

3) Числовые типы данных

- **BIT**: хранит значение 0 или 1. Фактически является аналогом булевого типа в языках программирования. Занимает 1 байт.
- **TINYINT**: хранит числа от 0 до 255. Занимает 1 байт. Хорошо подходит для хранения небольших чисел.
- **SMALLINT**: хранит числа от -32 768 до 32 767. Занимает 2 байта
- **INT**: хранит числа от -2 147 483 648 до 2 147 483 647. Занимает 4 байта. Наиболее используемый тип для хранения чисел.
- **BIGINT**: хранит очень большие числа от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807, которые занимают в памяти 8 байт.
- **DECIMAL**: хранит числа с фиксированной точностью. Занимает от 5 до 17 байт в зависимости от количества чисел после запятой.

Данный тип может принимать два параметра precision и scale: `DECIMAL(precision, scale)`.

Параметр precision представляет максимальное количество цифр, которые может хранить число. Это значение должно находиться в диапазоне от 1 до 38. По умолчанию оно равно 18.

Параметр scale представляет максимальное количество цифр, которые может содержать число после запятой. Это значение должно находиться в диапазоне от 0 до значения параметра precision. По умолчанию оно равно 0.

- **NUMERIC**: данный тип аналогичен типу DECIMAL.
- **SMALLMONEY**: хранит дробные значения от -214 748.3648 до 214 748.3647. Предназначено для хранения денежных величин. Занимает 4 байта. Эквивалентен типу `DECIMAL(10, 4)`.
- **MONEY**: хранит дробные значения от -922 337 203 685 477.5808 до 922 337 203 685 477.5807. Представляет денежные величины и занимает 8 байт. Эквивалентен типу `DECIMAL(19, 4)`.
- **FLOAT**: хранит числа от -1.79E+308 до 1.79E+308. Занимает от 4 до 8 байт в зависимости от дробной части.

Может иметь форму определения в виде `FLOAT(n)`, где n представляет число бит, которые используются для хранения десятичной части числа (мантииссы). По умолчанию n = 53.

- **REAL**: хранит числа от -340E+38 to 3.40E+38. Занимает 4 байта. Эквивалентен типу `FLOAT(24)`.

Примеры числовых столбцов:

- 1 Salary MONEY,
- 2 TotalWeight DECIMAL(9,2),
- 3 Age INT,
- 4 Surplus FLOAT

Типы данных, представляющие дату и время

- **DATE**: хранит даты от 0001-01-01 (1 января 0001 года) до 9999-12-31 (31 декабря 9999 года). Занимает 3 байта.
- **TIME**: хранит время в диапазоне от 00:00:00.0000000 до 23:59:59.9999999. Занимает от 3 до 5 байт.

Может иметь форму `TIME (n)`, где `n` представляет количество цифр от 0 до 7 в дробной части секунд.

- **DATETIME**: хранит даты и время от 01/01/1753 до 31/12/9999. Занимает 8 байт.
- **DATETIME2**: хранит даты и время в диапазоне от 01/01/0001 00:00:00.0000000 до 31/12/9999 23:59:59.9999999. Занимает от 6 до 8 байт в зависимости от точности времени.

Может иметь форму `DATETIME2 (n)`, где `n` представляет количество цифр от 0 до 7 в дробной части секунд.

- **SMALLDATETIME**: хранит даты и время в диапазоне от 01/01/1900 до 06/06/2079, то есть ближайшие даты. Занимает от 4 байта.
- **DATETIMEOFFSET**: хранит даты и время в диапазоне от 0001-01-01 до 9999-12-31. Сохраняет детальную информацию о времени с точностью до 100 наносекунд. Занимает 10 байт.

Распространенные форматы дат:

- `yyyy-mm-dd` - 2017-07-12
- `dd/mm/yyyy` - 12/07/2017
- `mm-dd-yy` - 07-12-17

В таком формате двузначные числа от 00 до 49 воспринимаются как даты в диапазоне 2000-2049. А числа от 50 до 99 как диапазон чисел 1950 - 1999.

