微软 SQL 服务器
专业人士笔记

Microsoft® SQL Server®

专业人士笔记

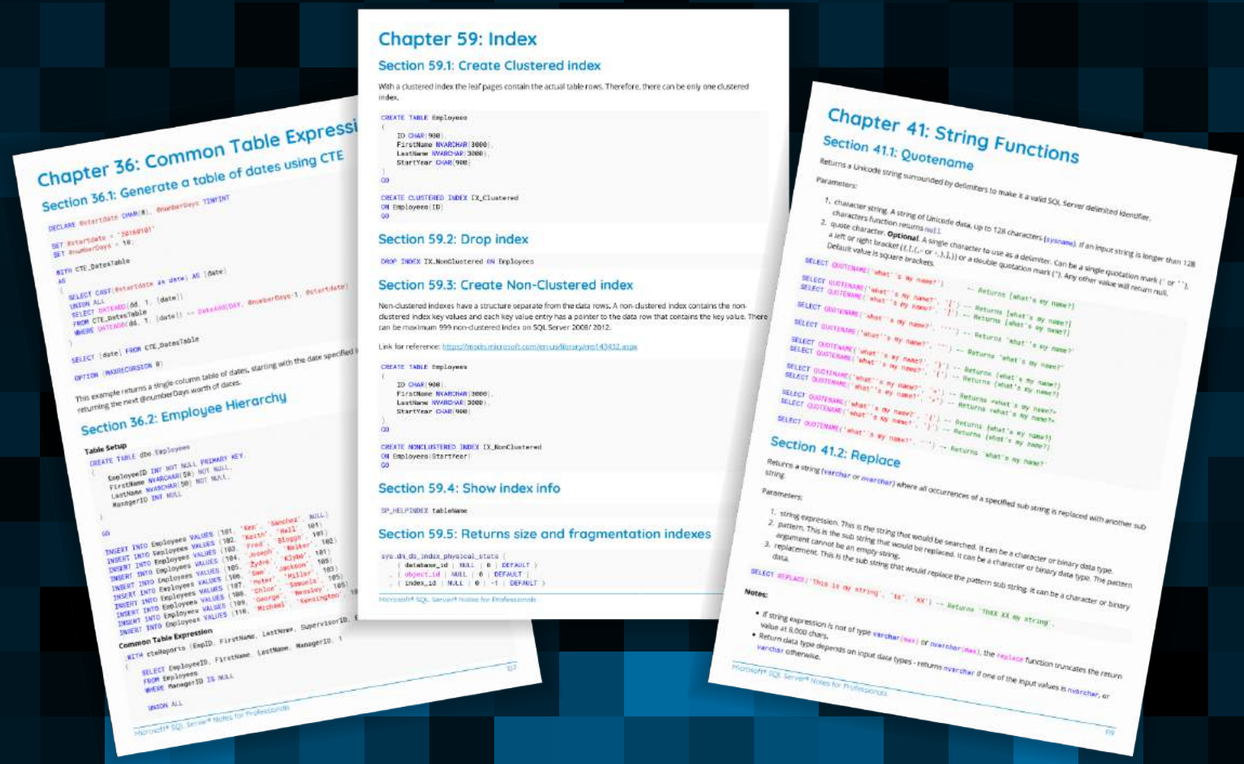
200 多页
专业提示和技巧

GoalKicker.com
免费编程书籍

免责声明
这是一本非官方的免费书籍，旨在教育用途，且与官方 Microsoft® SQL Server® 组织或公司无关。所有商标和注册商标均为其各自所有者所有

Microsoft® SQL Server®

Notes for Professionals



200+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official Microsoft® SQL Server® group(s) or company(s). All trademarks and registered trademarks are the property of their respective owners

目录

Contents

关于

请随意免费分享此PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/MicrosoftSQLServerBook

本书所提供的信息不保证正确或准确，使用风险自负

请将反馈和更正发送至web@petercv.com

About

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

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：开始使用Microsoft SQL Server

版本	发布日期
SQL Server 2017	2017-10-01
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

第1.1节：INSERT / SELECT / UPDATE / DELETE：数据操作语言的基础

数据操作语言（简称DML）包括INSERT、UPDATE和DELETE等操作：

```
-- 创建表 HelloWorld

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


-- DML操作INSERT，向表中插入一行数据
INSERT INTO HelloWorld (Description) VALUES ('Hello World')


-- DML操作SELECT，显示表内容
SELECT * FROM HelloWorld


-- 从表中选择特定列
SELECT Description FROM HelloWorld


-- 显示表中的记录数
SELECT Count(*) FROM HelloWorld


-- DML操作UPDATE，更新表中指定行
UPDATE HelloWorld SET Description = 'Hello, World!' WHERE Id = 1


-- 从表中选择行（更新后描述有何变化？）
SELECT * FROM HelloWorld


-- DML 操作 - DELETE，从表中删除一行
DELETE FROM HelloWorld WHERE Id = 1


-- 选择表。查看 DELETE 操作后的表内容
```


Chapter 1: Getting started with Microsoft SQL Server

Version	Release Date
SQL Server 2017	2017-10-01
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

Data **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
```

在此脚本中，我们正在创建一个表以演示一些基本查询。

以下示例展示了如何查询表：

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

将选择数据库Northwind中Customer表按CompanyName列排序的前10条记录（Northwind 是微软的示例数据库之一，可以从 here 下载）：

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

注意，Use Northwind; 会更改所有后续查询的默认数据库。您仍然可以通过使用完全限定的语法以 [数据库].[架构].[表] 的形式引用数据库：

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

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

如果你从不同的数据库查询数据，这非常有用。请注意，位于“中间”的 dbo 被称为模式（schema），在使用完全限定语法时需要指定。你可以把它看作数据库中的一个文件夹。dbo 是默认模式。默认模式可以省略。所有其他用户定义的模式都需要指定。

如果数据库表包含像保留字一样命名的列，例如 `Date`，你需要用方括号将列名括起来，像这样：

```
-- 降序排列
SELECT TOP 10 [Date] FROM dbo.MyLogTable
ORDER BY [Date] DESC
```

如果列名中包含空格（不推荐），同样适用。另一种语法是使用双引号代替方括号，例如：

```
-- 降序排列
SELECT top 10 "Date" from dbo.MyLogTable
order by "Date" desc
```

等价但不太常用。注意双引号和单引号的区别：单引号用于字符串，即

```
-- 降序排列
SELECT top 10 "Date" from dbo.MyLogTable
where UserId='johndoe'
```



```
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
```

是有效的语法。请注意，T-SQL 对 NChar 和 NVarchar 数据类型有一个 N 前缀，例如：

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

返回所有公司名称以 AL 开头的公司（% 是通配符，使用方法类似于 DOS 命令行中的星号，例如 DIR AL*）。对于 LIKE，有几个通配符可用，here 可以查看更多详情。

连接（Joins）

如果你想查询的字段不在单个表中，而是在多个表中，连接非常有用。例如：
你想查询 Northwind 数据库中 Region 表的所有列。但你发现你还需要 RegionDescription，该字段存储在另一个表 Region 中。不过，有一个公共键 RegionID，可以用来将这些信息合并到一个查询中，如下所示（Top 5 只返回前 5 行，省略它则返回所有行）：

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

将显示Territories表中的所有列以及Region表中的RegionDescription列。结果按TerritoryDescription排序。

表别名

当查询需要引用两个或多个表时，使用表别名可能会很有用。表别名是对表的简写引用，可以代替完整的表名，减少输入和编辑。使用别名的语法是：

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

其中as是可选关键字。例如，之前的查询可以重写为：

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

别名在查询中必须对所有表唯一，即使你使用同一个表两次。例如，如果你的员工表（Employee）包含一个SupervisorId字段，你可以使用此查询返回员工及其主管的姓名：

```
SELECT e.*,
s.姓名 as 主管姓名 -- 重命名字段以便输出
FROM 员工 e
INNER JOIN 员工 s
    ON e.主管编号 = s.员工编号
```



```
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.员工编号 = 111
```

联合

正如我们之前所见，Join 会添加来自不同表的列。但如果你想合并来自不同来源的行呢？在这种情况下，你可以使用UNION。假设你正在筹划一个聚会，想邀请不仅是员工，还有客户。那么你可以运行以下查询来实现：

```
SELECT 名字+' '+姓氏 as 联系人姓名, 地址, 城市 FROM 员工
UNION
SELECT 联系人姓名, 地址, 城市 FROM 客户
```

它将返回员工和客户的姓名、地址和城市，合并在一个表中。注意，重复的行（如果有的话）会被自动去除（如果不想去除，使用 UNION ALL）。所有参与联合的 SELECT 语句的列数、列名、顺序和数据类型必须匹配——这就是为什么第一个 SELECT 将员工的 名字 和 姓氏 合并成

联系人姓名。

表变量

如果需要处理临时数据（尤其是在存储过程里），使用表变量会很有用：
“真实”表和表变量的区别在于，表变量仅存在于内存中用于临时处理。

示例：

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

在内存中创建一个表。在这种情况下，前缀@是必须的，因为它是一个变量。你可以执行上述所有DML操作来插入、删除和选择行，例如：

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

但通常，你会基于真实表来填充它，比如

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

它将从真实表 dbo.Region 中读取过滤后的值，并将其插入到内存表 @Region 中——这样可以用于后续处理。例如，您可以在连接中使用它，如

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

这将在本例中返回所有Northern和Southern地区。更详细的信息可以在here找到。
临时表在here中讨论，如果你有兴趣了解更多关于该主题的内容。

注意：微软仅建议在表变量中的数据行数少于100时使用表变量。如果你将处理更多数据，请改用临时表或临时表（temp table）。


```
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.

第1.2节：从表中选择所有行和列

语法：

```
SELECT *
FROM table_name
```

使用星号操作符*作为选择表中所有列的快捷方式。所有行也将被选择，因为该SELECT语句没有WHERE子句来指定任何过滤条件。

如果你给表添加别名，比如本例中的e，这也同样适用：

```
SELECT *
FROM 员工 AS e
```

或者如果你想从特定表中选择所有内容，可以使用别名 + " .* "：

```
SELECT e.*, d.部门名称
FROM 员工 AS e
    INNER JOIN 部门 AS d
        ON e.部门ID = d.部门ID
```

数据库对象也可以使用完全限定名访问：

```
SELECT * FROM [服务器名称].[数据库名称].[模式名称].[表名称]
```

这并不一定推荐，因为更改服务器和/或数据库名称会导致使用完全限定名的查询因对象名称无效而无法执行。

请注意，如果查询分别在单个服务器、数据库和模式上执行，通常可以省略表名称之前的字段。但是，数据库通常有多个模式，在这些情况下应尽可能不省略模式名称。

警告： 在生产代码或存储过程中使用 SELECT * 可能会导致后续问题（例如表中添加新列，或列顺序被调整），尤其是当代码对列的顺序或返回的列数做出简单假设时。因此，在生产代码中，最好始终在SELECT语句中显式指定列名。

```
SELECT 列1, 列2, 列3
FROM 表名称
```

第1.3节：更新特定行

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

上述代码将 HelloWorlds 表中 "Id = 5" 的记录的字段 "HelloWorld" 的值更新为 "HELLO WORLD!!!"。

注意：在更新语句中，建议使用 "where" 子句以避免更新整个表，除非您的需求不同。

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.

第1.4节：删除所有行

```
DELETE
FROM Helloworlds
```

这将删除表中的所有数据。执行此代码后，表中将不包含任何行。与`DROP TABLE`不同，此操作保留表本身及其结构，您可以继续向该表插入新行。

删除表中所有行的另一种方法是截断表，具体如下：

```
TRUNCATE TABLE HelloWords
```

与 DELETE 操作的区别有几个：

1. 截断操作不会存储在事务日志文件中
2. 如果存在`IDENTITY`字段，该字段将被重置
3. TRUNCATE 可以应用于整个表，而不能应用于部分表（相比之下，使用DELETE命令可以关联WHERE子句）

TRUNCATE 的限制

1. 如果存在FOREIGN KEY引用，则不能对表执行 TRUNCATE
2. 如果表参与了INDEXED视图
3. 如果表通过TRANSACTIONAL复制或MERGE REPLICATION发布
4. 它不会触发表中定义的任何触发器

[sic]

第1.5节：代码中的注释

Transact-SQL支持两种注释写法。注释会被数据库引擎忽略，供人阅读。

注释以--开头，直到遇到新行之前都会被忽略：

```
-- 这是一个注释
SELECT *
FROM MyTable -- 这是另一个注释
WHERE Id = 1;
```

斜杠星号注释以/*开始，以*/结束。所有位于这两个定界符之间的文本都被视为注释块。

```
/* 这是
一个多行
注释块。 */
SELECT Id = 1, [Message] = '第一行'
UNION ALL
SELECT 2, '第二行'
/* 这是单行注释 */
SELECT '更多';
```

斜杠星号注释的优点是即使SQL语句丢失换行符，注释仍然有效。这种情况可能发生在故障排查时捕获SQL语句时。

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.

斜杠星号注释可以嵌套，斜杠星号注释内部的起始/*需要用*/结束才有效。以下代码将导致错误

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

即使斜杠星号在引号内，也被视为注释的开始。因此需要用另一个结束星号斜杠结束。正确的写法是

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

第1.6节：PRINT

向输出控制台显示一条消息。使用 SQL Server Management Studio 时，这条消息将显示在消息标签页，而不是结果标签页：

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

第1.7节：选择符合条件的行

通常，语法如下：

```
SELECT <列名>
FROM <表名>
WHERE <条件>
```

例如：

```
SELECT 名字, 年龄
FROM 用户
WHERE 姓氏 = '史密斯'
```

条件可以很复杂：

```
SELECT 名字, 年龄
FROM 用户
WHERE 姓氏 = '史密斯' AND (城市 = '纽约' OR 城市 = '洛杉矶')
```

第1.8节：更新所有行

更新的一种简单形式是增加表中某个字段的所有值。为此，我们需要定义该字段和增量值

下面是一个示例，将Score字段的值增加1（所有行）：

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


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
```

这可能很危险，因为如果你不小心对表中的特定行执行了更新，而实际上是对所有行执行了更新，可能会破坏数据。

第1.9节：TRUNCATE TABLE

```
TRUNCATE TABLE Helloworlds
```

这段代码将删除表Helloworlds中的所有数据。Truncate table与Delete from Table代码几乎相同。区别在于Truncate不能使用where子句。Truncate table被认为比delete更好，因为它使用更少的事务日志空间。

注意，如果存在标识列，则会重置为初始种子值（例如，自增ID将从1重新开始）。如果标识列被用作另一张表的外键，这可能导致不一致。

第1.10节：检索基本服务器信息

```
SELECT @@VERSION
```

返回实例上运行的 MS SQL Server 版本。

```
SELECT @@SERVERNAME
```

返回 MS SQL Server 实例的名称。

```
SELECT @@SERVICENAME
```

返回 MS SQL Server 作为 Windows 服务运行时的服务名称。

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

返回运行 SQL Server 的机器的物理名称。用于识别故障转移集群中的节点。

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

在故障转移集群中返回 SQL Server 可以运行的所有节点。如果不是集群，则不返回任何内容。

第 1.11 节：创建新表并从旧表插入记录

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

创建一个具有旧表结构的新表，并将所有行插入到新表中。

一些限制

1. 您不能将表变量或表值参数指定为新表。
2. 即使源表是分区表，也不能使用 SELECT...INTO 来创建分区表
   已分区。SELECT...INTO 不使用源表的分区方案；相反，新表是在默认文件组中创建的。要向分区表中插入行，必须先创建分区表，然后使用 INSERT INTO...SELECT FROM 语句。
3. 源表中定义的索引、约束和触发器不会转移到新表中，

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,

这些对象也不能在 SELECT...INTO 语句中指定。如果需要这些对象，可以在执行 SELECT...INTO 语句后创建它们。

4. 指定 ORDER BY 子句并不保证行会按照指定顺序插入。

当选择列表中包含稀疏列时，稀疏列属性不会转移到新表中的该列。如果新表中需要此属性，请在执行 SELECT...INTO 语句后修改列定义以包含此属性。

5. 当计算列包含在选择列表中时，新表中的相应列不是计算列。新列中的值是执行 SELECT...INTO 时计算得到的值。

[sic]

第1.12节：使用事务安全地更改数据

每当你更改数据时，在数据操作语言（DML）命令中，你可以将更改包装在一个事务中。DML包括UPDATE、TRUNCATE、INSERT和DELETE。确保更改正确数据的方法之一是使用事务。

DML更改会对受影响的行加锁。当你开始一个事务时，必须结束该事务，否则所有在DML中被更改的对象将继续被启动事务的人锁定。你可以用ROLLBACK或COMMIT结束事务。 ROLLBACK会将事务内的所有内容恢复到原始状态。

COMMIT将数据置于最终状态，之后无法撤销更改，除非执行另一个DML语句。

示例：

```
--创建测试表

使用[你的数据库]
GO
创建表 test_transaction (column_1 varchar(10))
GO

插入到
dbo.test_transaction
        ( column_1 )
值
        ( 'a' )

开始事务 --这是事务的开始

更新 dbo.test_transaction
设置 column_1 = 'B'
输出 INSERTED.*
条件 column_1 = 'A'


回滚事务  --回滚将撤销你的更改
          --或者，使用提交来保存结果

选择 * 从 dbo.test_transaction    --查看更改后表的内容

删除表 dbo.test_transaction
```

备注：

- 这是一个 简化的示例，不包含错误处理。但任何数据库操作都可能失败并

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

因此抛出异常。 这里是一个示例，展示了所需的错误处理可能的样子。你绝对不要在没有错误处理的情况下使用事务，否则可能会使事务处于未知状态。

- 根据 隔离级别，事务会对被查询或更改的数据加锁。你需要确保事务不会运行过长时间，因为它们会锁定数据库中的记录并可能导致与其他并行运行事务的 死锁。将事务中的操作尽可能保持简短，并尽量减少锁定数据的量，以降低影响。

第1.13节：获取表行数

以下示例可用于查找数据库中特定表的总行数，前提是将 table_name 替换为你想查询的表名：

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

也可以通过基于表的 HEAP（index_id = 0）或聚集索引（index_id = 1）连接回表的分区，使用以下脚本获取所有表的行数：

```
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' /* 取消注释以查找特定表 */
GROUP BY    [Tables].name;
```

这是可能的，因为每个表本质上都是单分区表，除非向其添加了额外的分区。该脚本的另一个好处是不会干扰对表行的读写操作。

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'.

第2章：数据类型

本节讨论 SQL Server 可以使用的数据类型，包括它们的数据范围、长度及限制（如果有的话）。

第2.1节：精确数值

精确数值数据类型有两大类——整数（Integer）和定点精度与小数位数（Fixed Precision and Scale）。

整数数据类型

- 位
- tinyint
- smallint
- int
- bigint

*整数*是永远不包含小数部分的数值，并且总是使用固定的存储空间。
整数数据类型的范围和存储大小如表所示：

数据类型	范围	存储
位	0 或 1	1 位 **
tinyint	0 到 255	1 字节
smallint	-2^15（-32,768）到 2^15-1（32,767）	2 字节
整数	-2^31 (-2,147,483,648) 到 2^31-1 (2,147,483,647)	4 字节
大整数	-2^63 (-9,223,372,036,854,775,808) 到 2^63-1 (9,223,372,036,854,775,807)	8 字节

定点精度和标度数据类型

- 数值型
- 十进制
- 小金额
- 金额

这些数据类型适用于精确表示数字。只要数值能在数据类型可存储的范围内，数值就不会出现舍入问题。这对于任何财务计算都非常有用，因为舍入错误会导致会计师的极大困扰。

注意，decimal 和 numeric 是同一数据类型的同义词。

数据类型	范围	存储
十进制 [(p [, s])] 或 数值型 [(p [, s])]	-10^38 + 1 到 10^38 - 1	详见精度表

定义decimal或numeric数据类型时，可能需要指定精度 [p] 和小数位数 [s]。

精度是可以存储的数字位数。例如，如果需要存储1到999之间的值，则需要精度为3（以容纳100中的三位数字）。如果未指定精度，默认精度为18。

小数位数是小数点后的数字位数。如果需要存储0.00到999.99之间的数字，则需要指定精度为5（五位数字）和小数位数为2（小数点后两位）。必须指定精度才能指定小数位数。默认小数位数为零。

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.

decimal或numeric数据类型的精度定义了存储该值所需的字节数，如下所示：

精度表

精度	存储字节数
1 - 9	5
10-19	9
20-28	13
29-38	17

货币固定数据类型

这些数据类型专门用于会计和其他货币数据。这些类型具有固定的小数位数为4——小数点后始终显示四位数字。对于大多数使用大多数货币的系统来说，使用小数位数为2的数值类型通常就足够了。请注意，存储的值中不包含任何关于所表示货币类型的信息。

数据类型	范围	存储
money	-922,337,203,685,477.5808 到 922,337,203,685,477.5807	8字节
smallmoney	-214,748.3648 到 214,748.3647	4字节

第2.2节：近似数值类型

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

这些数据类型用于存储浮点数。由于这些类型仅用于存储近似数值，因此在任何舍入误差不可接受的情况下不应使用它们。然而，如果您需要处理非常大的数字，或小数位数不确定的数字，这些类型可能是您的最佳选择。

数据类型	范围	大小
浮点数	-1.79E+308 到 -2.23E-308，0 和 2.23E-308 到 1.79E+308，取决于下表中的 n	
实数	-3.40E + 38 到 -1.18E - 38，0 和 1.18E - 38 到 3.40E + 38	4 字节

n 浮点数 float 的值表。如果在声明 float 时未指定值，则默认使用 53。注意 float(24) 等同于 real 值。

n 值	精度	大小
1-24	7 位数字	4 字节
25-53	15 位数字	8 字节

第 2.3 节：日期和时间

这些类型存在于所有版本的 SQL Server 中

- datetime
- smalldatetime

这些类型存在于 SQL Server 2012 之后的所有版本中

- date

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

第2.4节：字符字符串

- char
- varchar
- text

第2.5节：Unicode字符字符串

- nchar
- nvarchar
- ntext

第2.6节：二进制字符串

- binary
- varbinary
- image

第2.7节：其他数据类型

- 游标
- 时间戳
- 层次ID
- 唯一标识符
- SQL变体
- XML
- 表
- 空间类型

- 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

第3章：数据类型转换

第3.1节：TRY_PARSE

版本 ≥ SQL Server 2012

它将字符串数据类型转换为目标数据类型（日期或数值）。

例如，源数据是字符串类型，我们需要转换为日期类型。如果转换尝试失败，则返回NULL值。

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

String_value – 该参数是源值，类型为 NVARCHAR(4000)。
Data_type – 此参数是目标数据类型，可以是日期或数字。
Culture – 这是一个可选参数，用于将值转换为指定的文化格式。假设您想以法语显示日期，则需要传入文化类型为'Fr-FR'。如果不传入任何有效的文化名称，PARSE将会报错。

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

SELECT TRY_PARSE(@fakeDate AS DATE); --解析失败返回NULL

SELECT TRY_PARSE(@realDate AS DATE); -- 类型不匹配返回NULL

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

第3.2节：TRY_CONVERT

版本 ≥ SQL Server 2012

它将值转换为指定的数据类型，如果转换失败则返回NULL。例如，源值是字符串格式，我们需要日期/整数格式，这时它可以帮助我们实现转换。

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

TRY_CONVERT() 如果转换成功，则返回转换为指定数据类型的值；否则返回null。

Data_type - 要转换成的数据类型。这里的length是可选参数，用于指定结果的长度。

Expression - 要转换的值
样式 - 这是一个可选参数，用于确定格式。假设你想要的日期格式是"2013年5月18日"
那么你需要传入样式为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); -- 2015年9月13日
```


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
```

第3.3节：TRY_CAST

版本 ≥ SQL Server 2012

它将值转换为指定的数据类型，如果转换失败则返回NULL。例如，源值是字符串格式，我们需要将其转换为双精度/整数格式。此时该函数可以帮助我们实现。

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

TRY_CAST() 如果转换成功，则返回转换为指定数据类型的值；否则返回null。

表达式 - 将要转换的源值。
数据类型 - 源值将转换成的目标数据类型。
长度 - 这是一个可选参数，指定目标数据类型的长度。

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

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

第3.4节：转换（Cast）

Cast()函数用于将一种数据类型的变量或数据转换为另一种数据类型。

语法

CAST ([表达式] AS 数据类型)

你要转换的表达式的数据类型是目标类型。你转换的表达式的数据类型是源类型。

```
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' 被转换为 25（字符串转整数）
--'15' 被转换为 15（字符串转整数）

--结果
 --40

DECLARE @C varchar(2)  = 'a'

select CAST(@C as int) as Result
--结果
 --将 varchar 值 'a' 转换为 int 数据类型时转换失败。
```

失败时抛出错误

第3.5节：转换

当你将表达式从一种类型转换为另一种类型时，通常在存储过程或其他例程中需要将数据从 datetime 类型转换为 varchar 类型。Convert 函数用于

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

此类操作。CONVERT() 函数可用于以各种格式显示日期/时间数据。语法

CONVERT(data_type(length), expression, style)

Style - 用于将 datetime 或 smalldatetime 转换为字符数据的样式值。将样式值加 100 可获得包含世纪的四位年份（yyyy）。

```
select convert(varchar(20),GETDATE(),108)

13:27:16
```

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
```

第4章：用户定义表类型

用户定义表类型（简称UDT）是一种允许用户定义表结构的数据类型。用户定义表类型支持主键、唯一约束和默认值。

第4.1节：创建一个包含单个int列且该列为主键的UDT

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

第4.2节：创建一个包含多列的UDT

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

第4.3节：创建一个带有唯一约束的UDT：

```
创建类型 MyUniqueNamesType 作为表
(
名字 varchar(10),
    姓氏 varchar(10),
    唯一约束 (名字,姓氏)
)
```

注意：用户定义的表类型中的约束不能命名。

第4.4节：创建带有主键和带默认值列的用户定义类型：

```
创建类型 MyUniqueNamesType 作为表
(
名字 varchar(10),
    姓氏 varchar(10),
    创建日期 datetime 默认 GETDATE()
    主键 (名字,姓氏)
)
```


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)
)
```

第5章：SELECT语句

在SQL中，`SELECT`语句从数据集合如表或视图中返回结果集。`SELECT`语句可以与其他子句如`WHERE`、`GROUP BY`或`ORDER BY`一起使用，以进一步细化所需结果。

第5.1节：从表中基本SELECT

从某个表（此处为系统表）选择所有列：

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

或者，只选择某些特定的列：

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

第5.2节：使用WHERE子句过滤行

WHERE子句仅过滤满足某些条件的行：

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

第5.3节：使用ORDER BY排序结果

ORDER BY子句按某列或表达式对返回的结果集中的行进行排序：

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

第5.4节：使用GROUP BY分组结果

GROUP BY子句按某个值对行进行分组：

```
SELECT type, count(*) as c
FROM sys.objects
GROUP BY type
```

你可以对每个分组应用某些函数（聚合函数）来计算该分组中记录的总和或计数。

类型	c
SQ	3
S	72
IT	16
PK	1
U	5

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

第5.5节：使用 HAVING 子句过滤分组

HAVING 子句移除不满足条件的分组：

```
SELECT type, count(*) as c
FROM sys.objects
按类型分组
筛选计数(*) < 10
```

类型 c
SQ3
PK1
U 5

第5.6节：只返回前N行

TOP子句只返回结果中的前N行：

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

第5.7节：使用OFFSET FETCH进行分页

OFFSET FETCH子句是TOP的更高级版本。它允许你跳过N1行并获取接下来的N2行：

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

你可以只使用OFFSET而不使用FETCH来跳过前50行：

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

第5.8节：无FROM的SELECT（无数据源）

SELECT语句可以在没有FROM子句的情况下执行：

```
declare @var int = 17;

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

在这种情况下，将返回一行包含表达式值/结果的数据。

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.

第6章：SQL Server中的别名

以下是为SQL Server中的列提供别名的几种不同方式

第6.1节：在派生表名后给出别名

这是一种奇怪的方法，大多数人甚至不知道它的存在。

```
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)
```

- 演示

第6.2节：使用 AS

这是ANSI SQL方法，适用于所有关系型数据库管理系统。广泛使用的方法。

```
创建表 AliasNameDemo (id INT,firstname VARCHAR(20),lastname VARCHAR(20))

插入数据到 AliasNameDemo
VALUES      (1,'MyFirstName','MyLastName')

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

第6.3节：使用 =

这是我偏好的方法。与性能无关，仅是个人选择。它使代码看起来
更简洁。如果表达式很长，你可以轻松看到结果列名，而不必滚动代码。

```
创建表 AliasNameDemo (id INT,firstname VARCHAR(20),lastname VARCHAR(20))

插入数据到 AliasNameDemo
VALUES      (1,'MyFirstName','MyLastName')

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

第6.4节：不使用AS

此语法与使用AS关键字类似。只是我们不必使用AS关键字

```
创建表 AliasNameDemo (id INT,firstname VARCHAR(20),lastname VARCHAR(20))

插入数据到 AliasNameDemo
VALUES      (1,'我的名字','我的姓氏')

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


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
```

第7章：NULL值

在SQL Server中，NULL表示缺失或未知的数据。这意味着NULL实际上不是一个值；更准确地说，它是一个值的占位符。这也是为什么你不能将NULL与任何值比较，甚至不能与另一个NULL比较的原因。

第7.1节：COALESCE（）

COALESCE（）按顺序计算参数，并返回第一个初始值不为NULL的表达式的当前值。

```
DECLARE @MyInt int -- 变量在赋值前为null。
DECLARE @MyInt2 int -- 变量在赋值前为null。
DECLARE @MyInt3 int -- 变量在赋值前为null。

SET @MyInt3  = 3

SELECT COALESCE (@MyInt, @MyInt2 ,@MyInt3 ,5) -- 返回 3 ：@MyInt3 的值。
```

虽然 ISNULL() 的操作方式类似于 COALESCE()，但 ISNULL() 函数只接受两个参数——一个用于检查，另一个在第一个参数为 NULL 时使用。另见下面的 ISNULL。

第7.2节：ANSI NULLS

来自 MSDN

> 在未来版本的 SQL Server 中，ANSI_NULLS 将始终开启，任何显式将该选项设置为关闭的应用程序都会产生错误。避免在新开发工作中使用此功能，并计划修改当前使用此功能的应用程序。

ANSI NULLS 设置为关闭时，允许对 null 值进行 =/<> 比较。

给定以下数据：

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

开启 ANSI NULLS 时，以下查询：

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

不会返回任何结果。然而相同的查询，在关闭 ANSI NULLS 时：

```
set ansi_nulls off

SELECT id
FROM table
```


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
```

将返回 id 0。

第7.3节：ISNULL()

IsNull() 函数接受两个参数，如果第一个参数为 null，则返回第二个参数。

参数：

1. 检查表达式。任何数据类型的表达式。
2.替代值。如果检查表达式为null，则返回此值。替代值必须是可以隐式转换为检查表达式数据类型的数据类型。

函数IsNull()返回与检查表达式相同的数据类型。

```
DECLARE @MyInt int -- 所有变量在赋值前均为null。

SELECT ISNULL(@MyInt, 3) -- 返回3。
```

另见上文的COALESCE函数

第7.4节：Is null / Is not null

由于null不是一个值，不能使用比较运算符与null进行比较。
要检查某列或变量是否为null，需要使用is null：

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

以下语句将选择值6，因为所有与null值的比较结果均为false或未知：

```
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
```

将 @Date 变量的内容设置为 null 并重试，以下语句将返回 5：

```
SET @Date = NULL -- 注意这里的 '=' 是赋值运算符！

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
```

第7.5节：NULL 比较

在比较时，NULL 是一个特殊情况。

假设以下数据。


```
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
```

使用查询：

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

将返回 id 1

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

将返回 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.

如果你想让 NULL 在 =, <> 比较中被"计数"为值，必须先将其转换为可计数的数据类型：

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

OR

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

返回 0 和 2

或者你可以更改你的 ANSI Null 设置。

第7.6节：带有NOT IN子查询的NULL

在处理带有子查询中包含NULL的NOT IN子查询时，我们需要排除NULL以获得预期结果

```
创建表 #outertable (i int)
创建表 #innertable (i int)
```



```
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)
```

```
向 #outertable 插入数据(i) 值为 (1), (2),(3),(4), (5)
向 #innertable 插入数据(i) 值为 (2), (3), (null)

选择 * 从 #outertable 其中 i 在 (选择 i 从 #innertable)
--2
--3
--到目前为止一切正常

选择 * 从 #outertable 其中 i 不在 (选择 i 从 #innertable)
--这里的预期是得到1,4,5，但结果不是。由于NULL的存在，结果为空
执行时相当于 {select * from #outertable where i not in (null)}

--为了解决这个问题
select * from #outertable where i not in (select i from #innertable where i is not null)
--你将获得预期的结果
--1
--4
--5
```

在处理子查询中不包含null时，请注意你的预期输出


```
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

第8章：变量

第8.1节：声明表变量

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

当你创建普通表时，使用 `CREATE TABLE 名称 (列)` 语法。创建表变量时，使用 `DECLARE @名称 TABLE (列)` 语法。

要在SELECT语句中引用表变量，SQL Server要求必须给表变量指定一个别名，否则会报错：

> 必须声明标量变量"@TableVariableName"。

即

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

/*
-- 以下两个被注释的语句会产生错误：
SELECT *
FROM @Table1
INNER JOIN @Table2 ON @Table1.Example = @Table2.Example

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

-- 但以下语句可以正常工作：
SELECT *
FROM @Table1 T1
INNER JOIN @Table2 T2 ON T1.Example = T2.Example

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

第8.2节：使用SELECT更新变量

使用SELECT，可以一次更新多个变量。

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


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

当使用SELECT从表列更新变量时，如果有多个值，将使用最后一个值。
（适用正常顺序规则——如果未指定排序，顺序不保证。）

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

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

打印 @Variable
```

> 2

```
SELECT TOP 1 @Variable = 示例
FROM #Test
ORDER BY 示例 升序

打印 @Variable
```

> 1

如果查询未返回任何行，变量的值将不会改变：

```
SELECT TOP 0 @Variable = 示例
FROM #Test
ORDER BY 示例 升序

打印 @Variable
```

> 1

第8.3节：一次声明多个变量，并赋初始值

```
声明
  @Var1 整数 = 5,
  @Var2 NVARCHAR(50) = N'Hello World',
  @Var3 日期时间 = GETDATE()
```

第8.4节：使用SET更新变量

```
声明 @VariableName 整数
设置 @VariableName = 1
打印 @VariableName
```


> 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

使用SET时，您一次只能更新一个变量。

第8.5节：通过从表中选择更新变量

根据数据的结构，您可以创建动态更新的变量。

```
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)
```

在大多数情况下，使用此方法时，您需要确保查询只返回一个值。

第8.6节：复合赋值运算符

版本 ≥ SQL Server 2008 R2

支持的复合运算符：

> `+=` 加并赋值
>
> `-=` 减并赋值
>
> `*=` 乘并赋值
>
> `/=` 除并赋值
>
> `%=` 取模并赋值
>
> `&=` 按位与并赋值
>
> `^=` 按位异或并赋值
>
> `|=` 按位或并赋值

示例用法：

```
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
```


> 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
```

第9章：日期

第9.1节：使用CONVERT进行日期和时间格式化

您可以使用 CONVERT 函数将 datetime 数据类型转换为格式化的字符串。

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

您也可以使用一些内置代码转换为特定格式。以下是 SQL Server 内置的选项：

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

@convert_code	结果
100	"2016年7月21日 7:56AM"
101	"07/21/2016"
102	"2016.07.21"
103	2016年7月21日
104	2016年7月21日
105	2016年7月21日
106	2016年7月21日
107	2016年7月21日
108	07:57:05
109	2016年7月21日 7:57:45:707AM
110	2016年7月21日
111	2016年7月21日
112	"20160721"
113	"2016年7月21日 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] -- 2016年7月21日 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] -- 2016年7月21日 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
```


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
```

第9.2节：使用FORMAT进行日期和时间格式化

版本 ≥ SQL Server 2012

您可以使用新函数：FORMAT()。

使用此函数，您可以将DATETIME字段转换为自定义的VARCHAR格式。

示例

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

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

> 星期一，2016年9月5日 12:01:02 上午

参数

鉴于正在格式化的DATETIME为2016-09-05 00:01:02.333，以下图表显示了针对所提供参数的输出结果。

Argument	Output
yyyy	2016
yy	16
MMMM	九月
MM	09
M	9
dddd	星期一
ddd	周一
dd	05
d	5
HH	00
H	0
hh	12
h	12
毫米	01
米	1
秒	02
秒	2
tt	上午
t	A
fff	333
ff	33
f	3


```
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

您也可以向FORMAT()函数提供单个参数以生成预格式化输出：

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

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

> 2016年9月5日星期一 上午4:01:02

单个参数	输出
D	2016年9月5日，星期一
d	9/5/2016
F	星期一，2016年9月5日 12:01:02 上午
f	2016年9月5日 星期一 凌晨12:01
G	2016/9/5 凌晨12:01:02
g	2016/9/5 凌晨12:01
M	9月5日
O	2016-09-05T00:01:02.3330000
R	2016年9月5日 星期一 00:01:02 GMT
秒	2016-09-05T00:01:02
T	凌晨12:01:02
t	凌晨12:01
U	2016年9月5日星期一 上午4:01:02
u	2016-09-05 00:01:02Z
Y	2016年9月

注意：上述列表使用的是 en-US 文化。可以通过第三个参数为 FORMAT() 指定不同的文化：

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

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

> 2016年9月5日 4:01:02

第9.3节：用于添加和减去时间段的DATEADD

通用语法：

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

要添加时间量，number 必须为正数。要减去时间量，number 必须为负数。

示例

```
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
```


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年9月5日 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
```

注意：DATEADD 的 datepart 参数也接受缩写形式。一般不建议使用这些缩写，因为它们可能会引起混淆（如 m 与 mi，ww 与 w 等）。

第9.4节：创建函数以计算某人在特定日期的年龄

此函数将接受两个日期时间参数，出生日期（DOB）和一个用于检查年龄的日期

```
  CREATE FUNCTION [dbo].[Calc_Age]
    (
    @DOB datetime , @calcDate datetime
    )
    RETURNS int
    AS
   开始
声明 @age 整数

如果 (@calcDate < @DOB  )
返回 -1

-- 如果出生日期在比较日期之后，则返回 -1
SELECT @age = YEAR(@calcDate) - YEAR(@DOB) +
  CASE WHEN DATEADD(year,YEAR(@calcDate) - YEAR(@DOB)
  ,@DOB) > @calcDate THEN -1 ELSE 0 END


返回 @age


结束
```

例如，检查出生于2000年1月1日的人的当前年龄

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

第9.5节：获取当前日期时间

内置函数GETDATE和GETUTCDATE均返回当前日期和时间，且不带时区偏移。

这两个函数的返回值均基于运行SQL

Server实例的计算机的操作系统。

GETDATE的返回值表示与操作系统相同的时区的当前时间。GETUTCDATE的返回值表示当前的UTC时间。

这两个函数都可以包含在查询的SELECT子句中，或作为WHERE子句中布尔表达式的一部分。

示例：


```
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:

```
-- 示例查询，选择服务器时区和UTC的当前时间
SELECT GETDATE() as SystemDateTime, GETUTCDATE() as UTCDateTime

-- 示例查询，记录事件日期在过去的记录。
SELECT * FROM MyEvents WHERE EventDate < GETDATE()
```

还有一些其他内置函数返回当前日期时间的不同变体：

```
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
```

第9.6节：获取某个月的最后一天

使用DATEADD和DATEDIFF函数，可以返回某个月的最后日期。

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

版本 ≥ SQL Server 2012

EOMONTH函数提供了更简洁的方式来返回某个月的最后日期，并且有一个可选参数用于偏移月份。

```
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
```

第9.7节：跨平台日期对象

版本 ≥ SQL Server 2012

在 Transact SQL 中，您可以使用 [DATEFROMPARTS][1]（或 [DATETIMEFROMPARTS][1]）函数将对象定义为 Date（或 DateTime），如下所示：

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

您提供的参数是DATEFROMPARTS函数的年、月、日，对于DATETIMEFROMPARTS函数，您需要提供年、月、日、小时、分钟、秒和毫秒。

这些方法很有用且值得使用，因为使用普通字符串构建日期（或日期时间）可能会失败，这取决于主机的区域、位置或日期格式设置。

第9.8节：从日期时间中仅返回日期

有多种方法可以从日期时间对象中返回日期

1. SELECT CONVERT(Date, GETDATE())
2. SELECT DATEADD(dd, 0, DATEDIFF(dd, 0, GETDATE())) 返回 2016-07-21 00:00:00.000
3. SELECT CAST(GETDATE() AS DATE)
4. SELECT CONVERT(CHAR(10),GETDATE(),111)


```
-- 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')`

请注意，选项4和5返回的是字符串，而不是日期。

第9.9节：使用DATEDIFF计算时间段差异

通用语法：

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

如果 datetime_expr 相对于 datetime_expr2 是过去时间，则返回正数，否则返回负数。

示例

```
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
```

注意：DATEDIFF 的 datepart 参数也接受缩写形式。一般不建议使用这些缩写，因为它们可能会引起混淆（如m与 mi，ww 与 w 等）。

DATEDIFF 也可用于确定 UTC 与 SQL Server 本地时间之间的偏移。以下语句可用于计算 UTC 与本地时间（包括时区）之间的偏移。

```
select  DATEDIFF(hh, getutcdate(), getdate()) as 'CentralTimeOffset'
```

第9.10节：DATEPART 和 DATENAME

DATEPART 返回指定日期时间表达式的指定datepart的数值。

DATENAME 返回表示指定日期的指定datepart的字符串。实际上 DATENAME 主要用于获取月份名称或星期几的名称。

还有一些简写函数用于获取日期时间表达式的年、月或日，它们的行为类似于 DATEPART 使用各自的datepart单位。

语法：

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

示例：

```
DECLARE @now DATETIME2 = GETDATE();
```


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
-- DATEPART 和 DATENAME 之间的区别：
SELECT DATEPART(MONTH, @now)         --7
SELECT DATENAME(MONTH, @now)         --July
SELECT DATEPART(WEEKDAY, @now)       --5
SELECT DATENAME(WEEKDAY, @now)       --Thursday
--简写函数
SELECT DAY(@now)     --21
SELECT MONTH(@now)   --7
SELECT YEAR(@now)    --2016
```

注意：DATEPART 和 DATENAME 也接受 datepart 参数中的缩写。一般不建议使用这些缩写，因为它们可能会引起混淆（如 m 与 mi，ww 与 w 等）。

第9.11节：日期部分参考

以下是日期和时间函数可用的 datepart 值：

datepart	缩写
年	yy, yyyy
季度	qq, q
月	毫米，米
一年中的第几天	天数，年
天	日，天
周	周，周
工作日	工作日，周
小时	hh
分钟	分钟，n
秒	秒，s
毫秒ms	
微秒mcs	
纳秒ns	

注意：一般不建议使用缩写，因为它们可能会引起混淆（m 与 mi，ww 与 w 等）。datepart 表示法的完整版本有助于提高清晰度和可读性，应尽可能使用（month，minute，week，weekday 等）。

第9.12节：日期格式扩展

日期格式	SQL语句	示例输出
YY-MM-DD	SELECT RIGHT(CONVERT(VARCHAR(10), SYSDATETIME(), 20), 8) AS [YY-MM-DD] 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] SELECT REPLACE(CONVERT(VARCHAR(10), SYSDATETIME(), 111), '/', '-') AS [YYYY-MM-DD]	2011-06-08


```
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] 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] 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] 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] 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

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] 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] 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]	2011/6
年/月	SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 11) AS [YY/MM]	11/06
年/月	SELECT CONVERT(VARCHAR(7), SYSDATETIME(), 111) AS [YYYY/MM]	2011/06
年/月	SELECT RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YY/M]	11/6
年/月	SELECT CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [YYYY/M]	2011/6
月/日	SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 1) AS [MM/DD]	06/08
日/月	SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 3) AS [DD/MM]	08/06
月/日	SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) AS [M/D]	6/8
日/月	SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '/' + CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) AS [D/M]	8/6
月.日.年	SELECT REPLACE(CONVERT(VARCHAR(10), SYSDATETIME(), 101), '/', '.') AS [MM.DD.YYYY]	06.08.2011
月.日.年	SELECT REPLACE(CONVERT(VARCHAR(8), SYSDATETIME(), 1), '/', '.') AS [月.日.年]	06.08.11
月.日.年(四位数年份)	SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [月.日.年(四位数年份)]	6.8.2011
月.日.年(两位数年份)	SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + '.' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [月.日.年(两位数年份)]	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
年.月	SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)) AS [年.月]	6.2011
月.年	SELECT CAST(MONTH(SYSDATETIME()) AS VARCHAR(2)) + '.' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [月.年]	6.11
年.月	SELECT CONVERT(VARCHAR(7), SYSDATETIME(), 102) AS [年.月]	2011.06
年.月	SELECT CONVERT(VARCHAR(5), SYSDATETIME(), 2) AS [年.月]	11.06

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]	2011年6月8日
Mon YYYY	SELECT LEFT(DATENAME(MONTH, SYSDATETIME()), 3) + ' ' + DATENAME(YEAR, SYSDATETIME()) AS [Mon YYYY]	2011年6月
月份 年份	SELECT DATENAME(MONTH, SYSDATETIME()) + ' ' + DATENAME(YEAR, SYSDATETIME()) AS [月份 年份]	2011年6月
日 月份	SELECT RIGHT('0' + DATENAME(DAY, SYSDATETIME()), 2) + ' ' + DATENAME(MONTH, SYSDATETIME()) AS [日 月份]	06月08日
月份 日	SELECT DATENAME(MONTH, SYSDATETIME()) + ' ' + RIGHT('0' + DATENAME(DAY, SYSDATETIME()), 2) AS [月份 日]	6月08日
日 月份 年份（两位）	SELECT CAST(DAY(SYSDATETIME()) AS VARCHAR(2)) + ' ' + DATENAME(MM, SYSDATETIME()) + ' ' + RIGHT(CAST(YEAR(SYSDATETIME()) AS VARCHAR(4)), 2) AS [日 月 年]	08 六月 11
日 月 年	SELECT RIGHT('0' + DATENAME(DAY, SYSDATETIME()), 2) + ' ' + DATENAME(MONTH, SYSDATETIME())+ ' ' + DATENAME(YEAR, SYSDATETIME()) AS [日 月 年]	08 六月 2011
周-年	SELECT REPLACE(RIGHT(CONVERT(VARCHAR(9), SYSDATETIME(), 6), 6), ' ', '-') AS [周-年]	六月-08
周-年份	SELECT REPLACE(RIGHT(CONVERT(VARCHAR(11), SYSDATETIME(), 106), 8), ' ', '-') AS [周-年份]	2011年6月
日-月-年	SELECT REPLACE(CONVERT(VARCHAR(9), SYSDATETIME(), 6), ' ', '-') AS [日-月-年]	08-六月-11
日-月-年(四位数)	SELECT REPLACE(CONVERT(VARCHAR(11), SYSDATETIME(), 106), ' ', '-') AS [日-月-年(四位数)]	2011年6月8日

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

第10章：生成一系列日期

参数	详细信息
@FromDate	生成日期范围的包含下限。
@ToDate	生成日期范围的包含上限。

第10.1节：使用递归CTE生成日期范围

使用递归CTE，您可以生成一个包含起止日期的日期范围：

```
声明  @FromDate    日期 = '2014-04-21',
        @ToDate      日期 = '2014-05-02'

;使用 DateCte (日期) 作为
(
    选择  @FromDate 联合 全部
    选择  日期加(天, 1, 日期)
    来自    DateCte
    条件   日期 < @ToDate
)
选择  日期
来自    DateCte
选项  (最大递归 0)
```

默认的最大递归设置为100。使用此方法生成超过100个日期时，需要在查询中使用选项(最大递归 N)部分，其中N是所需的最大递归设置。将其设置为0将完全取消最大递归限制。

第10.2节：使用计数表生成日期范围

另一种生成日期范围的方法是利用计数表创建范围内的日期：

```
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 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
```

第11章：数据库快照

第11.1节：创建数据库快照

数据库快照是 SQL Server 数据库（源数据库）的只读、静态视图。它类似于备份，但它作为其他数据库一样可用，因此客户端可以查询快照数据库。

```
CREATE DATABASE MyDatabase_morning -- 快照名称
ON (
NAME=MyDatabase_data, -- 源数据库数据文件的逻辑名称
     FILENAME='C:\SnapShots\MySnapshot_Data.ss' -- 快照文件；
)
AS SNAPSHOT OF MyDatabase; -- 源数据库名称
```

你也可以创建包含多个文件的数据库快照：

```
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;
```

第11.2节：还原数据库快照

如果源数据库中的数据损坏或写入了错误数据，在某些情况下，将数据库恢复到早于损坏时间的数据库快照可能是恢复数据库备份的合适替代方案。

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

> 警告： 这将删除自快照创建以来对源数据库所做的所有更改！

第11.3节：删除快照

您可以使用 DELETE DATABASE 语句删除现有的数据库快照：

```
DROP DATABASE Mydatabase_morning
```

在此语句中，您应引用数据库快照的名称。

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.

