execute the query on second editor and you see that the record disappear (phantom read), this occurs because you tell, to the 2nd transaction to get all rows, also the uncommitteds.

This occurs when you change the isolation level with

```
set transaction isolation level READ UNCOMMITTED
```

# Chapter 106: Filestream

FILESTREAM integrates the SQL Server Database Engine with an NTFS file system by storing varbinary(max) binary large object (BLOB) data as files on the file system. Transact-SQL statements can insert, update, query, search, and back up FILESTREAM data. Win32 file system interfaces provide streaming access to the data.

## Section 106.1: Example

Source : MSDN https://technet.microsoft.com/en-us/library/bb933993(v=sql.105).aspx

# Chapter 107: bcp (bulk copy program) Utility

The bulk copy program utility (bcp) bulk copies data between an instance of Microsoft SQL Server and a data file in a user-specified format. The bcp utility can be used to import large numbers of new rows into SQL Server tables or to export data out of tables into data files.

## Section 107.1: Example to Import Data without a Format File(using Native Format )

```
REM Truncate table (for testing)
SQLCMD -Q "TRUNCATE TABLE TestDatabase.dbo.myNative;"

REM Import data
bcp TestDatabase.dbo.myNative IN D:\BCP\myNative.bcp -T -n

REM Review results
SQLCMD -Q "SELECT * FROM TestDatabase.dbo.myNative;"
```

# Chapter 108: SQL Server Evolution through different versions (2000 - 2016)

I am using SQL Server since 2004. I started with 2000 and now I am going to use SQL Server 2016. I created tables, views, functions, triggers, stored procedures and wrote many SQL queries but I did not use many new features from subsequent versions. I googled it but unfortunately, I did not find all the features in one place. So I gathered and validated these information from different sources and put here. I am just adding the high level information for all the versions starting from 2000 to 20

## Section 108.1: SQL Server Version 2000 - 2016

**The following features added in SQL Server 2000 from its previous version:**

1. New data types were added (BIGINT, SQL_VARIANT, TABLE)
2. Instead of and for Triggers were introduced as advancement to the DDL.
3. Cascading referential integrity.
4. XML support
5. User defined functions and partition views.
6. Indexed Views (Allowing index on views with computed columns).

**The following features added in version 2005 from its previous version:**

1. Enhancement in TOP clause with “WITH TIES” option.
2. Data Manipulation Commands (DML) and OUTPUT clause to get INSERTED and DELETED values
3. The PIVOT and UNPIVOT operators.
4. Exception Handling with TRY/CATCH block
5. Ranking functions
6. Common Table Expressions (CTE)
7. Common Language Runtime (Integration of .NET languages to build objects like stored procedures, triggers, functions etc.)
8. Service Broker (Handling message between a sender and receiver in a loosely coupled manner)
9. Data Encryption (Native capabilities to support encryption of data stored in user defined databases)
10. SMTP mail
11. HTTP endpoints (Creation of endpoints using simple T-SQL statement exposing an object to be accessed over the internet)
12. Multiple Active Result Sets (MARS).This allows a persistent database connection from a single client to have more than one active request per connection.
13. SQL Server Integration Services (Will be used as a primary ETL (Extraction, Transformation and Loading) Tool
14. Enhancements in Analysis Services and Reporting Services.
15. Table and index partitioning. Allows partitioning of tables and indexes based on partition boundaries as specified by a PARTITION FUNCTION with individual partitions mapped to file groups via a PARTITION SCHEME.

**The following features added in version 2008 from its previous version:**

1. Enhancement in existing DATE and TIME Data Types
2. New functions like – SYSUTCDATETIME() and SYSDATETIMEOFFSET()
3. Spare Columns – To save a significant amount of disk space.
4. Large User Defined Types (up to 2 GB in size)
5. Introduced a new feature to pass a table datatype into stored procedures and functions
6. New MERGE command for INSERT, UPDATE and DELETE operations

7. New HierarchyID datatype
8. Spatial datatypes - To represent the physical location and shape of any geometric object.
9. Faster queries and reporting with GROUPING SETS - An extension to the GROUP BY clause.
10. Enhancement to FILESTREAM storage option