- `Month dd, yyyy` - July 12, 2017

Распространенные форматы времени:

- `hh:mi` - 13:21
- `hh:mi am/pm` - 1:21 pm
- `hh:mi:ss` - 1:21:34
- `hh:mi:ss:mmm` - 1:21:34:12
- `hh:mi:ss:nnnnnnn` - 1:21:34:1234567

Строковые типы данных

- **CHAR**: хранит строку длиной от 1 до 8 000 символов. На каждый символ выделяет по 1 байту. Не подходит для многих языков, так как хранит символы не в кодировке Unicode.

Количество символов, которое может хранить столбец, передается в скобках. Например, для столбца с типом `CHAR(10)` будет выделено 10 байт. И если мы сохраним в столбце строку менее 10 символов, то она будет дополнена пробелами.

- **VARCHAR**: хранит строку. На каждый символ выделяется 1 байт. Можно указать конкретную длину для столбца - от 1 до 8 000 символов, например, `VARCHAR(10)`. Если строка должна иметь больше 8000 символов, то задается размер `MAX`, а на хранение строки может выделяться до 2 Гб: `VARCHAR(MAX)`.

Не подходит для многих языков, так как хранит символы не в кодировке Unicode.

В отличие от типа `CHAR` если в столбец с типом `VARCHAR(10)` будет сохранена строка в 5 символов, то в столбце будет сохранено именно пять символов.

- **NCHAR**: хранит строку в кодировке Unicode длиной от 1 до 4 000 символов. На каждый символ выделяется 2 байта. Например, `NCHAR(15)`
- **NVARCHAR**: хранит строку в кодировке Unicode. На каждый символ выделяется 2 байта. Можно задать конкретный размер от 1 до 4 000 символов: . Если строка должна иметь больше 4000 символов, то задается размер `MAX`, а на хранение строки может выделяться до 2 Гб.

Еще два типа **TEXT** и **NTEXT** являются устаревшими и поэтому их не рекомендуется использовать. Вместо них применяются `VARCHAR` и `NVARCHAR` соответственно.

Примеры определения строковых столбцов:

```
1 Email VARCHAR(30),
2 Comment NVARCHAR(MAX)
```

Бинарные типы данных

- **BINARY**: хранит бинарные данные в виде последовательности от 1 до 8 000 байт.
- **VARBINARY**: хранит бинарные данные в виде последовательности от 1 до 8 000 байт, либо до $2^{31}-1$ байт при использовании значения `MAX` (`VARBINARY(MAX)`).

Еще один бинарный тип - тип `IMAGE` является устаревшим, и вместо него рекомендуется применять тип `VARBINARY`.

Остальные типы данных

- **UNIQUEIDENTIFIER**: уникальный идентификатор GUID (по сути строка с уникальным значением), который занимает 16 байт.
- **TIMESTAMP**: некоторое число, которое хранит номер версии строки в таблице. Занимает 8 байт.
- **CURSOR**: представляет набор строк.
- **HIERARCHYID**: представляет позицию в иерархии.
- **SQL_VARIANT**: может хранить данные любого другого типа данных T-SQL.
- **XML**: хранит документы XML или фрагменты документов XML. Занимает в памяти до 2 Гб.
- **TABLE**: представляет определение таблицы.
- **GEOGRAPHY**: хранит географические данные, такие как широта и долгота.
- **GEOMETRY**: хранит координаты местонахождения на плоскости.

4) Для добавления данных применяется команда **INSERT**, которая имеет следующий формальный синтаксис:

```
1  INSERT [INTO] имя_таблицы [(список_столбцов)] VALUES (значение1, значение2, ...  
    значениеN)
```

Вначале идет выражение **INSERT INTO**, затем в скобках можно указать список столбцов через запятую, в которые надо добавлять данные, и в конце после слова **VALUES** скобках перечисляют добавляемые для столбцов значения.