第12章：COALESCE

第12.1节：使用 COALESCE 构建逗号分隔字符串

我们可以使用 COALESCE 从多行获取逗号分隔的字符串，如下所示。

由于使用了表变量，我们需要执行整个查询一次。为了便于理解，我添加了 BEGIN 和 END 块。

```
开始

    --声明表变量以存储示例记录
    DECLARE @Table TABLE (FirstName varchar(256), LastName varchar(256))

    --向表变量 @Table 插入示例记录
    INSERT INTO @Table (FirstName, LastName)
    值
    ('约翰','史密斯'),
    ('简','杜')

    --创建变量以存储结果
    DECLARE @Names varchar(4000)

    --使用 COALESCE 函数，将逗号分隔的 FirstName 连接到 @Names 变量中
    SELECT @Names = COALESCE(@Names + ', ', '') + FirstName
    FROM @Table

    --现在选择实际结果
    SELECT @Names
    结束
```

第12.2节：从一列值列表中获取第一个非空值

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

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

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

第12.3节：Coalesce 基本示例

COALESCE() 返回参数列表中的第一个 非 空 值。假设我们有一个包含电话号码和手机号码的表，并且想为每个用户只返回一个号码。为了只获取一个号码，我们可以获取第一个 非 空 值。

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


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 separated 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
```



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

第13章：IF...ELSE

第13.1节：单个IF语句

像大多数其他编程语言一样，T-SQL也支持IF..ELSE语句。

例如，在下面的例子中1=1是表达式，其计算结果为真，控制流进入BEGIN..END块，Print语句打印字符串'One is equal to One'

```
IF ( 1 = 1)  --<-- 某个表达式
 BEGIN
   PRINT 'One is equal to One'
 END
```

第13.2节：多个IF语句

我们可以使用多个IF语句来检查彼此完全独立的多个表达式。

在下面的示例中，每个IF语句的表达式都会被计算，如果为真，则执行BEGIN...END块中的代码。在这个特定示例中，第一和第三个表达式为真，只有那些打印语句会被执行。

```
IF (1 = 1)  --<-- 某个表达式      --<-- 这是正确的
BEGIN
    PRINT 'First IF is True'          --<-- 这将被执行
END

IF (1 = 2)  --<-- 某个表达式
BEGIN
    PRINT 'Second IF is True'
结束

IF (3 = 3)  --<-- 一些表达式        --<-- 这是真的
BEGIN
    PRINT '第三个 IF 为真'            --<-- 这将被执行
END
```

第13.3节：单个 IF..ELSE 语句

在单个 IF..ELSE 语句中，如果 IF 语句中的表达式计算结果为真，控制流进入第一个 BEGIN..END 块，只有该块内的代码会被执行，Else 块将被忽略。

另一方面，如果表达式计算结果为 False，ELSE BEGIN..END 块将被执行，控制流永远不会进入第一个 BEGIN..END 块。

在下面的示例中，表达式将计算为假，Else 块将被执行，打印字符串'第一个表达式不为真'

```
IF ( 1 <> 1)  --<-- 一些表达式
 BEGIN
     PRINT 'One is equal to One'
 END
ELSE
 开始
     打印'第一个表达式不为真'
```


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'
```

```
结束
```

第13.4节：带有最终ELSE语句的多个IF...ELSE

如果我们有多个IF...ELSE IF语句，但我们也想在所有表达式都未被评估为True时执行某段代码，那么我们可以简单地添加一个最终的ELSE块，该块仅在所有IF或ELSE IF表达式都未被评估为True时执行。

在下面的示例中，没有任何IF或ELSE IF表达式为True，因此仅执行ELSE块并打印 “没有其他表达式为真”

```
IF ( 1 = 1 + 1 )
    BEGIN
       PRINT '第一个If条件'
    END
ELSE IF (1 = 2)
    BEGIN
        PRINT '第二个If Else块'
    END
ELSE IF (1 = 3)
    开始
        PRINT '第三个If Else块'
    结束
ELSE
    开始
        PRINT '没有其他表达式为真'  --<-- 只有这条语句会被打印
    END
```

第13.5节：多个IF...ELSE语句

我们经常需要检查多个表达式，并根据这些表达式采取特定的操作。
这种情况通过多个IF...ELSE IF语句来处理。

在此示例中，所有表达式从上到下依次求值。一旦某个表达式求值为真，
该代码块内的代码就会执行。如果没有表达式求值为真，则不会执行任何代码。

```
IF (1 = 1 + 1)
BEGIN
    PRINT '第一个If条件'
END
ELSE IF (1 = 2)
开始
    PRINT '第二个If Else块'
END
ELSE IF (1 = 3)
开始
    PRINT '第三个If Else块'
END
ELSE IF (1 = 1)      --<-- 这是正确的
BEGIN
    PRINT '最后的Else块'  --<-- 只有这条语句会被打印
END
```



```
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
```

第14章：CASE语句

第14.1节：简单的CASE语句

在一个简单的 CASE 语句中，一个值或变量会被检查是否匹配多个可能的答案。下面的代码是一个简单 CASE 语句的示例：

```
SELECT CASE DATEPART(WEEKDAY, GETDATE())
    WHEN 1 THEN '星期日'
    WHEN 2 THEN '星期一'
    WHEN 3 THEN '星期二'
    WHEN 4 THEN '星期三'
    WHEN 4 THEN '星期四'
    WHEN 5 THEN '星期五'
    WHEN 6 THEN '星期六'
结束
```

第14.2节：搜索 CASE 语句

在搜索 CASE 语句中，每个选项可以独立测试一个或多个值。下面的代码是一个搜索 CASE 语句的示例：

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

SELECT CASE
    WHEN LEFT(@FirstName, 1) IN ('a','e','i','o','u')
        THEN '名字以元音字母开头'
    WHEN LEFT(@LastName, 1) IN ('a','e','i','o','u')
        THEN '姓氏以元音字母开头'
    ELSE
        '名字和姓氏都不以元音字母开头'
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
```

第15章：INSERT INTO

INSERT INTO 语句用于向表中插入新记录。

第15.1节：插入多行数据

在 SQL Server 2008 或更高版本中插入多行数据的方法：

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

在较早版本的 SQL Server 中插入多行数据，请使用“UNION ALL”，示例如下：

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

注意，INSERT 查询中的“INTO”关键字是可选的。另一个警告是 SQL Server 一次 INSERT 只支持 1000 行，因此你必须分批插入。

第15.2节：使用 OUTPUT 获取新生成的 Id

执行 INSERT 时，可以使用OUTPUT INSERTED.ColumnName 来获取新插入行的值，例如新生成的 Id——当你有IDENTITY列或任何默认值或计算值时，这非常有用。

当以编程方式调用此方法（例如，从ADO.net）时，您应将其视为普通查询，并读取值，就像您执行了一个SELECT语句一样。

```
-- 创建表 OutputTest ([Id] INT NOT NULL 主键 自增, [Name] NVARCHAR(50))

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

如果在同一组查询或存储过程中需要最近添加行的 ID。

```
-- 创建一个表变量，列的数据类型与 ID 相同

DECLARE @LastId TABLE ( id int);

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

SELECT id FROM @LastId

-- 我们可以将值设置到变量中，稍后在存储过程中使用

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


Chapter 15: INSERT INTO

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

Section 15.1: 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 15.2: 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 programmatically 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)
```

第15.3节：从 SELECT 查询结果插入

插入从 SQL 查询（单行或多行）检索的数据

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

注意，SELECT 中的 'student' 是一个字符串常量，将插入到每一行中。

如果需要，可以从同一张表中选择并插入数据

第15.4节：插入单行数据

单行数据可以通过两种方式插入：

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

或者

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

注意，第二个插入语句只允许值的顺序与表的列完全相同
而第一个插入语句中，值的顺序可以像下面这样更改：

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

第15.5节：针对特定列的插入

要对特定列进行插入（而不是全部列），必须指定要更新的列。

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

只有当您未列出的列是可空的、标识列、时间戳数据类型或计算列；或者具有默认值约束的列时，这才有效。因此，如果其中任何列是非空、非标识、非时间戳、非计算、无默认值的列……那么尝试这种插入将触发错误消息，告诉您必须为相关字段提供值。

第15.6节：将 Hello World 插入表中

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

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


Section 15.3: 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 15.4: 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 15.5: 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 15.6: 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
```

第16章：合并（MERGE）

从 SQL Server 2008 开始，可以使用 MERGE 语句在单条语句中执行插入、更新或删除操作。

MERGE 语句允许你将数据源与目标表或视图连接，然后根据连接结果对目标执行多种操作。

第16.1节：使用 MERGE 进行插入 / 更新 / 删除

```
MERGE INTO 目标表

USING 源表
ON (目标表.PKID = 源表.PKID)

WHEN MATCHED AND (目标表.PKID > 100) THEN
    DELETE

WHEN MATCHED AND (目标表.PKID <= 100) THEN
    UPDATE SET
目标表.ColumnA = 源表.ColumnA,
       目标表.ColumnB = 源表.ColumnB

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

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

描述：

- MERGE INTO targetTable - 要修改的表
- USING sourceTable - 数据来源（可以是表、视图或表值函数）
- ON ... - targetTable 和 sourceTable 之间的连接条件。
- WHEN MATCHED - 找到匹配时要执行的操作
    - AND (targetTable.PKID > 100) - 额外必须满足的条件，才能执行该操作
- THEN DELETE - 从 targetTable 删除匹配的记录
- THEN UPDATE - 更新匹配记录中由 SET 指定的列
- WHEN NOT MATCHED - 当 targetTable 中未找到匹配时要执行的操作
- WHEN NOT MATCHED BY SOURCE - 当 sourceTable 中未找到匹配时要执行的操作

备注：

如果不需要特定操作，则省略该条件，例如移除 WHEN NOT MATCHED THEN INSERT 将阻止记录被插入

Merge 语句需要以分号结尾。

限制条件：

- WHEN MATCHED 不允许使用 INSERT 操作
- UPDATE 操作只能更新一行。这意味着连接条件必须产生唯一匹配。

Chapter 16: 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 16.1: 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.

第16.2节：使用CTE源进行合并

```
WITH SourceTableCTE AS
(
    SELECT * FROM SourceTable
)
MERGE
TargetTable 作为 target
使用 SourceTableCTE 作为 source
ON (target.PKID = source.PKID)
当匹配时 THEN
    UPDATE SET target.ColumnA = source.ColumnA
当 不 匹配时 THEN
    INSERT (ColumnA) VALUES (Source.ColumnA);
```

第16.3节：合并示例 - 同步源表和目标表

为说明MERGE语句，考虑以下两个表 -

1. dbo.Product：该表包含公司当前销售的产品信息
2. dbo.ProductNew：该表包含公司未来将销售的产品信息。

以下 T-SQL 将创建并填充这两个表

```
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)
```

现在，假设我们想要将 dbo.Product 目标表与 dbo.ProductNew 表同步。以下是该任务的标准：

Section 16.2: 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 16.3: 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. dbo.ProductNew 源表和 dbo.Product 目标表中都存在的产品，将在 dbo.Product 目标表中用新的产品信息进行更新。
2. dbo.ProductNew 源表中存在但 dbo.Product 目标表中不存在的任何产品，将插入到 dbo.Product 目标表中。
3. dbo.Product 目标表中存在但 dbo.ProductNew 源表中不存在的任何产品，必须从 dbo.Product 目标表中删除。以下是执行此任务的 MERGE 语句。

```
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
    然后删除
OUTPUT $action, INSERTED.*, DELETED.*;
```

注意：MERGE 语句末尾必须有分号。

	$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

第16.4节：使用派生源表的MERGE

```
MERGE INTO 目标表  AS 目标
USING (VALUES (1,'Value1'), (2, 'Value2'), (3,'Value3'))
        AS 源 (主键ID, 列A)
ON 目标.主键ID = 源.主键ID
WHEN  MATCHED  THEN
    UPDATE SET 目标.列A= 源.列A
WHEN NOT MATCHED THEN
    INSERT (主键ID, 列A) VALUES (源.主键ID, 源.列A);
```

第16.5节：使用EXCEPT的合并

使用EXCEPT防止对未更改记录的更新

```
MERGE 目标表 targ
USING 源表 AS src
    ON src.id = targ.id
WHEN  MATCHED
    并且存在 (
        选择 src.field
        除外
        选择 targ.field
        )
    那么
        更新
        设置 field = src.field
当 未 被目标匹配时
```


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 16.4: 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 16.5: 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
```

```
    那么
        插入（
id
            ,field
            )
        值（
src.id
            ,src.field
            )
当未被源匹配时
    则
        删除;
```



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

第17章：创建视图

第17.1节：创建带索引的视图

要创建带索引的视图，视图必须使用WITH SCHEMABINDING关键字创建：

```
创建视图 view_EmployeeInfo
WITH SCHEMABINDING
作为
    选择 EmployeeID,
       FirstName,
       LastName,
         HireDate
    来自 [dbo].Employee
GO
```

现在可以创建任何聚集或非聚集索引：

```
创建唯一聚集索引 IX_view_EmployeeInfo
在 view_EmployeeInfo
(
EmployeeID 升序
)
```

索引视图存在一些限制：

- 视图定义可以引用同一数据库中的一个或多个表。
- 一旦创建了唯一聚集索引，就可以针对该视图创建额外的非聚集索引。
- 您可以更新底层表中的数据-包括插入、更新、删除，甚至截断操作。
- 您不能修改底层的表和列。视图是使用 WITH SCHEMABINDING 选项创建的。
- 它不能包含 COUNT、MIN、MAX、TOP、外连接或其他一些关键字或元素。

有关创建索引视图的更多信息，您可以阅读这篇MSDN文章

第17.2节：CREATE VIEW

```
CREATE VIEW view_EmployeeInfo
AS
    SELECT 员工编号,
       名字,
       姓氏,
       入职日期
    FROM 员工
GO
```

可以像选择表一样选择视图中的行：

```
SELECT FirstName
FROM view_EmployeeInfo
```

您也可以创建带有计算列的视图。我们可以通过添加一个

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

计算列来修改上述视图：

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

此视图添加了一个额外的列，当你SELECT行时该列将会出现。该额外列中的值将依赖于表Employee中的字段FirstName和LastName，并且当这些字段更新时会在后台自动更新。

第17.3节：带加密的CREATE VIEW

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

第17.4节：带INNER JOIN的CREATE VIEW

```
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
```

视图可以使用连接从多个来源选择数据，如表、表函数，甚至其他视图。这个示例使用了Person表中的FirstName和LastName列，以及Employee表中的JobTitle列。

现在可以使用此视图查看数据库中所有对应的经理行：

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

第17.5节：分组视图

分组视图基于带有GROUP BY子句的查询。由于每个分组在其构建的基础中可能有多行，因此这些视图必然是只读的。这类视图通常包含一个或多个聚合函数，主要用于报表目的。它们也方便用于解决SQL中的一些不足。考虑一个显示每个州最大销售额的视图。查询非常直接：

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

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;
```

第17.6节：联合视图

基于 UNION 或 UNION ALL 操作的视图是只读的，因为没有单一的方法可以将更改映射到基表中的某一行。UNION 操作符会从结果中删除重复行。UNION 和 UNION ALL 操作符都会隐藏行来自哪个表。这样的视图必须使用别名，因为 UNION [ALL] 中的列本身没有名称。理论上，两个不相交且各自没有重复行的表的 UNION 应该是可更新的。

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

```
创建视图 DepTally2 (emp_nbr, dependent_cnt)
作为 (选择 emp_nbr, 计数(*)
      来自 Dependents
     按 emp_nbr分组)
   联合
   (选择 emp_nbr, 0
      来自 Personnel 作为 P2
     其中 不存在
          (选择 *
             来自 Dependents 作为 D2
            其中 D2.emp_nbr = P2.emp_nbr));
```



```
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));
```

第18章：视图

第18.1节：创建带有架构绑定的视图

如果视图是使用 WITH SCHEMABINDING 创建的，则底层表不能被删除或以会破坏视图的方式修改。例如，视图中引用的表列不能被删除。

```
创建视图 dbo.PersonsView
带有模式绑定
作为
SELECT
姓名,
    地址
来自 dbo.PERSONS  -- 当存在模式绑定时，必须指定数据库模式
```

没有模式绑定的视图如果其基础表发生更改或被删除，可能会损坏。查询损坏的视图会导致错误信息。可以使用 sp_refreshview 确保没有模式绑定的现有视图不会损坏。

第18.2节：创建视图

```
创建视图 dbo.PersonsView
作为
SELECT
姓名,
    地址
来自 persons;
```

第18.3节：创建或替换视图

此查询将删除视图——如果它已存在——并创建一个新的视图。

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

创建视图 dbo.PersonsView
作为
SELECT
姓名,
    地址
来自 persons;
```


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;
```

第19章：联合（UNION）

第19.1节：UNION 和 UNION ALL

联合（Union）操作将两个或多个查询的结果合并为一个包含所有查询中所有行的结果集，并且会忽略存在的任何重复行。UNION ALL 也执行相同的操作，但会包含重复的值。下面的示例将使联合操作的概念更加清晰。使用联合时需要考虑的几点是：

1. 所有查询中的列数和列的顺序必须相同。

2. 数据类型必须兼容。

示例：

我们有三张表：成绩单1（Marksheet1）、成绩单2（Marksheet2）和成绩单3（Marksheet3）。成绩单3是成绩单2的重复表，包含与成绩单2相同的值。

表1：成绩单1（Marksheet1）

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

表2：成绩单2（Marksheet2）

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

表3：成绩单3

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

对表 Marksheet1 和 Marksheet2 进行联合

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


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
```

注意：三个表的联合输出结果与 Marksheet1 和 Marksheet2 的联合结果相同，因为联合操作不包含重复值。

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

输出

	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

全部联合

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

输出

	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

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

您会注意到这里使用 union all 也显示了 Marksheet3 中的重复值。

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

第20章：TRY/CATCH

第20.1节：TRY/CATCH中的事务

由于无效的日期时间，这将回滚两个插入操作：

```
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
结束尝试
BEGIN CATCH
    ROLLBACK TRANSACTION -- 先回滚然后抛出异常。
    THROW
END CATCH
```

这将提交两个插入操作：

```
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
```


第20.2节：在try-catch块中引发错误

RAISERROR函数将在TRY CATCH块中生成错误：

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

RAISERROR 第二个参数大于 10（本例中为 11）时，会停止 TRY 块中的执行并引发错误，该错误将在 CATCH 块中处理。您可以使用 ERROR_MESSAGE() 函数访问错误信息。
此示例的输出为：

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


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!
```

第 20.3 节：在 try catch 块中引发信息消息

RAISERROR 严重性（第二个参数）小于或等于 10 时，不会抛出异常。

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

在RAISERROR语句之后，第三条语句将被执行，CATCH块不会被调用。执行结果是：

```
第一条语句
这里有个问题！
第二条语句
```

第20.4节：重新抛出由RAISERROR生成的异常

您可以使用THROW语句重新抛出在CATCH块中捕获的错误：

```
DECLARE @msg nvarchar(50) = '这里有个问题！区域："%s" 行："%i"'
BEGIN TRY
    print '第一条语句';
    RAISERROR(@msg, 11, 1, 'TRY BLOCK', 2);
    print '第二条语句';
结束尝试
BEGIN CATCH
    print '错误：' + ERROR_MESSAGE();
    THROW;
END CATCH
```

请注意，在这种情况下，我们使用格式化参数（第四和第五个参数）引发错误。如果您想在消息中添加更多信息，这可能会很有用。执行结果是：

```
第一条语句
错误：这里有一个问题！区域：'TRY BLOCK' 行：'2'
消息 50000，级别 11，状态 1，行 26
这里有一个问题！区域：'TRY BLOCK' 行：'2'
```

第20.5节：在TRY/CATCH块中抛出异常

你可以在try catch块中抛出异常：

```
DECLARE @msg nvarchar(50) = 'Here is a problem!'
BEGIN TRY
    print '第一条语句';
    THROW 51000, @msg, 15;
    print '第二条语句';
END TRY
BEGIN CATCH
```


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 '错误：' + ERROR_MESSAGE();
    THROW;
END CATCH
```

异常将在CATCH块中处理，然后使用无参数的THROW重新抛出。

```
First statement
Error: Here is a problem!
消息 51000，级别 16，状态 15，行 39
这里有一个问题！
```

THROW 类似于 RAISERROR，但有以下区别：

- 建议新应用程序应使用 THROW 而不是 RAISERROR。
- THROW 可以使用任意数字作为第一个参数（错误号），RAISERROR 只能使用 sys.messages
  视图中的 ID
- THROW 的严重性为 16（不可更改）
- THROW 不能像 RAISERROR 那样格式化参数。如果需要此功能，请使用 FORMATMESSAGE 函数作为 RAISERROR 的参数。


```
    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.

第21章：WHILE 循环

第21.1节：使用 While 循环

WHILE 循环可以作为 CURSORS 的替代。以下示例将打印从 0 到 99 的数字。

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

第21.2节：使用 min 聚合函数的 While 循环

```
DECLARE @ID AS INT;

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

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


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
```

第22章：OVER子句

参数	详细信息
PARTITION BY	紧跟PARTITION BY的字段是分组的依据

第22.1节：累计和

使用商品销售表，我们将尝试了解商品销售额如何随日期增长。为此
我们将计算*累计和*，即按销售日期排序的每个商品的总销售额。

```
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
```

第22.2节：使用带有OVER的聚合函数

使用汽车表，我们将计算每个客户花费的总金额、最大值、最小值和平均金额，以及她带车维修的次数（COUNT）。

编号 客户编号 机械师编号 车型 状态 总费用

```
SELECT 客户编号,
       SUM(总费用) OVER(按客户编号分区) AS 总计,
       AVG(总费用) OVER(按客户编号分区) AS 平均,
       COUNT(总费用) OVER(按客户编号分区) AS 次数,
       MIN(总费用) OVER(按客户编号分区) AS 最小,
       MAX(总费用) OVER(按客户编号分区) AS 最大
  FROM 汽车表
 WHERE 状态 = '已完成'
```

注意，以这种方式使用OVER不会对返回的行进行聚合。上述查询将返回
以下结果：

客户编号	总计	平均	次数	最小	最大
1	430	215	2	200	230
1	430	215	2	200	230

重复的行可能对报告用途没有太大帮助。

如果您只是想汇总数据，最好使用 GROUP BY 子句以及适当的聚合函数，例如：

```
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
```


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
```

第22.3节：使用 NTILE 将数据划分为等分区间

假设您有几场考试的成绩，想要将每场考试的成绩划分为四分位数。

```
-- 设置数据：
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), -- 考试1成绩
(91, 2), (88, 2), (83, 2), (91, 2), (78, 2), (67, 2), (77, 2) -- 考试2成绩

-- 每场考试分成四个区间：
选择 ExamId,
ntile(4) over (partition by 考试编号 order by [值] desc) as 四分位数,
        值, 编号
from @values
order by 考试编号, 四分位数
```

	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

当你确实需要固定数量的分组且每组大致相同时，ntile表现非常好。
注意，只需使用ntile(100)，就可以轻松地将这些分数划分为百分位数。

第22.4节：使用聚合函数查找最新记录

使用图书馆数据库，我们尝试查找每位作者最近添加到数据库的书籍。对于这个简单的示例，我们假设每条记录的编号始终递增。

```
SELECT 最近书籍.姓名, 最近书籍.标题
FROM ( SELECT 作者.姓名,
书籍.标题,
                排名() OVER (PARTITION BY 作者.编号 ORDER BY 书籍.编号 DESC) AS 最新排名
        FROM 作者
        JOIN Books ON Books.AuthorId = Authors.Id
      ) MostRecentBook
WHERE MostRecentBook.NewestRank = 1
```

除了 RANK，还可以使用另外两个函数进行排序。在前面的例子中结果相同，但当排序导致每个排名有多行时，它们会给出不同的结果。

- RANK()：重复值获得相同排名，下一排名会考虑前一排名中重复值的数量

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()：重复值获得相同排名，下一排名总是比前一排名大一
- ROW_NUMBER()：会给每行一个唯一的“排名”，对重复值随机排序

例如，如果表中有一个非唯一列 CreationDate，且排序基于该列，以下查询：

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

可能产生如下结果：

作者	标题	创建日期	RANK	DENSE_RANK	ROW_NUMBER
作者 1	书籍 1	22/07/2016	1	1	1
作者 1	书籍 2	22/07/2016	1	1	2
作者 1	书籍 3	21/07/2016	3	2	3
作者 1	书籍 4	21/07/2016	3	2	4
作者 1	书籍 5	21/07/2016	3	2	5
作者 1	书籍 6	04/07/2016	6	3	6
作者 2	书籍 7	04/07/2016	1	1	1

- 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

第23章：分组（GROUP BY）

第23.1节：简单分组

订单表

客户编号	产品编号	数量	价格
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

当按特定列分组时，只返回该列的唯一值。

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

返回值：

客户编号
1
2
3

123聚合函数如

count() 应用于每个分组，而不是整个表：

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

返回值：

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

第23.2节：按多列分组（GROUP BY）

可能想要按多列进行GROUP BY

```
声明 @temp 表(age int, name varchar(15))

插入到 @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
```


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 年龄, 姓名, count(1) 计数
FROM @temp
GROUP BY 年龄, 姓名
```

将按年龄和姓名分组，并生成：

年龄 姓名 计数
18 卢克 3
21 卢克 2
18 马特1
21 马特2

第23.3节：带有ROLLUP和CUBE的GROUP BY

ROLLUP 操作符在生成包含小计和总计的报表时非常有用。

- CUBE 生成一个结果集，显示所选列中所有值组合的汇总数据。
- ROLLUP 生成一个结果集，显示所选列中值的层级汇总数据。

物品 颜色 数量
桌子 蓝色124
桌子 红色223
椅子 蓝色101
椅子 红色210

```
SELECT CASE WHEN (GROUPING(物品) = 1) THEN '全部'
            ELSE ISNULL(物品, '未知')
       END AS 物品,
       CASE WHEN (GROUPING(颜色) = 1) THEN '全部'
            ELSE ISNULL(颜色, '未知')
       END AS 颜色,
       SUM(数量) AS 数量总和
FROM 库存
按项目颜色分组并使用汇总

项目                 颜色                数量总和
-------------------- ------------------- --------------------------
椅子                 蓝色                101.00
椅子                 红色                 210.00
椅子                 全部                 311.00
桌子                 蓝色                124.00
桌子                 红色                 223.00
桌子                 全部                 347.00
全部                   全部                  658.00
```

(7 行受影响)

如果查询中的 ROLLUP 关键字改为 CUBE，CUBE 结果集相同，除了最后返回这两行额外数据：

```
全部                 蓝色                 225.00
```



```
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
```

```
全部                红色                433.00
```

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

第23.4节：多表多列分组

分组通常与连接语句一起使用。假设我们有两张表。第一张是学生表：

编号	全名	年龄
1	马特·琼斯	20
2	弗兰克·布鲁	21
3	安东尼·安吉尔	18

第二个表是每个学生可以选修的科目表：

科目编号	科目
1	数学
2	体育
3	物理

因为一个学生可以选修多个科目，一个科目也可以被多个学生选修（因此是多对多关系），我们需要第三个“关联”表。我们称这个表为学生科目表：

科目编号	学生编号
1	1
2	2
2	1
3	2
1	3
1	1

现在假设我们想知道每个学生选修的科目数量。这里单独使用GROUP BY语句是不够的，因为单个表中没有该信息。因此我们需要使用GROUP使用JOIN语句的BY：

```
选择Students.FullName, COUNT(Subject Id) 作为SubjectNumber 从Students_Subjects
左连接Students
在Students_Subjects.Student_id =Students.Id
按Students.FullName分组
```

给定查询的结果如下：

全名	科目数量
马特·琼斯	3
弗兰克·布鲁	2
安东尼·安吉尔	1

对于GROUP BY用法的更复杂示例，假设学生可能会多次将同一科目分配给自己（如Students_Subjects表所示）。在这种情况下，我们可以通过对多个列进行分组来计算每个科目被分配给学生的次数：

```
SELECT Students.FullName, Subjects.Subject,
COUNT(Students_subjects.Subject_id) AS NumberOfOrders
```



```
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
```

该查询返回以下结果：

全名	科目	科目编号
马特·琼斯	数学2	
马特·琼斯	体育	1
弗兰克·布鲁	体育	1
弗兰克·布鲁	物理	1
安东尼·安吉尔	数学1	

第23.5节：HAVING

因为WHERE子句在GROUP BY之前被计算，所以不能使用WHERE来缩减分组结果（通常是聚合函数，如COUNT(*)）。为满足这一需求，可以使用HAVING子句。

例如，使用以下数据：

```
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' )
```

如果我们想要获取每个人下的订单数量，可以使用

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

并得到

Name	Orders
马太福音2	
约翰福音5	
路加福音4	

但是，如果我们想限制为下过超过两个订单的个人，我们可以添加一个HAVING子句。

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



```
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
```

将会得到

Name	Orders
John	5
Luke	4

注意，和`GROUP BY`类似，放在`HAVING`中的列必须与`SELECT`语句中的对应列完全匹配。如果在上面的例子中我们改为说

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

那么我们的HAVING子句必须写成

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



```
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
```

第24章：ORDER BY

第24.1节：简单的ORDER BY子句

使用员工表，下面是一个示例，返回Id、FName和LName列，按LName升序排列：

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

返回结果：

Id	FName	LName
2	John	Johnson
1	James	Smith
4	Johnathon	Smith
3	迈克尔	威廉姆斯

要按降序排序，请在字段参数后添加 DESC 关键字，例如，按姓氏降序排序的相同查询是：

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

第24.2节：按多个字段排序

可以为ORDER BY子句指定多个字段，排序顺序可以是升序（ASC）或降序（DESC）。

例如，使用 http://stackoverflow.com/documentation/sql/280/example-databases/1207/item-sales-table#t=201607211314066434211 表，我们可以返回一个按 SaleDate 升序、Quantity 降序排序的查询。

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

注意，ASC关键字是可选的，默认情况下结果按指定字段的升序排序。

第24.3节：自定义排序

如果您想按某列进行排序，但不使用字母顺序或数字顺序，可以使用`case`来指定您想要的排序顺序。

按组排序返回：

组	计数
未退休	6
已退休	4
总计	10

按case group排序，当'Total' 时返回1，当'Retired' 时返回2，否则返回3结束返回：

组	计数

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

总计 10
已退休 4
未退休 6

第24.4节：带复杂逻辑的ORDER BY

如果我们想对每个组的数据进行不同的排序，可以在ORDER BY中添加CASE语法。在这个例子中，我们希望对第1部门的员工按姓氏排序，对第2部门的员工按薪水排序。

编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
1	James	Smith	1234567890	空值	1	1000	01-01-2002
2	John	约翰逊	2468101214	1	1	400	23-03-2005
3	迈克尔	威廉姆斯	1357911131	1	2	600	12-05-2009
4	Johnathon	Smith	1212121212	2	1	500	24-07-2016
5	山姆	萨克森	1372141312	2	2	400	25-03-2015

以下查询将提供所需的结果：

```
SELECT Id, FName, LName, Salary FROM Employees
ORDER BY Case When DepartmentId = 1 then LName else Salary end
```


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
```

第25章：STUFF函数

参数	详细信息
character_expression	你数据中现有的字符串
start_position	在character_expression中的位置，删除length个字符，然后插入replacement_string
长度	从character_expression中删除的字符数
替换字符串	要插入到character_expression中的字符序列

第25.1节：使用FOR XML连接多行的值

FOR XML函数的一个常见用途是连接多行的值。

以下是使用Customers表的示例：

```
SELECT
    STUFF( (SELECT ';' + Email
        FROM Customers
        where (Email is not null and Email <> '')
        ORDER BY Email ASC
        FOR XML PATH('')),
    1, 1, '')
```

在上面的示例中，FOR XML PATH('')) 用于连接电子邮件地址，使用 ; 作为分隔符字符。同时，STUFF 的作用是去除连接字符串开头的 ; 。STUFF 还隐式地将连接的字符串从 XML 转换为 varchar。

注意：上述示例的结果将是 XML 编码的，意味着它会将 < 字符替换为 < 等。如果你不想要这种情况，可以将 FOR XML PATH('')) 改为 FOR XML PATH, TYPE).value('.[1]','varchar(MAX)')，例如：

```
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, '')
```

这可以用来实现类似 MySQL 中 GROUP_CONCAT 或 PostgreSQL 9.0+ 中 string_agg 的效果，尽管我们使用子查询而不是 GROUP BY 聚合。（作为替代方案，如果你想要更接近 GROUP_CONCAT 功能的实现，可以安装用户定义的聚合函数，例如 this one）。

第 25.2 节：使用 STUFF() 进行基本字符替换

STUFF() 函数通过先删除指定数量的字符，然后插入一个字符串到另一个字符串中。下面的示例删除了 "Svr" 并替换为 "Server"。这是通过指定替换的 start_position 和 length 实现的。

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

执行此示例将返回SQL Server文档而不是SQL Svr 文档。

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.

第25.3节：STUFF()函数的基本示例

STUFF(原始表达式，起始位置，长度，替换表达式)STUFF()函数在指定

的起始位置插入替换表达式，同时删除使用长度参数指定的字符。

```
选择 名, 姓,邮箱, STUFF(邮箱, 2, 3, *******) 作为 替换邮箱 从 员工
```

执行此示例将返回下列表格

名	姓	邮箱	替换邮箱
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

第25.4节：SQL Server中逗号分隔的stu

使用FOR XML PATH和STUFF将多行连接成一行：

```
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;
```

第25.5节：获取以逗号分隔的列名（非列表）

```
/*
结果可用于快速在插入/更新中使用列。
适用于表和视图。

示例：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
内连接
    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, '')
```


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
```


GO

第26章：SQL Server中的JSON

参数	详细信息
表达式	通常是包含JSON文本的变量名或列名。
路径	指定要更新属性的JSON路径表达式。路径的语法如下： [append] [lax | strict] $.<json path>
jsonExpression	是包含 JSON 文本的 Unicode 字符表达式。

第 26.1 节：使用计算列对 JSON 属性建立索引

在 SQL Server 中存储 JSON 文档时，我们需要能够高效地根据 JSON 文档的属性过滤和排序查询结果。

```
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
```

基于上述表，如果我们想查找 Name = 'Adam' 的行，可以执行以下查询。

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

然而，这将要求 SQL Server 执行全表扫描，对于大表来说效率不高。

为了加快速度，我们想添加一个索引，但不能直接引用JSON文档中的属性。解决方案是在JSON路径$.Name上添加一个计算列，然后在计算列上添加索引。

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

CREATE INDEX idx_name
ON JsonTable(vName)
```

现在当我们执行相同的查询时，SQL Server不会进行全表扫描，而是使用索引在非聚集索引中查找满足指定条件的行。

注意：为了让SQL Server使用索引，必须用与查询中计划使用的表达式相同的表达式创建计算列——在本例中为JSON_VALUE(jsonInfo, '$.Name')，当然也可以使用计算列的名称vName

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`

第26.2节：使用CROSS APPLY OPENJSON连接父子JSON实体

连接父对象及其子实体，例如我们想要一个包含每个人及其爱好的关系表

```
DECLARE @json nvarchar(1000) =
N'[
    {
        "id":1,
"user":{"name":"约翰"},
        "hobbies":[
{"name": "阅读"},
          {"name": "冲浪"}
        ]
    },
    {
        "id":2,
"user":{"name":"简"},
        "hobbies":[
{"name": "编程"},
          {"name": "跑步"}
]
   }
 ]'
```

查询

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

或者，这个查询也可以使用 WITH 子句来编写。

```
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'
)
```

结果

Id	PersonName	Hobby
1	John	阅读
1	John	冲浪
2	简	编程
2	简	跑步

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

第26.3节：使用FOR JSON将查询结果格式化为JSON

输入表数据（人员表）

编号	姓名	年龄
1	约翰	23
2	简	31

查询

```
SELECT 编号, 姓名, 年龄
FROM 人员
FOR JSON PATH
```

结果

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

第26.4节：解析JSON文本

JSON_VALUE 和 JSON_QUERY 函数解析JSON文本并返回JSON文本中路径上的标量值或对象/数组。

```
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
```

结果

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

第26.5节：使用 FOR JSON 将一行表格格式化为单个 JSON 对象

FOR JSON 子句中的 WITHOUT_ARRAY_WRAPPER 选项将从 JSON 输出中移除数组括号。如果查询只返回单行数据，这非常有用。

> 注意：如果返回多行数据，此选项将生成无效的 JSON 输出。

输入表数据（人员表）

编号	姓名	年龄
1	约翰	23
2	简	31

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

查询

```
SELECT 编号, 姓名, 年龄
FROM 人员
WHERE Id = 1
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER
```

结果

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

第26.6节：使用 OPENJSON 函数解析 JSON 文本

OPENJSON 函数解析 JSON 文本并返回多个输出。应返回的值通过 WITH 子句中定义的路径指定。如果某列未指定路径，则使用列名作为路径。该函数将返回值转换为 WITH 子句中定义的 SQL 类型。如果某个对象/数组应被返回，必须在列定义中指定 AS JSON 选项。

```
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]')
```

结果

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

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#

第27章：OPENJSON

第27.1节：将JSON数组转换为行集合

OPENJSON函数解析JSON对象集合，并将JSON文本中的值作为行集合返回。

```
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
  )
```

在WITH子句中指定了OPENJSON函数的返回模式。JSON对象中的键通过列名获取。如果JSON中的某些键未在WITH子句中指定（例如本例中的Price），则会被忽略。
值会自动转换为指定的类型。

编号	日期	客户数量	
SO43659	2011-05-31T00:00:00	MSFT	1
SO43661	2011-06-01T00:00:00	Nokia	3

第27.2节：从JSON文本获取键值对

OPENJSON函数解析JSON文本并返回JSON第一层的所有键值对：

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

键	值	类型
姓名	乔	1
年龄	27	2
技能["C#","SQL"]		4

列类型描述值的类型，即 null(0)、字符串(1)、数字(2)、布尔值(3)、数组(4) 和对象(5)。

第27.3节：将嵌套的JSON字段转换为行集合

OPENJSON函数解析JSON对象集合，并将JSON文本中的值作为行集合返回。如果输入对象中的值是嵌套的，可以在WITH子句的每一列中指定额外的映射参数：

```
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}}
]'
```


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 (
编号   varchar(200) '$.data.num',
          日期     datetime '$.data.date',
        客户 varchar(200) '$.info.customer',
        数量 int '$.info.qty',
  )
```

在WITH子句中指定了OPENJSON函数的返回模式。类型指定后是JSON节点的路径，返回值应在该路径中找到。通过这些路径获取JSON对象中的键。值会自动转换为指定的类型。

编号	日期	客户	数量
SO43659	2011-05-31T00:00:00	MSFT	1
SO43661	2011-06-01T00:00:00	Nokia	3

第27.4节：提取内部JSON子对象

OPENJSON可以提取JSON文本中JSON对象的片段。在引用JSON子对象的列定义中，将类型设置为nvarchar(max)，并使用AS JSON选项：

```
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列将映射到"Info"对象。结果将是：

编号	日期	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}

第27.5节：处理嵌套的JSON子数组

JSON 可能具有包含内部数组的复杂结构。在此示例中，我们有一个订单数组，里面嵌套了子数组 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}]}
]'
```

我们可以使用 OPENJSON 解析根级属性，它将返回 Items 数组作为 JSON 片段。然后我们可以


```
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

再次对 Items 数组应用 OPENJSON 并打开内部 JSON 表。第一级表和内部表将像标准表之间的 JOIN 一样“连接”：

```
SELECT    *
FROM
OPENJSON (@json)
    WITH (  Number varchar(200), Date datetime,
            Items nvarchar(max) AS JSON )
        CROSS APPLY
OPENJSON (Items)
            WITH ( Price float, Quantity int)
```

结果：

编号	日期	项目	价格	数量
销售订单43659	2011-05-31 00:00:00.000	[{"Price":11.99,"Quantity":1},{"Price":12.99,"Quantity":5}]	11.99	1
销售订单43659	2011-05-31 00:00:00.000	[{"Price":11.99,"Quantity":1},{"Price":12.99,"Quantity":5}]	12.99	5
销售订单43661	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
销售订单43661	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
销售订单43661	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

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

第28章：FOR JSON

第28.1节：FOR JSON PATH

将SELECT查询的结果格式化为JSON文本。FOR JSON PATH子句添加在查询之后：

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

列名将作为JSON中的键，单元格的值将生成JSON值。查询结果
将是一个JSON对象数组：

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

principal_id列中的NULL值将被忽略（不会生成）。

第28.2节：带列别名的FOR JSON PATH

FOR JSON PATH允许您使用列别名控制输出JSON的格式：

```
SELECT top 3 object_id as id, name as [data.name], type as [data.type]
FROM sys.objects
FOR JSON PATH
```

列别名将用作键名。以点分隔的列别名（data.name 和 data.type）将生成嵌套对象。如果两个列在点表示法中具有相同的前缀，它们将被分组到单个对象中（本例中的 data）：

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

第28.3节：FOR JSON子句无数组包装器（输出为单个对象）

WITHOUT_ARRAY_WRAPPER选项允许你生成单个对象而不是数组。如果你确定只会返回单行/单个对象，请使用此选项：

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

此情况下将返回单个对象：

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


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 "}
```

第28.4节：INCLUDE_NULL_VALUES

FOR JSON子句会忽略单元格中的NULL值。如果你想为包含NULL值的单元格生成"key": null对，请在查询中添加INCLUDE_NULL_VALUES选项：

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

principal_id 列中的 NULL 值将被生成：

```
[
  {"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}
]
```

第28.5节：使用 ROOT 对象包装结果

将返回的 JSON 数组包装在带有指定键的额外根对象中：

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

查询结果将是包装对象内的 JSON 对象数组：

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

第28.6节：FOR JSON AUTO

自动将第二个表的值嵌套为 JSON 对象的嵌套子数组：

```
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
```

查询结果将是 JSON 对象数组：

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


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"}
     ]
  }
]
```

第28.7节：创建自定义嵌套JSON结构

如果您需要一些复杂的JSON结构，无法使用FOR JSON PATH或FOR JSON AUTO创建，您可以通过将FOR JSON子查询作为列表达式来自定义JSON输出：

```
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
```

每个子查询将生成JSON结果，并包含在主JSON内容中。


```
    "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.