**The following features added in version 2008 R2 from its previous version:**

1. PowerPivot – For processing large data sets.
2. Report Builder 3.0
3. Cloud ready
4. StreamInsight
5. Master Data Services
6. SharePoint Integration
7. DACPAC (Data-tier Application Component Packages)
8. Enhancement in other features of SQL Server 2008

**The following features added in version 2012 from its previous version:**

1. Column store indexes - reduces I/O and memory utilization on large queries.
2. Pagination - pagination can be done by using “OFFSET” and “FETCH’ commands.
3. Contained database – Great feature for periodic data migrations.
4. AlwaysOn Availability Groups
5. Windows Server Core Support
6. User-Defined Server Roles
7. Big Data Support
8. PowerView
9. SQL Azure Enhancements
10. Tabular Model (SSAS)
11. DQS Data quality services
12. File Table - an enhancement to the FILESTREAM feature which was introduced in 2008.
13. Enhancement in Error Handling including THROW statement
14. Improvement to SQL Server Management Studio Debugging a. SQL Server 2012 introduces more options to control breakpoints. b. Improvements to debug-mode windows
    c. Enhancement in IntelliSense - like Inserting Code Snippets.

**The following features added in version 2014 from its previous version:**

1. In-Memory OLTP Engine – Improves performance up to 20 times.
2. AlwaysOn Enhancements
3. Buffer Pool Extension
4. Hybrid Cloud Features
5. Enhancement in Column store Indexes (like Updatable Column store Indexes)
6. Query Handling Enhancements (like parallel SELECT INTO)
7. Power BI for Office 365 Integration
8. Delayed durability
9. Enhancements for Database Backups

**The following features added in version 2016 from its previous version:**

1. Always Encrypted - Always Encrypted is designed to protect data at rest or in motion.
2. Real-time Operational Analytics
3. PolyBase into SQL Server
4. Native JSON Support

5. Query Store
6. Enhancements to AlwaysOn
7. Enhanced In-Memory OLTP
8. Multiple TempDB Database Files
9. Stretch Database
10. Row Level Security
11. In-Memory Enhancements

> T-SQL Enhancements or new additions in SQL Server 2016

1. TRUNCATE TABLE with PARTITION
2. DROP IF EXISTS
3. STRING_SPLIT and STRING_ESCAPE Functions
4. ALTER TABLE can now alter many columns while the table remains online, using WITH (ONLINE = ON | OFF).
5. MAXDOP for DBCC CHECKDB, DBCC CHECKTABLE and DBCC CHECKFILEGROUP
6. ALTER DATABASE SET AUTOGROW_SINGLE_FILE
7. ALTER DATABASE SET AUTOGROW_ALL_FILES
8. COMPRESS and DECOMPRESS Functions
9. FORMATMESSAGE Statement
10. 2016 introduces 8 more properties with SERVERPROPERTY

a. InstanceDefaultDataPath

b. InstanceDefaultLogPath

c. ProductBuild

d. ProductBuildType

e. ProductMajorVersion

f. ProductMinorVersion

g. ProductUpdateLevel

h. ProductUpdateReference

# Chapter 109: SQL Server Management Studio (SSMS)

SQL Server Management Studio (SSMS) is a tool to manage and administer SQL Server and SQL Database.

SSMS is offered free of charge by Microsoft.

SSMS Documentation is available.

## Section 109.1: Refreshing the IntelliSense cache

When objects are created or modified they are not automatically available for IntelliSense. To make them available to IntelliSense the local cache has to be refreshed.

Within an query editor window either press `Ctrl + Shift + R` or select `Edit | IntelliSense | Refresh Local Cache` from the menu.

After this all changes since the last refresh will be available to IntelliSense.

# Chapter 110: Managing Azure SQL Database

## Section 110.1: Find service tier information for Azure SQL Database

Azure SQL Database has different editions and performance tiers.

You can find version, edition (basic, standard, or premium), and service objective (S0,S1,P4,P11, etc.) of SQL Database that is running as a service in Azure using the following statements:

```
SELECT @@version
SELECT DATABASEPROPERTYEX('Wwi', 'EDITION')
SELECT DATABASEPROPERTYEX('Wwi', 'ServiceObjective')
```

## Section 110.2: Change service tier of Azure SQL Database

You can scale-up or scale-down Azure SQL database using ALTER DATABASE statement:

```
ALTER DATABASE WWI
MODIFY (SERVICE_OBJECTIVE = 'P6')
-- or
ALTER DATABASE CURRENT
MODIFY (SERVICE_OBJECTIVE = 'P2')
```

If you try to change service level while changing service level of the current database is still in progress you wil get the following error:

> Msg 40802, Level 16, State 1, Line 1 A service objective assignment on server '.....' and database '.....' is already in progress. Please wait until the service objective assignment state for the database is marked as 'Completed'.