Например, пусть ранее была создана следующая база данных:

```
1  CREATE DATABASE productsdb;  
2  GO  
3  USE productsdb;  
4  CREATE TABLE Products  
5  (  
6      Id INT IDENTITY PRIMARY KEY,  
7      ProductName NVARCHAR(30) NOT NULL,  
8      Manufacturer NVARCHAR(20) NOT NULL,  
9      ProductCount INT DEFAULT 0,  
10     Price MONEY NOT NULL  
11 )
```

Добавим в нее одну строку с помощью команды INSERT:

```
1  INSERT Products VALUES ('iPhone 7', 'Apple', 5, 52000)
```

После удачного выполнения в SQL Server Management Studio в поле сообщений должно появиться сообщение "1 row(s) affected":

Стоит учитывать, что значения для столбцов в скобках после ключевого слова VALUES передаются по порядку их объявления. Например, в выражении CREATE TABLE выше можно увидеть, что первым столбцом идет Id. Но так как для него задан атрибут IDENTITY, то значение этого столбца автоматически генерируется, и его можно не указывать. Второй столбец представляет ProductName, поэтому первое значение - строка "iPhone 7" будет передано именно этому столбцу. Второе значение - строка "Apple" будет передана третьему столбцу Manufacturer и так далее. То есть значения передаются столбцам следующим образом:

- ProductName: 'iPhone 7'
- Manufacturer: 'Apple'
- ProductCount: 5
- Price: 52000

Также при вводе значений можно указать непосредственные столбцы, в которые будут добавляться значения:

```
1  INSERT INTO Products (ProductName, Price, Manufacturer)
2  VALUES ('iPhone 6S', 41000, 'Apple')
```

Здесь значение указывается только для трех столбцов. Причем теперь значения передаются в порядке следования столбцов:

- ProductName: 'iPhone 6S'
- Manufacturer: 'Apple'
- Price: 41000

Для неуказанных столбцов (в данном случае ProductCount) будет добавляться значение по умолчанию, если задан атрибут DEFAULT, или значение NULL. При этом неуказанные столбцы должны допускать значение NULL или иметь атрибут DEFAULT.

Также мы можем добавить сразу несколько строк:

```
1  INSERT INTO Products
2  VALUES
3  ('iPhone 6', 'Apple', 3, 36000),
4  ('Galaxy S8', 'Samsung', 2, 46000),
5  ('Galaxy S8 Plus', 'Samsung', 1, 56000)
```

В данном случае в таблицу будут добавлены три строки.

Также при добавлении мы можем указать, чтобы для столбца использовалось значение по умолчанию с помощью ключевого слова DEFAULT или значение NULL:

```
1  INSERT INTO Products (ProductName, Manufacturer, ProductCount, Price)
2  VALUES ('Mi6', 'Xiaomi', DEFAULT, 28000)
```

В данном случае для столбца ProductCount будет использовано значение по умолчанию (если оно установлено, если его нет - то NULL).

Если все столбцы имеют атрибут DEFAULT, определяющий значение по умолчанию, или допускают значение NULL, то можно для всех столбцов вставить значения по умолчанию:

```
1  INSERT INTO Products
2  DEFAULT VALUES
```

Но если брать таблицу Products, то подобная команда завершится с ошибкой, так как несколько полей не имеют атрибута DEFAULT и при этом не допускают значение NULL.

5)

IDENTITY

Атрибут **IDENTITY** позволяет сделать столбец идентификатором. Этот атрибут может назначаться для столбцов числовых типов INT, SMALLINT, BIGINT, TINYINT, DECIMAL и NUMERIC. При добавлении новых данных в таблицу SQL Server будет инкрементировать на единицу значение этого столбца у последней записи. Как правило, в роли идентификатора выступает тот же столбец, который является первичным ключом, хотя в принципе это необязательно.

```
1 CREATE TABLE Customers
2 (
3     Id INT PRIMARY KEY IDENTITY,
4     Age INT,
5     FirstName NVARCHAR(20),
6     LastName NVARCHAR(20),
7     Email VARCHAR(30),
8     Phone VARCHAR(20)
9 )
```

Также можно использовать полную форму атрибута:

```
1 IDENTITY(seed, increment)
```