第29章：带有JSON数据的查询

第29.1节：在查询中使用来自JSON的值

JSON_VALUE 函数使您能够从 JSON 文本中提取第二个参数指定路径的数据，
并在查询的任何部分使用该值：

```
select ProductID, Name, Color, Size, Price, JSON_VALUE(Data, '$.Type') as Type
from Product
where JSON_VALUE(Data, '$.Type') = 'part'
```

第29.2节：在报表中使用 JSON 值

一旦从 JSON 文本中提取出 JSON 值，您可以在查询的任何部分使用它们。您可以基于 JSON 数据创建某种报表，进行分组聚合等操作：

```
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
```

第29.3节：从查询结果中过滤错误的 JSON 文本

如果某些 JSON 文本可能格式不正确，您可以使用 ISJSON 函数将这些条目从查询中移除。

```
select ProductID, Name, Color, Size, Price, JSON_VALUE(Data, '$.Type') as Type
from Product
where JSON_VALUE(Data, '$.Type') = 'part'
and ISJSON(Data) > 0
```

第29.4节：更新JSON列中的值

JSON_MODIFY函数可用于更新某路径上的值。您可以使用此函数在UPDATE语句中修改JSON单元格的原始值：

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

JSON_MODIFY函数将更新或创建Price键（如果不存在）。如果新值为NULL，则该键将被移除。JSON_MODIFY函数会将新值视为字符串（转义特殊字符，用双引号包裹以创建正确的JSON字符串）。如果您的新值是JSON片段，应使用JSON_QUERY函数包裹：

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

不带第二个参数的JSON_QUERY函数表现得像“转换为JSON”。由于JSON_QUERY的结果是有效的JSON片段（对象或数组），JSON_MODIFY在修改输入JSON时不会对该值进行转义。

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.

第29.5节：向JSON数组追加新值

JSON_MODIFY函数可用于向JSON中的某个数组追加新值：

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

新值将被追加到数组末尾，或者创建一个包含值["sales"]的新数组。
JSON_MODIFY函数会将新值视为字符串（转义特殊字符，用双引号包裹以创建正确的JSON字符串）。如果您的新值是JSON片段，应使用JSON_QUERY函数包裹：

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

不带第二个参数的JSON_QUERY函数表现得像"转换为JSON"。由于JSON_QUERY的结果是有效的JSON片段（对象或数组），JSON_MODIFY在修改输入JSON时不会对该值进行转义。

第29.6节：使用内嵌JSON集合的JOIN表

如果您有一个格式为JSON集合的"子表"，并以JSON列的形式存储在行内，您可以展开此集合，将其转换为表并与父行连接。您应使用CROSS APPLY代替标准的JOIN操作符。在此示例中，产品部件格式为JSON对象集合，存储在Data列中：

```
select ProductID, Name, Size, Price, Quantity, PartName, Code
from Product
    CROSS APPLY OPENJSON(Data, '$.Parts') WITH (PartName varchar(20), Code varchar(5))
```

查询结果等同于Product表和Part表之间的连接。

第29.7节：查找包含JSON数组中某值的行

在此示例中，Tags数组可能包含各种关键字，如["promo", "sales"]，因此我们可以打开此数组并过滤值：

```
select ProductID, Name, Color, Size, Price, Quantity
from Product
    CROSS APPLY OPENJSON(Data, '$.Tags')
where value = 'sales'
```

OPENJSON 会打开标签的内部集合并将其作为表返回。然后我们可以根据表中的某个值过滤结果。

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.

第30章：在SQL表中存储JSON

第30.1节：作为文本列存储的JSON

JSON是文本格式，因此它存储在标准的NVARCHAR列中。NoSQL集合相当于两列表的键值表：

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

使用 `nvarchar(max)` 是因为你不确定JSON文档的大小。 `nvarchar(4000)` 和 `varchar(8000)` 性能更好，但大小限制为8KB。

第30.2节：使用

ISJSON确保JSON格式正确

由于JSON存储为文本列，你可能想确保它格式正确。你可以在JSON列上添加CHECK约束，检查文本是否为格式正确的JSON：

```
CREATE TABLE ProductCollection (
  Id int identity primary key,
  Data nvarchar(max)
       约束[数据应格式化为JSON]
       检查(ISJSON(数据) > 0)
)
```

如果您已经有一个表，可以使用 ALTER TABLE 语句添加检查约束：

```
ALTER TABLE ProductCollection
    ADD CONSTRAINT [数据应格式化为JSON]
        CHECK (ISJSON(Data) > 0)
```

第30.3节：将JSON文本中的值作为计算列公开

您可以将JSON列中的值作为计算列公开：

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

如果添加了PERSISTED计算列，JSON文本中的值将被物化到该列中。这样，您的查询可以更快地读取JSON文本中的值，因为不需要解析。每当该行中的JSON发生变化时，值将被重新计算。

第30.4节：在JSON路径上添加索引

通过JSON列中的某个值过滤或排序数据的查询通常会使用全表扫描。

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'
```

为了优化这类查询，您可以添加一个非持久化计算列，该列暴露用于过滤或排序的 JSON 表达式（在此示例中为 JSON_VALUE(Data, '$.Color')），并在该列上创建索引：

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

CREATE INDEX idx_JsonColor
ON ProductCollection(vColor)
```

查询将使用索引，而不是简单的表扫描。

第30.5节：存储在内存表中的 JSON

如果您可以使用内存优化表，则可以将 JSON 作为文本存储：

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

内存中 JSON 的优点：

- JSON 数据始终驻留在内存中，因此无需磁盘访问
- 处理 JSON 时没有锁和闩锁


```
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

第31章：修改JSON文本

第31.1节：修改指定路径上的JSON文本中的值

JSON_MODIFY函数使用JSON文本作为输入参数，并使用第三个参数修改指定路径上的值：

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

结果是，我们将得到新的JSON文本，其中"Price":39.99，其他值不会被更改。如果指定路径上的对象不存在，JSON_MODIFY将插入键值对。

要删除键值对，请将新值设为NULL：

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

JSON_MODIFY默认情况下如果没有值会删除键，因此你可以用它来删除键。

第31.2节：向JSON数组追加标量值

JSON_MODIFY 有一个“append”模式，可以将值追加到数组中。

```
declare @json nvarchar(4000) = N'{"Id":1,"Name":"玩具车","Tags":["玩具","游戏"]}'
set @json = JSON_MODIFY(@json, 'append $.Tags', '销售')
print @json -- 输出: {"Id":1,"Name":"玩具车","Tags":["玩具","游戏","销售"]}
```

如果指定路径上的数组不存在，JSON_MODIFY(append) 将创建一个包含单个元素的新数组：

```
declare @json nvarchar(4000) = N'{"Id":1,"Name":"玩具车","Price":34.99}'
set @json = JSON_MODIFY(@json, 'append $.Tags', '销售')
print @json -- 输出 {"Id":1,"Name":"玩具车","Tags":["销售"]}
```

第31.3节：在JSON文本中插入新的JSON对象

JSON_MODIFY 函数允许你将JSON对象插入到JSON文本中：

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

由于第三个参数是文本，您需要使用 JSON_QUERY 函数将其“转换”为 JSON。没有这个“转换”，JSON_MODIFY 会将第三个参数视为普通文本，并在插入为字符串值之前转义字符。没有 JSON_QUERY，结果将是：

```
{"Id":1,"Name":"玩具车","Price":'{\"Min\":34.99,\"Recommended\":45.49}'}
```

如果该对象不存在，JSON_MODIFY 会插入该对象；如果第三个参数的值为 NULL，则会删除该对象。

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.

第 31.4 节：插入使用 FOR

JSON 查询生成的新 JSON 数组

您可以使用带有 FOR JSON 子句的标准 SELECT 查询生成 JSON 对象，并将其作为第三个参数插入到 JSON 文本中：

```
声明 @json nvarchar(4000) = N'{"Id":17,"Name":"第一次世界大战"}'
设置 @json = JSON_MODIFY(@json, '$.tables',
                        (选择 name 从 sys.tables FOR JSON PATH) )
打印 @json
```

```
(影响 1 行)
```

```
{"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 会识别带有 FOR JSON 子句的 SELECT 查询生成有效的 JSON 数组，并将其直接插入到 JSON 文本中。

> 您可以在 SELECT 查询中使用所有 FOR JSON 选项，除了 WITHOUT_ARRAY_WRAPPER，该选项会生成单个对象而非 JSON 数组。请参见本主题中的其他示例，了解如何插入单个 JSON 对象。

第 31.5 节：插入使用 FOR

JSON 子句生成的单个 JSON 对象

您可以使用带有 FOR JSON 子句和 WITHOUT_ARRAY_WRAPPER

选项的标准 SELECT 查询生成 JSON 对象，并将其作为第三个参数插入到 JSON 文本中：

```
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
```

(影响 1 行)

```
        {"Id":17,"Name":"WWI","table":{"name":"Colors","create_date":"2016-06-02T10:04:03.280","schema_id":
13}}
```


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}}
```

使用 WITHOUT_ARRAY_WRAPPER 选项的 FOR JSON 可能会生成无效的 JSON 文本，如果 SELECT 查询返回多个结果（此时应使用 TOP 1 或按主键过滤）。因此，如果不使用 JSON_QUERY 函数包装，JSON_MODIFY 会假定返回结果只是普通文本，并像处理其他文本一样对其进行转义。

> 你应该使用 JSON_QUERY 函数包装FOR JSON，WITHOUT_ARRAY_WRAPPER查询，以便将结果转换为 JSON。

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.

第32章：FOR XML PATH

第32.1节：使用 FOR XML PATH 连接值

FOR XML PATH 可用于将值连接成字符串。下面的示例将值连接成一个csv字符串：

```
声明 @DataSource 表
(
    [rowID] TINYINT
   ,[FirstName] NVARCHAR(32)
);

向 @DataSource ([rowID], [FirstName])
插入值 (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
    ,''
);
```

几点重要说明：

- 可以使用ORDER BY子句以首选方式对值进行排序
- 如果使用较长的值作为连接分隔符，STUFF函数的参数也必须相应更改；

```
SELECT STUFF
(
    (
        SELECT '---' + [FirstName]
        FROM @DataSource
        ORDER BY [rowID] DESC
        FOR XML PATH(''), TYPE
    ).value('.', 'NVARCHAR(MAX)')
    ,1
    ,3 -- “3”也可以用：LEN('---') 来表示以增加清晰度
    ,''
);
```

- 当使用TYPE选项和.value函数时，连接操作适用于NVARCHAR(MAX)字符串

第32.2节：指定命名空间

版本 ≥ SQL Server 2008

```
WITH XMLNAMESPACES (
    DEFAULT 'http://www.w3.org/2000/svg',
    'http://www.w3.org/1999/xlink' 作为 xlink
```


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' 作为 'image/@xlink:href',
    '50px' 作为 'image/@width',
    '50px' 作为 '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>
```

第32.3节：使用XPath表达式指定结构

```
SELECT
    'XPath示例' 作为 'head/title',
    '此示例演示 ' 作为 'body/p',
    'https://www.w3.org/TR/xpath/' 作为 'body/p/a/@href',
    'XPath表达式' 作为 'body/p/a'
FOR XML PATH('html')
```

```
<html>
    <head>
        <title>XPath 示例</title>
    </head>
    <body>
        <p>此示例演示了<a href="https://www.w3.org/TR/xpath/">XPath
表达式</a></p>
    </body>
</html>
```

在FOR XML PATH中，没有名称的列会变成文本节点。NULL或''因此会变成空文本节点。
注意：您可以使用AS*将命名列转换为未命名列

```
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">你好，世界</FLD>
    <FLD NAME="QTY">2</FLD>
  </LIN>
</row>
```



```
)
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>
```

使用（空）文本节点有助于将之前输出的节点与下一个节点分开，这样 SQL Server就知道为下一列开始一个新元素。否则，当属性已经存在于它认为的“当前”元素上时，它会感到困惑。

例如，如果在 `SELECT` 语句中的元素和属性之间没有空字符串，SQL Server 会报错：

> 属性中心列 'FLD/@NAME' 不能出现在 XML
>
> 层次结构中非属性中心的兄弟节点之后，使用 FOR XML PATH 时会出现此错误。

还要注意，这个示例还将 XML 包裹在一个名为 row 的根元素中，由 ROOT('row') 指定

第 32.4 节：你好，世界 XML

```
SELECT '你好，世界' FOR XML PATH('example')
```

```
<example>你好，世界</example>
```


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>
```

第33章：连接

在结构化查询语言（SQL）中，JOIN 是一种在单个查询中连接两个数据表的方法，允许数据库一次性返回包含两个表数据的集合，或使用一个表中的数据作为第二个表的筛选条件。ANSI SQL 标准中定义了几种类型的 JOIN。

第33.1节：内连接

内连接只返回基于一个或多个条件（使用ON关键字指定）在两个表中匹配/存在的记录/行。它是最常见的连接类型。内连接的一般语法为：

```
SELECT *
FROM table_1
INNER JOIN table_2
  ON table_1.column_name = table_2.column_name
```

它也可以简化为仅使用JOIN：

```
SELECT *
FROM table_1
JOIN table_2
  ON table_1.column_name = table_2.column_name
```

示例

```
/* 示例数据。 */
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 ('狗');
INSERT INTO @Animal (Animal) VALUES ('猫');
INSERT INTO @Animal (Animal) VALUES ('大象');

INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (1, '吠叫');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (2, '喵喵');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (3, '象鸣');
/* 样本数据已准备好。 */

SELECT
    *
FROM
    @Animal
    JOIN @AnimalSound
        ON @Animal.AnimalId = @AnimalSound.AnimalId;
```

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           狗                  1             1           吠叫
2           猫                  2             2           喵喵
3           大象                3             3           象鸣
```


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
```

使用内连接和左外连接（替代 Not exists）

此查询将返回表1中的数据，其中字段与表2通过键匹配，且在与表2比较时数据不在表1中，满足条件和键。

```
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;
```

第33.2节：外连接

左外连接

LEFT JOIN 返回左表中的所有行，当 ON 子句条件满足时，与右表中的行匹配。未满足 ON 子句的行在右表的所有列中均为 NULL。 LEFT JOIN 的语法是：

```
SELECT * FROM table_1 AS t1
LEFT JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

右外连接

RIGHT JOIN 返回右表中的所有行，并将其与满足 ON 子句条件的左表行匹配。对于不满足 ON 子句的行，左表的所有列均为 NULL。 RIGHT JOIN 的语法是：

```
SELECT * FROM table_1 AS t1
RIGHT JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

全外连接

FULL JOIN 结合了 LEFT JOIN 和 RIGHT JOIN。无论 ON 子句的条件是否满足，都会返回两个表中的所有行。不满足 ON 子句的行，其对方表的所有列均为 NULL（即左表中的行，其右表所有列为 NULL，反之亦然）。 FULL JOIN 的语法是：

```
SELECT * FROM table_1 AS t1
FULL JOIN table_2 AS t2 ON t1.ID_Column = t2.ID_Column
```

示例

```
/* 示例测试数据。 */
DECLARE @Animal table (
    AnimalId Int IDENTITY,
    Animal Varchar(20)
);

DECLARE @AnimalSound table (
    AnimalSoundId Int IDENTITY,
    AnimalId Int,
Sound Varchar(20)
```


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 ('狗');
INSERT INTO @Animal (Animal) VALUES ('猫');
INSERT INTO @Animal (Animal) VALUES ('大象');
INSERT INTO @Animal (Animal) VALUES ('青蛙');

INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (1, '吠叫');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (2, '喵喵');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (3, '象鸣');
INSERT INTO @AnimalSound (AnimalId, Sound) VALUES (5, '咆哮');
/* 示例数据已准备好。 */
```

左外连接

```
SELECT *
FROM @Animal As t1
LEFT JOIN @AnimalSound As t2 ON t1.AnimalId = t2.AnimalId;
```

LEFT JOIN 的结果

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           狗                  1             1           吠叫
2           猫                  2             2           喵喵声
3           大象                3             3           喇叭声
4           青蛙                NULL          NULL        NULL
```

RIGHT OUTER JOIN

```
SELECT *
FROM @Animal As t1
RIGHT JOIN @AnimalSound As t2 ON t1.AnimalId = t2.AnimalId;
```

RIGHT JOIN 的结果

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           狗                  1             1           吠叫
2           猫                  2             2           喵喵声
3           大象                3             3           喇叭声
NULL        NULL                4             5           吼叫声
```

FULL OUTER JOIN

```
SELECT *
FROM @Animal As t1
FULL JOIN @AnimalSound As t2 ON t1.AnimalId = t2.AnimalId;
```

FULL JOIN 的结果

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           狗                  1             1           吠叫
2           猫                  2             2           喵喵声
3           大象                3             3           喇叭声
```



```
);

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           青蛙                NULL        NULL        NULL
NULL        NULL                4           5           吼叫声
```

第33.3节：在更新中使用连接

连接也可以用于UPDATE语句：

```
创建表 Users (
    UserId int 非空,
    AccountId int 非空,
    RealName nvarchar(200) 非空
)

创建表 Preferences (
    UserId int 非空,
    SomeSetting bit 非空
)
```

按如下方式根据Users表上的谓词过滤，更新Preferences表的SomeSetting列：

```
更新 p
设置 p.SomeSetting = 1
来自 Users u
JOIN Preferences p ON u.UserId = p.UserId
WHERE u.AccountId = 1234
```

p 是语句中 FROM 子句定义的 Preferences 的别名。只有与 Users 表中匹配的 AccountId 的行才会被更新。

使用左外连接语句进行更新

```
更新 t
设置 t.Column1=100
从 Table1 t 左连接 Table12 t2
连接条件 t2.ID=t.ID
```

使用内连接和聚合函数更新表

```
更新 t1
设置 t1.field1 = t2.field2Sum
从 table1 t1
内连接 (选择 field3, 求和(field2) 作为 field2Sum
从 table2
按 field3分组) 作为 t2
连接条件 t2.field3 = t1.field3
```

第33.4节：对子查询进行连接

当你想从子表/明细表获取聚合数据（例如计数、平均值、最大值或最小值）并将其与父表/主表的记录一起显示时，通常会使用子查询连接。例如，你可能想根据日期或编号检索排名第一的子行，或者你想要所有子行的计数或平均值。

此示例使用别名，使得在涉及多个表时查询更易读。在本例中，我们检索父表采购订单的所有行，并仅从子表采购订单明细中检索最后（或最新）的子行。此示例假设子表使用递增的数字编号。


```
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
```

第33.5节：交叉连接

交叉连接是笛卡尔连接，意味着两个表的笛卡尔积。此连接不需要任何条件来连接两个表。左表中的每一行都会与右表中的每一行连接。交叉连接的语法如下：

```
SELECT * FROM table_1
CROSS JOIN table_2
```

示例：

```
/* 示例数据。 */
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;
```

结果：

```
AnimalId    Animal              AnimalSoundId AnimalId    Sound
----------- ------------------- ------------- ----------- --------------------
1           狗                   1             1           吠叫
2           猫                  1             1           叫声
3           大象                  1             1           叫声
1           狗                  2             2           喵喵声
2           猫                  2             2           喵喵声
3           大象                   2             2           喵喵声
1           狗                  3             3           喇叭声
2           猫                  3             3           喇叭声
```



```
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          大象                3           3          喇叭声
```

注意，CROSS JOIN 还有其他应用方式。这是一种"旧式"连接（自 ANSI SQL-92 起已弃用），没有条件，结果是交叉/笛卡尔积连接：

```
SELECT *
FROM @Animal, @AnimalSound;
```

该语法也因"始终为真"的连接条件而有效，但不推荐使用，应避免，建议使用明确的CROSS JOIN语法，以提高可读性。

```
SELECT *
FROM
    @Animal
    JOIN @AnimalSound
        ON 1=1
```

第33.6节：自连接

表可以与自身连接，这称为自连接，将表中的记录与同一表中的其他记录组合。自连接通常用于定义表中列层级的查询。

考虑一个名为Employees的表中的示例数据：

ID	名称	上司_ID
1	鲍勃	3
2	吉姆	1
3	萨姆	2

每个员工的Boss_ID映射到另一个员工的ID。要检索带有各自上司姓名的员工列表，可以使用此映射将表自身连接。请注意，以这种方式连接表时，需要在第二次引用该表时使用别名（此处为Bosses），以区分原始表。

```
SELECT Employees.Name,
    Bosses.Name AS Boss
FROM Employees
INNER JOIN Employees AS Bosses
    ON Employees.Boss_ID = Bosses.ID
```

执行此查询将输出以下结果：

姓名	上司
鲍勃	萨姆
吉姆	鲍勃
萨姆	吉姆

第33.7节：意外将外连接变为内连接

外连接返回一个或两个表中的所有行，以及匹配的行。

```
表 人员
人员ID 名字
       1 爱丽丝
```



```
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鲍勃
       3伊芙

表 分数
人员ID 科目 分数
       1数学    100
       2数学     54
       2科学  98
```

左连接表：

```
选择 * 从 People a
左 连接 Scores b
在 a.PersonID = b.PersonID
```

返回结果：

```
PersonID 名字 PersonID 科目 分数
       1爱丽丝           1 数学    100
       2鲍勃             2 数学     54
       2鲍勃             2 科学  98
       3伊芙          NULL NULL   NULL
```

如果你想返回所有人及其相关的数学成绩，一个常见错误是写成：

```
选择 * 从 People a
左 连接 Scores b
在 a.PersonID = b.PersonID
哪里 科目 = '数学'
```

这会将伊芙从结果中移除，同时也会移除鲍勃的科学成绩，因为她的科目是NULL。

正确的语法是在保留People表中所有人的同时，去除非数学记录，应为：

```
选择 * 从 People a
左 连接 Scores b
在 a.PersonID = b.PersonID
并且 b.Subject = '数学'
```

第33.8节：使用Join进行删除

Join也可以用于DELETE语句。给定如下模式：

```
创建表 Users (
    UserId int 非空,
    AccountId int 非空,
    RealName nvarchar(200) 非空
)

创建表 Preferences (
    UserId int 非空,
    SomeSetting bit 非空
)
```

我们可以从Preferences表中删除行，通过Users表上的谓词进行过滤，示例如下：

```
DELETE p
```



```
       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
```

这里p是语句中FROM子句定义的Preferences的别名，我们只删除在Users表中具有匹配AccountId的行。


```
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.

第34章：cross apply

第34.1节：将表行与从单元格动态生成的行连接

CROSS APPLY使您能够将表中的行与某些表值函数返回的动态生成的行“连接”。

假设你有一个Company表，其中有一列包含产品数组（ProductList列），还有一个函数可以解析这些值并返回一组产品。你可以从Company表中选择所有行，对ProductList列应用该函数，并将生成的结果与父Company行“连接”：

```
SELECT *
FROM Companies c
     CROSS APPLY dbo.GetProductList( c.ProductList ) p
```

对于每一行，ProductList单元格的值将被传递给该函数，函数将返回这些产品作为一组行，可以与父行连接。

第34.2节：将表行与存储在单元格中的JSON数组连接

CROSS APPLY使你能够将表中的行与存储在列中的JSON对象集合“连接”。

假设你有一个Company表，其中有一列包含格式为JSON数组的产品数组（ProductList列）。OPENJSON表值函数可以解析这些值并返回产品集合。你可以从Company表中选择所有行，使用OPENJSON解析JSON产品，并将生成的结果与父Company行“连接”：

```
SELECT *
FROM Companies c
     CROSS APPLY OPENJSON( c.ProductList )
                 WITH ( Id int, Title nvarchar(30), Price money)
```

对于每一行，ProductList单元格的值将被传递给OPENJSON函数，OPENJSON函数会将JSON对象转换为WITH子句中定义的模式的行。

第34.3节：按数组值过滤行

如果你将标签列表以逗号分隔的值存储在一行中，STRING_SPLIT 函数可以将标签列表转换为值的表格。CROSS APPLY 使你能够将由STRING_SPLIT函数解析的值与父行“连接”。

假设你有一个产品表，其中有一列包含以逗号分隔的标签数组（例如 promo,sales,new）。STRING_SPLIT 和 CROSS APPLY 使你能够将产品行与其标签连接，从而可以按标签筛选产品：

```
SELECT *
FROM Products p
     CROSS APPLY STRING_SPLIT( p.Tags, ',' ) tags
WHERE tags.value = 'promo'
```

对于每一行，*Tags* 单元格的值将传递给 STRING_SPLIT 函数，该函数返回标签值。然后你可以根据这些值筛选行。

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.

注意：STRING_SPLIT 函数在 SQL Server 2016 之前不可用

Note: *STRING_SPLIT* function is not available before **SQL Server 2016**

第35章：计算列

第35.1节：列由表达式计算得出

计算列是由一个表达式计算得出的，该表达式可以使用同一表中的其他列。表达式可以是非计算列名、常量、函数，以及由一个或多个运算符连接的这些元素的任意组合。

创建带有计算列的表

```
创建表 净利润
(
员工薪资          int,
    分发奖金          int,
    业务运营成本       int,
    业务维护成本       int,
    业务收入          int,
    业务净收入
                作为 业务收入 - (员工薪资          +
                                分发奖金          +
                                业务运营成本      +
                                业务维护成本      )

)
```

该值在插入其他值时自动计算并存储在计算列中。

```
插入到 净利润
    (员工薪资,
    分发奖金,
    业务运营成本,
    业务维护成本,
    业务收入)
值
    (0000000,
     10000,
     0000000,
     50000,
     2500000)
```

第35.2节：我们通常在对数表中使用的简单示例

```
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())))
```

这将给出运行时间的分钟差，非常实用。

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..

第36章：公共表表达式

第36.1节：使用CTE生成日期表

```
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)
```

此示例返回一个单列表，包含从@startdate变量指定的日期开始，接下来的@numberDays天数的日期。

第36.2节：员工层级

表结构设置

```
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, 'Žydrė', 'Klybė', 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)
```

公共表表达式

```
;WITH cteReports (EmpID, FirstName, LastName, SupervisorID, EmpLevel) AS
(
    SELECT 员工编号, 名字, 姓氏, 经理编号, 1
    FROM 员工
    WHERE ManagerID IS NULL

    UNION ALL
```


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, 'Žydrė', 'Klybė', 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:

全名	EmpLevel	ManagerName
Ken Sánchez	1	*null*
基思·霍尔	2	肯·萨切斯á
弗雷德·布洛格斯	2	Ken Sánchez
为伊德雷·克莱布	2	肯·萨切斯á
约瑟夫·沃克	3	基思·霍尔
彼得·米勒	3	弗雷德·布洛格斯
萨姆·杰克逊	3	为伊德雷·克莱布
克洛伊·萨缪尔斯	3	为伊德雷·克莱布
乔治·韦斯莱	3	为伊德雷·克莱布
迈克尔·肯辛顿 4		萨姆·杰克逊

第36.3节：递归公共表表达式（CTE）

此示例展示如何获取从今年到2011年（2012 - 1）的每一年。

```
WITH yearsAgo
(
myYear
)
作为
(
    -- 基础情况：递归从这里开始
    SELECT DATEPART(year, GETDATE()) AS myYear

    UNION ALL  -- 这里必须是 UNION ALL（不能是 UNION）

    -- 递归部分：这就是我们递归调用时所做的操作
    SELECT yearsAgo.myYear - 1
    FROM yearsAgo
    WHERE yearsAgo.myYear >= 2012
)
    SELECT myYear FROM yearsAgo;  -- 单个 SELECT、INSERT、UPDATE 或 DELETE
```

myYear
2018


```
    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

您可以通过将 MAXRECURSION 作为查询选项来控制递归（类似代码中的堆栈溢出），该选项将限制递归调用的次数。

```
WITH yearsAgo
(
myYear
)
作为
(
      -- 基本情况
      SELECT DATEPART(year , GETDATE()) AS myYear
      UNION ALL
      -- 递归部分
      SELECT yearsAgo.myYear - 1
      FROM yearsAgo
      WHERE yearsAgo.myYear >= 2002
)
      SELECT * FROM yearsAgo
      OPTION (MAXRECURSION 10);
```

> 消息 530，级别 16，状态 1，第 2 行 语句终止。最大递归次数 10 在语句完成前已被耗尽。

第 36.4 节：使用 CTE 删除重复行

员工表：

```
|  ID  | 名字    | 姓氏     | 性别   | 薪水   |
+------+---------+---------+--------+--------+
|  1   | 马克    | 黑斯廷斯 | 男     | 60000  |
|  1   | 马克    | 黑斯廷斯 | 男     | 60000  |
|  2   | 玛丽    | 兰贝斯   | 女     | 30000  |
|  2   | 玛丽    | 兰贝斯   | 女     | 30000  |
|  3   | 本      | 霍斯金斯 | 男     | 70000  |
|  3   | 本      | 霍斯金斯 | 男     | 70000  |
|  3   | 本      | 霍斯金斯 | 男     | 70000  |
+------+---------+---------+--------+--------+
```

CTE（公共表表达式）：

```
WITH EmployeesCTE AS
(
   SELECT *, ROW_NUMBER()OVER(PARTITION BY ID ORDER BY ID) AS RowNumber
   FROM Employees
)
DELETE FROM EmployeesCTE WHERE RowNumber > 1
```

执行结果：

```
|  ID  | 名字 | 姓氏 | 性别 | 薪水 |
+------+-----------+----------+--------+--------+
|  1   | Mark      | Hastings | 男   | 60000  |
|  2   | Mary      | Lambeth  | 女 | 30000  |
|  3   | Ben       | Hoskins  | 男   | 70000  |
+------+-----------+----------+--------+--------+
```


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  |
+------+-----------+----------+--------+--------+
```

第36.5节：带有多个AS语句的CTE

```
;WITH cte_query_1
AS
(
    SELECT *
    FROM database.table1
),
cte_query_2
作为
(
    SELECT *
    FROM database.table2
)
SELECT *
FROM cte_query_1
WHERE cte_query_one.fk IN
(
    SELECT PK
    FROM cte_query_2
)
```

使用公共表表达式，可以通过逗号分隔的 AS 语句创建多个查询。然后，一个查询可以以多种不同方式引用其中的任何一个或全部查询，甚至可以将它们连接起来。

第36.6节：使用CTE查找第n高薪资

员工表：

```
|  ID  | 名字      | 姓氏      | 性别   | 薪资   |
+------+----------+----------+-------+-------+
| 1    | 贾汉吉尔  | 阿拉姆    | 男     | 70000  |
| 2    | 阿里弗尔  | 拉赫曼    | 男     | 60000  |
| 3    | 奥利      | 阿哈迈德  | 男     | 45000  |
| 4    | 西玛      | 苏尔塔娜  | 女     | 70000  |
| 5    | 苏迪普塔  | 罗伊      | 男     | 80000  |
+------+----------+----------+-------+-------+
```

CTE（公共表表达式）：

```
 WITH RESULT AS
(
    SELECT 薪资,
DENSE_RANK() OVER (ORDER BY 薪资 DESC) AS DENSERANK
    FROM 员工
)
SELECT TOP 1 薪水
FROM RESULT
WHERE DENSERANK = 1
```

要查找第二高的薪水，只需将 N 替换为 2。同理，要查找第三高的薪水，只需将 N 替换为 3。

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.

第37章：移动和复制表中的数据

第37.1节：将数据从一个表复制到另一个表

此代码从表中选择数据并在查询工具（通常是 SSMS）中显示

```
SELECT Column1, Column2, Column3 FROM MySourceTable;
```

此代码将数据插入到表中：

```
INSERT INTO MyTargetTable (Column1, Column2, Column3)
SELECT Column1, Column2, Column3 FROM MySourceTable;
```

第37.2节：将数据复制到表中，同时动态创建该表

此代码从表中选择数据：

```
SELECT Column1, Column2, Column3 FROM MySourceTable;
```

此代码创建了一个名为MyNewTable的新表，并将数据放入其中

```
SELECT Column1, Column2, Column3
INTO MyNewTable
FROM MySourceTable;
```

第37.3节：将数据移动到表中（假设唯一键方法）

要*移动*数据，首先将其插入目标表，然后从源表中删除你插入的数据。这不是一个常规的SQL操作，但可能具有启发性

你插入了什么？通常在数据库中你需要有一个或多个列来唯一标识行，因此我们将假设这一点并加以利用。

该语句选择了一些行

```
SELECT Key1, Key2, Column3, Column4 FROM MyTable;
```

首先我们将这些插入目标表：

```
INSERT INTO TargetTable (Key1, Key2, Column3, Column4)
SELECT Key1, Key2, Column3, Column4 FROM MyTable;
```

现在假设两个表中的记录在Key1和Key2上都是唯一的，我们可以利用这一点来查找并删除源表中的数据

```
DELETE MyTable
WHERE EXISTS (
    SELECT * FROM TargetTable
    WHERE TargetTable.Key1 = SourceTable.Key1
```


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
);
```

这只有在Key1和Key2在两个表中都是唯一的情况下才能正确工作最后

，我们不想让工作半途而废。如果我们将其包装在一个事务中，那么要么所有数据都会被移动，要么什么都不会发生。这确保了我们不会先插入数据然后发现无法从源中删除数据。

```
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;
```



```
    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;
```

第38章：限制结果集

参数	详细信息
TOP	限制关键字。与数字一起使用。
PERCENT	百分比关键字。位于TOP和限制数字之后。

随着数据库表的增长，通常需要将查询结果限制为固定的数量或百分比。这可以通过使用SQL Server的TOP关键字或OFFSET FETCH子句来实现。

第38.1节：使用PERCENT限制

此示例将SELECT结果限制为总行数的15%。

```
SELECT TOP 15 PERCENT *
FROM table_name
```

第38.2节：使用FETCH限制

版本 ≥ SQL Server 2012

FETCH通常更适用于分页，但也可以作为TOP的替代方案：

```
SELECT *
FROM table_name
ORDER BY 1
OFFSET 0 ROWS
FETCH NEXT 50 ROWS ONLY
```

第38.3节：使用TOP进行限制

此示例将SELECT结果限制为100行。

```
SELECT TOP 100 *
FROM table_name;
```

也可以使用变量来指定行数：

```
DECLARE @CountDesiredRows int = 100;
SELECT TOP (@CountDesiredRows) *
FROM table_name;
```


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;
```

第39章：检索实例信息

第39.1节：关于数据库、表、存储过程的一般信息及其搜索方法

查询数据库中最后执行的存储过程

```
SELECT execquery.last_execution_time AS [日期 时间], execsql.text AS [脚本]
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
```

查询存储过程

```
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
```

查询数据库中所有表的列

```
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;
```

查询还原详情

```
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
```

查询日志的语句

```
select top 100 * from databaselog
Order by Posttime desc
```

查询存储过程详情的语句

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 类型 = 'P'
按 modify_date 降序排列
```

第39.2节：获取当前会话和查询执行的信息

```
sp_who2
```

此存储过程可用于查找当前SQL服务器会话的信息。由于它是一个存储过程，通常将结果存储到临时表或表变量中会很有帮助，这样可以根据需要对结果进行排序、过滤和转换。

下面可用于查询版本的 sp_who2：

```
-- 创建一个变量表以保存 sp_who2 的结果以便查询

DECLARE @who2 TABLE (
SPID INT NULL,
      状态 VARCHAR(1000) NULL,
     登录名 SYSNAME NULL,
     主机名 SYSNAME NULL,
     被阻塞者 SYSNAME NULL,
     数据库名 SYSNAME NULL,
     命令 VARCHAR(8000) NULL,
     CPU时间 INT NULL,
磁盘IO INT NULL,
最后批处理 VARCHAR(250) NULL,
     程序名 VARCHAR(250) NULL,
     第二SPID INT NULL, -- 不知为何的第二个SPID...?
请求ID INT NULL
)

INSERT INTO @who2
EXEC sp_who2

SELECT    *
FROM    @who2 w
WHERE    1=1
```

示例：

```
-- 查找特定用户会话：
SELECT  *
FROM    @who2 w
WHERE   1=1
    and  login = 'userName'

-- 查找最长CPU时间的查询：
SELECT  top 5 *
FROM    @who2 w
WHERE   1=1
order   by CPUTime desc
```



```
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
```

第39.3节：关于SQL Server版本的信息

要查询SQL Server的版本、产品级别和版本号，以及主机名和服务器类型：

```
SELECT    SERVERPROPERTY('机器名') AS 主机,
          SERVERPROPERTY('实例名') AS 实例,
          DB_NAME() AS 当前数据库,
          SERVERPROPERTY('版本') AS 版本,
          SERVERPROPERTY('产品级别') AS 产品级别,
          CASE SERVERPROPERTY('是否集群')
            WHEN 1 THEN '集群'
            ELSE '独立' END AS 服务器类型,
          @@VERSION AS 版本号;
```

第39.4节：检索实例的版本和版本号

```
SELECT    SERVERPROPERTY('产品版本') AS 产品版本,
          SERVERPROPERTY('产品级别') AS 产品级别,
          SERVERPROPERTY('版本') AS 版本,
          SERVERPROPERTY('引擎版本') AS 引擎版本;
```

第39.5节：检索实例运行天数

```
SELECT  DATEDIFF(DAY, 登录时间, getdate()) 运行天数
FROM    master..sysprocesses
WHERE   spid = 1
```

第39.6节：检索本地和远程服务器

要检索实例上注册的所有服务器列表：

```
EXEC sp_helpserver;
```


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;
```

第40章：带有并列选项

第40.1节：测试数据

```
创建表 #TEST
(
编号 INT,
名称 VARCHAR(10)
)

插入到 #Test
选择 1,'A'
联合 全部
选择 1,'B'
联合 全部
选择 1,'C'
联合 全部
选择 2,'D'
```

以下是上述表的输出，正如您所见，编号列重复了三次。

```
编号    名称
1     A
1     B
1     C
2     D
```

现在让我们用简单的排序来检查输出..

```
Select Top (1)  Id,Name From
#test
Order By Id ;
```

输出 :(上述查询的输出每次不保证相同)

```
编号    名称
1     B
```

让我们运行带有 Ties 选项的相同查询..

```
Select Top (1) With Ties Id,Name
 From
#test
按编号排序
```

输出：

```
编号    名称
1     A
1     B
1     C
```

如您所见，SQL Server 会输出所有与排序列相同的行。让我们再看一个例子
以便更好地理解。

```
Select Top (1) With Ties Id,Name
```


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
```

```
 来自
#test
按 Id ,Name 排序
```

Output:

```
编号    名称
1     A
```

总结，当我们使用 With Ties 选项时，SQL Server 会输出所有绑定的行，而不管我们设置的限制是多少。


```
 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

第41章：字符串函数

第41.1节：Quotename

返回一个由定界符包围的Unicode字符串，使其成为有效的SQL Server定界标识符。

参数：

1. 字符字符串。一个最多包含128个字符的Unicode数据字符串（sysname）。如果输入字符串超过128个字符，函数将返回null。字符函数返回null。
2. 引号字符。可选。用于作为分隔符的单个字符。可以是单引号（'或``），左括号或右括号（{,[,(,<或>,),],}）或双引号（"）。任何其他值将返回null。默认值是方括号。

```
SELECT QUOTENAME('what''s my name?')      -- 返回 [what's my name?]

SELECT QUOTENAME('what''s my name?', '[') -- 返回 [what's my name?]
SELECT QUOTENAME('what''s my name?', ']') -- 返回 [what's my name?]

SELECT QUOTENAME('what''s my name?', '''') -- 返回 'what''s my name?'

SELECT QUOTENAME('what''s my name?', '"') -- 返回 "what's my name?"

SELECT QUOTENAME('what''s my name?', ')') -- 返回 (what's my name?)
SELECT QUOTENAME('what''s my name?', '(') -- 返回 (what's my name?)

SELECT QUOTENAME('what''s my name?', '<') -- 返回 <what's my name?>
SELECT QUOTENAME('what''s my name?', '>') -- 返回 <what's my name?>

SELECT QUOTENAME('what''s my name?', '{') -- 返回 {what's my name?}
SELECT QUOTENAME('what''s my name?', '}') -- 返回 {what's my name?}

SELECT QUOTENAME('what''s my name?', '`') -- 返回 `what's my name?`
```

第41.2节：替换

返回一个字符串（varchar或nvarchar），其中所有指定子字符串的出现都被替换为另一个子字符串。

参数：

1.字符串表达式。要搜索的字符串。可以是字符或二进制数据类型。
2.模式。要被替换的子字符串。可以是字符或二进制数据类型。pattern参数不能为空字符串。
3.替换。这是将替换模式子字符串的子字符串。它可以是字符或二进制数据。

```
SELECT REPLACE('This is my string', 'is', 'XX') -- 返回 'ThXX XX my string'。
```

备注：

- 如果字符串表达式的类型不是varchar(max)或nvarchar(max)，replace函数会将返回值截断为8,000个字符。
- 返回数据类型取决于输入数据类型——如果其中一个输入值是nvarchar，则返回nvarchar，否则返回varchar。

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.