Re-run your ALTER DATABASE statement when transition period finishes.

## Section 110.3: Replication of Azure SQL Database

You can create a secondary replica of database with the same name on another Azure SQL Server, making the local database primary, and begins asynchronously replicating data from the primary to the new secondary.

```
ALTER DATABASE <<mydb>>
ADD SECONDARY ON SERVER <<secondaryserver>>
WITH ( ALLOW_CONNECTIONS = ALL )
```

Target server may be in another data center (usable for geo-replication). If a database with the same name already exists on the target server, the command will fail. The command is executed on the master database on the server hosting the local database that will become the primary. When ALLOW_CONNECTIONS is set to ALL (it is set to NO by default), secondary replica will be a read-only database that will allow all logins with the appropriate permissions to connect.

Secondary database replica might be promoted to primary using the following command:

```
ALTER DATABASE mydb FAILOVER
```

You can remove the secondary database on secondary server:

```
ALTER DATABASE <<mydb>>
REMOVE SECONDARY ON SERVER <<testsecondaryserver>>
```

## Section 110.4: Create Azure SQL Database in Elastic pool

You can put your azure SQL Database in SQL elastic pool:

```
CREATE DATABASE wwi
( SERVICE_OBJECTIVE = ELASTIC_POOL ( name = mypool1 ) )
```

You can create copy of an existing database and place it in some elastic pool:

```
CREATE DATABASE wwi
AS COPY OF myserver.WideWorldImporters
( SERVICE_OBJECTIVE = ELASTIC_POOL ( name = mypool1 ) )
```

# Chapter 111: System database - TempDb

## Section 111.1: Identify TempDb usage

Following query will provide information about TempDb usage. Analyzing the counts you can identify which thing is impacting TempDb

```
SELECT
 SUM (user_object_reserved_page_count)*8 AS usr_obj_kb,
 SUM (internal_object_reserved_page_count)*8 AS internal_obj_kb,
 SUM (version_store_reserved_page_count)*8  AS version_store_kb,
 SUM (unallocated_extent_page_count)*8 AS freespace_kb,
 SUM (mixed_extent_page_count)*8 AS mixedextent_kb
FROM sys.dm_db_file_space_usage
```

| Attribute | Meaning |
|---|---|
| Higher number of user objects | More usage of Temp tables , cursors or temp variables |
| Higher number of internal objects | Query plan is using a lot of database. Ex: sorting, Group by etc. |
| Higher number of version stores | Long running transaction or high transaction throughput |

## Section 111.2: TempDB database details

Below query can be used to get TempDB database details:

```
USE [MASTER]
SELECT * FROM sys.databases WHERE database_id = 2
```

OR

```
USE [MASTER]
SELECT * FROM sys.master_files WHERE database_id = 2
```

With the help of below DMV, you can check how much TempDb space does your session is using. This query is quite helpful while debugging TempDb issues

```
SELECT * FROM sys.dm_db_session_space_usage WHERE session_id = @@SPID
```

# Appendix A: Microsoft SQL Server Management Studio Shortcut Keys

## Section A.1: Shortcut Examples

1. Open a new Query Window with current connection ( `Ctrl` + `N` )
2. Toggle between opened tabs ( `Ctrl` + `Tab` )
3. Show/Hide Results pane ( `Ctrl` + `R` )
4. Execute highlighted query ( `Ctrl` + `E` )
5. Make selected text uppercase or lowercase ( `Ctrl` + `Shift` + `U` , `Ctrl` + `Shift` + `L` )
6. Intellisense list member and complete word ( `Ctrl` + `Space` , `Tab` )
7. Go to line ( `Ctrl` + `G` )
8. close a tab in SQL Server Managament Studio ( `Ctrl` + `F4` )