- 如果任何输入参数为NULL，则返回NULL

第41.3节：子字符串

返回一个子字符串，该子字符串从指定的起始索引处的字符开始，长度不超过指定的最大长度。

参数：

1. 字符表达式。字符表达式可以是任何可以隐式转换的数据类型 varchar 或 nvarchar，除 text 或 ntext 外。
2. 起始索引。一个数字（int 或 bigint），用于指定所请求子字符串的起始索引。（注意：字符串在 SQL Server 中，索引是从 1 开始的，意味着字符串的第一个字符的索引是 1）。该数字可以小于 1。在这种情况下，如果起始索引与最大长度的和大于 0，返回的字符串将是从字符表达式的第一个字符开始，长度为（起始索引 + 最大长度 - 1）的字符串。如果小于 0，则返回空字符串。
3. 最大长度。一个介于0和bigint最大值（9,223,372,036,854,775,807）之间的整数。如果最大长度参数为负数，将引发错误。如果最大长度参数为负数，将引发错误。

```
SELECT SUBSTRING('这是我的字符串', 6, 5) -- 返回 'is my'
```

如果最大长度加起始索引超过字符串中的字符数，则返回整个字符串。

```
SELECT SUBSTRING('你好，世界',1,100) -- 返回 '你好，世界'
```

如果起始索引大于字符串中的字符数，则返回空字符串。

```
SELECT SUBSTRING('你好，世界',15,10) -- 返回 ''
```

第41.4节：String_Split

版本 ≥ SQL Server 2016

使用字符分隔符拆分字符串表达式。注意，STRING_SPLIT()是一个表值函数，因此必须在FROM子句中使用。

参数：

1. 字符串。任何字符类型表达式（char，nchar，varchar 或 nvarchar）
2. 分隔符。任何类型的单字符表达式（char(1)，nchar(1)，varchar(1)或nvarchar(1)）。

返回一个单列表格，每行包含字符串的一个片段。列名为value，且如果任一参数是nchar或nvarchar，则数据类型为nvarchar，否则为varchar。

以下示例使用空格作为分隔符拆分字符串：

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

结果：

```
value
-----
Lorem
ipsum
dolor
sit
```


- 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.
```

备注：

> STRING_SPLIT函数仅在兼容级别130下可用。如果您的数据库兼容级别低于130，SQL Server将无法找到并执行STRING_SPLIT函数。您可以使用以下命令更改数据库的兼容级别：

```
ALTER DATABASE [database_name] SET COMPATIBILITY_LEVEL = 130
```

版本 < SQL Server 2016

较旧版本的 SQL Server 没有内置的字符串拆分函数。有许多用户定义的函数可以解决字符串拆分的问题。你可以阅读 Aaron Bertrand 的文章"正确拆分字符串"–或者"次佳方法"，了解它们的全面比较。

第 41.5 节：Left

返回从字符串最左边开始的子字符串，长度最多为指定的最大长度。

参数：

1. 字符表达式。字符表达式可以是任何可以隐式转换为varchar或nvarchar的数据类型，但不包括text或ntext
2. 最大长度。一个介于 0 和bigint最大值（9,223,372,036,854,775,807）之间的整数。如果最大长度参数为负数，将引发错误。

```
SELECT LEFT('This is my string', 4) -- 结果：'This'
```

如果最大长度超过字符串中的字符数，则返回整个字符串。

```
SELECT LEFT('这是我的字符串', 50) -- 结果: '这是我的字符串'
```

第41.6节：Right函数

返回字符串的最右部分的子字符串，长度不超过指定的最大长度。

参数：

1. 字符表达式。字符表达式可以是任何可以隐式转换为varchar或nvarchar的数据类型，但不包括text或ntext
2. 最大长度。一个介于0和bigint最大值（9,223,372,036,854,775,807）之间的整数。如果最大长度参数为负数，将引发错误。

```
SELECT RIGHT('This is my string', 6) -- 返回 'string'
```

如果最大长度超过字符串中的字符数，则返回整个字符串。

```
SELECT RIGHT('This is my string', 50) -- 返回 'This is my string'
```



```
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'
```

第41.7节：Soundex函数

返回一个四字符代码（varchar），用于评估两个字符串的语音相似度。

参数：

1.字符表达式。一个字母数字字符数据表达式。

soundex函数创建一个基于字符表达式发音的四字符代码。第一个字符是参数第一个字符的大写形式，后面3个字符是表示表达式中字母的数字（忽略a、e、i、o、u、h、w和y）。

```
SELECT SOUNDEX ('Smith') -- 返回 'S530'

SELECT SOUNDEX ('Smythe') -- 返回 'S530'
```

第41.8节：格式

版本 ≥ SQL Server 2012

返回一个NVARCHAR值，按指定的格式和文化（如果指定）进行格式化。主要用于
将日期时间类型转换为字符串。

参数：

1. value。一个支持的数据类型表达式，用于格式化。有效类型列于下方。
2. format。一个NVARCHAR格式模式。有关标准和自定义格式，请参见微软官方文档
   字符串。
3. culture。可选。nvarchar参数，指定文化。默认值为当前
   会话的文化。

日期

使用标准格式字符串：

```
DECLARE @d DATETIME = '2016-07-31';

SELECT
    FORMAT ( @d, 'd', 'en-US' ) 作为 'US English Result' -- 返回 '7/31/2016'
   ,FORMAT ( @d, 'd', 'en-gb' ) 作为 'Great Britain English Result' -- 返回 '31/07/2016'
   ,FORMAT ( @d, 'd', 'de-de' ) 作为 'German Result' -- 返回 '31.07.2016'
   ,FORMAT ( @d, 'd', 'zh-cn' ) AS '简体中文（中国）结果' -- 返回 '2016/7/31'
   ,FORMAT ( @d, 'D', 'en-US' ) AS '美国英语结果' -- 返回 'Sunday, July 31, 2016'
   ,FORMAT ( @d, 'D', 'en-gb' ) AS '英国英语结果' -- 返回 '31 July 2016'
   ,FORMAT ( @d, 'D', 'de-de' ) AS '德语结果' -- 返回 'Sonntag, 31. Juli 2016'
```

使用自定义格式字符串：

```
SELECT FORMAT( @d, 'dd/MM/yyyy', 'en-US' ) AS '日期时间结果' -- 返回 '31/07/2016'
      ,FORMAT(123456789,'###-##-####') AS '自定义数字结果' -- 返回 '123-45-6789',
      ,FORMAT( @d,'dddd, MMMM dd, yyyy hh:mm:ss tt','en-US') AS '美国' -- 返回 'Sunday, July 31,
2016 12:00:00 AM'
      ,FORMAT( @d,'dddd, MMMM dd, yyyy hh:mm:ss tt','hi-IN') AS '印地语' -- 返回 रविवार, जुलाई
31, 2016 12:00:00 पूर्वाह्न
      ,FORMAT ( @d, 'dddd', 'en-US' )  AS '美国' -- 返回 'Sunday'
      ,FORMAT ( @d, 'dddd', 'hi-IN' )  AS '印地语' -- 返回 'रविवार'
```


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 रविवार, जुलाई
31, 2016 12:00:00 पूर्वाह्न
      ,FORMAT ( @d, 'dddd', 'en-US' )  AS 'US' -- Returns 'Sunday'
      ,FORMAT ( @d, 'dddd', 'hi-IN' )  AS 'Hindi' -- Returns 'रविवार'
```

FORMAT 也可用于格式化 货币、百分比 和 数字。

货币

```
DECLARE @Price1 INT = 40
SELECT FORMAT(@Price1,'c','en-US') AS '美国文化中的货币' -- 返回 '$40.00'
       ,FORMAT(@Price1,'c','de-DE') AS '德国文化中的货币' -- 返回 '40,00 €'
```

我们可以指定小数点后的位数。

```
DECLARE @Price DECIMAL(5,3) = 40.356
SELECT FORMAT( @Price, 'C') AS '默认', -- 返回 '$40.36'
       FORMAT( @Price, 'C0') AS '无小数位', -- 返回 '$40'
       FORMAT( @Price, 'C1') AS '保留1位小数', -- 返回 '$40.4'
       FORMAT( @Price, 'C2') AS '保留2位小数', -- 返回 '$40.36'
```

百分比

```
   DECLARE @Percentage float = 0.35674
   SELECT FORMAT( @Percentage, 'P') AS '% 默认', -- 返回 '35.67 %'
   FORMAT( @Percentage, 'P0') AS '% 无小数位', -- 返回 '36 %'
   FORMAT( @Percentage, 'P1') AS '% 保留1位小数' -- 返回 '35.7 %'
```

数字

```
DECLARE @Number AS DECIMAL(10,2) = 454545.389
SELECT FORMAT( @Number, 'N','en-US') AS '美国数字格式', -- 返回 '454,545.39'
FORMAT( @Number, 'N','en-IN') AS '印度数字格式', -- 返回 '4,54,545.39'
FORMAT( @Number, '#.0') AS '保留1位小数', -- 返回 '454545.4'
FORMAT( @Number, '#.00') AS '保留2位小数', -- 返回 '454545.39'
FORMAT( @Number, '#,##.00') AS '带千分位和2位小数', -- 返回 '454,545.39'
FORMAT( @Number, '##.00') AS '无千分位且保留2位小数', -- 返回 '454545.39'
FORMAT( @Number, '000000000') AS '左侧补零至九位数' -- 返回 '000454545'
```

有效值类型列表：（source）

类别	类型	.Net 类型
数值型	bigint	Int64
数值型	int	Int32
数值型	smallint	Int16
数值型	tinyint	Byte
数值型	decimal	SqlDecimal
数值型	numeric	SqlDecimal
数值型	float	Double
数值型	real	Single
数值型	smallmoney	Decimal
数值型	money	Decimal
日期和时间	date	DateTime
日期和时间	time	TimeSpan
日期和时间	datetime	DateTime
日期和时间	smalldatetime	DateTime
日期和时间	datetime2	DateTime
日期和时间	datetimeoffset	DateTimeOffset

重要说明：

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 对于除无效文化之外的错误返回NULL。例如，如果格式中指定的值无效，则返回NULL。

- FORMAT依赖于.NET框架公共语言运行时（CLR）的存在。
- FORMAT依赖于CLR格式化规则，该规则规定冒号和句点必须进行转义。因此，当格式字符串（第二个参数）包含冒号或句点时，如果输入值（第一个参数）是时间数据类型，则冒号或句点必须用反斜杠进行转义。

另请参见使用FORMAT的日期和时间格式化文档示例。

第41.9节：String_escape

版本 ≥ SQL Server 2016

转义文本中的特殊字符并返回带有转义字符的文本（nvarchar(max)）。

参数：

1. text。是一个表示应被转义字符串的nvarchar表达式。
2. type。将应用的转义规则。目前唯一支持的值是'json'。

```
SELECT STRING_ESCAPE('\   /  \\
  "     ', 'json') -- 返回 '\\\/\\\\\"'
```

将被转义的字符列表：

```
特殊字符     编码序列
-----------------------------------
引号 (")    \"
反斜杠 (\)   \\
斜杠 (/)          \/
退格键           \b
换页符           \f
换行符
回车符           \r
水平制表符


控制字符     编码序列
-----------------------------------
CHAR(0)             \u0000
CHAR(1)           \u0001
...               ...
CHAR(31)            \u001f
```

第41.10节：ASCII

返回一个整数值，表示字符串最左边字符的ASCII码。

```
SELECT ASCII('t') -- 返回116
SELECT ASCII('T') -- 返回84
SELECT ASCII('This') -- 返回84
```

如果字符串是Unicode且最左边的字符不是ASCII，但可以在当前排序规则中表示，则可以返回大于127的值：

- 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') -- 当 `SERVERPROPERTY('COLLATION') =
'SQL_Latin1_General_CP1_CI_AS`` 时返回239
```

如果字符串是Unicode且最左边的字符无法在当前排序规则中表示，则返回整数值63（在ASCII中表示问号）：

```
SELECT ASCII(N' )' -- 返回 63
SELECT ASCII(nchar(2039)) -- 返回 63
```

第41.11节：字符

返回由整数ASCII码表示的字符。

```
SELECT CHAR(116) -- 返回 't'
SELECT CHAR(84)  -- 返回 'T'
```

这可以用来引入换行符/换行符CHAR(10)、回车符CHAR(13)等。参考请见AsciiTable.com。

如果参数值不在0到255之间，CHAR函数返回NULL。
CHAR函数的返回数据类型是char(1)

第41.12节：连接

版本 ≥ SQL Server 2012

返回两个或多个字符串连接后的结果字符串。CONCAT接受两个或更多参数。

```
SELECT CONCAT('This', ' is', ' my', ' string') -- 返回 'This is my string'
```

注意：与使用字符串连接运算符（+）连接字符串不同，当向concat函数传递null值时，它会隐式转换为一个空字符串：

```
SELECT CONCAT('This', NULL, ' is', ' my', ' string'), -- 返回 'This is my string'
       'This' + NULL + ' is' + ' my' + ' string' -- 返回 NULL。
```

非字符串类型的参数也会被隐式转换为字符串：

```
SELECT CONCAT('This', ' is my ', 3, 'rd string') -- 返回 'This is my 3rd string'
```

非字符串类型的变量也会被转换为字符串格式，无需手动转换或强制转换为字符串：

```
DECLARE @Age INT=23;
SELECT CONCAT('Ram is ', @Age,' years old');  -- 返回 'Ram is 23 years old'
```

版本 < SQL Server 2012

旧版本不支持CONCAT函数，必须使用字符串连接运算符（+）。非字符串类型必须先强制转换或转换为字符串类型，才能以这种方式连接。

```
SELECT 'This is the number ' + CAST(42 AS VARCHAR(5)) -- 返回 'This is the number 42'
```

第41.13节：LTrim

返回一个字符表达式（varchar 或 nvarchar），该表达式在去除所有前导空格后，即空格


```
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

从左侧开始直到第一个非空白字符。

参数：

1. 字符表达式。任何可以隐式转换为varchar的字符或二进制数据表达式，除text、ntext和image之外。

```
SELECT LTRIM('    This is my string') -- 返回 'This is my string'
```

第41.14节：RTrim

返回一个字符表达式（varchar或nvarchar），去除所有尾部空格，即从字符串右端开始直到第一个非空白字符左侧的所有空格。

参数：

1. 字符表达式。任何可以隐式转换为varchar的字符或二进制数据表达式，除text、ntext和image之外。

```
SELECT RTRIM('This is my string     ') -- 返回 'This is my string'
```

第41.15节：PatIndex

返回指定表达式中指定模式首次出现的位置。

参数：

1.模式。包含要查找序列的字符表达式。长度限制为最多8000个字符。模式中可以使用通配符（%，_）。如果模式不以通配符开头，则只能匹配表达式开头的内容。如果不以通配符结尾，则只能匹配表达式结尾的内容。

2.表达式。任何字符串数据类型。

```
SELECT PATINDEX('%ter%', 'interesting') -- 返回 3。

SELECT PATINDEX('%t_r%t%', 'interesting') -- 返回 3。

SELECT PATINDEX('ter%', 'interesting') -- 返回 0，因为 'ter' 不在开头。

SELECT PATINDEX('inter%', 'interesting') -- 返回 1。

SELECT PATINDEX('%ing', 'interesting') -- 返回 9。
```

第 41.16 节：空格

返回一个重复空格的字符串（varchar）。

参数：

1. 整数表达式。任何整数表达式，最大为 8000。如果为负，返回 null。如果为 0，返回空字符串。（要返回长度超过 8000 个空格的字符串，请使用 Replicate。）

```
SELECT SPACE(-1) -- 返回 NULL
SELECT SPACE(0)  -- 返回空字符串
```


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) -- 返回 '   '（包含3个空格的字符串）
```

第41.17节：差异

返回一个整数（int）值，表示两个字符

表达式的soundex值之间的差异。

参数：

1.字符表达式1。
2.字符表达式2。

两个参数都是字符数据的字母数字表达式。

返回的整数是参数soundex值中相同字符的数量，因此4表示表达式非常相似，0表示它们非常不同。

```
SELECT  SOUNDEX('Green'),  -- G650
        SOUNDEX('Greene'),  -- G650
        DIFFERENCE('Green','Greene') -- 返回4

SELECT  SOUNDEX('Blotchet-Halls'),  -- B432
        SOUNDEX('Greene'),  -- G650
        DIFFERENCE('Blotchet-Halls', 'Greene') -- 返回0
```

第41.18节：Len

返回字符串的字符数。
注意：LEN 函数会忽略尾部空格：

```
SELECT LEN('My string'), -- 返回 9
       LEN('My string   '), -- 返回 9
       LEN('   My string') -- 返回 12
```

如果需要包括尾部空格的长度，有几种方法可以实现，尽管每种方法都有其缺点。一种方法是向字符串追加一个字符，然后使用 LEN 减去一：

```
DECLARE @str varchar(100) = 'My string   '
SELECT LEN(@str + 'x') - 1 -- 返回 12
```

缺点是如果字符串变量或列的类型是最大长度，追加的额外字符会被丢弃，结果长度仍然不会计算尾部空格。为了解决这个问题，下面的修改版本解决了该问题，并在所有情况下都能给出正确结果，代价是稍微增加了一点执行时间，因此（包括代理对的正确结果和合理的执行速度）似乎是最佳的使用方法：

```
SELECT LEN(CONVERT(NVARCHAR(MAX), @str) + 'x') - 1
```

另一种方法是使用DATALENGTH函数。

```
DECLARE @str varchar(100) = 'My string   '
SELECT DATALENGTH(@str) -- 返回 12
```

但需要注意的是，DATALENGTH 返回的是字符串在内存中的字节长度。对于 varchar 和 nvarchar 来说，这个值是不同的。


```
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 theDATALENGTH 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) -- 返回 24
```

您可以通过将字符串的数据长度除以单个字符的数据长度（两者必须是相同类型）来进行调整。下面的示例演示了这一点，同时也处理了目标字符串为空的情况，从而避免除以零的错误。

```
DECLARE @str nvarchar(100) = 'My string    '
SELECT DATALENGTH(@str) / DATALENGTH(LEFT(LEFT(@str, 1) + 'x', 1)) -- 返回 12
```

不过，即使这样，在 SQL Server 2012 及以上版本中也存在问题。当字符串包含代理对（某些字符在同一字符串中占用的字节数多于其他字符）时，它会产生错误的结果。

另一种方法是使用REPLACE将空格替换为非空格字符，然后取结果的LEN。这种方法在所有情况下都能得到正确结果，但对于长字符串执行速度非常慢。

第41.19节：Lower

将所有大写字符转换为小写后，返回字符表达式（varchar或nvarchar）。

参数：

1. 字符表达式。任何可以隐式转换为varchar的字符或二进制数据表达式。

```
SELECT LOWER('This IS my STRING') -- 返回 'this is my string'

DECLARE @String nchar(17) = N'This IS my STRING';
SELECT LOWER(@String) -- 返回 'this is my string'
```

第41.20节：Upper

返回一个字符表达式（varchar 或 nvarchar），将所有小写字符转换为大写后得到。

参数：

1. 字符表达式。任何可以隐式转换为varchar的字符或二进制数据表达式。

```
SELECT UPPER('这是我的字符串') -- 返回 'THIS IS MY STRING'

DECLARE @String nchar(17) = N'这是我的字符串';
SELECT UPPER(@String) -- 返回 'THIS IS MY STRING'
```

第41.21节：Unicode

返回表示输入表达式第一个字符的Unicode值的整数值。

参数：

1. Unicode字符表达式。任何有效的 nchar 或 nvarchar 表达式。

```
SELECT UNICODE(N'Ɛ') -- 返回400

DECLARE @Unicode nvarchar(11) = N'Ɛ 是一个字符'
SELECT UNICODE(@Unicode) -- 返回400
```



```
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'Ɛ') -- Returns 400

DECLARE @Unicode nvarchar(11) = N'Ɛ is a char'
SELECT UNICODE(@Unicode) -- Returns 400
```

第41.22节：NChar

返回对应于其接收的整数参数的Unicode字符（nchar(1) 或 nvarchar(2)），根据Unicode标准定义。

参数：

1.整数表达式。任何介于0到65535之间的正整数表达式，或者如果数据库的排序规则支持补充字符（CS）标志，支持的范围为0到1114111。如果整数表达式不在此范围内，则返回 null。

```
SELECT NCHAR(257) -- 返回 'ā'
SELECT NCHAR(400) -- 返回 'Ɛ'
```

第41.23节：Str

返回从数值数据转换而来的字符数据（varchar）。

参数：

1. 浮点表达式。带小数点的近似数值数据类型。
2. 长度。可选。 返回的字符串表达式的总长度，包括数字、小数点和前导空格（如需要）。默认值为10。
3. 小数位数。可选。 小数点右侧的位数。如果超过16，结果将被截断为小数点右侧16位。

```
SELECT STR(1.2) -- 返回 '         1'

SELECT STR(1.2, 3) -- 返回 '  1'

SELECT STR(1.2, 3, 2) -- 返回 '1.2'

SELECT STR(1.2, 5, 2) -- 返回 ' 1.20'

SELECT STR(1.2, 5, 5) -- 返回 '1.200'

SELECT STR(1, 5, 2) -- 返回 ' 1.00'

SELECT STR(1) -- 返回 '         1'
```

第41.24节：Reverse

返回一个字符串值的反转顺序。

参数：

1. 字符串表达式。任何可以隐式转换为varchar的字符串或二进制数据。

```
选择REVERSE('Sql Server') -- 返回 'revreS lqS'
```

第41.25节：复制

将字符串值重复指定的次数。

参数：

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 'ā'
SELECT NCHAR(400) -- Returns 'Ɛ'
```

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.字符串表达式。字符串表达式可以是字符字符串或二进制数据。
2. 整数表达式。任何整数类型，包括bigint。如果为负，则返回null。如果为0，则返回空字符串
。

```
SELECT REPLICATE('a', -1)  -- 返回 NULL

SELECT REPLICATE('a', 0)  -- 返回 ''

SELECT REPLICATE('a', 5)  -- 返回 'aaaaa'

SELECT REPLICATE('Abc', 3) -- 返回 'AbcAbcAbc'
```

注意：如果字符串表达式不是varchar(max)或nvarchar(max)类型，返回值不会超过8000
字符。复制将在添加会导致返回值超过该限制的字符串之前停止：

```
SELECT LEN(REPLICATE('a b c d e f g h i j k l', 350)) -- 返回 7981

SELECT LEN(REPLICATE(cast('a b c d e f g h i j k l' as varchar(max)), 350)) -- 返回 8050
```

第41.26节：CharIndex

返回字符串表达式中另一个字符串表达式首次出现的起始索引。

参数列表：

1. 要查找的字符串（最多8000个字符）
2. 要搜索的字符串（任何有效的字符数据类型和长度，包括二进制）3.（可选）
起始索引。类型为int或bigint的数字。如果省略或小于1，则从字符串开头开始搜索。

如果要搜索的字符串是varchar(max)、nvarchar(max)或varbinary(max)，CHARINDEX函数将返回一个bigint值。否则，将返回一个int值。

```
SELECT CHARINDEX('is', 'this is my string') -- 返回3
SELECT CHARINDEX('is', 'this is my string', 4) -- 返回6
SELECT CHARINDEX(' is', 'this is my string') -- 返回5
```


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
```

第42章：逻辑函数

第42.1节：CHOOSE

版本 ≥ SQL Server 2012

返回指定索引处的值列表中的项。如果index超出values的范围，则返回NULL。

参数：

1. index：整数，values中项的索引。基于1计数。
2. values: 任意类型，逗号分隔列表

```
SELECT CHOOSE (1, '苹果', '梨', '橙子', '香蕉') AS chosen_result

chosen_result
-------------
苹果
```

第42.2节：IIF

版本 ≥ SQL Server 2012

根据给定的布尔表达式是否为真，返回两个值中的一个。

参数：

1. boolean_expression 用于判断返回哪个值的布尔表达式
2. true_value 如果 boolean_expression 计算结果为真，则返回该值
3. false_value 如果 boolean_expression 计算结果为假，则返回该值

```
SELECT IIF (42 > 23, '我早就知道了！', '那不是真的。') AS iif_result

iif_result
------------
我早就知道了！
```

版本 < SQL Server 2012

IIF可以被CASE语句替代。上述示例可以写成

```
SELECT CASE WHEN 42 > 23 THEN '我早就知道了！' ELSE '那不是真的。' END AS iif_result

iif_result
------------
我早就知道了！
```


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!
```

第43章：聚合函数

SQL Server中的聚合函数对一组值进行计算，返回单个值。

第43.1节：SUM()

返回指定列中数值的总和。

我们有如图所示的表，将用于执行不同的聚合函数。表名为*成绩单*。

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

```
选择 SUM(MarksObtained) 从 成绩单
```

sum函数不会考虑作为参数使用的字段中值为NULL的行

在上述示例中，如果我们有另一行如下：

```
106意大利语    NULL
```

该行不会被计入求和计算

第43.2节：AVG()

返回指定列中数值的平均值。

我们有如图所示的表，将用于执行不同的聚合函数。表名为*成绩单*。

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

```
选择 AVG(MarksObtained) From Marksheet
```

average函数不会考虑作为参数使用的字段中值为NULL的行

在上述示例中，如果我们有另一行如下：

```
106意大利语    NULL
```

该行不会被计入平均值计算

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

第43.3节：MAX()

返回指定列中的最大值。

我们有如图所示的表，将用于执行不同的聚合函数。表名为
成绩单。

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

```
选择 MAX(MarksObtained) From Marksheet
```

第43.4节：MIN()

返回给定列中的最小值。

我们有如图所示的表，将用于执行不同的聚合函数。表名为
成绩单。

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

```
Select MIN(MarksObtained) From Marksheet
```

第43.5节：COUNT()

返回给定列中的值的总数。

我们有如图所示的表，将用于执行不同的聚合函数。表名为
成绩单。

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

```
Select COUNT(MarksObtained) From Marksheet
```

函数count不会计算参数字段中值为NULL的行。通常count
参数是*（所有字段），因此只有当行的所有字段均为NULL时，该行才不会被计入。

在上述示例中，如果我们有另一行如下：

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意大利语    NULL
```

该行不会被计入计数计算中

注意

函数COUNT(*)返回表中的行数。该值也可以通过使用不包含列引用的常量非空表达式获得，例如COUNT(1)。

示例

```
Select COUNT(1) From Marksheet
```

第43.6节：使用GROUP BY的COUNT(Column_Name) Column_Name

大多数时候，我们希望获得表中某列值的总出现次数，例如：

表名：REPORTS

报告名称	报告价格
测试	10.00 $
测试	10.00 $
测试	10.00 $
测试 2	11.00 $
测试	10.00 $
测试 3	14.00 $
测试 3	14.00 $
测试 3	14.00 $
测试 4	100.00 $

```
选择
报告名称 作为 报告名称,
    计数(报告名称) 作为 计数
来自
报告
分组依据
报告名称
```

报告名称	计数
测试	4
测试 2	1
测试 3	2
测试 4	1


```
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 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

第44章：SQL Server中的字符串聚合函数

第44.1节：使用STUFF进行字符串聚合

我们有一个包含SubjectId的学生表。这里的需求是基于SubjectId进行连接。

所有 SQL Server 版本

```
创建表 #yourstudent (subjectid int, studentname varchar(10))

插入数据到 #yourstudent (subjectid, studentname) 值为
 ( 1       ,'Mary'    )
,( 1       ,'John'    )
,( 1       ,'Sam'     )
,( 2       ,'Alaina')
,( 2       ,'Edward')

选择 subjectid,  stuff(( 选择 concat( ',', studentname) 从 #yourstudent y 哪里 y.subjectid =
u.subjectid for xml path('')),1,1, '')
     从 #yourstudent u
     group by subjectid
```

第44.2节：用于字符串聚合的 String_Agg

在 SQL Server 2017 或 vnext 的情况下，我们可以使用内置的 STRING_AGG 来进行此聚合。对于相同的学生表，

```
创建表 #yourstudent (subjectid int, studentname varchar(10))

插入数据到 #yourstudent (subjectid, studentname) 值为
 ( 1       ,'Mary'    )
,( 1       ,'John'    )
,( 1       ,'Sam'     )
,( 2       ,'Alaina')
,( 2       ,'Edward')

select subjectid, string_agg(studentname, ',') from #yourstudent
    group by subjectid
```


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
```

第45章：排名函数

参数	详细信息
<partition_by_clause>	将由FROM子句生成的结果集划分为分区，DENSE_RANK函数应用于这些分区。DENSE_RANK函数应用于分区。有关PARTITION BY语法，请参见OVER子句 (Transact-SQL)。
<order_by_clause>	确定DENSE_RANK函数应用于分区中行的顺序。
OVER ([partition_by_clause] order_by_clause)	partition_by_clause将由FROM子句生成的结果集划分为函数应用的分区。如果未指定，函数将查询结果集的所有行视为单个组。order_by_clause确定函数应用前数据的顺序。order_by_clause是必需的。 对于RANK函数，OVER子句中的<rows或range子句>不能指定。更多信息，请参见OVER子句 (Transact-SQL)。

第45.1节：DENSE_RANK（）

与RANK（）相同。它返回无间隙的排名：

```
Select  Studentid, Name,Subject,Marks,
DENSE_RANK() over(partition by name order by Marks desc)Rank
From Exam
按 name排序

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瑞安    科学    50      3
103Tanvi   数学      90      1
103Tanvi   科学      90      1
103Tanvi   社会      80      2
```

第45.2节：RANK()

RANK() 返回分区列结果集中每行的排名。

例如：

```
Select Studentid,Name,Subject,Marks,
RANK() over(partition by name order by Marks desc)Rank
From Exam
按 name,subject排序

Studentid    姓名    科目    分数    排名

    101Ivan    数学      70       2
    101伊万    科学     80       1
    101伊万    社会      60       3
    102瑞安    数学      60       2
    102瑞安    科学     50       3
    102瑞安    社会      70       1
    103谭维   数学      90       1
    103谭维   科学     90       1
    103谭维   社会      80       3
```


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
```

第46章：窗口函数

第46.1节：中心移动平均

计算价格的6个月（126个交易日）居中移动平均值：

```
SELECT 交易日期, AVG(价格) OVER (ORDER BY 交易日期 ROWS BETWEEN 63 PRECEDING AND 63 FOLLOWING) AS
价格移动平均
FROM 历史价格
```

注意，因为每个返回的行前后最多会取63行，所以在交易日期范围的开始和结束处，窗口不会居中：当达到最大交易日期时，只能找到63个之前的值来计算平均值。

第46.2节：查找带时间戳事件列表中最新的单个项目

在记录事件的表中，通常有一个日期时间字段记录事件发生的时间。查找最新的单个事件可能很困难，因为总有可能两个事件的时间戳完全相同。你可以使用 row_number() over (order by ...) 来确保所有记录都有唯一排名，然后选择排名第一的记录（即 my_ranking=1）。

```
select *
来自 (
    选择
        *,
        row_number() over (order by 创建日期 desc) as 我的排名
    from sys.sysobjects
) g
where my_ranking=1
```

同样的技术可以用来从任何可能包含重复值的数据集中返回单行数据。

第46.3节：最近30项的移动平均

最近30项销售的移动平均

```
SELECT
value_column1,
    (   SELECT
            AVG(value_column1) AS moving_average
        FROM Table1 T2
        WHERE ( SELECT
                    COUNT(*)
                来自 Table1 T3
                WHERE date_column1 BETWEEN T2.date_column1 AND T1.date_column1
                ) BETWEEN 1 AND 30
    ) as MovingAvg
FROM Table1 T1
```


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
```

第47章：PIVOT / UNPIVOT

第47.1节：动态PIVOT

PIVOT查询的一个问题是，如果你想将它们作为列显示，必须在IN选择中指定所有值。绕过这个问题的一个快速方法是创建一个动态的IN选择，使你的PIVOT变得动态。

为了演示，我们将使用一个名为Books的表，位于Bookstore数据库中。我们假设该表相当非规范化，并具有以下列

```
表：Books
-----------------------------
BookId（主键列）
名称
语言
页数
版本号
印刷年份
入库年份
ISBN
作者姓名
价格
销售单位数
```

表的创建脚本如下：

```
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
```

现在，如果我们需要查询数据库，统计每年购入书店的英语、俄语、德语、印地语、拉丁语书籍数量，并以简短报告格式呈现输出，可以使用如下PIVOT查询

```
SELECT * FROM
  (
   SELECT YearBoughtIntoStore AS [Year Bought],[Language], NumberOfBooks
   FROM BookList
```


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
```

特殊情况是当我们没有完整的语言列表时，我们将使用如下的动态SQL

```
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)
```

第47.2节：简单的PIVOT和UNPIVOT（T-SQL）

下面是一个简单的示例，展示了每个商品在每个工作日的平均价格。

首先，假设我们有一个表，记录了所有商品的每日价格。

```
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);
```

该表应如下所示：

```
+========+==========+=======+
|   item | weekday | price |
+========+==========+=======+
|  Item1 |    Mon  |   110 |
+--------+---------+-------+
|  Item2 |    Mon  |   230 |
+--------+---------+-------+
|  Item3 |    Mon  |   150 |
+--------+---------+-------+
|  Item1 |    Tue  |   115 |
+--------+---------+-------+
|  Item2 |    Tue  |   231 |
+--------+---------+-------+
|  Item3 |    Tue  |   162 |
+--------+---------+-------+
|          . . .          |
+--------+---------+-------+
|  Item2 |    Wed  |   220 |
+--------+---------+-------+
```



```
  ) 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 |
+--------+---------+-------+
```

为了执行聚合操作，即计算每个商品在每个工作日的平均价格，我们将使用关系运算符PIVOT，将表值表达式中的weekday列旋转为聚合的行值，如下所示：

```
SELECT * FROM tbl_stock
PIVOT (
    AVG(price) 针对工作日 在 ([周一], [周二], [周三], [周四], [周五])
) pvt;
```

结果：

```
+--------+------+------+------+------+------+
| 商品  | 周一 | 周二 | 周三 | 周四 | 周五 |
+--------+------+------+------+------+------+
| 商品1 |  116 |  112 |  117 |  109 |  120 |
| 商品2 |  227 |  233 |  230 |  228 |  210 |
| 商品3 |  145 |  158 |  152 |  145 |  125 |
+--------+------+------+------+------+------+
```

最后，为了执行PIVOT的逆操作，我们可以使用关系运算符UNPIVOT将列旋转为行，如下所示：

```
SELECT * FROM tbl_stock
PIVOT (
    AVG(price) 针对工作日 在 ([周一], [周二], [周三], [周四], [周五])
) pvt
UNPIVOT (
price 针对工作日 在 ([周一], [周二], [周三], [周四], [周五])
) unpvt;
```

结果：

```
+=======+========+=========+
|  商品 |  价格  | 工作日  |
+=======+========+=========+
| 商品1 |    116 |    周一 |
+-------+--------+---------+
| 商品1 |    112 |    周二 |
+-------+--------+---------+
| 商品1 |    117 |    周三 |
+-------+--------+---------+
| 商品1 |    109 |    周四 |
+-------+--------+---------+
| 商品1 |    120 |    周五 |
+-------+--------+---------+
| 商品2 |    227 |    周一 |
+-------+--------+---------+
| 商品2 |    233 |    周二 |
+-------+--------+---------+
| 商品2 |    230 |    周三 |
+-------+--------+---------+
| 商品2 |    228 |    周四 |
+-------+--------+---------+
| 商品2 |    210 |    周五 |
```



```
| 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 |
```

```
+-------+--------+---------+
| 商品3 |    145 |    周一 |
+-------+--------+---------+
| 商品3 |    158 |    周二 |
+-------+--------+---------+
| 商品3 |    152 |    周三 |
+-------+--------+---------+
| 商品3 |    145 |    周四 |
+-------+--------+---------+
| 商品3 |    125 |    周五 |
+-------+--------+---------+
```

第47.3节：简单透视 - 静态列

使用示例数据库中的商品销售表，让我们计算并显示每种产品的总销售数量。

这可以通过分组（group by）轻松完成，但假设我们要以一种"旋转"结果表的方式，使每个产品编号（Product Id）对应一列。

```
SELECT [100], [145]
  FROM (SELECT ItemId , Quantity
          FROM #ItemSalesTable
       ) AS pivotIntermediate
 PIVOT (   SUM(Quantity)
           FOR ItemId IN ([100], [145])
       ) AS pivotTable
```

由于我们的"新"列是数字（在源表中），我们需要使用方括号 [ ]

这将给我们如下输出

```
100 145
45  18
```



```
+-------+--------+---------+
| 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
```

第48章：动态SQL透视

本主题介绍如何在SQL Server中进行动态透视。

第48.1节：基本动态SQL透视

```
如果 object_id('tempdb.dbo.#temp') 不为空 则删除表 #temp
创建表 #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)


声明
    @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 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
```

第49章：分区

第49.1节：检索分区边界值

```
SELECT        ps.name AS 分区方案
            , fg.name AS [文件组]
            , 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
```

第49.2节：切换分区

根据此[TechNet微软页面][1]，

> **分区数据**使您能够快速高效地管理和访问数据的子集，同时保持整个数据集合的完整性。

当您调用以下查询时，数据不会被物理移动；只有关于数据位置的元数据发生变化。

```
ALTER TABLE [SourceTable] SWITCH TO [TargetTable]
```

表必须具有相同的列、相同的数据类型和NULL设置，且需要位于相同的文件组中，新的目标表必须为空。有关切换分区的更多信息，请参见上述页面链接。

[1]: https://technet.microsoft.com/en-us/library/ms191160(v=sql.105).aspx 列的 IDENTITY 属性可能不同。

第49.3节：使用单个查询检索分区表、列、方案、函数、总数及最小-最大边界值

```
SELECT DISTINCT
    object_name(i.object_id) AS [对象名称],
    c.name AS [分区列],
    s.name AS [分区方案],
    pf.name AS [分区函数],
   prv.tot AS [分区计数],
    prv.miVal AS [最小边界值],
    prv.maVal AS [最大边界值]
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
```


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)
```

只需取消注释where子句并将table_name替换为实际表名即可查看所需对象的详细信息。


```
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.

第50章：存储过程

在 SQL Server 中，存储过程是一个可以传入参数的存储程序。它不像函数那样返回一个值。然而，它可以向调用它的过程返回成功/失败状态。

第 50.1 节：创建和执行基本存储过程

使用图书馆数据库中的Authors表

```
CREATE PROCEDURE GetName
(
    @input_id INT = NULL,       --输入参数，人员的 ID，默认 NULL
    @name VARCHAR(128) = NULL   --输入参数，人员的姓名，默认 NULL
)
作为
开始
    SELECT Name + ' 来自 ' + Country
    FROM Authors
    WHERE Id = @input_id OR Name = @name
END
GO
```

你可以用几种不同的语法来执行存储过程。首先，你可以使用EXECUTE或EXEC

```
EXECUTE GetName @id = 1
EXEC Getname @name = 'Ernest Hemingway'
```

此外，您可以省略 EXEC 命令。同时，您不必指定传入的是哪个参数，因为您会传入所有参数。

```
GetName NULL, 'Ernest Hemingway'
```

当您想以与存储过程声明顺序不同的顺序指定输入参数时，可以指定参数名并赋值。例如

```
 CREATE PROCEDURE dbo.sProcTemp
 (
    @Param1 INT,
    @Param2 INT
)
作为
开始

    SELECT
Param1 = @Param1,
        Param2 = @Param2

END
```

执行此存储过程的正常顺序是先为 @Param1 指定值，然后为 @Param2 指定值。所以它看起来大致如下

```
  EXEC dbo.sProcTemp @Param1 = 0,@Param2=1
```

但也可以使用以下方式

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
```

在这里，您首先为@param2指定值，然后为@Param1指定值。这意味着您不必保持与存储过程声明时相同的顺序，可以按任意顺序进行。但您需要明确指定要为哪个参数设置值。

从任何数据库访问存储过程

你也可以创建一个带有前缀sp_的存储过程，这些存储过程就像所有系统存储过程一样，可以在不指定数据库的情况下执行，这是由于SQL Server的默认行为。当你执行一个以"sp_"开头的存储过程时，SQL Server会先在master数据库中查找该存储过程。如果在master中找不到该存储过程，则会在当前活动数据库中查找。如果你有一个想要从所有数据库访问的存储过程，请将其创建在master数据库中，并使用包含"sp_"前缀的名称。

```
使用 Master

创建存储过程 sp_GetName
(
    @input_id INT = NULL,       --输入参数，人员的 ID，默认 NULL
    @name VARCHAR(128) = NULL   --输入参数，人员的姓名，默认 NULL
)
作为
开始
    SELECT Name + ' 来自 ' + Country
    FROM Authors
    WHERE Id = @input_id OR Name = @name
END
GO
```

第50.2节：带有If...Else和Insert Into操作的存储过程

创建示例表Employee：

```
创建表 Employee
(
Id INT,
EmpName VARCHAR(25),
    EmpGender VARCHAR(6),
    EmpDeptId INT
)
```

创建一个存储过程，用于检查传入存储过程的值是否非空或非空字符串，并在Employee表中执行插入操作。

```
创建存储过程 spSetEmployeeDetails
(
    @ID int,
    @Name VARCHAR(25),
    @Gender VARCHAR(6),
    @DeptId INT
)
作为
开始
    IF (
        (@ID IS NOT NULL AND LEN(@ID) !=0)
        AND (@Name IS NOT NULL AND LEN(@Name) !=0)
```



```
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)
    )
    开始
        INSERT INTO Employee
        (
Id,
            EmpName,
            EmpGender,
          EmpDeptId
        )
        值
        (
            @ID,
            @Name,
            @Gender,
            @DeptId
        )
    结束
ELSE
    打印'参数错误'
结束
GO
```

执行存储过程

```
DECLARE @ID INT,
    @Name VARCHAR(25),
    @Gender VARCHAR(6),
    @DeptId INT

EXECUTE spSetEmployeeDetails
    @ID = 1,
    @Name = 'Subin Nepal',
    @Gender = '男',
    @DeptId = 182666
```


第50.3节：存储过程中的动态SQL

动态SQL使我们能够在运行时生成并执行SQL语句。当我们的SQL
语句包含在不同编译时可能变化的标识符时，就需要使用动态SQL。

动态SQL的简单示例：

```
CREATE PROC sp_dynamicSQL
@table_name       NVARCHAR(20),
@col_name         NVARCHAR(20),
@col_value        NVARCHAR(20)
AS
开始
DECLARE  @Query  NVARCHAR(max)
SET      @Query = 'SELECT * FROM ' + @table_name
SET      @Query = @Query + ' WHERE ' + @col_name + ' = ' + ''''+@col_value+''''
EXEC     (@Query)
结束
```

在上述 SQL 查询中，我们可以看到通过在运行时定义 @table_name、@col_name 和 @col_value 的值来使用该查询。查询在运行时生成并执行。这是一种可以将整个脚本作为字符串存储在变量中并执行的技术。我们可以使用动态 SQL 创建更复杂的查询


```
        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

和连接的概念。当你想创建一个可以在多种条件下使用的脚本时，这个概念非常强大。

执行存储过程

```
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
```

我使用的表

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

输出

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

第 50.4 节：带有 OUT 参数的存储过程

存储过程可以使用参数列表中的 OUTPUT 关键字返回值。

创建带有单个输出参数的存储过程

```
创建存储过程 SprocWithOutParams
(
    @InParam VARCHAR(30),
    @OutParam VARCHAR(30) OUTPUT
)
作为
开始
    SELECT @OutParam = @InParam + ' must come out'
    RETURN
END
GO
```

执行存储过程

```
DECLARE @OutParam VARCHAR(30)
EXECUTE SprocWithOutParams 'what goes in', @OutParam OUTPUT
PRINT @OutParam
```

创建带有多个输出参数的存储过程

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
)
作为
开始
    SELECT @OutParam = @InParam +' must come out'
    SELECT @OutParam2 = @InParam +' must come out'
    RETURN
结束
GO
```

执行存储过程

```
声明 @OutParam VARCHAR(30)
声明 @OutParam2 VARCHAR(30)
执行 SprocWithOutParams2 '传入内容', @OutParam 输出, @OutParam2 输出
打印 @OutParam
打印 @OutParam2
```

第50.5节：简单循环

首先，将一些数据放入名为#systables的临时表中，并添加一个递增的行号，以便我们可以一次查询一条记录

```
选择
o.name,
    row_number() over (order by o.name) as rn
into
    #systables
来自
    sys.objects as o
where
o.type = 'S'
```

接下来我们声明一些变量来控制循环，并在此示例中存储表名

```
declare
    @rn int = 1,
    @maxRn int = (
                    选择
                        max(rn)
                    来自
#systables as s
                    )
declare    @tablename sys name
```

现在我们可以使用简单的 while 循环。我们在 select 语句中递增 @rn，但这也可以是一个单独的语句，例如 set @rn = @rn + 1，这取决于你的需求。我们还使用 @rn 递增前的值从 #systables 中选择单条记录。最后我们打印表名。

```
while @rn <= @maxRn
    begin

        选择
            @tablename = name,
            @rn = @rn + 1
        from
```



```
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
```

第50.6节：简单循环

```
创建存储过程 SprocWithSimpleLoop
(
    @SayThis VARCHAR(30),
    @ThisManyTimes INT
)
作为
开始
    当 @ThisManyTimes > 0
    开始
        打印 @SayThis;
        设置 @ThisManyTimes = @ThisManyTimes - 1;
    结束

    返回;
结束
GO
```



```
            #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
```

第51章：检索数据库信息

第51.1节：检索所有存储过程的列表

以下查询将返回数据库中所有存储过程的列表，以及每个存储过程的基本信息：

版本 ≥ SQL Server 2005

```
SELECT *
FROM INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_TYPE = 'PROCEDURE'
```

ROUTINE_NAME、ROUTINE_SCHEMA 和 ROUTINE_DEFINITION 列通常是最有用的。

版本 ≥ SQL Server 2005

```
SELECT *
FROM sys.objects
WHERE type = 'P'
```

版本 ≥ SQL Server 2005

```
SELECT *
FROM sys.procedures
```

请注意，此版本相较于从 sys.objects 选择具有优势，因为它包含了额外的列 is_auto_executed, is_execution_replicated, is_repl_serializable, 和 skips_repl_constraints。

版本 < SQL Server 2005

```
SELECT *
FROM sysobjects
WHERE type = 'P'
```

请注意，输出包含许多与存储过程无关的列。

下一组查询将返回数据库中包含字符串 'SearchTerm' 的所有存储过程：

版本 < 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%'
```

版本 ≥ 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%'
```

第51.2节：获取服务器上所有数据库的列表

方法1： 以下查询适用于 SQL Server 2000 及以上版本（包含12列）

```
SELECT * FROM dbo.sysdatabases
```


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
```

方法2： 以下查询提取有关数据库的更多信息（例如：状态、隔离级别、恢复模型等）

注意：这是一个目录视图，仅在SQL SERVER 2005及更高版本中可用

```
SELECT * FROM sys.databases
```

方法3： 若只想查看数据库名称，可以使用未公开的存储过程sp_MSForEachDB

```
EXEC sp_MSForEachDB 'SELECT "?" AS DatabaseName'
```

方法4： 以下存储过程将帮助您提供数据库大小以及数据库名称、所有者、状态等信息

```
EXEC sp_helpdb
```

方法5 同样，以下存储过程将提供数据库名称、数据库大小和备注

```
EXEC sp_databases
```

第51.3节：计算数据库中的表数量

此查询将返回指定数据库中的表数量。

```
USE YourDatabaseName
SELECT COUNT(*) from INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE'
```

以下是另一种方法，可用于 SQL Server 2008 及以上版本的所有用户表。参考文献在 here。

```
SELECT COUNT(*) FROM sys.tables
```

第 51.4 节：数据库文件

显示所有数据库的所有数据文件及其大小和增长信息

```
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',
增长 AS GrowthUnit,
        ROUND(((CAST(Size AS FLOAT)*8)/1024)/1024,2) [SizeGB], -- 将8k页面转换为GB
        Maxsize,
        filename AS 物理文件名

FROM    Master.SYS.SYSALTFILES SF
Join    Master.SYS.Databases d on sf.fileid = d.database_id

Order by d.name
```


Method 2: Below query extract information about databases with more information (eg: 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
```

第51.5节：查看是否使用了企业版特定功能

有时验证您在开发者版上的工作是否引入了对任何仅限企业版功能的依赖是很有用的。

您可以使用 sys.dm_db_persisted_sku_features 系统视图来实现，如下所示：

```
SELECT * FROM sys.dm_db_persisted_sku_features
```

针对数据库本身。

这将列出正在使用的功能（如果有的话）。

第51.6节：确定Windows登录的权限路径

这将显示用户类型和权限路径（用户从哪个Windows组获得权限）。

```
xp_logininfo 'DOMAIN\user'
```

第51.7节：搜索并返回包含指定列值的所有表和列

此脚本来自 here 和 here，将返回所有包含指定值的表和列。这对于查找数据库中某个特定值的位置非常有用。此操作可能较为耗费资源，建议先在备份/测试环境中执行。

```
DECLARE @SearchStr nvarchar(100)
SET @SearchStr = '## YOUR STRING HERE ##'


    -- 版权所有 © 2002 Narayana Vyas Kondreddi。保留所有权利。
    -- 目的：搜索所有表的所有列中给定的搜索字符串
    -- 作者：Narayana Vyas Kondreddi
    -- 网站：http://vyaskn.tripod.com
    -- 更新和测试者：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+Databa
se+Updated.aspx
    -- 测试环境：SQL Server 7.0、SQL Server 2000、SQL Server 2005 和 SQL Server 2010
    -- 修改日期：2011年3月3日 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

    开始
        SET @ColumnName = ''
        SET @TableName =
        (
            SELECT MIN(QUOTENAME(TABLE_SCHEMA) + '.' + QUOTENAME(TABLE_NAME))
            FROM     INFORMATION_SCHEMA.TABLES
```


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+Databa
se+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
        )

        当 @TableName 不为 NULL 且 @ColumnName 不为 NULL 时

        开始
            设置 @ColumnName =
            (
                选择 最小(QUOTENAME(COLUMN_NAME))
                来自     INFORMATION_SCHEMA.COLUMNS
                条件          TABLE_SCHEMA     = PARSENAME(@TableName, 2)
                    且     TABLE_NAME     = PARSENAME(@TableName, 1)
                    且     DATA_TYPE 属于 ('char', 'varchar', 'nchar', 'nvarchar', 'int', 'decimal')
                    且    QUOTENAME(COLUMN_NAME) > @ColumnName
            )

            如果 @ColumnName 不为 NULL

            开始
                插入到 #Results
                执行
                (
                    '选择 ''' + @TableName + '.' + @ColumnName + ''', LEFT(' + @ColumnName + ',
3630) 从 ' + @TableName + ' (NOLOCK) ' +
                    ' WHERE ' + @ColumnName + ' LIKE ' + @SearchStr2
                )
            结束
                )
            结束
        结束
    结束

    选择 列名, 列值 从 #Results

删除表 #Results
-- 更多内容见：
http://thesitedoctor.co.uk/blog/search-every-table-and-field-in-a-sql-server-database-updated#sthas
h.bBEqfJVZ.dpuf
```

第51.8节：获取所有架构、表、列和索引

```
SELECT
s.名称 作为 [架构],
    t.对象ID 作为 [表对象ID],
    t.名称 作为 [表名],
c.列ID,
    c.名称 作为 [列名],
    i.名称 作为 [索引名],
    i.类型描述 作为 [索引类型]
从 sys.schemas 作为 s
内连接 sys.表 作为 t
    连接条件 s.架构ID = t.架构ID
内连接 sys.列 作为 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
```



```
            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#sthas
h.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;
```

第51.9节：返回包含计划信息的SQL代理作业列表

```
USE msdb
Go


SELECT dbo.sysjobs.Name AS '作业名称',
      '作业启用' = CASE dbo.sysjobs.Enabled
            WHEN 1 THEN '是'
            WHEN 0 THEN '否'
      END,
      '频率' = CASE dbo.sysschedules.freq_type
            WHEN 1 THEN '一次'
            WHEN 4 THEN '每日'
            WHEN 8 THEN 'Weekly'
            WHEN 16 THEN '每月'
            WHEN 32 THEN '每月相对'
            WHEN 64 THEN 'SQLServer 代理启动时'
      END,
      '开始日期' = 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,
      '开始时间' = 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'
                  + 左(右(运行时长,5),1)
```


    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 '子日频率'
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

联合

SELECT dbo.sysjobs.Name AS '作业名称',
      '作业启用' = CASE dbo.sysjobs.Enabled
            WHEN 1 THEN '是'
            WHEN 0 THEN '否'
END,
'频率' = CASE dbo.sysschedules.freq_type
            WHEN 1 THEN '一次'
            WHEN 4 THEN '每日'
            WHEN 8 THEN 'Weekly'
            WHEN 16 THEN '每月'
            WHEN 32 THEN '每月相对'
            WHEN 64 THEN 'SQLServer 代理启动时'
END,
'开始日期' = 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,
'开始时间' = 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)
                        +':' + 右侧(next_run_time,2) 作为字符 (8))
            当 5 时 则 cast('0' + 左侧(右侧(next_run_time,5),1)
                        +':' + 左侧(右侧(next_run_time,4),2)
                        +':' + 右侧(next_run_time,2) 作为字符 (8))
            当 6 时 则 cast(左侧(右侧(next_run_time,6),2)
```



```
                        +':' + 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)
```

```
                    +':' + 左侧(右侧(next_run_time,4),2)
                    +':' + 右侧(next_run_time,2) 作为字符 (8))
    结束,

    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'
                    + 左(右(运行时长,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 '子日频率'
FROM dbo.sysjobs
左外连接 dbo.sysjobschedules 在 dbo.sysjobs.job_id = dbo.sysjobschedules.job_id
内连接 dbo.sysschedules 在 dbo.sysjobschedules.schedule_id = dbo.sysschedules.schedule_id
左外连接 (选择 job_id, 最大值(run_duration) 作为 run_duration
            来自 dbo.sysjobhistory
            按 job_id分组) Q1
在 dbo.sysjobs.job_id = Q1.job_id
条件 Next_run_time <> 0

排序依据 [开始日期],[开始时间]
```

第51.10节：检索包含已知列的表

此查询将返回所有 列 及其关联的 表，针对给定的列名。它旨在显示哪些表（未知）包含指定的列（已知）

```
SELECT
c.名称 作为 列名,
    t.名称 作为 表名
来自
    sys.columns c
    JOIN sys.tables t ON c.object_id = t.object_id
WHERE
c.name LIKE '%MyName%'
```



```
                    +':' + 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%'
```

第51.11节：显示当前数据库中所有表的大小

```
SELECT
s.name + '.' + t.NAME AS TableName,
    SUM(a.used_pages)*8 AS 'TableSizeKB'  --SQL Server中的一个页大小为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
分组依据
s.name, t.name
ORDER BY
    --按名称排序：
s.name + '.' + t.NAME
    --或按大小从大到小排序：
    --SUM(a.used_pages) desc
```

第51.12节：检索数据库选项

以下查询返回数据库选项和元数据：

```
select * from sys.databases WHERE name = 'MyDatabaseName';
```

第51.13节：查找数据库中字段的所有引用

```
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
```

将查找 SProcs、视图等中对myField的引用

第51.14节：检索备份和还原操作的信息

获取当前数据库实例上执行的所有备份操作列表：

```
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
```

获取当前数据库实例上执行的所有还原操作列表：

```
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]
```


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
```



```
    INNER JOIN msdb.dbo.backupmediafamily bmf ON [bs].[media_set_id] = [bmf].[media_set_id]
ORDER BY [d].[name], [r].restore_date DESC
```

第52章：SQL Server中的字符串拆分函数

第52.1节：使用XML在Sql Server 2008/2012/2014中拆分字符串

由于没有STRING_SPLIT函数，我们需要使用XML技巧将字符串拆分成多行：

示例：

```
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);
```

结果：

```
-------
|Value|
-------
|A    |
-------
|B    |
-------
|C    |
+-----+
```

第52.2节：在Sql Server 2016中拆分字符串

在**SQL Server 2016**中，他们终于引入了拆分字符串函数：`STRING_SPLIT`

参数：它接受两个参数

字符串：

> 是任何字符类型的表达式（即nvarchar、varchar、nchar或char）。

separator：

> 是任何字符类型（例如 nvarchar(1)、varchar(1)、nchar(1) 或 char(1)）的单字符表达式，
> 用于作为连接字符串的分隔符。

注意： 你应始终检查表达式是否为非空字符串。

示例：

```
选择值
来自 STRING_SPLIT('a|b|c','|')
```

在上述示例中

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

```
字符串    : 'a|b|c'
分隔符 : '|'
```

结果 ：

```
-------
|Value|
-------
|a    |
-------
|b    |
-------
|c    |
+-----+
```

如果是空字符串：

```
SELECT value
FROM STRING_SPLIT('','')
```

结果 ：

```
+-----+
|Value|
-------
1 |     |
-------
```

你可以通过添加WHERE子句来避免上述情况

```
SELECT value
FROM STRING_SPLIT('','')
WHERE LTRIM(RTRIM(value))<>''
```

第52.3节：T-SQL表变量和XML

```
声明@userList 表(UserKey VARCHAR(60))
向@userList 插入值('bill'),('jcom'),('others')
--声明了一个表变量并插入了3条记录

声明@text XML
选择@text = (
        select UserKey  from @userList for XML Path('user'), root('group')
)
--从表中设置XML值

Select @text

--查看变量值
XML:
\<group>\<user>\<UserKey>bill\</UserKey>\</user>\<user>\<UserKey>jcom\</UserKey>\</user>\<user>\<Us
erKey>others\</UserKey>\</user>\</group>
```



```
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>
```

第53章：插入

第53.1节：向名为Invoices的表中添加一行

```
INSERT INTO Invoices [ /* 这里可以写列名 */ ]
VALUES (123, '1234abc', '2016-08-05 20:18:25.770', 321, 5, '2016-08-04');
```

- 如果插入的表中包含带有IDENTITY属性的列，则必须指定列名。

```
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 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');
```

第54章：主键

第54.1节：创建带有标识列作为主键的表

```
-- 标识主键 - 唯一的任意递增数字
 创建表 person (
id int identity(1,1) 主键 非空,
 firstName varchar(100) 非空,
lastName varchar(100) 非空,
 dob DateTime 非空,
ssn varchar(9) 非空
 )
```

第54.2节：创建带有GUID主键的表

```
-- GUID主键 - 表的任意唯一值
 创建表 person (
id uniqueIdentifier 默认 (newId()) 主键,
 firstName varchar(100) 非空,
lastName varchar(100) 非空,
 dob DateTime 非空,
ssn varchar(9) 非空
 )
```

第54.3节：创建带有自然键的表

```
-- 自然主键 - 使用表中已有的唯一标识记录的数据

 创建表 person (
 firstName varchar(100) 非空,
 lastName varchar(100) 非空,
 dob DateTime 非空,
ssn varchar(9) 主键 非空
 )
```

第54.4节：创建带复合键的表

```
-- 复合键 - 使用表中两个或多个现有列来创建主键
 创建表 person (
firstName varchar(100) 非空,
 lastName varchar(100) 非空,
 dob DateTime 非空,
ssn varchar(9) 非空,
 主键 (firstName, lastName, dob)
 )
```

第54.5节：向现有表添加主键

```
修改表 person
 添加约束 pk_PersonSSN 主键 (ssn)
```

注意，如果主键列（本例中为ssn）存在多个具有相同候选键的行，上述语句将失败，因为主键值必须唯一。

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 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.

第54.6节：删除主键

```
ALTER TABLE Person
 DROP CONSTRAINT pk_PersonSSN
```


Section 54.6: Delete primary key

```
ALTER TABLE Person
 DROP CONSTRAINT pk_PersonSSN
```

第55章：外键

第55.1节：外键关系/约束

外键使您能够定义两个表之间的关系。一个（父）表需要有主键以唯一标识表中的行。另一个（子）表可以在其中一列中包含父表主键的值。FOREIGN KEY REFERENCES约束确保子表中的值必须存在于父表的主键值中。

在此示例中，我们有一个带有CompanyId主键的父表Company，以及一个子表Employee，子表中包含该员工所在公司的ID。

```
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 确保插入到Employee.CompanyId列的值也必须存在于Company.CompanyId列中。此外，如果子表中至少有一个员工的companyId匹配，则不能删除公司表中的公司。

FOREIGN KEY关系确保两个表中的行不能“断开连接”。

第55.2节：维护父子行之间的关系

假设我们在公司表（Company）中有一行，companyId为1。我们可以在员工表（Employee）中插入一行，其companyId为1：

```
insert into Employee values (17, 'John', 1)
```

但是，我们不能插入companyId不存在的员工：

```
insert into Employee values (17, 'John', 111111)
```

消息 547，级别 16，状态 0，第 12 行 INSERT 语句与外键约束 "FK__Employee__Compan__1EE485AA" 冲突。冲突发生在数据库 "MyDb"、表 "dbo.Company"、列 'CompanyId'。语句已终止。

此外，只要员工表中至少有一行子记录引用该公司，我们就不能删除公司表中的父行。

```
delete from company where CompanyId = 1
```

消息 547，级别 16，状态 0，第 14 行 DELETE 语句与引用约束 "FK__Employee__Compan__1EE485AA" 冲突。冲突发生在数据库 "MyDb"、表 "dbo.Employee"、列 'CompanyId'。语句已终止。

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.

外键关系确保公司和员工行不会“断开关联”。

第55.3节：在现有表上添加外键关系

可以在尚未建立关系的现有表上添加外键约束。假设我们有Company（公司）和Employee（员工）表，其中Employee表有CompanyId列，但没有外键关系。
ALTER TABLE语句允许你在现有列上添加引用其他表及该表主键的外键约束：

```
alter table Employee
    add  foreign key (CompanyId) references Company(CompanyId)
```

第55.4节：在现有表上添加外键

可以在尚未建立关系的现有表上添加带约束的外键列。假设我们有Company（公司）和Employee（员工）表，其中Employee表没有CompanyId列。ALTER TABLE语句允许你添加带有外键约束的新列，该约束引用其他表及该表的主键：

```
alter table Employee
    add CompanyId int foreign key references Company(CompanyId)
```

第55.5节：获取外键约束信息

sys.foreignkeys系统视图返回数据库中所有外键关系的信息：

```
select name,
 OBJECT_NAME(referenced_object_id) 作为[父表],
 OBJECT_NAME(parent_object_id) 作为[子表],
delete_referential_action_desc,
 update_referential_action_desc
来自 sys.foreign_keys
```


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
```

第56章：最后插入的标识

第56.1节：@@IDENTITY 和 MAX(ID)

```
SELECT MAX(Id) FROM Employees -- 显示 Employees 表中最后一行的 Id 值。
GO
INSERT INTO Employees (FName, LName, PhoneNumber) -- 插入一行新数据
VALUES ('John', 'Smith', '25558696525')
GO
SELECT @@IDENTITY
GO
SELECT MAX(Id) FROM Employees -- 显示新插入行的 Id 值。
GO
```

最后两个 SELECT 语句的值是相同的。

第56.2节：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
作为
    INSERT INTO dbo.logging_table(log_message)
    VALUES('有人向person表中添加了数据')
GO;

INSERT INTO dbo.person(person_name)
VALUES('约翰·多伊')

SELECT SCOPE_IDENTITY();
```

这将返回在同一连接、当前作用域内最近添加的标识值。
在本例中，对于dbo.person表的第一行，返回值为1。

第56.3节：@@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
作为
    INSERT INTO dbo.logging_table(log_message)
    VALUES('有人向person表中添加了数据')
GO;

INSERT INTO dbo.person(person_name)
```


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('约翰·多伊')

SELECT @@IDENTITY;
```

这将返回同一连接上最近添加的标识值，无论作用域如何。在这种情况下，
假设SQL Server实例上没有其他活动且此插入操作没有触发其他触发器，
无论logging_table的标识列当前值是多少。

第56.4节：IDENT_CURRENT('tablename')

```
SELECT IDENT_CURRENT('dbo.person');
```

这将选择所选表上最近添加的标识值，无论连接或作用域如何。


```
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.

第57章：SCOPE_IDENTITY()

第57.1节：简单示例介绍

SCOPE_IDENTITY() 返回在同一作用域内插入到标识列的最后一个标识值。作用域是一个模块：存储过程、触发器、函数或批处理。因此，如果两个语句位于同一存储过程、函数或批处理中，则它们处于同一作用域。

```
INSERT INTO  ([column1],[column2]) VALUES (8,9);
GO
SELECT SCOPE_IDENTITY() AS [SCOPE_IDENTITY];
GO
```


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
```

第58章：序列

第58.1节：创建序列

```
CREATE SEQUENCE [dbo].[CustomersSeq]
AS INT
START WITH 10001
INCREMENT BY 1
MINVALUE -1;
```

第58.2节：在表中使用序列

```
CREATE TABLE [dbo].[Customers]
(
CustomerID INT DEFAULT (NEXT VALUE FOR [dbo].[CustomersSeq]) NOT NULL,
    CustomerName VARCHAR(100),
);
```

第58.3节：使用序列插入表中

```
INSERT INTO [dbo].[Customers]
       ([CustomerName])
 值
       ('Jerry'),
       ('Gorge')

SELECT * FROM [dbo].[Customers]
```

结果

客户编号	客户名称
10001	杰瑞
10002	乔治

第58.4节：删除并插入新数据

```
DELETE FROM [dbo].[Customers]
WHERE CustomerName = 'Gorge';

INSERT INTO [dbo].[Customers]
       ([CustomerName])
 VALUES ('George')

SELECT * FROM [dbo].[Customers]
```

结果

客户编号	客户名称
10001	杰瑞
10003	乔治

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

第59章：索引

第59.1节：创建聚集索引

使用聚集索引时，叶子页包含实际的表行。因此，聚集索引只能有一个。

```
创建表 员工
(
ID 字符(900),
    名字 NVARCHAR(3000),
    姓氏 NVARCHAR(3000),
    入职年份 字符(900)
)
GO

创建聚集索引 IX_Clustered
在 员工(ID)
GO
```

第59.2节：删除索引

```
删除索引 IX_NonClustered 在 员工
```

第59.3节：创建非聚集索引

非聚集索引的结构与数据行分开。非聚集索引包含非聚集索引键值，每个键值条目都有一个指向包含该键值的数据行的指针。在 SQL Server 2008/2012 中，最多可以有 999 个非聚集索引。

参考链接：https://msdn.microsoft.com/en-us/library/ms143432.aspx

```
创建表 员工
(
ID 字符(900),
    名字 NVARCHAR(3000),
    姓氏 NVARCHAR(3000),
    入职年份 字符(900)
)
GO

CREATE NONCLUSTERED INDEX IX_NonClustered
ON Employees(StartYear)
GO
```

第 59.4 节：显示索引信息

```
SP_HELPINDEX tableName
```

第 59.5 节：返回大小和碎片索引

```
sys.dm_db_index_physical_stats (
    { database_id | NULL | 0 | DEFAULT }
  , { object_id | NULL | 0 | DEFAULT }
  , { index_id | NULL | 0 | -1 | DEFAULT }
```


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 }
)
```

示例：

```
SELECT * FROM sys.dm_db_index_physical_stats
    (DB_ID(N'DBName'), OBJECT_ID(N'IX_NonClustered '), NULL, NULL , 'DETAILED');
```

第59.6节：重组和重建索引

avg_fragmentation_in_percent 值 纠正说明

>5% 且 <= 30% 重组

>30% 重建

```
ALTER INDEX IX_NonClustered ON tableName REORGANIZE;


ALTER INDEX ALL ON Production.Product
 REBUILD WITH (FILLFACTOR = 80, SORT_IN_TEMPDB = ON,
          STATISTICS_NORECOMPUTE = ON);
```

第59.7节：重建或重组表上的所有索引

重建索引使用以下语句完成

```
ALTER INDEX All ON tableName REBUILD;
```

这将删除索引并重新创建它，消除碎片，回收磁盘空间并重新排序索引页。

也可以使用以下命令重新组织索引

```
ALTER INDEX All ON tableName REORGANIZE;
```

该命令将使用最少的系统资源，通过物理重新排序叶级页，使其与叶节点的逻辑从左到右顺序相匹配，从而对表和视图上的聚集和非聚集索引的叶级进行碎片整理

第59.8节：重建所有索引数据库

```
EXEC sp_MSForEachTable 'ALTER INDEX ALL ON ? REBUILD'
```

第59.9节：视图上的索引

```
CREATE VIEW  View_Index02
WITH SCHEMABINDING
作为
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)
```



```
  , { 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)
```

第59.10节：索引调查

你可以使用 "SP_HELPINDEX Table_Name"，但Kimberly Tripp 有一个存储过程（可以在 here 找到），这是更好的示例，因为它显示了更多关于索引的信息，包括列和过滤器定义，例如：
用法：

```
USE Adventureworks
EXEC sp_SQLskills_SQL2012_helpindex 'dbo.Product'
```

另外，Tibor Karaszi 有一个存储过程（在 here 找到）。后者还会显示索引使用情况，并可选择提供索引建议列表。用法：

```
USE Adventureworks
EXEC sp_indexinfo 'dbo.Product'
```


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'
```

第60章：全文索引

第60.1节：A. 创建唯一索引、全文目录，和全文索引

以下示例在AdventureWorks2012示例数据库的HumanResources.JobCandidate表的JobCandidateID列上创建唯一索引。然后示例创建一个默认的全文目录ft。最后，示例在Resume列上创建全文索引，使用ft目录和系统停用词列表。

```
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

第60.2节：在多个表列上创建全文索引

```
USE AdventureWorks2012;
GO
CREATE FULLTEXT CATALOG production_catalog;
GO
CREATE FULLTEXT INDEX ON Production.ProductReview
 (
审稿人姓名
      语言 1033,
电子邮件地址
      语言 1033,
评论
      语言 1033
 )
  主键索引 PK_ProductReview_ProductReviewID
      在 production_catalog;
GO
```

第60.3节：创建带有搜索属性列表但不填充的全文索引

```
USE AdventureWorks2012;
GO
在 Production.Document 上创建全文索引
  (
标题
      语言 1033,
  文档摘要
      语言 1033,
  文档
```


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
```

```
      类型列 文件扩展名
      语言 1033
  )
  关键索引 PK_Document_DocumentID
          使用停用词表 = 系统, 搜索属性列表 = DocumentPropertyList, 关闭变更跟踪,
不填充;
   GO
```

然后稍后填充它

```
修改全文索引 ON Production.Document 设置变更跟踪 自动;
GO
```

第60.4节：全文搜索

```
选择 product_id
来自 products
其中包含(product_description, "Snap Happy 100EZ" 或 形式为(同义词词库,'Snap Happy') 或 '100EZ')

且 product_cost < 200 ;


选择 candidate_name,SSN
来自 candidates
其中包含(candidate_resume,"SQL Server") 且 candidate_division =DBA;
```

欲了解更多详细信息 https://msdn.microsoft.com/en-us/library/ms142571.aspx


```
      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

第61章：触发器

触发器是一种特殊类型的存储过程，在事件发生后自动执行。触发器有两种类型：数据定义语言触发器和数据操作语言触发器。

它通常绑定到表，并自动触发。你不能显式调用任何触发器。

第61.1节：DML触发器

DML触发器是在响应DML语句（insert、update或delete）时触发的。
可以为单个表或视图创建一个DML触发器，以处理一个或多个DML事件。这意味着单个DML触发器可以处理对特定表或视图的插入、更新和删除记录，但只能处理该单个表或视图上的数据更改。

DML触发器提供对inserted和deleted表的访问，这些表包含了由触发触发器的插入、更新或删除语句所影响的数据的信息。

注意DML触发器是基于语句的，而非基于行的。这意味着如果语句影响了多行，inserted或deleted表将包含多行。

示例：

```
CREATE TRIGGER tblSomething_InsertOrUpdate ON tblSomething
FOR INSERT
作为

    INSERT INTO tblAudit (TableName, RecordId, Action)
    SELECT 'tblSomething', Id, 'Inserted'
    FROM 插入

GO

创建触发器 tblSomething_InsertOrUpdate 在 tblSomething
用于更新
作为

    插入到 tblAudit (表名, 记录ID, 操作)
    选择 'tblSomething', Id, '已更新'
    FROM 插入

GO

创建触发器 tblSomething_InsertOrUpdate 在 tblSomething
用于删除
作为

    插入到 tblAudit (表名, 记录ID, 操作)
    选择 'tblSomething', Id, '已删除'
    FROM 删除

GO
```

以上所有示例将在tblSomething中添加、删除或更新记录时，向tblAudit添加记录。

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.

第61.2节：触发器的类型和分类

在SQL Server中，触发器分为两类：DDL触发器和DML触发器。

DDL触发器响应数据定义语言（DDL）事件触发。这些事件主要对应以关键字CREATE、ALTER和DROP开头的Transact-SQL语句。

DML 触发器是在响应数据操作语言（DML）事件时触发的。这些事件对应于以关键字INSERT、UPDATE和DELETE开头的Transact-SQL语句。

DML 触发器分为两大类：

1. 事后触发器（After Triggers）

    - 插入后触发器（AFTER INSERT Trigger）。
    - 更新后触发器（AFTER UPDATE Trigger）。
    - 删除后触发器（AFTER DELETE Trigger）。

2. 替代触发器（Instead of triggers）

    - 替代插入触发器（INSTEAD OF INSERT Trigger）。
    - 替代更新触发器（INSTEAD OF UPDATE Trigger）。
    - 替代删除触发器（INSTEAD OF DELETE Trigger）。

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.

第62章：游标

第62.1节：基本的仅向前游标

通常你会想避免使用游标，因为它们可能对性能产生负面影响。然而，在某些特殊情况下，你可能需要逐条循环处理数据记录并执行某些操作。

```
DECLARE @orderId AS INT

-- 这里我们创建游标，作为本地游标并且只允许
-- 向前操作
DECLARE rowCursor CURSOR LOCAL FAST_FORWARD FOR
     -- 这是我们想要逐条循环的查询
     SELECT [OrderId]
     FROM [dbo].[Orders]

-- 首先我们需要打开游标
OPEN rowCursor

-- 现在我们将通过获取第一条数据来初始化游标，在本例中是
[OrderId] 列，
-- 并将值存储到名为 @orderId 的变量中
FETCH NEXT FROM rowCursor INTO @orderId

-- 开始循环，直到没有更多记录可循环
WHILE @@FETCH_STATUS = 0
开始

     打印@orderId

     -- 这很重要，因为它告诉 SQL Server 获取下一条记录并将 [OrderId]
列的值存储到 @orderId 变量中
     FETCH NEXT FROM rowCursor INTO @orderId

结束

-- 这将释放游标使用的任何内存
CLOSE rowCursor
DEALLOCATE rowCursor
```

第62.2节：基本游标语法

一个简单的游标语法，操作几个示例测试行：

```
/* 准备测试数据 */
DECLARE @test_table TABLE
(
Id INT,
     Val VARCHAR(100)
);
INSERT INTO @test_table(Id, Val)
VALUES
     (1, 'Foo'),
     (2, 'Bar'),
     (3, 'Baz');
/* 测试数据已准备 */

/* 迭代变量 @myId，仅作示例 */
```


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;

/* 用于遍历行并将值赋给变量的游标 */
DECLARE myCursor CURSOR FOR
    SELECT Id
    FROM @test_table;

/* 开始遍历行 */
OPEN myCursor;
FETCH NEXT FROM myCursor INTO @myId;

/* @@FETCH_STATUS 全局变量在没有更多行可取时将为 1 / true */
WHILE @@FETCH_STATUS = 0
开始

    /* 在循环中执行的写操作。此处以简单的SELECT为例 */
    SELECT Id, Val
    FROM @test_table
    WHERE Id = @myId;

    /* 将变量设置为迭代器返回的下一个值；否则游标将无限循环。*/

    FETCH NEXT FROM myCursor INTO @myId;
结束
/* 完成所有操作后，进行清理 */
CLOSE myCursor;
DEALLOCATE myCursor;
```

来自 SSMS 的结果。请注意，这些都是独立的查询，彼此之间没有任何统一。注意查询引擎如何逐条处理每次迭代，而不是作为一个集合处理。

Id	Val
1	Foo

(影响 1 行)

Id	Val
2	Bar

(影响 1 行)

Id	Val
3	Baz

(影响 1 行)


```
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)

第63章：事务隔离级别

第63.1节：已提交读

版本 ≥ SQL Server 2008 R2

```
设置事务隔离级别为已提交读
```

该隔离级别是第二宽松的。它防止脏读。已提交读的行为取决于READ_COMMITTED_SNAPSHOT的设置：

- 如果设置为关闭（默认设置），事务使用共享锁来防止其他事务修改当前事务使用的行，同时阻止当前事务读取其他事务修改的行。
- 如果设置为开启，可以使用READCOMMITTEDLOCK表提示来请求共享锁，而不是对运行在已提交读模式下的事务使用行版本控制。

注意：已提交读是SQL Server的默认行为。

第63.2节："脏读"是什么？

脏读（或未提交读）是指读取正在被未完成事务修改的行。

这种行为可以通过使用两个独立的查询来复制：一个用于开启事务并向表中写入一些数据但不提交，另一个使用此隔离级别选择尚未提交的数据。

查询 1 - 准备一个事务但不完成它：

```
CREATE TABLE dbo.demo (
    col1 INT,
col2 VARCHAR(255)
);
GO
--这行将正常提交：
BEGIN TRANSACTION;
    INSERT INTO dbo.demo(col1, col2)
    VALUES (99, '正常事务');
COMMIT TRANSACTION;
--这行将"卡"在一个未完成的事务中，导致脏读
BEGIN TRANSACTION;
    INSERT INTO dbo.demo(col1, col2)
    VALUES (42, '脏读');
--此处不要提交事务（COMMIT TRANSACTION）或回滚事务（ROLLBACK TRANSACTION）
```

查询 2 - 读取包含未提交事务的行：

```
设置事务隔离级别为 READ UNCOMMITTED;
选择 * 从 dbo.demo;
```

返回结果：

```
col1        col2
----------- ---------------------------------------
99          正常事务
```


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          脏读
```

附注：别忘了清理这些演示数据：

```
提交事务（COMMIT TRANSACTION）；
删除表 dbo.demo;
GO
```

第63.3节：读取未提交数据

版本 ≥ SQL Server 2008 R2

```
设置事务隔离级别为读取未提交
```

这是最宽松的隔离级别，因为它根本不会引起任何锁。它规定语句可以读取所有行，包括那些已在事务中写入但尚未提交的行（即仍处于事务中）。该隔离级别可能会发生“脏读”。

第63.4节：可重复读

版本 ≥ SQL Server 2008 R2

```
设置事务隔离级别为可重复读
```

该事务隔离级别比已提交读（READ COMMITTED）稍微严格一些，因为对事务中每个语句读取的所有数据都会加共享锁，并且该锁会一直持有直到事务完成，而不是在每个语句后释放。

注意：仅在必要时使用此选项，因为它比已提交读（READ COMMITTED）更可能导致数据库性能下降以及死锁。

第63.5节：快照

版本 ≥ SQL Server 2008 R2

```
设置事务隔离级别为快照
```

指定事务中任何语句读取的数据都是事务开始时存在的事务一致性版本的数据，即只读取事务开始前已提交的数据。

快照（SNAPSHOT）事务不会请求或引起对读取数据的任何锁，因为它只读取事务开始时存在的数据版本（或快照）。

在快照（SNAPSHOT）隔离级别下运行的事务只读取其自身的数据更改。例如，事务可以更新某些行然后读取更新后的行，但该更改仅对当前事务可见，直到它被提交。

注意：必须先将ALLOW_SNAPSHOT_ISOLATION数据库选项设置为ON，才能使用SNAPSHOT隔离级别。

第63.6节：可串行化

版本 ≥ SQL Server 2008 R2


```
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

设置事务隔离级别 SERIALIZEABLE

此隔离级别是最严格的。它请求范围锁锁定事务中每个语句读取的键值范围。这也意味着，如果要插入的行位于当前事务锁定的范围内，来自其他事务的INSERT语句将被阻塞。

此选项的效果等同于在事务中所有SELECT语句的所有表上设置HOLDLOCK。

注意：此事务隔离级别的并发性最低，应仅在必要时使用。


```
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.

第64章：高级选项

第64.1节：启用并显示高级选项

```
执行 sp_configure 'show advanced options' ,1
RECONFIGURE
GO
-- 显示所有配置
sp_configure
```

第64.2节：启用备份压缩默认设置

```
执行 sp_configure 'backup compression default',1
GO
RECONFIGURE;
```

第64.3节：启用cmd权限

```
执行 sp_configure 'xp_cmdshell', 1
GO
RECONFIGURE
```

第64.4节：设置默认填充因子百分比

```
sp_configure 'fill factor', 100;
GO
RECONFIGURE;
```

必须重启服务器后更改才能生效。

第64.5节：设置系统恢复间隔

```
USE master;
GO
-- 设置每3分钟恢复一次
EXEC sp_configure 'recovery interval', '3';
RECONFIGURE WITH OVERRIDE;
```

第64.6节：设置最大服务器内存大小

```
USE master
EXEC sp_configure 'max server memory (MB)', 64
RECONFIGURE WITH OVERRIDE
```

第64.7节：设置检查点任务数量

```
EXEC sp_configure "number of checkpoint tasks", 4
```


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
```

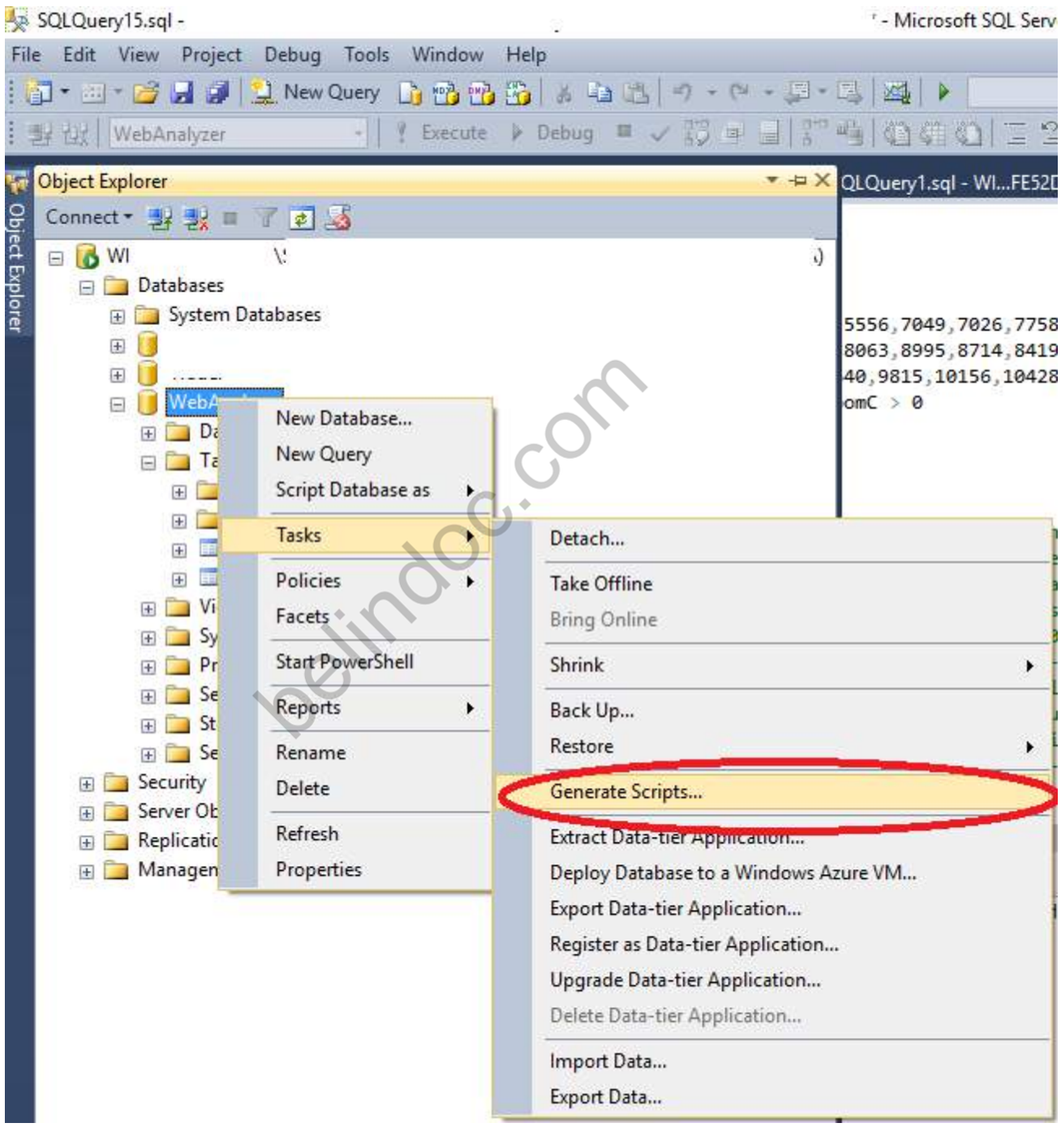
Section 64.7: Set number of checkpoint tasks

```
EXEC sp_configure "number of checkpoint tasks", 4
```

第65章：迁移

第65.1节：如何生成迁移脚本

1. 点击 右键鼠标 在您想要迁移的数据库上，依次点击 -> 任务 -> 生成脚本...