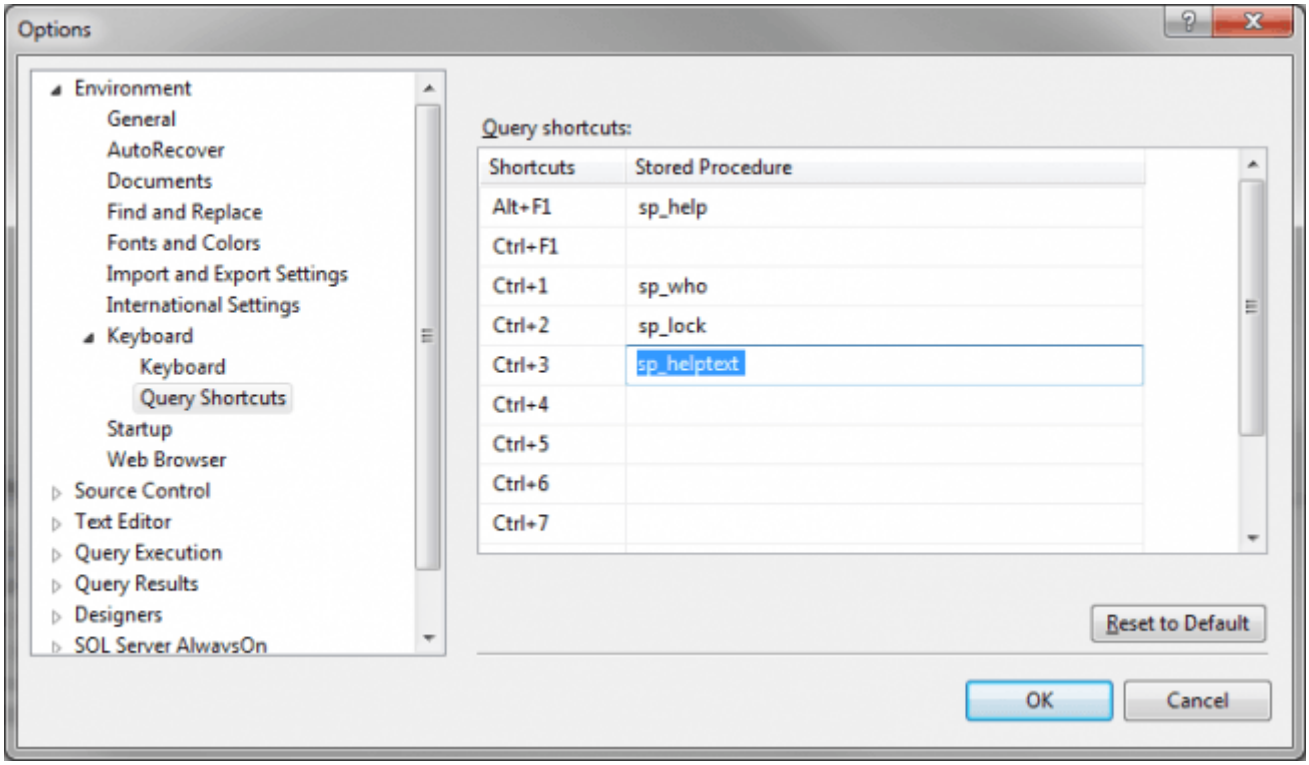
## Section A.2: Menu Activation Keyboard Shortcuts

1. Move to the SQL Server Management Studio menu bar ( `ALT` )
2. Activate the menu for a tool component ( `ALT` + `HYPHEN` )
3. Display the context menu ( `SHIFT` + `F` )
4. Display the New File dialog box to create a file ( `CTRL` + `N` )
5. Display the Open Project dialog box to open an existing project ( `CTRL` + `SHIFT` + `0` )
6. Display the Add New Item dialog box to add a new file to the current project ( `CTRL` + `SHIFT` + `A` )
7. Display the Add Existing Item dialog box to add an existing file to the current project ( `CTRL` + `SHIFT` + `A` )
8. Display the Query Designer ( `CTRL` + `SHIFT` + `Q` )
9. Close a menu or dialog box, canceling the action ( `ESC` )

## Section A.3: Custom keyboard shortcuts

Go to Tools -> Options. Go to Environment -> Keyboard -> Query Shortcuts

On the right side you can see some shortcuts which are by default in SSMS. Now if you need to add a new one, just click on any column under Stored Procedure column.