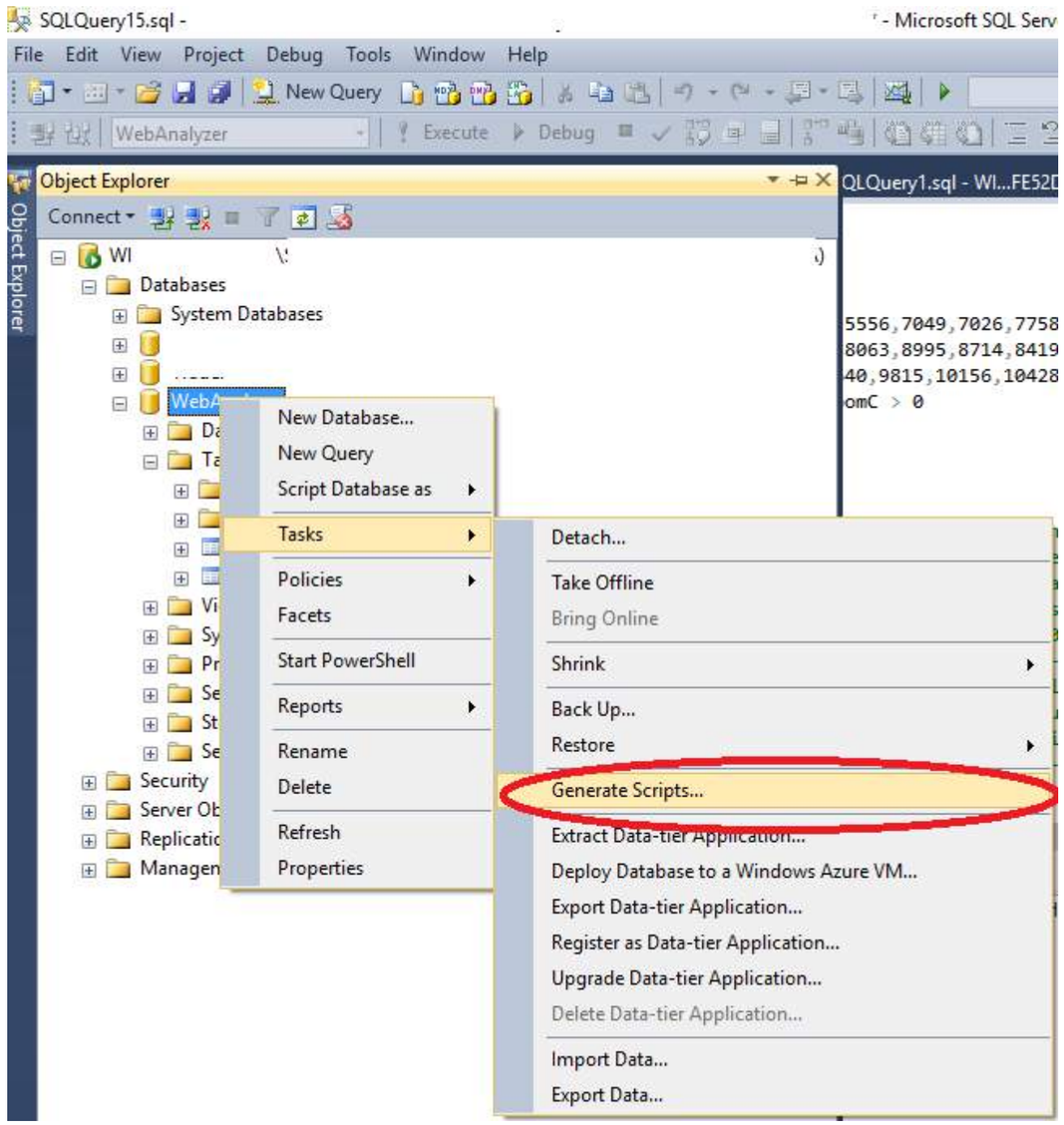


2. 向导将打开，点击下一步，然后选择您想要迁移的对象，再次点击下一步，然后点击高级向下滚动一点，在数据类型中选择脚本，选择模式和数据（除非你只想要结构）

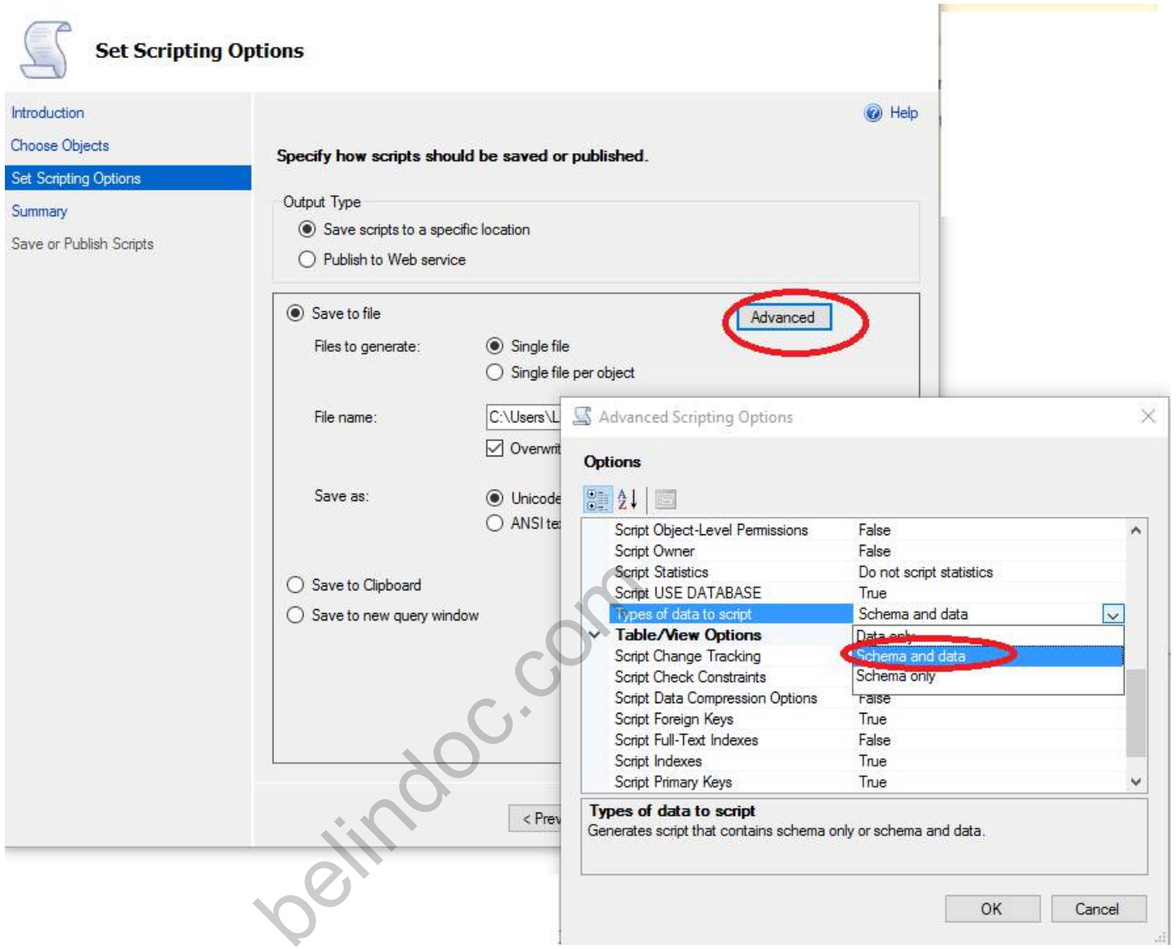
Chapter 65: Migration

Section 65.1: How to generate migration scripts

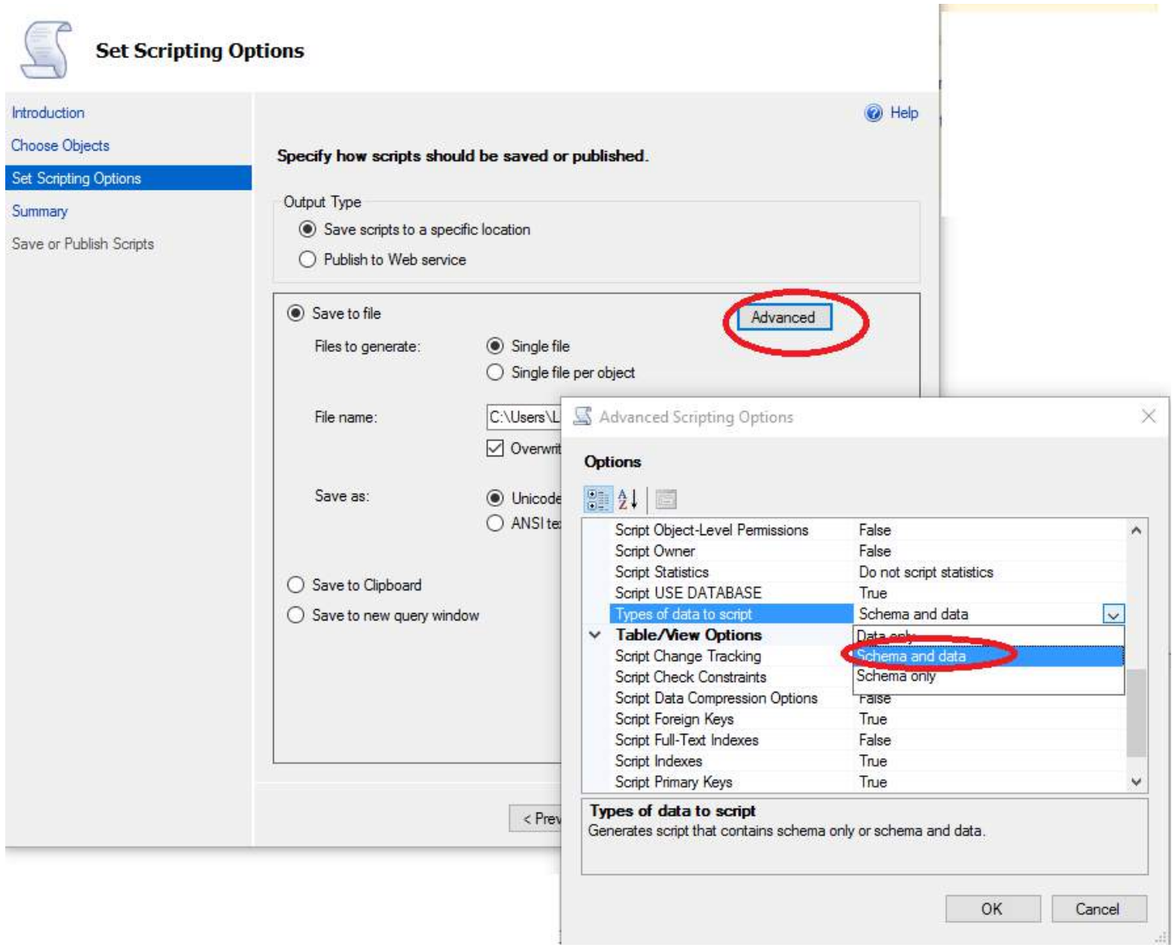
1. Click `Right Mouse` on Database you want to migrate then -> `Tasks` -> `Generate Scripts...`



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. 点击几次下一步和完成，你就应该能将数据库脚本保存为.sql文件。
4. 在您的新服务器上运行.sql 文件，您就完成了。



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.

第66章：表值参数

第66.1节：使用表值参数向表中插入多行

首先，定义一个用户自定义的表类型以供使用：

```
创建类型 names 作为表
(
FirstName varchar(10),
    LastName varchar(10)
)
GO
```

创建存储过程：

```
创建存储过程 prInsertNames
(
    @Names dbo.Names 只读 -- 注意：必须指定为只读
)
作为

插入到 dbo.TblNames (FirstName, LastName)
选择 FirstName, LastName
来自 @Names
GO
```

执行存储过程：

```
声明 @names dbo.Names
插入到 @Names 值
('Zohar', 'Peled'),
('First', 'Last')

执行 dbo.prInsertNames @Names
```


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
```

第67章：DBMAIL

第67.1节：发送简单邮件

此代码向 recipient@someaddress.com 发送一封仅包含文本的简单邮件

```
EXEC msdb.dbo.sp_send_dbmail
    @profile_name = '配置文件名称',
    @recipients = 'recipient@someaddress.com',
    @body = '这是一封由 SQL Server 发送的简单邮件。',
    @subject = '简单邮件'
```

第67.2节：发送查询结果

这会将查询 SELECT * FROM Users 的结果作为附件发送给 recipient@someaddress.com

```
EXEC msdb.dbo.sp_send_dbmail
    @profile_name = '配置文件名称',
    @recipients = 'recipient@someaddress.com',
    @query = 'SELECT * FROM Users',
    @subject = '用户列表',
    @attach_query_result_as_file = 1;
```

第67.3节：发送HTML邮件

HTML内容必须传递给 sp_send_dbmail

版本 ≥ SQL Server 2012

```
DECLARE @html VARCHAR(MAX);
SET @html = CONCAT
(
    '<html><body>',
    '<h1>Some Header Text</h1>',
    '<p>Some paragraph text</p>',
    '</body></html>'
)
```

版本 < SQL Server 2012

```
DECLARE @html VARCHAR(MAX);
SET @html =
    '<html><body>' +
    '<h1>Some Header Text</h1>' +
    '<p>Some paragraph text</p>' +
    '</body></html>';
```

然后使用@html变量作为@body参数。HTML字符串也可以直接传递给@body，
尽管这可能会使代码更难阅读。

```
EXEC msdb.dbo.sp_send_dbmail
    @recipients='recipient@someaddress.com',
    @subject = 'Some HTML content',
    @body = @html,
    @body_format = 'HTML';
```


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';
```

第68章：内存中OLTP（Hekaton）

第68.1节：声明内存优化表变量

为了更快的性能，您可以对表变量进行内存优化。以下是传统表变量的T-SQL：

```
DECLARE @tvp TABLE
(
col1   INT NOT NULL ,
    Col2   CHAR(10)
);
```

要定义内存优化变量，必须先创建一个内存优化表类型，然后声明一个来自该类型的变量：

```
CREATE TYPE dbo.memTypeTable
AS TABLE
(
Col1  INT NOT NULL INDEX ix1,
    Col2  CHAR(10)
)
WITH
    (MEMORY_OPTIMIZED = ON);
```

然后我们可以这样使用该表类型：

```
DECLARE @tvp memTypeTable
insert INTO @tvp
values (1,'1'),(2,'2'),(3,'3'),(4,'4'),(5,'5'),(6,'6')

SELECT * FROM @tvp
```

结果：

```
Col1    Col2
1       1
2       2
3       3
4       4
5       5
6       6
```

第68.2节：创建内存优化表

```
-- 创建演示数据库
CREATE DATABASE SQL2016_Demo
 ON  PRIMARY
(
名称 = N'SQL2016_Demo',
    文件名 = N'C:\Dump\SQL2016_Demo.mdf',
    大小 = 5120KB,
    文件增长 = 1024KB
 )
 日志 开启
 (
```


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
 (
```

```
名称 = N'SQL2016_Demo_log',
    文件名 = N'C:\Dump\SQL2016_Demo_log.ldf',
    大小 = 1024KB,
    文件增长 = 10%
 )
GO

使用 SQL2016_Demo
go

-- 通过 MEMORY_OPTIMIZED_DATA 类型添加文件组
修改数据库 SQL2016_Demo
    添加文件组 MemFG 包含 MEMORY_OPTIMIZED_DATA
GO


-- 向定义的文件组添加文件
修改数据库 SQL2016_Demo 添加文件
    (
名称 = MemFG_File1,
        FILENAME = N'C:\Dump\MemFG_File1' -- 你的文件路径，执行此代码前请检查目录是否存在

    )
TO FILEGROUP MemFG
GO

--对象资源管理器 -- 检查数据库是否已创建
GO

-- 创建内存优化表 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);  --或 DURABILITY = SCHEMA_ONLY
go

-- 创建内存优化表 2
创建表 MemOptTable2
(
ID INT 非空 主键 非聚集 哈希，参数为 (BUCKET_COUNT = 10000),
    FullName NVARCHAR(200) 非空,
DateAdded DATETIME 非空
) 参数为 (MEMORY_OPTIMIZED = 开启, DURABILITY = SCHEMA_AND_DATA)
GO
```

第68.3节：显示为内存优化表创建的.dll文件和表

```
SELECT
    OBJECT_ID('MemOptTable1') 作为 MemOptTable1_ObjectID,
```



```
    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') 作为 MemOptTable2_ObjectID
GO

SELECT
name,description
来自 sys.dm_os_loaded_modules
条件 name LIKE '%XTP%'
GO
```

显示所有内存优化表：

```
SELECT
name,type_desc,durability_desc,Is_memory_Optimized
来自 sys.tables
    条件 Is_memory_Optimized = 1
GO
```

第68.4节：创建内存优化的系统版本化时间表

```
创建表 [dbo].[MemOptimizedTemporalTable]
(
    [BusinessDocNo] [bigint] 不允许为空,
    [ProductCode] [int] 不允许为空,
    [UnitID] [tinyint] 不允许为空,
    [PriceID] [tinyint] 不允许为空,
    [SysStartTime] [datetime2](7) 始终生成 作为行开始时间 不允许为空,
    [SysEndTime] [datetime2](7) 始终生成 作为行结束时间 不允许为空,
   用于系统时间的期间 ([SysStartTime], [SysEndTime]),

    约束 [PK_MemOptimizedTemporalTable]  主键 非聚集索引
    (
        [BusinessDocNo] 升序,
        [ProductCode] 升序
    )
)
使用 (
    MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA, -- 内存优化选项开启
   SYSTEM_VERSIONING = ON (HISTORY_TABLE = [dbo].[MemOptimizedTemporalTable_History] ,
DATA_CONSISTENCY_CHECK = ON )
)
```

更多信息

第68.5节：内存优化表类型和临时表

例如，这是传统的基于tempdb的表类型：

```
CREATE TYPE dbo.testTableType AS TABLE
(
col1 INT NOT NULL,
   col2 CHAR(10)
);
```

要内存优化此表类型，只需添加选项memory_optimized=on，并且如果原始类型上没有索引，则添加一个索引：


```
    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 information

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);
```

全局临时表如下：

```
CREATE TABLE ##tempGlobalTabel
(
Col1   INT    NOT NULL ,
    Col2   NVARCHAR(4000)
);
```

内存优化的全局临时表：

```
CREATE TABLE dbo.tempGlobalTabel
(
Col1   INT    NOT NULL   INDEX ix NONCLUSTERED,
    Col2   NVARCHAR(4000)
)
    WITH
        (MEMORY_OPTIMIZED = ON,
         DURABILITY = SCHEMA_ONLY);
```

针对内存优化的全局临时表（##temp）：

1. 创建一个新的SCHEMA_ONLY内存优化表，其架构与全局##temp表相同
    - 确保新表至少有一个索引
2. 将所有Transact-SQL语句中对##temp的引用更改为新的内存优化表temp
3. 将代码中DROP TABLE ##temp语句替换为DELETE FROM temp，以清理内容
4. 从代码中移除CREATE TABLE ##temp 语句– 这些现在已多余

更多信息


```
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 information

第69章：时间表

第69.1节：创建时间表

```
创建表 dbo.员工
(
  [员工ID] int 不允许为空 主键聚集索引
  , [姓名] nvarchar(100) 不允许为空
  , [职位] varchar(100) 不允许为空
  , [部门] varchar(100) 不允许为空
  , [地址] nvarchar(1024) 不允许为空
  , [年薪] decimal (10,2) 不允许为空
  , [有效起始时间] datetime2 (2) 始终生成 作为行开始时间
  , [有效结束时间] datetime2 (2) 始终生成 作为行结束时间
  , 时间段 用于 系统时间 (有效起始时间, 有效结束时间)
 )
 使用 (系统版本控制 = 开启 (历史表 = dbo.员工历史));
```

插入： 在插入时，系统根据系统时钟将有效起始时间列的值设置为当前事务的开始时间（UTC时区），并将有效结束时间列的值设置为最大值9999-12-31。此操作标记该行为开放状态。

更新： 在更新时，系统将该行的先前值存储到历史表中，并根据系统时钟将有效结束时间列的值设置为当前事务的开始时间（UTC时区）。此操作标记该行已关闭，并记录该行有效的时间段。在当前表中，该行被更新为新值，系统将有效起始时间列的值设置为当前事务的开始时间（UTC时区）。当前表中更新行的有效结束时间列值仍为最大值9999-12-31。

删除： 在删除时，系统将该行的先前值存储到历史表中，并根据系统时钟将有效结束时间列的值设置为当前事务的开始时间（UTC时区）。
这将该行标记为已关闭，并记录该行之前有效的时间段。在当前表中，该行被移除。对当前表的查询将不会返回该行。只有处理历史数据的查询才会返回已关闭行的数据。

MERGE：在**MERGE**操作中，行为完全等同于执行多达三条语句（一个**INSERT**、一个**UPDATE**和/或一个**DELETE**），具体取决于**MERGE**语句中指定的操作。

提示：系统datetime2列中记录的时间基于事务本身的开始时间。例如，在单个事务中插入的所有行将在对应于SYSTEM_TIME期间开始的列中记录相同的UTC时间。

第69.2节：FOR SYSTEM_TIME ALL

返回属于当前表和历史表的行的并集。

```
SELECT * FROM Employee
    FOR SYSTEM_TIME ALL
```

第69.3节：创建内存优化的系统版本化时态表及清理SQL Server历史表

创建带有默认历史表的时态表是一个方便的选项，当您想控制命名并

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

仍依赖系统以默认配置创建历史表时。在下面的示例中，创建了一个新的系统版本化内存优化时态表，关联到一个新的基于磁盘的历史表。

```
创建模式 History
GO
创建表 dbo.Department
(
DepartmentNumber char(10) 非空 主键 非聚集,
    DepartmentName varchar(50) 非空,
ManagerID int  可空,
ParentDepartmentNumber char(10) 可空,
    SysStartTime datetime2 始终生成 作为行开始 隐藏 非空,
    SysEndTime datetime2 始终生成 作为行结束 隐藏 非空,
    PERIOD 用于 系统时间 (SysStartTime,SysEndTime)
)
使用
    (
        内存优化 = 开启, 持久性 = 模式和数据,
       系统版本控制 = 开启 ( 历史表 = History.DepartmentHistory )
    );
```

清理 SQL Server 历史表 随着时间推移，历史表可能会显著增长。由于不允许对历史表进行插入、更新或删除操作，清理历史表的唯一方法是先禁用系统版本控制：

```
ALTER TABLE dbo.Employee
设置 (系统版本控制 = 关闭);
GO
```

删除历史表中不必要的数据：

```
DELETE FROM dbo.EmployeeHistory
WHERE EndTime <= '2017-01-26 14:00:29';
```

然后重新启用系统版本控制：

```
ALTER TABLE dbo.Employee
SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = [dbo].[EmployeeHistory], DATA_CONSISTENCY_CHECK =
ON));
```

在 Azure SQL 数据库中清理历史表稍有不同，因为 Azure SQL 数据库内置了对历史表清理的支持。首先，需要在数据库级别启用时间历史保留清理：

```
ALTER DATABASE CURRENT
SET TEMPORAL_HISTORY_RETENTION ON
GO
```

然后为每个表设置保留期限：

```
ALTER TABLE dbo.Employee
SET (SYSTEM_VERSIONING = ON (HISTORY_RETENTION_PERIOD = 90 DAYS));
```

这将删除历史表中所有超过 90 天的数据。SQL Server 2016 本地数据库不支持 TEMPORAL_HISTORY_RETENTION 和 HISTORY_RETENTION_PERIOD，如果在 SQL Server 2016 本地数据库上执行上述任一查询，将会出现以下错误。

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.

对于 TEMPORAL_HISTORY_RETENTION，错误信息为：

```
Msg 102, Level 15, State 6, Line 34
```

'TEMPORAL_HISTORY_RETENTION' 附近的语法不正确。

对于 HISTORY_RETENTION_PERIOD 错误将是：

```
Msg 102，级别 15，状态 1，第 39 行
```

在 'HISTORY_RETENTION_PERIOD' 附近的语法不正确。

第 69.4 节：FOR SYSTEM_TIME BETWEEN <start_date_time> AND <end_date_time>

与上述 FOR SYSTEM_TIME FROM <start_date_time> TO <end_date_time> 描述相同，区别在于返回的行表包括在<end_date_time>端点定义的上边界上变为活动状态的行。

```
SELECT * FROM Employee
    FOR SYSTEM_TIME BETWEEN  '2015-01-01' AND '2015-12-31'
```

第 69.5 节：FOR SYSTEM_TIME FROM <start_date_time> TO <end_date_time>

返回一个表，包含在指定时间范围内所有处于活动状态的行版本的值，无论它们是否在 FROM 参数的 <start_date_time> 值之前开始活动，或在 TO 参数的 <end_date_time> 值之后停止活动。内部执行了时间表与其历史表之间的联合，并过滤结果以返回在指定时间范围内任何时间处于活动状态的所有行版本的值。包括在 FROM 端点定义的下边界上恰好变为活动状态的行，但不包括在 TO 端点定义的上边界上恰好变为活动状态的记录。

```
SELECT * FROM Employee
    FOR SYSTEM_TIME FROM '2015-01-01' TO '2015-12-31'
```

第69.6节：FOR SYSTEM_TIME 包含于 (<start_date_time> , <end_date_time>)

返回一个表，包含在由CONTAINED IN参数的两个日期时间值定义的指定时间范围内打开和关闭的所有行版本的值。恰好在下边界变为活动状态或恰好在上边界停止活动的行也包括在内。

```
SELECT * FROM Employee
    FOR SYSTEM_TIME CONTAINED IN ('2015-04-01', '2015-09-25')
```

第69.7节：如何查询时间数据？

```
SELECT * FROM Employee
    FOR SYSTEM_TIME
```


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;
```

第69.8节：返回指定时间点的实际值（FOR SYSTEM_TIME AS OF <date_time>）

返回一个表，包含在过去指定时间点实际（当前）存在的值的行。

```
SELECT * FROM Employee
    FOR SYSTEM_TIME AS  OF '2016-08-06 08:32:37.91'
```



```
        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'
```

第70章：TEMP表的使用

第70.1节：删除临时表

临时表必须具有唯一的ID（对于本地临时表，在会话内唯一；对于全局临时表，在服务器内唯一）。尝试使用已存在的名称创建表时，会返回以下错误：

```
数据库中已经存在名为 '#tempTable' 的对象。
```

如果您的查询会生成临时表，并且您想多次运行它，则需要在尝试重新生成之前删除这些表。基本语法如下：

```
drop table #tempTable
```

在表不存在时（例如语法首次运行时）执行此语法会导致另一个错误：

```
无法删除表 '#tempTable'，因为它不存在或您没有权限。
```

为避免此情况，您可以在删除表之前检查表是否已存在，方法如下：

```
IF OBJECT_ID ('tempdb..#tempTable', 'U') is not null DROP TABLE #tempTable
```

第70.2节：本地临时表

- 只要用户当前连接持续存在，该表将可用。

  用户断开连接时自动删除。

  名称应以 # 开头（#temp）

```
CREATE TABLE #LocalTempTable(
               StudentID      int,
               StudentName    varchar(50),
               StudentAddress varchar(150))
```

```
insert into #LocalTempTable values ( 1, 'Ram','India');

select * from #LocalTempTable
```

执行完所有这些语句后，如果关闭查询窗口再重新打开并尝试插入和查询，
将显示错误信息

```
“无效的对象名称 #LocalTempTable”
```

第70.3节：全局临时表

- 名称将以 ## 开头（##temp）。

  只有当用户断开所有连接时才会被删除。

  它的行为类似于永久表。

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
```

注意：这些表对数据库的所有用户可见，无论权限级别如何。


```
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.

第71章：计划任务或作业

SQL Server代理使用SQL Server来存储作业信息。作业包含一个或多个作业步骤。每个步骤包含其自己的任务，例如：备份数据库。SQL Server代理可以按计划运行作业，响应特定事件，或按需运行。

第71.1节：创建计划作业

创建作业

- 要添加作业，首先必须使用名为sp_add_job的存储过程

```
USE msdb ;
GO
EXEC dbo.sp_add_job
@job_name = N'Weekly Job' ;  -- 任务名称
```

- 然后我们必须使用名为sp_add_jobStep的存储过程添加一个作业步骤

```
EXEC sp_add_jobstep
@job_name = N'Weekly Job',  -- 要添加步骤的作业名称
@step_name = N'将数据库设置为只读',  -- 步骤名称
@subsystem = N'TSQL',  -- 步骤类型
@command = N'ALTER DATABASE SALES SET READ_ONLY',   -- 命令
@retry_attempts = 5,  -- 重试次数
@retry_interval = 5 ; -- 间隔时间（分钟）
```

- 将作业指向服务器

```
EXEC dbo.sp_add_jobserver
@job_name = N'Weekly Sales Data Backup',
@server_name = 'MyPC\data;   -- 默认是 LOCAL
GO
```

使用 SQL 创建计划

要创建计划，我们必须使用一个名为sp_add_schedule的系统存储过程

```
USE msdb
GO

EXEC sp_add_schedule
    @schedule_name = N'NightlyJobs' ,  -- 指定计划名称
    @freq_type = 4,   -- 表示作业执行时间的值（4）表示每日
    @freq_interval = 1,  -- 作业执行的天数，取决于
`freq_type` 的值。
    @active_start_time = 010000 ;   -- 作业可以开始执行的时间
GO
```

还有更多参数可以与sp_add_schedule一起使用，您可以在上面提供的链接中了解更多内容。

将计划附加到作业

要将计划附加到SQL代理作业，您必须使用名为sp_attach_schedule的存储过程

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

```sql
-- 将计划附加到作业 BackupDatabase
EXEC sp_attach_schedule
   @job_name = N'BackupDatabase',  -- 要附加的作业名称
   @schedule_name = N'NightlyJobs' ;  -- 计划名称
GO
```



```sql
-- 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
```

第72章：隔离级别和锁定

第72.1节：设置隔离级别的示例

设置隔离级别的示例：

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM Products WHERE ProductId=1;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; --恢复为默认级别
```

1. READ UNCOMMITTED - 意味着当前事务中的查询不能访问来自另一个尚未提交的事务中被修改的数据 - 不会发生脏读！但是，可能会发生不可重复读和幻读，因为数据仍可能被其他事务修改。

2. REPEATABLE READ - 意味着当前事务中的查询不能访问来自另一个尚未提交的事务中被修改的数据 - 不会发生脏读！在当前事务完成之前，其他事务不能修改当前事务正在读取的数据，从而消除了不可重复读。但是，如果另一个事务插入了新行，并且查询被执行多次，则从第二次读取开始可能出现幻读（如果新行符合查询的where条件）。

3. SNAPSHOT - 只能返回查询开始时存在的数据。确保数据的一致性。它防止脏读、不可重复读和幻读。要使用该功能，需要进行数据库配置：

```
ALTER DATABASE DBTestName SET ALLOW_SNAPSHOT_ISOLATION ON;GO;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

4. READ COMMITTED - SQL 服务器的默认隔离级别。它防止读取被另一个事务更改但尚未提交的数据。事务直到提交才允许读取。它在表上使用共享锁和行版本控制，防止脏读。它依赖于数据库配置 READ_COMMITTED_SNAPSHOT - 如果启用，则使用行版本控制。启用方法如下：

```
ALTER DATABASE DBTestName SET ALLOW_SNAPSHOT_ISOLATION ON;GO;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; --恢复默认隔离级别
```

5. SERIALIZABLE - 使用物理锁，这些锁会被获取并保持到事务结束，防止脏读、幻读和不可重复读。但是，它会影响数据库性能，因为并发事务被序列化，依次执行。

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
```


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 ;
```

第73章：排序/排序行

第73.1节：基础知识

首先，设置示例表。

```
-- 创建一个示例表
CREATE TABLE SortOrder
(
ID INT IDENTITY 主键,
    [Text] VARCHAR(256)
)
GO

-- 向表中插入行
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
```

请记住，当检索数据时，如果不指定行排序子句（ORDER BY），SQL Server 不保证任何时候的排序（列的顺序）。真的，任何时候都不保证。对此争论毫无意义，这已经在互联网上被成千上万次地证明了。

没有 ORDER BY == 没有排序。故事结束。

```
-- 可能看起来行是按标识符排序的，
-- 但实际上无法确定它是否总是有效。
-- 如果你在生产环境中就这样放着，墨菲定律会100%保证它不会正常工作。
SELECT * FROM SortOrder
GO
```

数据可以按两种方向排序：

- 升序（向上移动），使用 ASC
- 降序（向下移动），使用 DESC

```
-- 升序 - 向上
SELECT * FROM SortOrder ORDER BY ID ASC
GO

-- 升序是默认排序方式
SELECT * FROM SortOrder ORDER BY ID
GO

-- 降序 - 向下
SELECT * FROM SortOrder ORDER BY ID DESC
GO
```

当按文本列（(n)char 或 (n)varchar）排序时，请注意排序遵循排序规则。有关排序规则的更多信息，请查阅相关主题。

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 literally 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 won't.
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.

数据的排序和整理可能会消耗资源。这时，合理创建的索引就非常有用。有关索引的更多信息，请查阅相关主题。

可以伪随机化结果集中行的顺序。只需强制排序表现为非确定性即可。

```
SELECT * FROM SortOrder ORDER BY CHECKSUM(NEWID())
GO
```

排序可以保存在存储过程里，如果这是在向最终用户展示之前操作行集的最后一步，那么你应该这样做。

```
CREATE PROCEDURE GetSortOrder
AS
    SELECT *
    FROM SortOrder
    ORDER BY ID DESC
GO

EXEC GetSortOrder
GO
```

SQL Server 视图对排序的支持有限且不太可靠，但建议不要使用它。

```
/* 这可能有效也可能无效，取决于你的 SQL Server 及其更新的安装方
式 */
CREATE VIEW VwSortOrder1
AS
    SELECT TOP 100 PERCENT *
    FROM SortOrder
    ORDER BY ID DESC
GO

SELECT * FROM VwSortOrder1
GO

-- 这可以工作，但嘿.....你真的应该使用它吗？
CREATE VIEW VwSortOrder2
AS
    SELECT TOP 99999999 *
    FROM SortOrder
    ORDER BY ID DESC
GO

SELECT * FROM VwSortOrder2
GO
```

对于排序，你可以使用列名、别名或ORDER BY中的列号。

```
SELECT *
FROM SortOrder
ORDER BY [Text]

-- 新的结果集列别名为 'Msg'，可以随意用它来排序
SELECT ID, [Text] + ' (' + CAST(ID AS nvarchar(10)) + ')' AS Msg
FROM SortOrder
ORDER BY Msg

-- 如果你了解你的表，这可能很方便，但真的不适合生产环境
```


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
```

我建议不要在代码中使用数字，除非你想在执行后立刻忘记它。

第73.2节：按条件排序

如果你想按数字或字母顺序排序数据，可以简单地使用order by[column]。如果你想使用自定义层级排序，则使用case语句。

```
组别
-----
总计
年轻
中年
老年
男性
女性
```

使用基本的`order by`：

```
Select * from MyTable
Order by Group
```

返回的是字母顺序排序，这并不总是理想的：

```
组别
-----
女性
男性
中年
老年
Total
Young
```

添加一个“case”语句，按你想要的数据排序顺序分配递增的数值：

```
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
```

返回按指定顺序排列的数据：

```
Group
-----
Total
Male
```



```
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
```



```
Female
Young
MiddleAge
Old
```

第74章：权限或许可

第74.1节：简单规则

授予创建表的权限

```
USE AdventureWorks;
GRANT CREATE TABLE TO MelanieK;
GO
```

授予应用程序角色SHOWPLAN权限

```
USE AdventureWorks2012;
GRANT SHOWPLAN TO AuditMonitor;
GO
```

授予CREATE VIEW权限并带有授权选项

```
USE AdventureWorks2012;
GRANT CREATE VIEW TO CarmineEs WITH GRANT OPTION;
GO
```

授予用户在特定数据库上的所有权限

```
use YourDatabase
go
exec sp_addrolemember 'db_owner', 'UserName'
go
```


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
```

第75章：SQLCMD

第75.1节：从批处理文件或命令行调用SQLCMD.exe

```
echo off

cls

sqlcmd.exe -S "你的服务器名称" -U "sql用户名" -P "sql密码" -d "数据库名称" -Q "这里
you可以写入你的查询/存储过程"
```

像这样的批处理文件可以用来自动化任务，例如在指定时间备份数据库
（可以用任务计划程序调度），适用于无法使用代理作业的SQL Server Express版本。

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.

第76章：资源管理器

第76.1节：读取统计信息

```
select *
from sys.dm_resource_governor_workload_groups

select *
from sys.dm_resource_governor_resource_pools
```


Chapter 76: Resource Governor

Section 76.1: Reading the Statistics

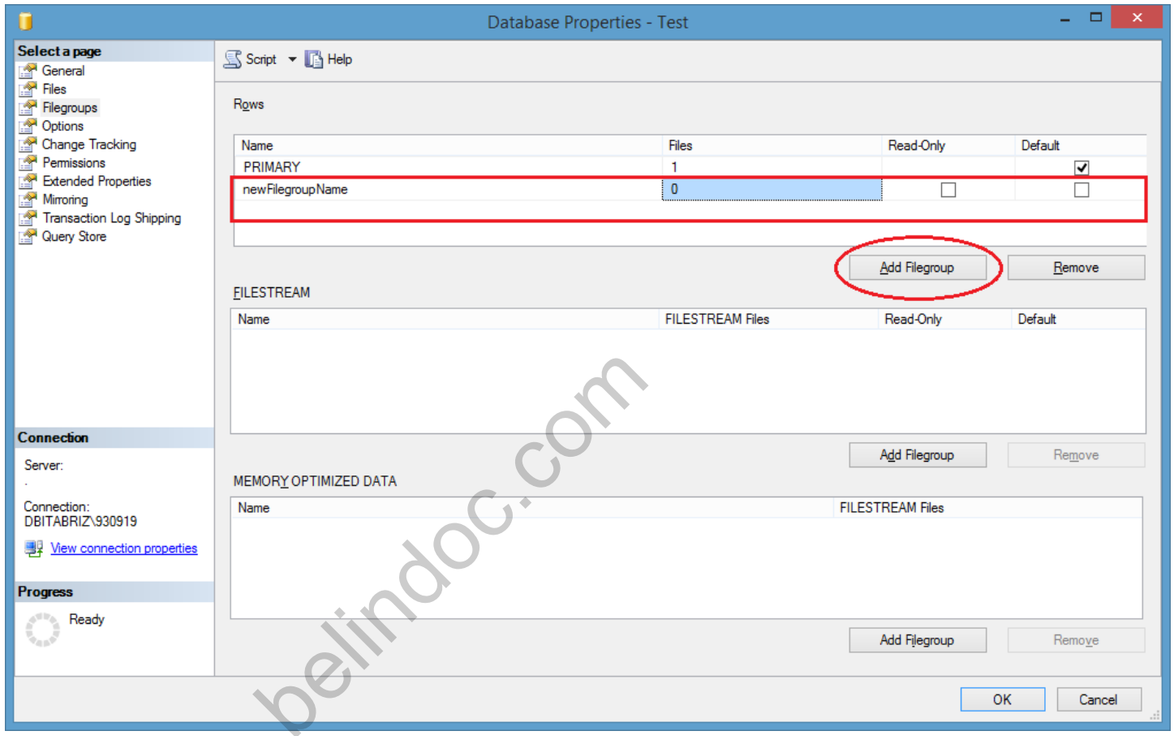
```
select *
from sys.dm_resource_governor_workload_groups

select *
from sys.dm_resource_governor_resource_pools
```

第77章：文件组

第77.1节：在数据库中创建文件组

我们可以通过两种方式创建它。第一种是通过数据库属性设计器模式：



通过 SQL 脚本：

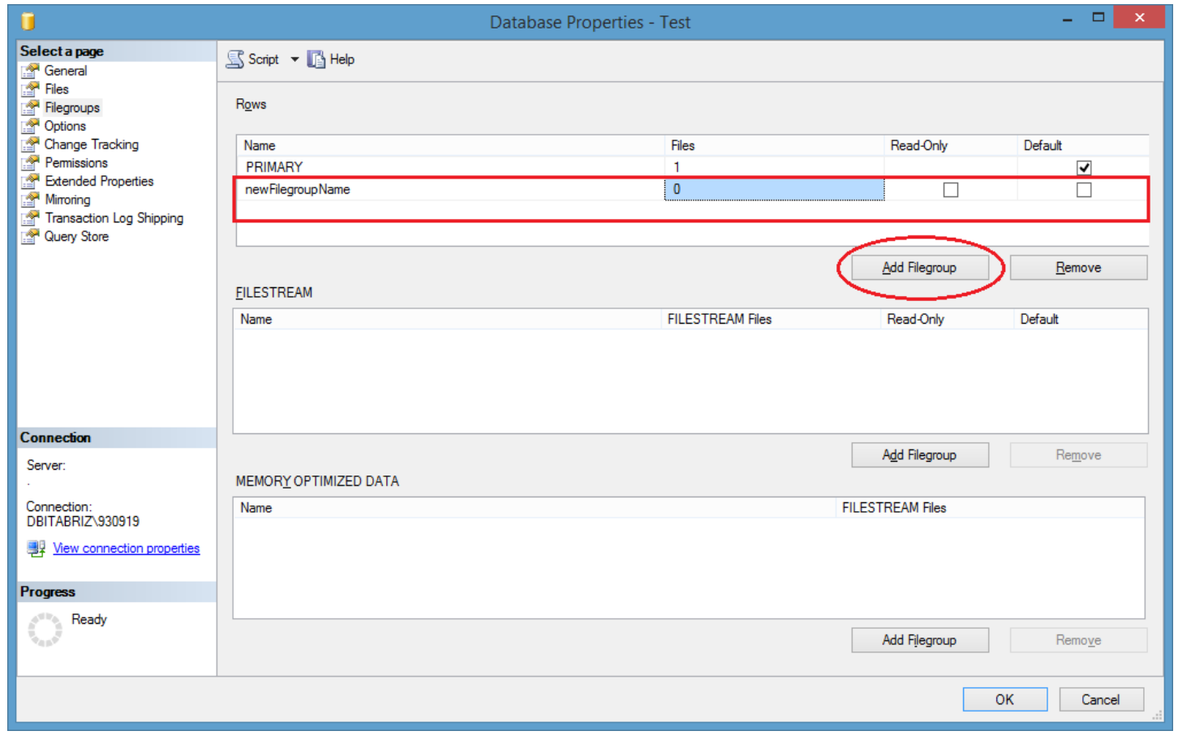
```
USE master;
GO
-- 使用默认数据创建数据库
-- 文件组和日志文件。指定
-- 增长增量和最大尺寸
-- 主要数据文件。

创建数据库 TestDB 在主文件组
(
    名称 = 'TestDB_Primary',
    文件名 = 'C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\TestDB_Prm.mdf',
    大小 = 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
),
```


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

-- 在用户定义的文件组中创建表。
USE TestDB;
Go

CREATE TABLE MyTable
(
col1 INT PRIMARY KEY,
    col2 CHAR(8)
)
ON TestDB_FG1;
GO
```



```
(
    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
```

第78章：MS SQL Server中的基本DDL操作

第78.1节：入门

本节介绍一些基本的DDL（="数据定义语言"）命令，用于创建数据库、数据库中的表、视图以及存储过程。

创建数据库

以下 SQL 命令在当前服务器上创建一个名为Northwind的新数据库，使用路径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
```

注意： 一个 T-SQL 数据库由两个文件组成，数据库文件 *.mdf 和其事务日志 *.ldf。创建新数据库时，两者都需要指定。

创建表

下面的 SQL 命令在当前数据库中使用模式 dbo 创建一个新表 Categories（你可以通过 Use <DatabaseName> 切换数据库上下文）：

```
CREATE TABLE dbo.Categories(
    CategoryID int IDENTITY NOT NULL,
    CategoryName nvarchar(15) NOT NULL,
    Description ntext NULL,
Picture image NULL,
      约束 PK_Categories 主键聚集索引
      (
CategoryID 升序
      )
      使用 (PAD_INDEX = 关闭, STATISTICS_NORECOMPUTE = 关闭, IGNORE_DUP_KEY = 关闭,
```


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
```

创建视图

以下 SQL 命令在当前数据库中创建一个新视图Summary_of_Sales_by_Year，使用模式dbo（你可以通过Use <DatabaseName>切换数据库上下文）：

```
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
```

这将连接表Orders和[Order Subtotals]，显示列ShippedDate、OrderID和Subtotal。因为在 Northwind 数据库中表[Order Subtotals]的名称中有空格，所以需要用方括号括起来。

创建存储过程

以下 SQL 命令在当前数据库中创建一个新的存储过程CustOrdersDetail，使用模式dbo（你可以通过Use <DatabaseName>切换数据库上下文）：

```
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
结束
```

该存储过程创建后，可以按如下方式调用：

```
exec dbo.MyCustOrdersDetail 10248
```

这将返回所有 @OrderId=10248（且数量默认>=0）的订单详情。或者你可以指定可选参数

```
exec dbo.MyCustOrdersDetail 10248, 10
```

这将只返回数量不少于10（或更多）的订单。


```
            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).

第79章：子查询

第79.1节：子查询

子查询是嵌套在另一个 SQL 查询中的查询。子查询也称为内查询或内选择，包含子查询的语句称为外查询或外选择。

注意

1. 子查询必须用括号括起来，
2. 子查询中不能使用 ORDER BY。
3. 子查询中不允许使用图像类型，如 BLOB、数组、文本数据类型。

子查询可以与 select、insert、update 和 delete 语句一起使用，出现在 where、from、select 子句中，配合 IN 、比较运算符等使用。

我们有一个名为 ITCompanyInNepal 的表，将在其上执行查询以展示子查询示例：

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

示例：**使用 select 语句的子查询**

配合 **IN** 运算符和 **where** 子句：

```
SELECT *
FROM ITCompanyInNepal
WHERE Headquarter IN (SELECT Headquarter
                      FROM ITCompanyInNepal
                      WHERE Headquarter = 'USA');
```

配合 比较运算符 和 where 子句

```
SELECT *
FROM ITCompanyInNepal
WHERE NumberOfEmployee < (SELECT AVG(NumberOfEmployee)
                          FROM ITCompanyInNepal
                         )
```

带有select子句

```
SELECT   CompanyName,
         CompanyAddress,
         Headquarter,
         (Select SUM(NumberOfEmployee)
         FROM ITCompanyInNepal
```


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'
```

带有插入语句的子查询

我们需要将 IndianCompany 表中的数据插入到 ITCompanyInNepal 表中。IndianCompany 表如下所示：

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
```

带有更新语句的子查询

假设所有总部位于美国的公司决定因美国公司的某些政策变动，从所有位于尼泊尔的美国公司裁员50人。

```
更新 ITCompanyInNepal
设置 NumberOfEmployee = NumberOfEmployee - 50
条件 Headquarter 在 (选择 Headquarter
                    来自 ITCompanyInNepal
                    条件 Headquarter = 'USA')
```

带有删除语句的子查询

假设所有总部位于丹麦的公司决定关闭其在尼泊尔的公司。

```
从 ITCompanyInNepal 删除
条件 Headquarter 在 (选择 Headquarter
                    来自 ITCompanyInNepal
                    条件 Headquarter = 'Denmark')
```



```
        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')
```

第80章：分页

SQL Server 各版本中的行偏移和分页

第80.1节：使用OFFSET FETCH进行分页

版本 ≥ SQL Server 2012

OFFSET FETCH子句以更简洁的方式实现分页。通过它，可以跳过N1行（在OFFSET中指定）并返回接下来的N2行（在FETCH中指定）：

```
SELECT *
FROM sys.objects
ORDER BY object_id
OFFSET 40 ROWS FETCH NEXT 10 ROWS ONLY
```

必须使用ORDER BY子句以确保结果的确定性。

第80.2节：使用内查询进行分页

在早期版本的SQL Server中，开发者必须使用双重排序结合TOP关键字来返回分页中的行：

```
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
```

内层查询将返回按name排序的前50行。然后外层查询将反转这50行的顺序，并选择前10行（这些将是反转前组中的最后10行）。

第80.3节：SQL Server各版本中的分页

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 (
```


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
```

第80.4节：SQL Server 2012/2014 使用 ORDER BY OFFSET 和 FETCH NEXT

要获取接下来的10行，只需运行此查询：

```
SELECT * FROM TableName ORDER BY id OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

使用时需要注意的要点：

- 使用 OFFSET 和 FETCH 子句时，ORDER BY 是必需的。
- OFFSET 子句必须与 FETCH 一起使用。不能单独使用 ORDER BY ... FETCH。
- TOP 不能与 OFFSET 和 FETCH 在同一查询表达式中结合使用。

第80.5节：使用带有公共表表达式的 ROW_NUMBER 进行分页

版本 ≥ SQL Server 2008

ROW_NUMBER函数可以为结果集中的每一行分配一个递增的编号。结合使用带有BETWEEN运算符的公共表表达式，可以创建结果集的“分页”。例如：第一页包含结果1-10，第二页包含结果11-20，第三页包含结果21-30，依此类推。

```
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
```

注意：无法在 WHERE 子句中使用 ROW_NUMBER，如下所示：

```
SELECT object_id,
    name,
    type,
```



```
    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
```

虽然这样会更方便，但在这种情况下，SQL Server 会返回以下错误：

> 消息 4108，级别 15，状态 1，第 6 行
>
> 窗口函数只能出现在 SELECT 或 ORDER BY 子句中。


```
    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.

第81章：聚集列存储

第81.1节：在现有表上添加聚集列存储索引

CREATE CLUSTERED COLUMNSTORE INDEX 使您能够以列格式组织表：

```
如果存在则删除表 Product
GO
创建表 Product (
    Name nvarchar(50) NOT NULL,
    Color nvarchar(15),
     尺寸 nvarchar(5) NULL,
    价格 money NOT NULL,
    数量 int
)
GO
创建聚集列存储索引 cci 在 Product 上
```

第81.2节：重建聚集列存储索引

如果有大量已删除的行，可以重建聚集列存储索引：

```
ALTER INDEX cci 在 Products 上
重建分区 = 全部
```

重建聚集列存储索引将“重新加载”当前表中的数据到新表中，并重新应用压缩，删除已删除的行等。

您可以重建一个或多个分区。

第81.3节：带聚集列存储索引的表

如果想让表以列存储格式而非行存储格式组织，在表定义中添加索引 cci 聚集列存储：

```
如果存在则删除表 Product
GO
创建表 Product (
    ProductID int,
    Name nvarchar(50) 不允许为空,
    Color nvarchar(15),
     Size nvarchar(5) 允许为空,
     Price money 不允许为空,
    Quantity int,
     索引 cci 聚集列存储
)
```

列存储表更适合预期进行全表扫描和报表的表，而行存储表更适合读取或更新较小数据集的表。

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
)
```

COLUMNSTORE 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.

第82章：Parsename

‘object_name’	object_piece
要检索指定对象部分的对象名称。object_name 是 sysname。此参数是一个可选限定的对象名称。如果对象名称的所有部分都被限定，则该名称可以包含四部分：服务器名称、数据库名称、所有者名称和对象名称。	要返回的对象部分。object_piece 是 int 类型，可以取以下值：1 = 对象名称 2 = 架构名称 3 = 数据库名称 4 = 服务器名称

第82.1节：PARSENAME

```
声明 @ObjectName nVarChar(1000)
设置 @ObjectName = 'HeadOfficeSQL1.Northwind.dbo.Authors'

选择
 PARSENAME(@ObjectName, 4) 作为服务器
,PARSENAME(@ObjectName, 3) 作为数据库
,PARSENAME(@ObjectName, 2) 作为所有者
,PARSENAME(@ObjectName, 1) 作为对象
```

返回结果：

服务器	数据库
总部SQL1	Northwind

所有者对象	
dbo	作者

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

第83章：在Windows上安装SQL Server

第83.1节：介绍

以下是SQL Server的可用版本，由版本矩阵说明：

- Express：入门级免费数据库。包含核心关系数据库管理系统功能。磁盘大小限制为10G。适合开发和测试。
- 标准版：标准授权版本。包含核心功能和商业智能功能。
- 企业版：功能齐全的 SQL Server 版本。包括高级安全性和数据仓库功能。
- 开发者版：包含企业版的所有功能且没有限制，且仅限于开发用途免费下载和使用。

下载/获取 SQL Server 后，安装程序通过 SQLSetup.exe 执行，该程序提供图形界面（GUI）和命令行两种形式。

通过任一方式安装时，您需要指定产品密钥并运行一些初始配置，包括启用功能、独立服务以及为每个服务设置初始参数。可以随时通过运行 SQLSetup.exe 程序（命令行或 GUI 版本）启用额外的服务和功能。

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.

第 84 章：查询分析

第 84.1 节：扫描与查找

查看执行计划时，您可能会看到 SQL Server 决定执行查找（Seek）或扫描（Scan）。

当 SQL Server 知道需要去哪里并且只抓取特定项时，就会发生查找。这通常发生在查询中使用了良好的过滤条件，例如 where name = 'Foo'。

扫描是指 SQL Server 不确切知道所需数据的位置，或者决定如果选择的数据量足够大，扫描比查找更高效。

查找通常更快，因为它们只抓取数据的一个子集，而扫描则选择大部分数据。

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.

第85章：查询提示

第85.1节：JOIN提示

当你连接两个表时，SQL Server查询优化器（QO）可以选择不同类型的连接用于查询：

- HASH连接
- LOOP连接
- MERGE连接

QO会探索执行计划并选择连接表的最优操作符。然而，如果你确定知道最优的连接操作符，可以指定应使用哪种JOIN。内部LOOP连接将强制QO选择嵌套循环连接来连接两个表：

```
select top 100 *
from Sales.Orders o
    inner loop join Sales.OrderLines ol
    on o.OrderID = ol.OrderID
```

内部MERGE连接将强制使用MERGE连接操作符：

```
select top 100 *
from Sales.Orders o
    inner merge join Sales.OrderLines ol
    on o.OrderID = ol.OrderID
```

内部HASH连接将强制使用HASH连接操作符：

```
select top 100 *
from Sales.Orders o
    内连接哈希连接 Sales.OrderLines ol
    on o.OrderID = ol.OrderID
```

第85.2节：GROUP BY提示

当你使用GROUP BY子句时，SQL Server查询优化器（QO）可以选择不同类型的分组操作符：

- 哈希聚合，创建用于分组条目的哈希映射
- 流聚合，适用于预排序的输入

如果你知道哪种聚合操作是最优的，可以显式要求QO选择其中一种聚合操作符。使用OPTION (ORDER GROUP)时，QO将始终选择流聚合，并在输入未排序时在流聚合前添加排序操作符：

```
select OrderID, AVG(Quantity)
from Sales.OrderLines
group by OrderID
OPTION (ORDER GROUP)
```

使用OPTION (HASH GROUP)时，QO将始终选择哈希聚合：

```
select OrderID, AVG(Quantity)
```


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
选项（哈希组）
```

第85.3节：快速行提示

指定查询针对快速检索前number_rows行进行了优化。该值为非负整数。在返回前number_rows行后，查询继续执行并生成完整的结果集。

```
select OrderID, AVG(Quantity)
from Sales.OrderLines
按订单号分组
选项（快速20）
```

第85.4节：UNION提示

当你对两个查询结果使用UNION操作符时，查询优化器（QO）可以使用以下操作符来创建两个结果集的联合：

- 合并（Union）
- 连接（Union）
- 哈希匹配（Union）

你可以使用OPTION()提示显式指定应使用的操作符：

```
选择订单号，订单日期，预计交货日期，备注
来自销售.订单
条件订单日期>DATEADD(天, -1, getdate())
UNION
select PurchaseOrderID as OrderID, OrderDate, ExpectedDeliveryDate, Comments
from Purchasing.PurchaseOrders
where OrderDate > DATEADD(day, -1, getdate())
OPTION(HASH UNION)
-- 或 OPTION(CONCAT UNION)
-- 或 OPTION(MERGE UNION)
```

第85.5节：MAXDOP 选项

指定使用此选项的查询的最大并行度。

```
SELECT OrderID,
    AVG(Quantity)
FROM Sales.OrderLines
GROUP BY OrderID
OPTION (MAXDOP 2);
```

此选项会覆盖 sp_configure 和资源管理器(Resource Governor)的 MAXDOP 配置选项。如果 MAXDOP 设置为零，则服务器会选择最大并行度。

第85.6节：索引提示

索引提示用于强制查询使用特定索引，而不是让 SQL Server 的查询优化器选择其认为最优的索引。在某些情况下，通过指定查询必须使用的索引可以获得性能提升。通常，SQL Server 的查询优化器会选择最适合查询的最佳索引，但由于统计信息缺失/过时或特定需求，您可以强制指定索引。


```
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'
```



```sql
SELECT *
FROM mytable WITH (INDEX (ix_date))
WHERE field1 > 0
    AND CreationDate > '20170101'
```

第86章：查询存储

第86.1节：在数据库上启用查询存储

可以使用以下命令在数据库上启用查询存储：

```
ALTER DATABASE tpch SET QUERY_STORE = ON
```

SQL Server/Azure SQL数据库将收集已执行查询的信息，并在
sys.query_store 视图中提供相关信息：

- 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

第86.2节：获取SQL查询/计划的执行统计信息

以下查询将返回有关查询、其执行计划及其平均统计信息，包括持续时间、CPU时间、物理和逻辑IO读取的相关信息。

```
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
```

第86.3节：从查询存储中删除数据

如果您想从查询存储中删除某个查询或查询计划，可以使用以下命令：

```
EXEC sp_query_store_remove_query 4;
EXEC sp_query_store_remove_plan 3;
```

这些存储过程的参数是从系统视图中检索到的查询/计划ID。

您也可以仅删除特定计划的执行统计信息，而不从存储中删除该计划：

```
EXEC sp_query_store_reset_exec_stats 3;
```

此过程提供的参数是计划ID。

第86.4节：查询的强制计划

SQL 查询优化器将为某个查询选择他能找到的最佳执行计划。如果你能找到某个查询的最优执行计划，

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

你可以使用以下存储过程强制查询优化器始终使用该计划：

```
EXEC sp_query_store_unforce_plan @query_id, @plan_id
```

从此时起，查询优化器将始终使用为该查询提供的计划。

如果你想取消此绑定，可以使用以下存储过程：

```
EXEC sp_query_store_force_plan @query_id, @plan_id
```

从此时起，查询优化器将再次尝试找到最佳执行计划。

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.

第87章：按页查询结果

第87.1节：Row_Number()

```
SELECT Row_Number() OVER(ORDER BY 用户名) As 行号, 用户名首字母, 用户姓氏
FROM 用户表
```

由此将生成一个包含行号字段的结果集，您可以使用该字段进行分页。

```
SELECT *
FROM
    ( SELECT Row_Number() OVER(ORDER BY 用户名) As 行号, 用户名首字母, 用户姓氏
      FROM 用户表
    ) As 行结果
WHERE 行号 Between 5 AND 10
```


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
```

第88章：模式

第88.1节：目的

模式指的是特定的数据库表及其相互关系。它提供了数据库构建的组织蓝图。实施数据库模式的额外好处是，模式可以作为限制或授予对数据库中特定表访问权限的方法。

第88.2节：创建模式

```
CREATE SCHEMA dvr AUTHORIZATION 所有者
    CREATE TABLE sat_Sales (来源 int, 成本 int, 部件ID int)
    GRANT SELECT ON SCHEMA :: dvr TO  用户1
    DENY SELECT ON SCHEMA :: dvr to 用户2
GO
```

第88.3节：修改模式

```
ALTER SCHEMA dvr
    TRANSFER dbo.tbl_Staging;
GO
```

这将把 tbl_Staging 表从 dbo 模式转移到 dvr 模式

第88.4节：删除模式

```
DROP SCHEMA dvr
```


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
```

第89章：数据库备份与恢复

参数	详细信息
database	要备份或恢复的数据库名称
backup_device	用于备份或恢复数据库的设备，如 {DISK 或 TAPE}。可以用逗号 (,)
with_options	执行操作时可用的各种选项，例如格式化存放备份将被放置的位置或使用替换选项恢复数据库。

第89.1节：无选项的基本磁盘备份

下面的命令将'Users'数据库备份到'D:\DB_Backup'文件。最好不要指定扩展名。

```
BACKUP DATABASE Users TO DISK = 'D:\DB_Backup'
```

第89.2节：无选项的基本磁盘恢复

下面的命令从'D:\DB_Backup'文件恢复'Users'数据库。

```
RESTORE DATABASE Users FROM DISK = 'D:\DB_Backup'
```

第89.3节：使用REPLACE的数据库恢复

当你尝试从另一台服务器恢复数据库时，可能会遇到以下错误：

> 错误3154：备份集包含的数据库备份与现有数据库不同。

在这种情况下，您应该使用 WITH REPLACE 选项，用备份中的数据库替换现有数据库：

```
还原数据库 WWIDW
从磁盘 = 'C:\Backup\WideWorldImportersDW-Full.bak'
使用 REPLACE
```

即使在这种情况下，你也可能会收到错误，提示某些路径上的文件无法找到：

> 消息 3156，级别 16，状态 3，第 1 行 文件 'WWI_Primary' 无法还原到 'D:\Data\WideWorldImportersDW.mdf'。请使用 WITH MOVE 指定文件的有效位置。

此错误可能是因为你的文件未放置在新服务器上存在的相同文件夹路径中。
在这种情况下，你应该将单个数据库文件移动到新位置：

```
还原数据库 WWIDW
从磁盘 = 'C:\Backup\WideWorldImportersDW-Full.bak'
使用 REPLACE,
移动 'WWI_Primary' 到 'C:\Data\WideWorldImportersDW.mdf',
移动 'WWI_UserData' 到 'C:\Data\WideWorldImportersDW_UserData.ndf',
移动 'WWI_Log' 到 'C:\Data\WideWorldImportersDW.ldf',
移动 'WWIDW_InMemory_Data_1' 到 'C:\Data\WideWorldImportersDW_InMemory_Data_1'
```

通过此语句，你可以替换数据库，并将所有数据库文件移动到新位置。

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.

第90章：事务处理

参数	详细信息
transaction_name	用于命名您的事务——配合参数[with mark]使用非常有用，该参数允许有意义的日志记录——区分大小写（！）

可以向[transaction_name]添加标记['description']，并将在日志中存储该标记

第90.1节：带错误处理的基本事务框架

```
BEGIN TRY -- 开始错误处理
    BEGIN TRANSACTION; -- 从这里开始事务（修改）尚未最终提交
        -- 开始您的语句
        select 42/0 as ANSWER  -- 简单的带错误的SQL查询
        -- 结束您的语句
    COMMIT TRANSACTION; -- 提交所有事务（修改）
END TRY   -- 结束错误处理——跳转到结尾
BEGIN CATCH -- 如果发生错误则执行此处
        ROLLBACK TRANSACTION; -- 撤销所有事务（修改）
-- 组合一些信息作为查询
    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;  -- 错误处理的最后一行
GO -- 执行前面的代码
```


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
```

第91章：本地编译模块（Hekaton）

第91.1节：本地编译存储过程

在本地编译的存储过程中，T-SQL代码被编译为dll并作为本地C代码执行。要创建一个本地编译存储过程，您需要：

- 使用标准的CREATE PROCEDURE语法
- 在存储过程定义中设置NATIVE_COMPILATION选项
- 在存储过程定义中使用SCHEMABINDING选项
- 在存储过程定义中定义EXECUTE AS OWNER选项

您需要使用BEGIN ATOMIC块代替标准的BEGIN END块：

```
BEGIN ATOMIC
   WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT, LANGUAGE='us_english')
   -- T-Sql代码写在这里
结束
```

示例：

```
CREATE PROCEDURE usp_LoadMemOptTable (@maxRows INT, @FullName NVARCHAR(200))
WITH
    NATIVE_COMPILATION,
    SCHEMABINDING,
    EXECUTE AS OWNER
作为
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
```

第91.2节：本地编译的标量函数

本地编译函数中的代码将被转换为C代码并编译为dll。要创建本地编译的标量函数，您需要：

- 使用标准的CREATE FUNCTION语法
- 在函数定义中设置NATIVE_COMPILATION选项
- 在函数定义中使用SCHEMABINDING选项

您需要使用BEGIN ATOMIC块代替标准的BEGIN END块：

```
BEGIN ATOMIC
   WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT, LANGUAGE='us_english')
   -- T-Sql代码写在这里
结束
```


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
```

示例：

```
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);
结束

-- 使用示例：
SELECT dbo.udfMultiply(10, 12)
```

第91.3节：本地内联表值函数

本地编译的表值函数返回表作为结果。原生编译函数中的代码将被转换为C代码并编译为dll。2016版本仅支持内联表值函数。要创建本地表值函数，您需要：

- 使用标准的CREATE FUNCTION语法
- 在函数定义中设置NATIVE_COMPILATION选项
- 在函数定义中使用SCHEMABINDING选项

您需要使用BEGIN ATOMIC块代替标准的BEGIN END块：

```
BEGIN ATOMIC
   WITH (TRANSACTION ISOLATION LEVEL=SNAPSHOT, LANGUAGE='us_english')
   -- T-Sql代码写在这里
结束
```

示例：

```
CREATE FUNCTION [dbo].[udft_NativeGetBusinessDoc]
(
   @RunDate VARCHAR(25)
)
返回表
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
```


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
```

```
);
```


);

第92章：空间数据

有2种空间数据类型

几何 用于平面表面的X/Y坐标系

地理 用于曲面（地球）的纬度/经度坐标系。曲面有多种投影，因此每个地理空间数据必须让SQL Server知道使用哪种投影。常用的空间参考ID（SRID）是4326，单位为公里。这是大多数网络地图中使用的默认SRID

第92.1节：点（POINT）

创建一个点。根据所使用的类，这将是几何点或地理点。

参数	详细信息
纬度或X	表示正在生成的点的x坐标的浮点表达式
经度或Y	是表示正在生成的点的y坐标的浮点表达式
字符串	几何/地理形状的知名文本（WKT）
二进制	几何/地理形状的知名二进制（WKB）
空间参考ID（SRID）	表示几何/地理实例的空间参考ID（SRID）的整数表达式您希望返回的

```
--显式构造函数
DECLARE @gm1 GEOMETRY = GEOMETRY::Point(10,5,0)

DECLARE @gg1 GEOGRAPHY = GEOGRAPHY::Point(51.511601,-0.096600,4326)

--隐式构造函数（使用WKT - Well Known Text）
DECLARE @gm1 GEOMETRY = GEOMETRY::STGeomFromText('POINT(5 10)', 0)

DECLARE @gg1 GEOGRAPHY= GEOGRAPHY::STGeomFromText('POINT(-0.096600 51.511601)', 4326)

--隐式构造函数（使用WKB - Well Known Binary）
DECLARE @gm1 GEOMETRY = GEOMETRY::STGeomFromWKB(0x0101000000000000000000144000000000000002440, 0)

DECLARE @gg1 GEOGRAPHY= GEOGRAPHY::STGeomFromWKB(0x01010000005F29CB10C7BAB8BFEACC3D247CC14940,
4326)
```


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)
```

第93章：动态SQL

第93.1节：执行作为字符串提供的SQL语句

在某些情况下，您需要执行放在字符串中的SQL查询。EXEC、EXECUTE或系统存储过程sp_executesql可以执行任何作为字符串提供的SQL查询：

```
sp_executesql N'SELECT * FROM sys.objects'
-- 或者
sp_executesql @stmt = N'SELECT * FROM sys.objects'
-- 或者
EXEC sp_executesql N'SELECT * FROM sys.objects'
-- 或者
EXEC('SELECT * FROM sys.columns')
-- 或者
EXECUTE('SELECT * FROM sys.tables')
```

此存储过程将返回与作为语句文本提供的SQL查询相同的结果集。sp_executesql可以执行作为字符串字面量、变量/参数甚至表达式提供的SQL查询：

```
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
```

你需要使用QUOTENAME函数来转义@table变量中的特殊字符。如果没有此函数，当@table变量包含空格、括号或任何其他特殊字符时，将会出现语法错误。

第93.2节：以不同用户身份执行动态SQL

你可以使用AS USER = '数据库用户名称'以不同用户身份执行SQL查询

```
EXEC(N'SELECT * FROM product') AS USER = 'dbo'
```

SQL查询将以dbo数据库用户身份执行。所有适用于dbo用户的权限检查都将应用于该SQL查询。

第93.3节：使用动态SQL的SQL注入

动态查询是

```
SET @sql = N'SELECT COUNT(*) FROM AppUsers WHERE Username = ''' + @user + ''' AND Password = ''' +
@pass + ''''
EXEC(@sql)
```

如果用户变量的值是 myusername'' OR 1=1 --，则将执行以下查询：

```
SELECT COUNT(*)
FROM AppUsers
WHERE Username = 'myusername' OR 1=1 --' AND Password = ''
```

变量 @username 值末尾的注释符将注释掉查询的后续部分，条件 1=1 将被计算。应用程序检查该查询是否至少返回一个用户时，将返回大于 0 的计数，登录将成功。

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.

使用这种方法，攻击者即使不知道有效的用户名和密码，也能登录应用程序。

第 93.4 节：带参数的动态 SQL

为了避免注入和转义问题，动态SQL查询应使用参数执行，例如：

```
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
```

第二个参数是查询中使用的参数列表及其类型，紧接着是将用作参数值的变量。

sp_executesql 会转义特殊字符并执行 SQL 查询。

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.

第94章：动态数据掩码

第94.1节：在列上添加默认掩码

如果你在列上添加默认掩码，SELECT 语句中将显示掩码而非实际值：

```
ALTER TABLE  Company
ALTER COLUMN Postcode ADD MASKED WITH (FUNCTION = 'default()')
```

第94.2节：使用动态数据掩码掩盖电子邮件地址

如果你有电子邮件列，可以使用 email() 掩码进行掩盖：

```
ALTER TABLE  Company
ALTER COLUMN Email ADD MASKED WITH (FUNCTION = 'email()')
```

当用户尝试从 Company 表中选择电子邮件时，他将看到类似以下的值：

mXXX@XXXX.com

zXXX@XXXX.com

rXXX@XXXX.com

第94.3节：在列上添加部分掩码

您可以在列上添加部分掩码，显示字符串开头和结尾的几个字符
并在中间字符位置显示掩码：

```
ALTER TABLE  Company
ALTER COLUMN Phone ADD MASKED WITH (FUNCTION = 'partial(5,"XXXXXXX",2)')
```

在 partial 函数的参数中，您可以指定显示开头多少个字符，显示结尾多少个字符，以及中间显示的掩码模式。

当用户尝试从 Company 表中选择电子邮件时，他将看到类似以下的值：

(381)XXXXXXX39

(360)XXXXXXX01

(415)XXXXXXX05

第94.4节：使用

random() 掩码显示范围内的随机值

随机掩码将显示指定范围内的随机数，而非实际值：

```
ALTER TABLE  Product
ALTER COLUMN Price ADD MASKED WITH (FUNCTION = 'random(100,200)')
```

请注意，在某些情况下，显示的值可能与列中的实际值匹配（如果随机选择的数字与单元格中的值相同）。

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

值相同）。

第94.5节：控制谁可以查看未掩码数据

您可以使用以下语句授予特权用户查看未掩码值的权限：

```
GRANT UNMASK TO MyUser
```

如果某个用户已经拥有未掩码权限，您可以撤销该权限：

```
REVOKE UNMASK TO MyUser
```


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
```

第95章：使用

SQLCMD导出txt文件中的数据

第95.1节：在命令提示符下使用SQLCMD

命令结构为

sqlcmd -S yourservername\instancename -d database_name -o outputfilename_withpath -Q "your select query"

开关参数如下

`-S`

用于服务器名和实例名

`-d`

用于源数据库

`-o`

用于目标输出文件（将创建输出文件）

`-Q`

用于查询以获取数据

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

第96章：公共语言运行时集成

第96.1节：在数据库上启用CLR

CLR存储过程默认未启用。您需要运行以下查询以启用CLR：

```
sp_configure '显示高级选项', 1;
GO
RECONFIGURE;
GO
sp_configure '启用clr', 1;
GO
重新配置;
GO
```

此外，如果某些CLR模块需要外部访问，您应在数据库中将TRUSTWORTHY属性设置为ON：

```
ALTER DATABASE MyDbWithClr SET TRUSTWORTHY ON
```

第96.2节：添加包含Sql CLR模块的.dll

用.Net语言编写的存储过程、函数、触发器和类型存储在.dll文件中。一旦您创建了包含CLR存储过程的.dll文件，应将其导入到SQL Server中：

```
创建程序集 MyLibrary
从 'C:\lib\MyStoredProcedures.dll'
    使用权限集 = EXTERNAL_ACCESS
```

权限集默认是 Safe，意味着.dll中的代码不需要权限即可访问外部资源（例如文件、网站、其他服务器），且不会使用可访问内存的本机代码。

权限集 = EXTERNAL_ACCESS 用于标记包含将访问外部

资源代码的程序集。

你可以在 sys.assemblies 视图中找到当前 CLR 程序集文件的信息：

```
SELECT *
从 sys.assemblies asms
哪里 is_user_defined = 1
```

第 96.3 节：在 SQL Server 中创建 CLR 函数

如果你已经创建了 .Net 函数，将其编译成 .dll，并导入到 SQL Server 作为程序集，你可以创建引用该程序集内函数的用户自定义函数：

```
创建函数 dbo.TextCompress(@input nvarchar(max))
返回 varbinary(max)
作为外部 名称 MyLibrary.[Name.Space.ClassName].TextCompress
```

你需要指定函数名和输入参数及返回值签名，需与 .Net 函数匹配。在 AS EXTERNAL NAME 子句中，你需要指定程序集名称、命名空间/类名，其中包含此函数

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

函数所在的位置以及包含将作为函数暴露的代码的方法名称。

您可以使用以下查询来查找有关 CLR 函数的信息：

```
SELECT * FROM dbo.sysobjects WHERE TYPE ='FS'
```

第 96.4 节：在 SQL Server 中创建 CLR 用户自定义类型

如果您创建了表示某些用户自定义类型的 .Net 类，将其编译成 .dll，并作为程序集导入到 SQL server 中，您可以创建引用该类的用户自定义函数：

```
CREATE TYPE dbo.Point
EXTERNAL NAME MyLibrary.[Name.Space.Point]
```

您需要指定将在 T-SQL 查询中使用的类型名称。在 EXTERNAL NAME 子句中，您需要指定程序集名称、命名空间和类名。

第 96.5 节：在 SQL Server 中创建 CLR 存储过程

如果您在某个类中创建了 .Net 方法，将其编译成 .dll，并作为程序集导入到 SQL Server，您可以创建引用该程序集中的方法的用户自定义存储过程：

```
CREATE PROCEDURE dbo.DoSomethng(@input nvarchar(max))
AS EXTERNAL NAME MyLibrary.[Name.Space.ClassName].DoSomething
```

您需要指定过程的名称和带有与 .Net 方法匹配的输入参数的签名。在 ASEXTERNAL NAME 子句中，您需要指定程序集名称、过程所在的命名空间/类名以及包含将作为过程公开的代码的方法名称。

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.

第97章：定界特殊字符和保留字

第97.1节：基本方法

SQL Server 转义保留字的基本方法是使用方括号（[和]）。例如，Description 和 Name 是保留字；但是，如果有对象使用这两个作为名称，使用的语法是：

```
SELECT [Description]
FROM   dbo.TableName
WHERE  [Name] = 'foo'
```

SQL Server 唯一的特殊字符是单引号 '，转义方法是将其重复使用。例如，要在同一表中查找名称为 *O'Shea* 的记录，使用以下语法：

```
SELECT [Description]
FROM   dbo.TableName
WHERE  [Name] = 'O''Shea'
```


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'
```

第98章：DBCC

第98.1节：DBCC 语句

DBCC 语句作为 SQL Server 的数据库控制台命令。要获取指定 DBCC 命令的语法信息，请使用 DBCC HELP (...) 语句。

以下示例返回所有可用帮助的 DBCC 语句：

```
DBCC HELP ('?');
```

以下示例返回 DBCC CHECKDB 语句的选项：

```
DBCC HELP ('CHECKDB');
```

第 98.2 节：DBCC 维护命令

DBCC 命令使用户能够维护数据库空间、清理缓存、收缩数据库和表。

示例如下：

```
DBCC DROPCLEANBUFFERS
```

从缓冲池中移除所有干净的缓冲区，以及从列存储对象池中移除列存储对象。

```
DBCC FREEPROCCACHE
-- 或
DBCC FREEPROCCACHE (0x060006001ECA270EC0215D05000000000000000000000000);
```

清除计划缓存中的所有 SQL 查询。每个新计划将被重新编译：您可以指定计划句柄、查询句柄以清理特定查询计划或 SQL 语句的计划。

```
DBCC FREESYSTEMCACHE ('ALL', myresourcepool);
-- 或者
DBCC FREESYSTEMCACHE;
```

清理系统创建的所有缓存条目。它可以清理所有或某些指定资源池中的条目（上例中的**myresourcepool**）

```
DBCC FLUSHAUTHCACHE
```

清空包含登录信息和防火墙规则的数据库身份验证缓存。

```
DBCC SHRINKDATABASE (MyDB [, 10]);
```

将数据库 MyDB 缩小到 10%。第二个参数是可选的。您可以使用数据库 ID 代替名称。

```
DBCC SHRINKFILE (DataFile1, 7);
```

压缩当前数据库中名为 DataFile1 的数据文件。目标大小为 7 MB（此参数可选）。

```
DBCC CLEANTABLE (AdventureWorks2012,'Production.Document', 0)
```


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)
```

从指定表中回收空间

第98.3节：DBCC 验证语句

DBCC 命令允许用户验证数据库的状态。

```
ALTER TABLE Table1 WITH NOCHECK ADD CONSTRAINT chkTab1 CHECK (Col1 > 100);
GO
DBCC CHECKCONSTRAINTS(Table1);
--OR
DBCC CHECKCONSTRAINTS ('Table1.chkTable1');
```

约束以 nocheck 选项添加，因此不会检查现有数据。DBCC 将触发约束检查。

以下 DBCC 命令检查数据库、表或目录的完整性：

```
DBCC CHECKTABLE tablename1 | tableid
DBCC CHECKDB databasename1 | dbid
DBCC CHECKFILEGROUP filegroup_name | filegroup_id | 0
DBCC CHECKCATALOG databasename1 | database_id1 | 0
```

第98.4节：DBCC 信息性语句

DBCC 命令可以显示有关数据库对象的信息。

```
DBCC PROCCACHE
```

以表格格式显示有关过程缓存的信息。

```
DBCC OUTPUTBUFFER ( session_id [ , request_id ])
```

返回指定 session_id（和可选的

request_id）当前的输出缓冲区，格式为十六进制和 ASCII。

```
DBCC INPUTBUFFER ( session_id [ , request_id ])
```

显示从客户端发送到 Microsoft SQL Server 实例的最后一条语句。

```
DBCC SHOW_STATISTICS ( table_or_indexed_view_name , column_statistic_or_index_name)
```

第98.5节：DBCC 跟踪命令

SQL Server 中的跟踪标志用于修改 SQL Server 的行为，开启或关闭某些功能。DBCC 命令可以控制跟踪标志：

以下示例全局开启跟踪标志3205，并为当前会话开启3206：

```
DBCC TRACEON (3205, -1);
DBCC TRACEON (3206);
```

以下示例全局关闭跟踪标志3205，并为当前会话关闭3206：

```
DBCC TRACEON (3205, -1);
```


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);
```

以下示例显示跟踪标志2528和3205的状态：

```
DBCC TRACESTATUS (2528, 3205);
```



```
DBCC TRACEON (3206);
```

The following example displays the status of trace flags 2528 and 3205:

```
DBCC TRACESTATUS (2528, 3205);
```

第99章：批量导入

第99.1节：批量插入

BULK INSERT 命令可用于将文件导入到 SQL Server 中：

```
BULK INSERT People
FROM 'f:\orders\people.csv'
```

BULK INSERT 命令会将文件中的列映射到目标表中的列。

第 99.2 节：带选项的 BULK INSERT

您可以使用 WITH 子句中的不同选项自定义解析规则：

```
BULK INSERT PeopleF
ROM 'f:\orders\people.csv'  WITH
(  CODEPAGE = '65001',  FIELDTER
          MINATOR =',',  ROWTERMIN
          ATOR ='  );
```

在此示例中，CODEPAGE 指定源文件为 UTF-8 文件，终止符为逗号和换行符。

第 99.3 节：使用 OPENROWSET(BULK) 读取文件全部内容

您可以使用 OPENROWSET(BULK) 函数读取文件内容并将内容存储到某个表中：

```
INSERT INTO myTable(content)
   SELECT BulkColumn
          FROM OPENROWSET(BULK N'C:\Text1.txt', SINGLE_BLOB) AS Document;
```

SINGLE_BLOB 选项将把文件的全部内容作为单个单元读取。

第99.4节：使用 OPENROWSET(BULK) 和格式文件读取文件

您可以使用 FORMATFILE 选项定义将要导入文件的格式：

```
INSERT INTO mytable
SELECT a.*
FROM OPENROWSET(BULK 'c:est\values.txt',
   FORMATFILE = 'c:est\values.fmt') AS a;
```

格式文件 format_file.fmt 描述了 values.txt 中的列：

```
9.0
2
1SQLCHAR  0  10 ""        1  ID                SQL_Latin1_General_Cp437_BIN
2SQLCHAR  0  40 "\r"      2  Description       SQL_Latin1_General_Cp437_BIN
```


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
```

第99.5节：使用OPENROWSET(BULK)读取json文件

您可以使用OPENROWSET读取文件内容并将其传递给其他函数以解析结果。

下面的示例展示了如何使用OPENROWSET(BULK)读取整个JSON文件内容，然后将BulkColumn提供给OPENJSON函数，解析JSON并返回列：

```
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
```


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
```

第100章：服务代理

第100.1节：基础知识

服务代理是一种基于两个（或多个）实体之间异步通信的技术。服务代理包括：消息类型、合同、队列、服务、路由，以及至少一个实例端点

更多信息：https://msdn.microsoft.com/en-us/library/bb522893.aspx

第100.2节：在数据库上启用服务代理

```
ALTER DATABASE [MyDatabase] SET ENABLE_BROKER WITH ROLLBACK IMMEDIATE;
```

第100.3节：在数据库上创建基本的服务代理结构（单数据库通信）

```
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]

)

创建服务 TargetService
在队列 TargetQueue 上
(
     [//调用/合同]

)

授予发送权限于服务::[InitiatorService] 给所有用户
执行

授予发送权限于服务::[TargetService] 给所有用户
执行
```

单数据库通信不需要路由。

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.

第100.4节：如何通过
服务代理发送基本通信

本演示将使用本文件其他部分创建的服务代理结构。
提到的部分名为**3. 在数据库上创建基本服务代理结构（单数据库通信）**。

```
使用 [MyDatabase]

声明 @ch 唯一标识符 = NEWID()
声明 @msg XML

开始对话会话 @ch
    来自服务 [InitiatorService]
    发送至服务 '目标服务'
    基于合同 [//调用/合同]
    加密方式 = 关闭; -- 更多可能的选项

        SET @msg = (
                    SELECT 'HelloThere' "elementNum1"
                    FOR XML PATH(''), ROOT('ExampleRoot'), ELEMENTS XSINIL, TYPE
                    );

SEND ON CONVERSATION @ch MESSAGE TYPE [//initiator] (@msg);
END CONVERSATION @ch;
```

在此对话之后，您的消息将位于 TargetQueue 中

第100.5节：如何自动接收来自 TargetQueue 的对话

本演示将使用本文件其他部分创建的服务代理结构。
提到的部分称为**3. 在数据库上创建基本的服务代理结构（单数据库通信）**。

首先我们需要创建一个能够从队列读取和处理数据的存储过程

```
USE [MyDatabase]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

CREATE PROCEDURE [dbo].[p_RecieveMessageFromTargetQueue]

    作为
    开始

    declare
    @message_body xml,
    @message_type_name nvarchar(256),
    @conversation_handle uniqueidentifier,
    @messagetypename nvarchar(256);
```


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
         结束

      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

          结束


      IF (@messagetypename = 'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog')
         BEGIN
             END CONVERSATION @conversation_handle;
         END

  COMMIT TRANSACTION
 END


结束
```

第二步：允许你的目标队列自动运行你的存储过程：

```
使用 [MyDatabase]

ALTER QUEUE [dbo].[TargetQueue] WITH STATUS = ON , RETENTION = OFF ,
ACTIVATION
 (  STATUS = ON , --激活状态
    PROCEDURE_NAME = dbo.p_RecieveMessageFromTargetQueue , --存储过程名称
    MAX_QUEUE_READERS = 1 , --读取者数量
    EXECUTE AS SELF  )
```



```
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  )
```

第101章：权限与安全

第101.1节：为用户分配对象权限

在生产环境中，最好保护您的数据，并且只允许通过存储过程对其进行操作。这意味着您的应用程序不能直接对数据执行CRUD操作，从而避免潜在的问题。分配权限是一项耗时、繁琐且通常很繁重的任务。因此，通常更容易利用每个SQL Server数据库中包含的INFORMATION_SCHEMA架构中蕴含的（相当强大的）功能。

与其逐个为用户分配权限，不如运行下面的脚本，复制输出结果，然后在查询窗口中执行。

```
SELECT 'GRANT EXEC ON core.' + r.ROUTINE_NAME + ' TO ' + <MyDatabaseUsername>
FROM INFORMATION_SCHEMA.ROUTINES r
WHERE r.ROUTINE_CATALOG = '<MyDataBaseName>'
```


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>'
```

第102章：数据库权限

第102.1节：更改权限

```
授予 SELECT 权限于 [dbo].[someTable] 给 [aUser];

撤销 SELECT 权限于 [dbo].[someTable] 给 [aUser];
--REVOKE SELECT [dbo].[someTable] FROM [aUser]; 等同于此

拒绝 SELECT 权限于 [dbo].[someTable] 给 [aUser];
```

第102.2节：创建用户

```
--隐式将此用户映射到与用户名相同的登录名
创建用户 [aUser];