Click OK. Now please go to a query window and select the stored procedure then press CTRL+3, it will show the stored procedure result.

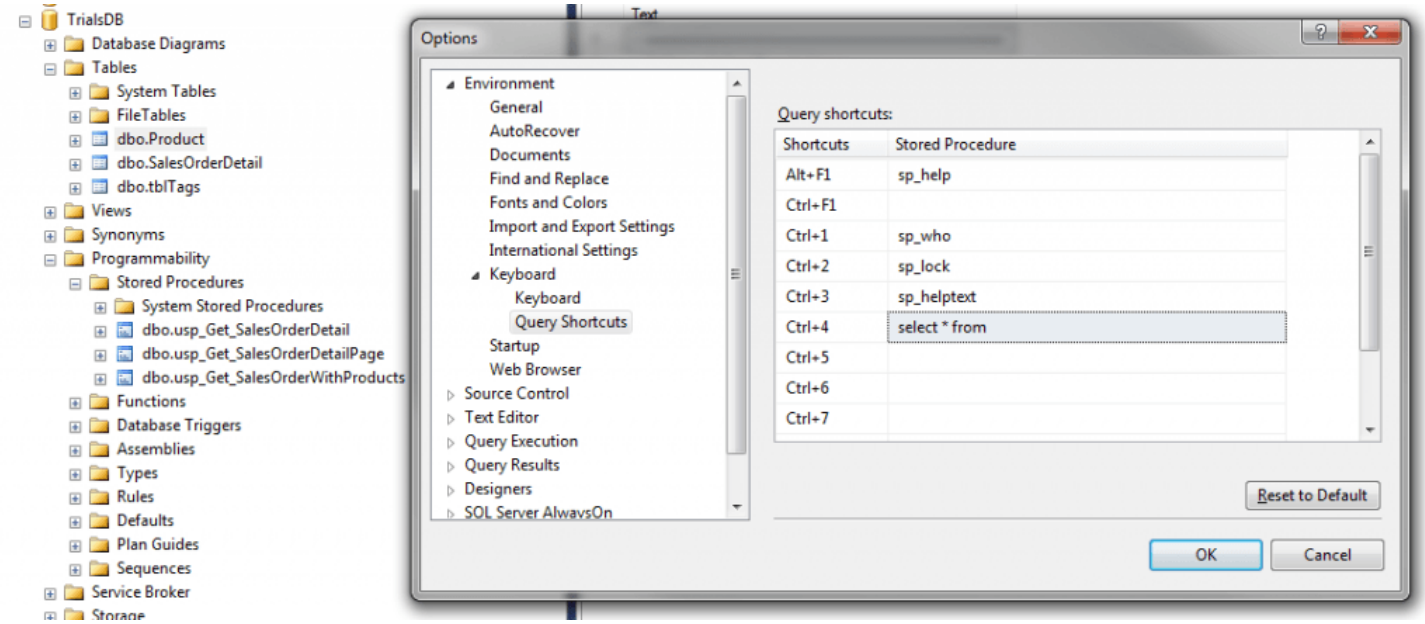
```
usp_Get_SalesOrderDetail
```

100 %

Results | Messages

| | Text |
|---|---|
| 1 | -- ============================================ |
| 2 | -- Author: <Author,Sibeesh Venu> |
| 3 | -- Create date: <Create Date, 18-Feb-2016> |
| 4 | -- Description: <Description,To fetch SalesOrderDetail> |
| 5 | -- ============================================ |
| 6 | CREATE PROCEDURE [dbo].[usp_Get_SalesOrderDetail] |
| 7 | AS |
| 8 | BEGIN |
| 9 | -- SET NOCOUNT ON added to prevent extra result sets ... |
| 10 | -- interfering with SELECT statements. |
| 11 | SET NOCOUNT ON; |
| 12 | |
| 13 | -- Select statements for procedure here |
| 14 | SELECT top(100) SalesOrderID,SalesOrderDetailID,Carri... |
| 15 | END |

Now if you need to select all the records from a table when you select the table and press CTRL+5(You can select any key). You can make the shortcut as follows.




Now go ahead and select the table name from the query window and press CTRL+4(The key we selected), it will give you the result.

# Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

| | |
|---|---|
| 5arx | Chapter 101 |
| Abhilash R Vankayala | Chapters 1 and 15 |
| Abhishek Jain | Chapter 1 |
| Adam Porad | Chapter 9 |
| Ahmad Aghazadeh | Chapters 59, 64 and 69 |
| Akash | Chapter 11 |
| Ako | Chapter 76 |
| Akshay Anand | Chapters 59 and 43 |
| alalp | Chapters 1 and 38 |
| Alex | Chapter 15 |
| Almir Vuk | Chapters 1 and 17 |
| Amir Pourmand | Chapter 7 |
| Andrea | Chapter 51 |
| Andy | Chapters 33 and 23 |
| andyabel | Chapter 46 |
| Anuj Tripathi | Chapters 51 and 111 |
| APH | Chapters 33, 8, 24, 70 and 73 |
| Arif | Chapter 36 |
| Arthur D | Chapter 1 |
| Athafoud | Chapters 22 and 47 |
| A_Arnold | Chapters 9 and 41 |
| Baodad | Chapter 51 |
| barcanoj | Chapter 16 |
| bassrek | Chapter 97 |
| bbrown | Chapter 36 |
| BeaglesEnd | Chapter 1 |
| beercohol | Chapter 24 |
| Behzad | Chapters 68 and 77 |
| Bellash | Chapter 9 |
| Ben Thul | Chapter 102 |
| Bharat Prasad Satyal | Chapter 12 |
| Biju jose | Chapter 1 |
| Bino Mathew Varghese | Chapters 33, 39, 50 and 112 |
| bluefeet | Chapter 25 |
| Brandon | Chapter 22 |
| Chetan Sanghani | Chapter 25 |
| chrisb | Chapter 38 |
| cnayak | Chapters 50, 35, 19, 79 and 43 |
| cteski | Chapters 33, 36, 9, 50, 17, 43, 80 and 85 |
| D M | Chapter 1 |
| dacohenii | Chapter 25 |
| Dan Guzman | Chapters 49 and 108 |
| Daniel Lemke | Chapter 18 |
| David Kaminski | Chapter 15 |
| dd4711 | Chapters 109 and 42 |
| Dean Ward | Chapter 33 |
| DForck42 | Chapters 36 and 84 |
| Dheeraj Kumar | Chapter 57 |
| DhruvJoshi | Chapters 47 and 33 |

| | |
|---|---|
| Dileep | Chapter 33 |
| DVJex | Chapter 16 |
| Edathadan Chief aka Arun | Chapters 9, 23, 12, 17, 60, 3 and 71 |
| ErikE | Chapter 38 |
| Eugene Niemand | Chapters 50 and 75 |
| feetwet | Chapters 51 and 46 |
| Gajendra | Chapter 33 |
| Gidil | Chapters 1 and 24 |
| gofr1 | Chapter 22 |
| Gordon Bell | Chapter 1 |
| gotqn | Chapter 32 |
| Hadi | Chapters 17, 41, 7 and 71 |
| Hamza Rabah | Chapter 34 |
| Hari K M | Chapter 50 |
| hatchet | Chapter 41 |
| Henrik Staun Poulsen | Chapter 59 |
| HK1 | Chapter 33 |
| Igor Micev | Chapter 41 |
| intox | Chapter 16 |
| Iztoksson | Chapters 1 and 33 |
| James | Chapter 10 |
| James Anderson | Chapters 51, 39 and 49 |
| JamieA | Chapters 51 and 9 |
| Jared Hooper | Chapters 1 and 9 |
| Jayasurya Satheesh | Chapter 50 |
| Jeffrey L Whitledge | Chapter 43 |
| Jeffrey Van Laethem | Chapters 51, 36 and 56 |
| Jenism | Chapter 23 |
| Jesse | Chapter 48 |
| Jibin Balachandran | Chapters 52 and 41 |
| Jivan | Chapter 4 |
| Joe Taras | Chapters 1 and 43 |
| John Odom | Chapters 1 and 49 |
| Jones Joseph | Chapter 89 |
| Josh B | Chapter 17 |
| Josh Morel | Chapter 58 |
| Jovan MSFT | Chapters 11, 26, 52, 41, 28, 5, 29, 30, 27, 20, 55, 34, 81, 89, 91, 93, 80, 85, 31, 103, 94, 110, 96, 98, 99 and 86 |
| juergen d | Chapter 23 |
| jyao | Chapter 51 |
| K48 | Chapter 1 |
| Kane | Chapter 62 |
| Kannan Kandasamy | Chapters 7, 35 and 44 |
| Karthikeyan | Chapter 12 |
| Keith Hall | Chapters 32, 36 and 8 |
| Kiran Ukande | Chapters 25 and 23 |
| kolunar | Chapters 47 and 45 |
| Kritner | Chapters 9, 39, 54 and 7 |
| Laughing Vergil | Chapters 1, 51, 7, 2 and 14 |
| lord5et | Chapter 21 |
| LowlyDBA | Chapters 51, 33 and 39 |
| Luis Bosquez | Chapter 83 |
| M.Ali | Chapters 13 and 108 |
| Mahesh Dahal | Chapter 1 |

| | |
|---|---|
| Malt | Chapter 1 |
| Mani | Chapter 82 |
| MarmiK | Chapters 46 and 107 |
| martinshort | Chapter 16 |
| MasterBob | Chapter 52 |
| Matas Vaitkevicius | Chapters 50, 23, 16, 21, 65 and 2 |
| Matej | Chapters 12 and 100 |
| Matt | Chapters 1 and 78 |
| Max | Chapters 1, 16, 18 and 105 |
| Merenix | Chapter 88 |
| Metanormal | Chapter 90 |
| Michael Stum | Chapter 16 |
| Mihai | Chapter 1 |
| Mono | Chapter 26 |
| Monty Wild | Chapter 36 |
| Moshiour | Chapter 15 |
| MrE | Chapter 25 |
| Mspaja | Chapter 69 |
| Mudassir Hasan | Chapter 1 |
| n00b | Chapters 1 and 16 |
| Nathan Skerl | Chapter 50 |
| Neil Kennedy | Chapter 92 |
| New | Chapters 45 and 70 |
| Nick | Chapters 1 and 9 |
| Nick.McDermaid | Chapter 37 |
| Oluwafemi | Chapters 61 and 74 |
| OzrenTkalcecKrznaric | Chapters 1, 33 and 73 |
| Pat | Chapter 87 |
| Paul Bambury | Chapter 22 |
| Peter Tirrell | Chapter 1 |
| Phrancis | Chapters 1, 51, 62, 33, 9, 8, 41, 67 and 63 |
| Pirate X | Chapter 50 |
| podiluska | Chapters 21 and 7 |
| Prateek | Chapter 1 |
| P□□□□□ | Chapters 52 and 6 |
| Raghu Ariga | Chapter 106 |
| Raidri | Chapter 41 |
| Ram Grandhi | Chapter 33 |
| Randall | Chapter 53 |
| ravindra | Chapter 20 |
| Rhumborl | Chapter 51 |
| Robert Columbia | Chapters 16 and 17 |
| Ross Presser | Chapter 41 |
| Rubenisme | Chapter 104 |
| S.Karras | Chapter 39 |
| Sam | Chapters 1 and 22 |
| scsimon | Chapters 51, 39, 50 and 12 |
| Sender | Chapter 80 |
| Shaneis | Chapter 1 |
| sheraz mirza | Chapter 95 |
| Sibeesh Venu | Chapter 112 |
| Siyual | Chapters 9 and 10 |
| Soukai | Chapter 9 |
| spaghettidba | Chapter 51 |
| sqlandmore.com | Chapter 72 |

| | |
|---|---|
| SQLMason | Chapter 36 |
| sqluser | Chapter 56 |
| Susang | Chapter 49 |
| Tab Alleman | Chapter 12 |
| takrl | Chapter 41 |
| Techie | Chapter 75 |
| TheGameiswar | Chapter 40 |
| Thuta Aung | Chapter 1 |
| Tom V | Chapter 59 |
| Tot Zam | Chapters 1, 51, 16 and 17 |
| TZHX | Chapter 51 |
| Uberzen1 | Chapters 1 and 20 |
| UnhandledExcepSean | Chapter 9 |
| user1690166 | Chapter 25 |
| user_0 | Chapters 1 and 85 |
| Vexator | Chapter 43 |
| Vikas Vaidya | Chapter 14 |
| Vladimir Oselsky | Chapter 92 |
| Wolfgang | Chapters 11 and 32 |
| Zohar Peled | Chapters 9, 41, 61, 7, 4 and 66 |

# You may also like