--显式映射用户应关联的登录名
创建用户 [aUser] 对应 登录 [aUser];
```

第102.3节：创建角色

```
创建角色 [myRole];
```

第102.4节：更改角色成员资格

```
-- SQL 2005及以上版本
执行 sp_addrolemember @rolename = 'myRole', @membername = 'aUser';
执行 sp_droprolemember @rolename = 'myRole', @membername = 'aUser';

-- SQL 2008+
ALTER ROLE [myRole] ADD MEMBER [aUser];
ALTER ROLE [myRole] DROP MEMBER [aUser];
```

注意：角色成员可以是任何数据库级主体。也就是说，您可以将一个角色作为成员添加到另一个角色中。此外，添加/删除角色成员是幂等的。也就是说，无论当前角色成员状态如何，尝试添加/删除操作都会导致成员在角色中存在/不存在（分别如此）。

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.

第103章：行级安全性

第103.1节：RLS过滤谓词

Sql Server 2016及以上版本和Azure Sql数据库允许您使用某些谓词自动过滤select语句返回的行。此功能称为行级安全性。

首先，您需要一个表值函数，其中包含描述允许用户从某个表读取数据的条件的谓词：

```
如果存在则删除函数 dbo.pUserCanAccessCompany
GO
创建函数

dbo.pUserCanAccessCompany(@CompanyID int)

    返回表
    使用 SCHEMABINDING
作为返回 (
    选择 1 作为 canAccess 条件

    转换(SESSION_CONTEXT(N'CompanyID') 为 int) = @CompanyID

)
```

在此示例中，谓词表示只有 SESSION_CONTEXT 中的值与输入参数匹配的用户才能访问该公司。您可以添加任何其他条件，例如检查当前用户的数据库角色或 database_id 等。

> 上面的代码大部分是模板，您将复制粘贴。这里唯一会变化的是谓词的名称和参数以及 WHERE 子句中的条件。现在您创建一个安全策略，将此谓词应用于某个表。

现在您可以创建一个安全策略，将谓词应用于某个表：

```
创建 安全策略 dbo.CompanyAccessPolicy
    添加 过滤谓词 dbo.pUserCanAccessCompany(CompanyID) 于 dbo.Company
    使用 (状态=开启)
```

此安全策略将谓词分配给公司表。每当有人尝试从 Company 表读取数据时，安全策略会对每一行应用谓词，将 CompanyID 列作为谓词的参数传递，谓词将评估该行是否应包含在 SELECT 查询的结果中。

第 103.2 节：修改 RLS 安全策略

安全策略是一组与表关联的谓词，可以一起管理。您可以添加或删除谓词，或开启/关闭整个策略。

您可以在现有的安全策略中为表添加更多谓词。

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy
    ADD FILTER PREDICATE dbo.pUserCanAccessCompany(CompanyID) ON dbo.Company
```


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
```

您可以从安全策略中删除某些谓词：

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy
    DROP FILTER PREDICATE ON dbo.Company
```

您可以禁用安全策略

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy WITH ( STATE = OFF );
```

您可以启用已禁用的安全策略：

```
ALTER SECURITY POLICY dbo.CompanyAccessPolicy WITH ( STATE = ON );
```

第103.3节：使用RLS阻止谓词防止更新

行级安全允许您定义一些谓词来控制谁可以更新表中的行。首先，您需要定义一些表值函数，表示将控制访问策略的谓词。

```
创建函数
dbo.pUserCanAccessProduct(@CompanyID int)
RETURNS TABLE
使用 SCHEMABINDING
作为返回 (
    选择 1 作为 canAccess 条件
    CAST(SESSION_CONTEXT(N'CompanyID') as int) = @CompanyID
)
```

在此示例中，谓词表示只有 SESSION_CONTEXT 中的值与输入参数匹配的用户才能访问该公司。您可以添加任何其他条件，例如检查当前用户的数据库角色或 database_id 等。

> 上面的代码大部分是模板，您将复制粘贴。这里唯一会变化的是谓词的名称和参数以及 WHERE 子句中的条件。现在您创建一个安全策略，将此谓词应用于某个表。

现在我们可以创建安全策略，使用谓词阻止对产品表的更新，如果表中的CompanyID列不满足谓词条件。

```
CREATE SECURITY POLICY dbo.ProductAccessPolicy
ADD BLOCK PREDICATE dbo.pUserCanAccessProduct(CompanyID) ON dbo.Product
```

该谓词将应用于所有操作。如果您只想将谓词应用于某些操作，可以写成如下形式：

```
CREATE SECURITY POLICY dbo.ProductAccessPolicy
ADD BLOCK PREDICATE dbo.pUserCanAccessProduct(CompanyID) ON dbo.Product AFTER INSERT
```

您可以在阻止谓词定义后添加的可能选项有：

```
[ { AFTER { INSERT | UPDATE } }
| { BEFORE { UPDATE | DELETE } } ]
```


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 } } ]
```

第104章：加密

可选参数	详细信息
WITH PRIVATE KEY	对于 CREATE CERTIFICATE，可以指定私钥：(FILE='D:\Temp\CertTest\private.pvk', DECRYPTION BY PASSWORD = 'password');

第 104.1 节：通过证书加密

```
CREATE CERTIFICATE My_New_Cert
FROM FILE = 'D:\Temp\CertTest\certificateDER.cer'
GO
```

创建证书

```
SELECT EncryptByCert(Cert_ID('My_New_Cert'),
'这段文本将被加密') encryption_test
```

通常，您会使用对称密钥进行加密，该密钥会被证书中的非对称密钥（公钥）加密。

另外，请注意加密长度受密钥长度限制，否则返回 NULL。
微软写道："限制如下：512 位 RSA 密钥最多可加密 53 字节，1024 位密钥最多可加密 117 字节，2048 位密钥最多可加密 245 字节。"

EncryptByAsymKey 具有相同的限制。对于 UNICODE，每个字符 16 位（2 字节），因此 1024 位密钥最多可加密 58 字符。

第104.2节：数据库加密

```
使用 TDE
创建数据库加密密钥
使用算法 = AES_256
通过服务器证书加密 My_New_Cert
GO

修改数据库 TDE
设置加密开启
GO
```

这使用了"透明数据加密"（TDE）

第104.3节：对称密钥加密

```
-- 创建密钥并用证书保护它
创建对称密钥 My_Sym_Key
使用算法 = AES_256
通过证书加密 My_New_Cert;
GO

-- 打开密钥
打开对称密钥 My_Sym_Key
通过证书解密 My_New_Cert;

-- 加密
SELECT EncryptByKey(Key_GUID('SSN_Key_01'), '这段文本将被加密');
```


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');
```

第104.4节：通过密码短语加密

```
SELECT EncryptByPassphrase('MyPassPhrase', '这段文本将被加密')
```

这也会加密，但使用的是密码短语，而不是非对称（证书）密钥或显式对称密钥。

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.

第105章：幻读

在数据库系统中，隔离性决定了事务完整性对其他用户和系统的可见性，因此它定义了一个操作所做更改何时以及如何对其他操作可见。当你获取尚未提交到数据库的数据时，可能会发生幻读。

第105.1节：隔离级别 READ UNCOMMITTED

在示例数据库上创建示例表

```
CREATE TABLE [dbo].[Table_1](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [title] [varchar](50) NULL,
 约束 [PK_Table_1] 主键聚集索引
(
    [Id] 升序
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) 在 [主键]
```

现在打开第一个查询编辑器（在数据库上），插入以下代码并执行（**不要触碰 --rollback**）
在这种情况下，你插入了一行数据到数据库，但**未**提交更改。

```
开始事务

插入到 Table_1 值('标题 1')

选择 * 从 [Test].[dbo].[Table_1]

--回滚
```

现在打开第二个查询编辑器（在数据库上），插入以下代码并执行

```
开始事务

设置事务隔离级别为 读未提交

选择 * 从 [Test].[dbo].[Table_1]
```

你可能会注意到，在第二个编辑器中可以看到第一笔事务中新创建的行（但未提交）。
在第一个编辑器中执行回滚（选择 rollback 词并执行）。

```
-- 回滚第一笔事务
回滚
```

在第二个编辑器上执行查询，你会看到记录消失（幻读），这是因为你告诉第二个事务获取所有行，包括未提交的行。

当你更改隔离级别时会发生这种情况

```
设置事务隔离级别为 READ UNCOMMITTED
```


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
```

第106章：Filestream

FILESTREAM通过将varbinary(max)二进制大对象（BLOB）数据作为文件存储在NTFS文件系统上，将SQL Server数据库引擎与NTFS文件系统集成。Transact-SQL语句可以插入、更新、查询、搜索和备份FILESTREAM数据。Win32文件系统接口提供对数据的流式访问。

第106.1节：示例

来源：MSDN https://technet.microsoft.com/en-us/library/bb933993(v=sql.105).aspx

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

第107章：bcp（批量复制程序）
实用工具

批量复制程序实用工具（bcp）在Microsoft SQL Server实例与用户指定格式的数据文件之间批量复制数据。bcp实用工具可用于将大量新行导入SQL Server表，或将表中的数据导出到数据文件中。

第107.1节：无格式文件导入数据示例（使用本机格式）

```
REM 截断表(用于测试)
SQLCMD -Q "截断表 TestDatabase.dbo.myNative;"

REM 导入数据
bcp TestDatabase.dbo.myNative IN D:\BCP\myNative.bcp -T -n

REM 审查结果
SQLCMD -Q "SELECT * FROM TestDatabase.dbo.myNative;"
```


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;"
```

第108章：SQL Server通过不同版本（2000 - 2016）的演变

我从2004年开始使用SQL Server。我从2000版本开始，现在准备使用SQL Server 2016。我创建了表、视图、函数、触发器、存储过程，并编写了许多SQL查询，但没有使用后续版本中的许多新功能。我在谷歌上搜索过，但遗憾的是，没有在一个地方找到所有功能。因此，我从不同来源收集并验证了这些信息，并放在这里。我只是为所有版本（从2000年开始到20）添加了高级别的信息

第108.1节：SQL Server版本2000 - 2016

以下功能是SQL Server 2000相较于其前一版本新增的：

1.新增数据类型（BIGINT、SQL_VARIANT、TABLE）
2.引入了Instead of和for触发器，作为DDL的改进。
3.级联参照完整性。
4.XML支持
5.用户定义函数和分区视图。
6. 索引视图（允许对包含计算列的视图建立索引）。

以下功能是从2005版本相较于之前版本新增的：

1. TOP子句的增强，增加了“WITH TIES”选项。
2. 数据操作命令（DML）和OUTPUT子句，用于获取插入和删除的值
3. PIVOT和UNPIVOT操作符。
4. 使用TRY/CATCH块进行异常处理
5. 排名函数
6. 公共表表达式（CTE）
7. 公共语言运行时（集成.NET语言以构建存储过程、触发器、函数等对象）
8. 服务代理（以松耦合方式处理发送方和接收方之间的消息）
9. 数据加密（原生支持对用户定义数据库中存储数据的加密）
10. SMTP邮件
11. HTTP端点（使用简单的T-SQL语句创建端点，使对象可通过互联网访问）
12. 多活动结果集（MARS）。这允许单个客户端的持久数据库连接在每个连接上有多个活动请求。
13.SQL Server 集成服务（将作为主要的 ETL（提取、转换和加载）工具使用）
14.分析服务和报表服务的增强功能。
15. 表和索引分区。允许根据由分区函数（PARTITION FUNCTION）指定的分区边界对表和索引进行分区，且通过分区方案（PARTITION SCHEME）将各个分区映射到文件组。

以下是2008版本相较于之前版本新增的功能：

1. 现有日期和时间数据类型的增强
2. 新增函数如– SYSUTCDATETIME() 和 SYSDATETIMEOFFSET()
3. 备用列– 用于节省大量磁盘空间。
4. 大型用户定义类型（最大可达2 GB）
5. 引入了将表数据类型传递给存储过程和函数的新功能
6. 用于插入、更新和删除操作的新MERGE命令

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. 新的HierarchyID数据类型
8. 空间数据类型 - 用于表示任何几何对象的物理位置和形状。
9. 使用GROUPING SETS实现更快的查询和报表 - GROUP BY子句的扩展。
10. FILESTREAM存储选项的增强

以下功能是2008 R2版本相较于其前一版本新增的：

1. PowerPivot – 用于处理大型数据集。
2.报表生成器3.0
3.云端支持
4. StreamInsight
5.主数据服务
6.SharePoint集成
7.DACPAC（数据层应用组件包）
8.SQL Server 2008其他功能的增强

以下功能是2012版本相较于其前一版本新增的：

1.列存储索引 - 减少大型查询的I/O和内存使用。
2. 分页 - 分页可以通过使用“OFFSET”和“FETCH”命令来实现。
3. 包含数据库–适用于定期数据迁移的强大功能。
4. AlwaysOn 可用性组
5. Windows 服务器核心支持
6. 用户定义的服务器角色
7. 大数据支持
8. PowerView
9. SQL Azure 增强功能
10. 表格模型（SSAS）
11. DQS 数据质量服务
12. 文件表——对 2008 年引入的 FILESTREAM 功能的增强。
13. 错误处理的增强，包括 THROW 语句14. SQL Server Manag
ement Studio 调试的改进 a. SQL Server 2012 引入了更多控制断点的选项。b. 调试模式窗口的改进

c. IntelliSense 的增强——如插入代码片段。

以下是 2014 版本相较于之前版本新增的功能：

1. 内存中OLTP引擎–性能提升高达20倍。
2. AlwaysOn增强功能
3. 缓冲池扩展
4. 混合云功能
5. 列存储索引的增强（如可更新的列存储索引）
6. 查询处理增强（如并行SELECT INTO）
7. Office 365的Power BI集成
8. 延迟持久性
9. 数据库备份的增强

以下功能是2016版本相较于之前版本新增的：

1. Always Encrypted（始终加密）- 旨在保护静态或传输中的数据。
2. 实时运营分析
3. 将PolyBase集成到SQL Server
4. 原生JSON支持

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.查询存储
6.AlwaysOn增强
7.增强的内存内OLTP
8.多个TempDB数据库文件
9.伸缩数据库
10.行级安全
11.内存增强

> SQL Server 2016中的T-SQL增强或新增功能

1.带分区的TRUNCATE TABLE

2.如果存在则删除（DROP IF EXISTS）

3.STRING_SPLIT和STRING_ESCAPE函数

4.ALTER TABLE现在可以在表保持联机状态时修改多列，使用WITH (ONLINE = ON | OFF)。

5.DBCC CHECKDB、DBCC CHECKTABLE和DBCC CHECKFILEGROUP的MAXDOP

6. ALTER DATABASE SET AUTOGROW_SINGLE_FILE

7. ALTER DATABASE SET AUTOGROW_ALL_FILES

8.COMPRESS和DECOMPRESS函数

9.FORMATMESSAGE语句

10.2016版本引入了8个更多的SERVERPROPERTY属性

a. 实例默认数据路径

b. 实例默认日志路径

c. 产品构建

d. 产品构建类型

e. 产品主版本号

f. 产品次版本号

g. 产品更新级别

h. 产品更新参考

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

第109章：SQL Server管理工作室（SSMS）

SQL Server管理工作室（SSMS）是用于管理和维护SQL Server及SQL数据库的工具。

SSMS由微软免费提供。

SSMS文档 可用。

第109.1节：刷新IntelliSense缓存

当对象被创建或修改时，它们不会自动对IntelliSense可用。要使它们对IntelliSense可用，必须刷新本地缓存。

在查询编辑器窗口中，按Ctrl+Shift+R，或从菜单中选择编辑|IntelliSense|刷新本地缓存。

完成后，自上次刷新以来的所有更改将对IntelliSense可用。

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.

第110章：管理Azure SQL数据库

第110.1节：查找Azure SQL数据库的服务层信息

Azure SQL数据库有不同的版本和性能层级。

您可以使用以下语句查找作为Azure服务运行的SQL数据库的版本、版本（基础版、标准版或高级版）和服务目标（S0、S1、P4、P11等）：

```
select @@version
SELECT DATABASEPROPERTYEX('Wwi', 'EDITION')
SELECT DATABASEPROPERTYEX('Wwi', 'ServiceObjective')
```

第110.2节：更改Azure SQL数据库的服务层

您可以使用ALTER DATABASE语句对Azure SQL数据库进行升级或降级：

```
ALTER DATABASE WWI
MODIFY (SERVICE_OBJECTIVE = 'P6')
-- 或者
ALTER DATABASE CURRENT
MODIFY (SERVICE_OBJECTIVE = 'P2')
```

如果您尝试在当前数据库的服务级别更改仍在进行时更改服务级别，将会收到以下错误：

> 消息 40802，级别 16，状态 1，第 1 行 服务器 '.....' 和数据库 '.....' 上的服务目标分配已在进行中。请等待数据库的服务目标分配状态标记为“已完成”。

在过渡期结束后，请重新运行您的 ALTER DATABASE 语句。

第 110.3 节：Azure SQL 数据库的复制

您可以在另一个 Azure SQL 服务器上创建同名数据库的辅助副本，使本地数据库成为主数据库，并开始将数据从主数据库异步复制到新的辅助副本。

```
ALTER DATABASE <<mydb>>
ADD SECONDARY ON SERVER <<secondaryserver>>
WITH ( ALLOW_CONNECTIONS = ALL )
```

目标服务器可能位于另一个数据中心（可用于地理复制）。如果目标服务器上已存在同名数据库，命令将失败。该命令在托管将成为主数据库的本地数据库的服务器上的主数据库中执行。当 ALLOW_CONNECTIONS 设置为 ALL（默认设置为 NO）时，辅助副本将是只读数据库，允许所有具有适当权限的登录连接。

可以使用以下命令将辅助数据库副本提升为主数据库：

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
```

您可以删除辅助服务器上的辅助数据库：

```
ALTER DATABASE <<mydb>>
在服务器上移除辅助<<testsecondaryserver>>
```

第110.4节：在弹性池中创建Azure SQL数据库

您可以将Azure SQL数据库放入SQL弹性池中：

```
CREATE DATABASE wwi
( SERVICE_OBJECTIVE = ELASTIC_POOL ( name = mypool1 ) )
```

您可以创建现有数据库的副本并将其放入某个弹性池中：

```
CREATE DATABASE wwi
AS COPY OF myserver.WideWorldImporters
( SERVICE_OBJECTIVE = ELASTIC_POOL ( name = mypool1 ) )
```



```
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 ) )
```

第111章：系统数据库 - TempDb

第111.1节：识别TempDb使用情况

以下查询将提供有关TempDb使用情况的信息。通过分析计数，您可以识别出影响TempDb的因素

```
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

第111.2节：TempDB数据库详情

以下查询可用于获取TempDB数据库详情：

```
USE [MASTER]
SELECT * FROM sys.databases WHERE database_id = 2
```

OR

```
USE [MASTER]
SELECT * FROM sys.master_files WHERE database_id = 2
```

借助以下DMV，您可以检查您的会话使用了多少TempDb空间。此查询在调试TempDb问题时非常有用

```
SELECT * FROM sys.dm_db_session_space_usage WHERE session_id = @@SPID
```


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
```

附录A：Microsoft SQL Server 管理工作室快捷键

第A.1节：快捷键示例

1. 使用当前连接打开新的查询窗口（ Ctrl + N ）
2. 在打开的标签页之间切换（ Ctrl + 标签页 ）
3. 显示/隐藏结果窗格（ Ctrl + R ）
4. 执行高亮查询（ Ctrl + E ）
5. 将选中文本转换为大写或小写（ Ctrl + Shift + U , Ctrl + Shift + L ）
6. 智能感知列表成员和完整单词（ Ctrl + 空格 , 标签页 ）
7. 转到行（ Ctrl + G ）
8. 在 SQL Server Management Studio 中关闭标签页（ Ctrl + F4 ）

第A.2节：菜单激活快捷键

1. 移动到SQL Server管理工作室菜单栏（ ALT ）
2. 激活工具组件的菜单（ ALT + 连字符 ）
3. 显示上下文菜单（ SHIFT + F ）
4. 显示“新建文件”对话框以创建文件（ CTRL + N ）
5. 显示“打开项目”对话框以打开现有项目（ CTRL + SHIFT + 0 ）
6. 显示“添加新项”对话框以向当前项目添加新文件（ CTRL + SHIFT + A ）
7. 显示“添加现有项”对话框以向当前项目添加现有文件 （CTRL + SHIFT + A ）
8. 显示“查询设计器”（ CTRL + SHIFT + Q ）
9. 关闭菜单或对话框，取消操作（ ESC ）

A.3节：自定义键盘快捷键

进入“工具” -> “选项”。进入“环境” -> “键盘” -> “查询快捷键”

右侧可以看到SSMS中默认的一些快捷键。如果需要添加新的快捷键，只需点击“存储过程”列下的任意一栏。

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)

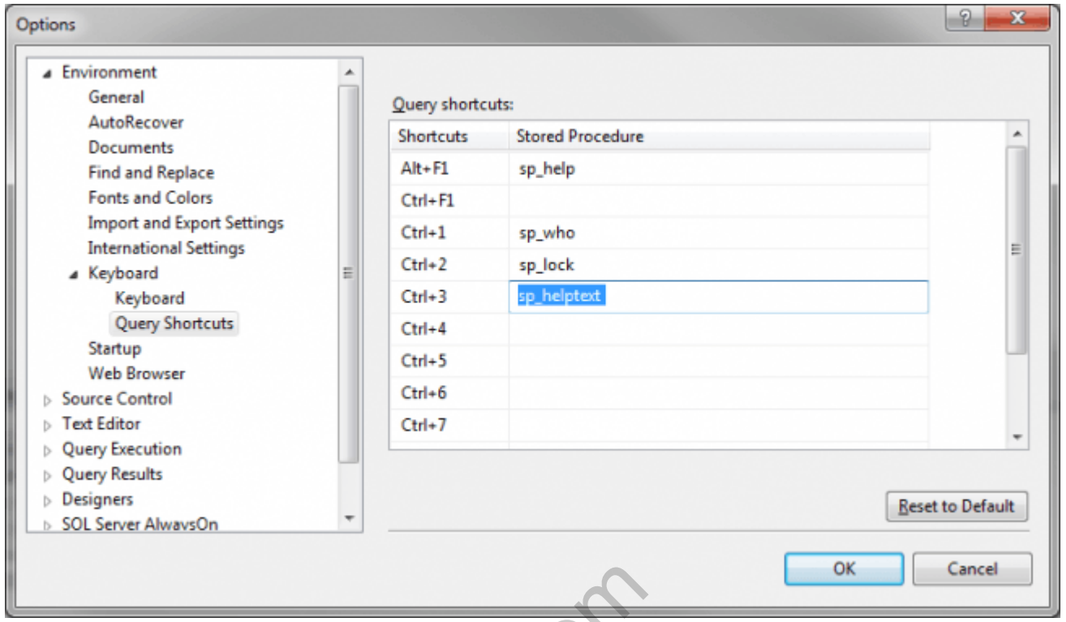
Section A.2: Menu Activation Keyboard Shortcuts

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.



点击确定。现在请进入查询窗口，选择存储过程，然后按下CTRL+3，将显示存储过程的结果。

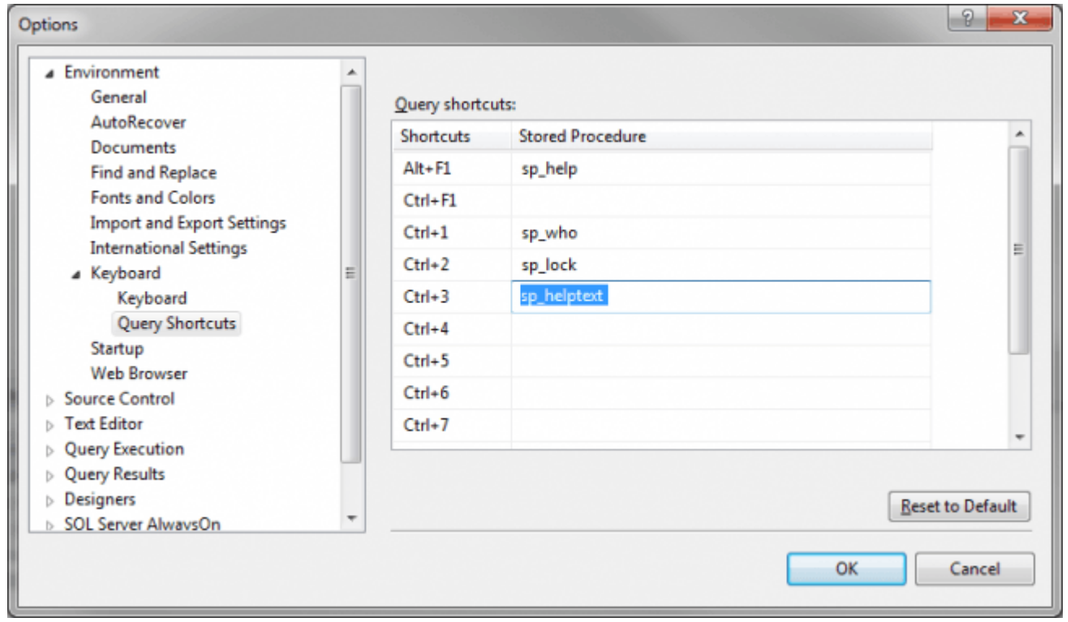
```
usp_Get_SalesOrderDetail
```

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

如果需要在选择表时按下CTRL+5（你可以选择任意键）来选择该表中的所有记录，可以按以下方式设置快捷键。



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.

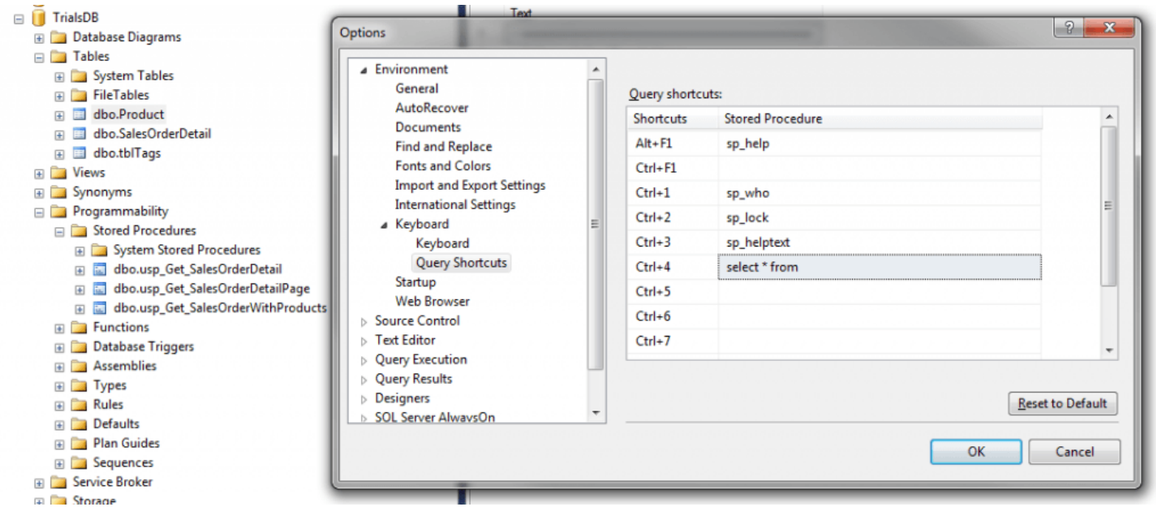
```
usp_Get_SalesOrderDetail
```

100 %

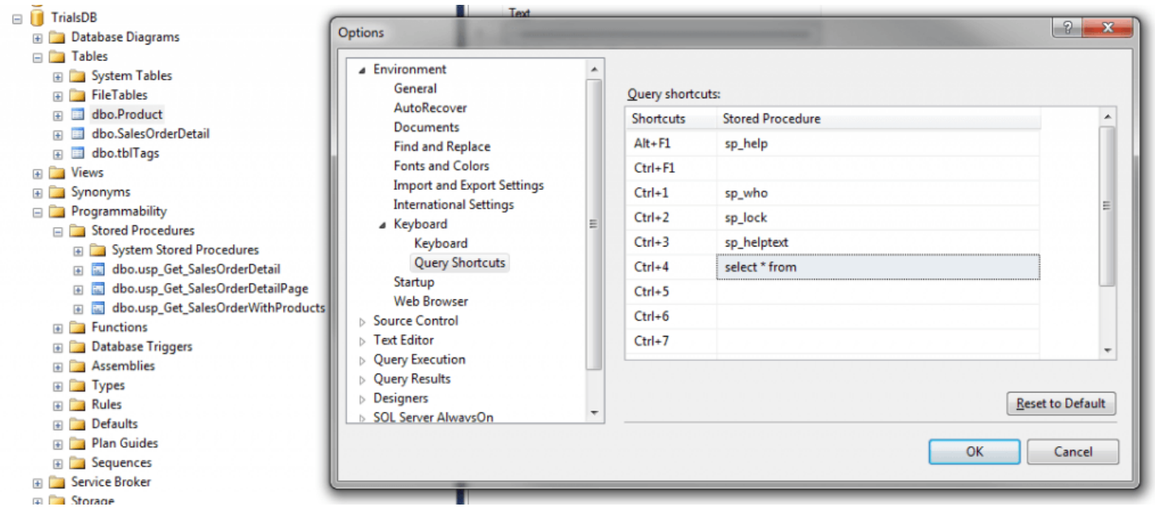
Results | Messages

	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.



现在继续，在查询窗口选择表名并按下CTRL+4（我们选择的键），将显示查询结果。



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.

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

5arx	第101章
阿比拉什·R·万卡亚拉	第1章和第16章
阿比谢克·贾因	第1章
亚当·波拉德	第9章
艾哈迈德·阿加扎德	第59章、第64章和第69章
阿卡什	第11章
阿科	第76章
阿克谢·阿南德	第43章和第59章
alalp	第1章和第38章
亚历克斯	第16章
阿尔米尔·武克	第1章和第17章
阿米尔·普尔曼德 امیر پورمند	第7章
安德烈亚	第51章
安迪	第23章和第33章
andyabel	第46章
阿努杰·特里帕蒂	第51章和第111章
APH	第8章、第24章、第33章、第70章和第73章
阿里夫	第36章
亚瑟·D	第1章
阿伦·普拉萨德·E·S	第3章、第9章、第12章、第17章、第23章、第60章和第71章
阿塔福德	第22章和第47章
A_Arnold	第9章和第41章
Baodad	第51章
barcanoj	第15章
bassrek	第97章
bbrown	第36章
BeaglesEnd	第1章
beercohol	第24章
贝赫扎德	第68章和第77章
贝拉什	第9章
本·图尔	第102章
巴拉特·普拉萨德·萨特亚尔	第12章
比朱·约瑟	第1章
比诺·马修·瓦尔格斯	第33、39、50和112章
布兰登	第22章
切坦·桑加尼	第25章
chrisb	第38章
cnayak	第19、35、43、50和79章
cteski	第9、17、33、36、43、50、80和85章
D M	第1章
dacohenii	第25章
丹·古兹曼	第49章和第108章
丹尼尔·莱姆克	第18章
大卫·卡明斯基	第16章
dd4711	第42章和第109章
迪恩·沃德	第33章

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 16
Abhishek Jain	Chapter 1
Adam Porad	Chapter 9
Ahmad Aghazadeh	Chapters 59, 64 and 69
Akash	Chapter 11
Ako	Chapter 76
Akshay Anand	Chapters 43 and 59
alalp	Chapters 1 and 38
Alex	Chapter 16
Almir Vuk	Chapters 1 and 17
Amir Pourmand امیر پورمند	Chapter 7
Andrea	Chapter 51
Andy	Chapters 23 and 33
andyabel	Chapter 46
Anuj Tripathi	Chapters 51 and 111
APH	Chapters 8, 24, 33, 70 and 73
Arif	Chapter 36
Arthur D	Chapter 1
Arun Prasad E S	Chapters 3, 9, 12, 17, 23, 60 and 71
Athafoud	Chapters 22 and 47
A_Arnold	Chapters 9 and 41
Baodad	Chapter 51
barcanoj	Chapter 15
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
Brandon	Chapter 22
Chetan Sanghani	Chapter 25
chrisb	Chapter 38
cnayak	Chapters 19, 35, 43, 50 and 79
cteski	Chapters 9, 17, 33, 36, 43, 50, 80 and 85
D M	Chapter 1
dacohenii	Chapter 25
Dan Guzman	Chapters 49 and 108
Daniel Lemke	Chapter 18
David Kaminski	Chapter 16
dd4711	Chapters 42 and 109
Dean Ward	Chapter 33

DForck42	第36章和第84章
迪拉杰·库马尔	第57章
德鲁夫·乔希	第33章和第47章
迪利普	第33章
DVJex	第15章
埃里克E	第38章
尤金·尼曼	第50章和第75章
feetwet	第46章和第51章
加杰德拉	第33章
吉迪尔	第1章和第24章
gofr1	第22章
戈登·贝尔	第1章
gotqn	第32章
哈迪	第7章、第17章、第41章和第71章
哈姆扎·拉巴赫	第34章
哈里·K·M	第50章
手斧	第41章
亨里克·斯陶恩·波尔森	第59章
HK1	第33章
伊戈尔·米采夫	第41章
intox	第15章
伊兹托克森	第1章和第33章
詹姆斯	第10章
詹姆斯·安德森	第39章、第49章和第51章
杰米A	第9章和第51章
贾里德·胡珀	第1章和第9章
贾亚苏里亚·萨蒂什	第50章
杰弗里·L·惠特利奇	第43章
杰弗里·范·拉瑟姆	第36章、第51章和第56章
杰尼斯姆	第23章
杰西	第48章
吉宾·巴拉钱德兰	第41章和第52章
吉万	第4章
乔·塔拉斯	第1章和第43章
约翰·奥多姆	第1章和第49章
琼斯·约瑟夫	第89章
乔什·B	第17章
乔什·莫雷尔	第58章
乔万 MSFT	第5、11、20、26、27、28、29、30、31、34、41、52、55、80、81、85、86、89、91、93、94、96、98、99、103 和 110 章
尤尔根·D	第23章
焦耀	第51章
K48	第1章
凯恩	第62章
坎南·坎达萨米	第7章、第35章和第44章
卡尔蒂凯扬	第12章
基思·霍尔	第8章、第32章和第36章
基兰·乌坎德	第23章和第25章
科卢纳尔	第45章和第47章
克里特纳	第7章、第9章、第39章和第54章
笑着的维吉尔	第1、2、7、14和51章
lord5et	第21章
LowlyDBA	第33、39和51章

DForck42	Chapters 36 and 84
Dheeraj Kumar	Chapter 57
DhruvJoshi	Chapters 33 and 47
Dileep	Chapter 33
DVJex	Chapter 15
ErikE	Chapter 38
Eugene Niemand	Chapters 50 and 75
feetwet	Chapters 46 and 51
Gajendra	Chapter 33
Gidil	Chapters 1 and 24
gofr1	Chapter 22
Gordon Bell	Chapter 1
gotqn	Chapter 32
Hadi	Chapters 7, 17, 41 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 15
Iztoksson	Chapters 1 and 33
James	Chapter 10
James Anderson	Chapters 39, 49 and 51
JamieA	Chapters 9 and 51
Jared Hooper	Chapters 1 and 9
Jayasurya Satheesh	Chapter 50
Jeffrey L Whitledge	Chapter 43
Jeffrey Van Laethem	Chapters 36, 51 and 56
Jenism	Chapter 23
Jesse	Chapter 48
Jibin Balachandran	Chapters 41 and 52
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 5, 11, 20, 26, 27, 28, 29, 30, 31, 34, 41, 52, 55, 80, 81, 85, 86, 89, 91, 93, 94, 96, 98, 99, 103 and 110
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 8, 32 and 36
Kiran Ukande	Chapters 23 and 25
kolunar	Chapters 45 and 47
Kritner	Chapters 7, 9, 39 and 54
Laughing Vergil	Chapters 1, 2, 7, 14 and 51
lord5et	Chapter 21
LowlyDBA	Chapters 33, 39 and 51

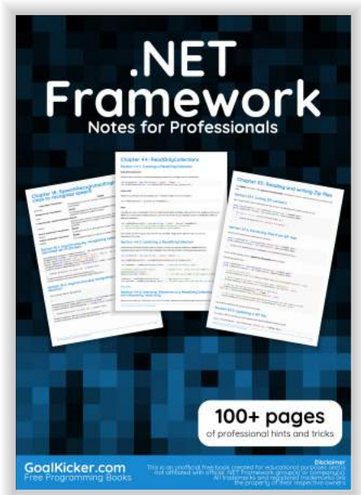
路易斯·博斯克斯	第83章
M.Ali	第13章和第108章
马赫什·达哈尔	第1章
马尔特	第1章
马尼	第82章
马尔米克	第46章和第107章
马丁肖特	第15章
鲍勃大师	第52章
马塔斯·瓦伊特凯维丘斯	第2、15、21、23、50和65章
马泰伊	第12和100章
马特	第1和78章
马克斯	第1、15、18和105章
梅雷尼克斯	第88章
梅塔诺玛尔	第90章
迈克尔·斯图姆	第15章
米哈伊	第1章
单声道	第26章
蒙蒂·怀尔德	第36章
莫希乌尔	第16章
MrE	第25章
Mspaja	第69章
穆达西尔·哈桑	第1章
新手	第1章和第15章
内森·斯克尔	第50章
尼尔·肯尼迪	第92章
新	第45章和第70章
尼克	第1章和第9章
Nick.McDermaid	第37章
奥卢瓦费米	第61章和第74章
奥兹伦·特卡尔塞茨·克尔兹纳里奇	第1章、第33章和第73章
帕特	第87章
保罗·班布里	第22章
彼得·蒂雷尔	第1章
弗朗西斯	第1章、第8章、第9章、第33章、第41章、第51章、第62章、第63章和第67章
海盗X	第50章
podiluska	第7章和第21章
Prateek	第1章
Pரத்யூ	第6章和第52章
Raghu Ariga	第106章
Raidri	第41章
Ram Grandhi	第33章
Randall	第53章
拉文德拉	第20章
朗博尔	第51章
罗伯特·哥伦比亚	第15章和第17章
罗斯·普雷瑟	第41章
鲁本主义	第104章
S.Karras	第39章
萨姆	第1章和第22章
scsimon	第12章、第39章、第50章和第51章
发送者	第80章
Shaneis	第1章
sheraz mirza	第95章

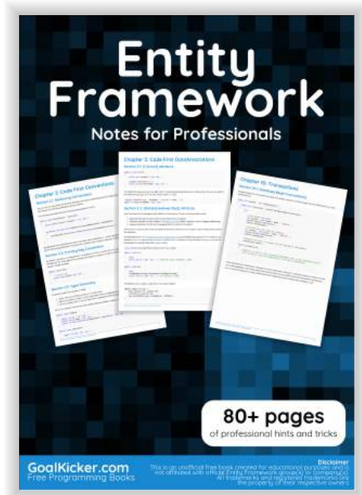
Luis Bosquez	Chapter 83
M.Ali	Chapters 13 and 108
Mahesh Dahal	Chapter 1
Malt	Chapter 1
Mani	Chapter 82
MarmiK	Chapters 46 and 107
martinshort	Chapter 15
MasterBob	Chapter 52
Matas Vaitkevicius	Chapters 2, 15, 21, 23, 50 and 65
Matej	Chapters 12 and 100
Matt	Chapters 1 and 78
Max	Chapters 1, 15, 18 and 105
Merenix	Chapter 88
Metanormal	Chapter 90
Michael Stum	Chapter 15
Mihai	Chapter 1
Mono	Chapter 26
Monty Wild	Chapter 36
Moshiour	Chapter 16
MrE	Chapter 25
Mspaja	Chapter 69
Mudassir Hasan	Chapter 1
n00b	Chapters 1 and 15
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, 8, 9, 33, 41, 51, 62, 63 and 67
Pirate X	Chapter 50
podiluska	Chapters 7 and 21
Prateek	Chapter 1
Pரத்யூ	Chapters 6 and 52
Raghu Ariga	Chapter 106
Raidri	Chapter 41
Ram Grandhi	Chapter 33
Randall	Chapter 53
ravindra	Chapter 20
Rhumborl	Chapter 51
Robert Columbia	Chapters 15 and 17
Ross Presser	Chapter 41
Rubenisme	Chapter 104
S.Karras	Chapter 39
Sam	Chapters 1 and 22
scsimon	Chapters 12, 39, 50 and 51
Sender	Chapter 80
Shaneis	Chapter 1
sheraz mirza	Chapter 95

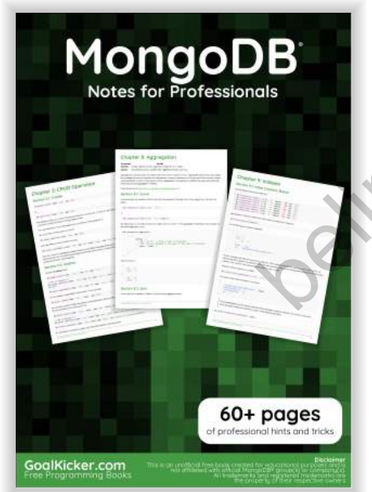
Sibeesh Venu	第112章
西尤尔	第9章和第10章
总会	第9章
spaghettidba	第51章
sqlandmore.com	第72章
SQL梅森	第36章
sqluser	第56章
苏桑	第49章
塔布·阿勒曼	第12章
takrl	第41章
塔琳	第25章
技术员	第75章
TheGameiswar	第40章
图塔·昂	第1章
汤姆·V	第59章
总计	第1章、第15章、第17章和第51章
TZHX	第51章
Uberzen1	第1章和第20章
UnhandledExcepSean	第9章
user1690166	第25章
user_0	第1章和第85章
Vexator	第43章
维卡斯·瓦伊迪亚	第14章
弗拉基米尔·奥塞尔斯基	第92章
沃尔夫冈	第11章和第32章
佐哈尔·佩莱德	第4、7、9、41、61和66章

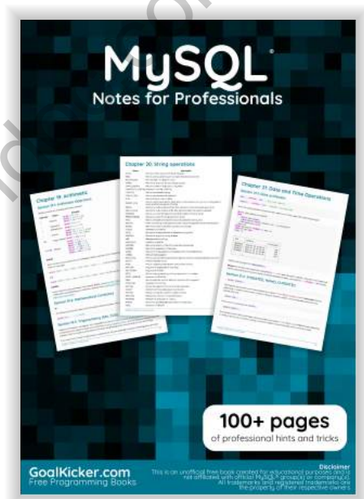
Sibeesh Venu	Chapter 112
Siyual	Chapters 9 and 10
Soukai	Chapter 9
spaghettidba	Chapter 51
sqlandmore.com	Chapter 72
SQLMason	Chapter 36
sqluser	Chapter 56
Susang	Chapter 49
Tab Alleman	Chapter 12
takrl	Chapter 41
Taryn	Chapter 25
Techie	Chapter 75
TheGameiswar	Chapter 40
Thuta Aung	Chapter 1
Tom V	Chapter 59
Tot Zam	Chapters 1, 15, 17 and 51
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 4, 7, 9, 41, 61 and 66

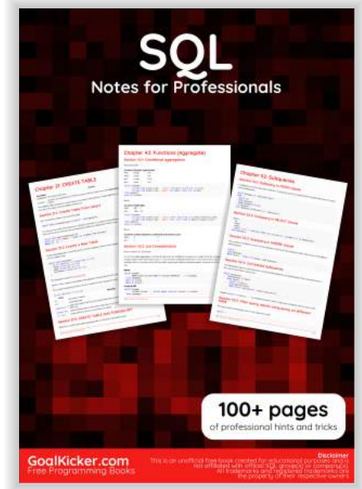
你可能还喜欢











You may also like