Здесь параметр `seed` указывает на начальное значение, с которого будет начинаться отсчет. А параметр `increment` определяет, насколько будет увеличиваться следующее значение. По умолчанию атрибут использует следующие значения:

```
1 IDENTITY(1, 1)
```

То есть отсчет начинается с 1. А последующие значения увеличиваются на единицу. Но мы можем это поведение переопределить. Например:

```
1 Id INT IDENTITY (2, 3)
```

В данном случае отсчет начнется с 2, а значение каждой последующей записи будет увеличиваться на 3. То есть первая строка будет иметь значение 2, вторая - 5, третья - 8 и т.д.

Также следует учитывать, что в таблице только один столбец должен иметь такой атрибут.

UNIQUE

Если мы хотим, чтобы столбец имел только уникальные значения, то для него можно определить атрибут **UNIQUE**.

```
1 CREATE TABLE Customers
2 (
3     Id INT PRIMARY KEY IDENTITY,
4     Age INT,
5     FirstName NVARCHAR(20),
6     LastName NVARCHAR(20),
7     Email VARCHAR(30) UNIQUE,
8     Phone VARCHAR(20) UNIQUE
9 )
```

В данном случае столбцы, которые представляют электронный адрес и телефон, будут иметь уникальные значения. И мы не сможем добавить в таблицу две строки, у которых значения для этих столбцов будут совпадать.

Также мы можем определить этот атрибут на уровне таблицы:

```
1 CREATE TABLE Customers
2 (
3     Id INT PRIMARY KEY IDENTITY,
4     Age INT,
5     FirstName NVARCHAR(20),
6     LastName NVARCHAR(20),
7     Email VARCHAR(30),
8     Phone VARCHAR(20),
9     UNIQUE (Email, Phone)
10 )
```

NULL и NOT NULL

Чтобы указать, может ли столбец принимать значение NULL, при определении столбца ему можно задать атрибут **NULL** или **NOT NULL**. Если этот атрибут явным образом

не будет использован, то по умолчанию столбец будет допускать значение NULL. Исключением является тот случай, когда столбец выступает в роли первичного ключа - в этом случае по умолчанию столбец имеет значение NOT NULL.

```
1 CREATE TABLE Customers
2 (
3     Id INT PRIMARY KEY IDENTITY,
4     Age INT,
5     FirstName NVARCHAR(20) NOT NULL,
6     LastName NVARCHAR(20) NOT NULL,
7     Email VARCHAR(30) UNIQUE,
8     Phone VARCHAR(20) UNIQUE
9 )
```

DEFAULT

Атрибут **DEFAULT** определяет значение по умолчанию для столбца. Если при добавлении данных для столбца не будет предусмотрено значение, то для него будет использоваться значение по умолчанию.

```
1 CREATE TABLE Customers
2 (
3     Id INT PRIMARY KEY IDENTITY,
4     Age INT DEFAULT 18,
5     FirstName NVARCHAR(20) NOT NULL,
6     LastName NVARCHAR(20) NOT NULL,
7     Email VARCHAR(30) UNIQUE,
8     Phone VARCHAR(20) UNIQUE
9 );
```

Здесь для столбца Age предусмотрено значение по умолчанию 18.

CHECK

Ключевое слово **CHECK** задает ограничение для диапазона значений, которые могут храниться в столбце. Для этого после слова CHECK указывается в скобках условие, которому должен соответствовать столбец или несколько столбцов. Например, возраст клиентов не может быть меньше 0 или больше 100:

```

1 CREATE TABLE Customers
2 (
3     Id INT PRIMARY KEY IDENTITY,
4     Age INT DEFAULT 18 CHECK(Age >0 AND Age < 100),
5     FirstName NVARCHAR(20) NOT NULL,
6     LastName NVARCHAR(20) NOT NULL,
7     Email VARCHAR(30) UNIQUE CHECK(Email != ''),
8     Phone VARCHAR(20) UNIQUE CHECK(Phone != '')
9 );

```

Здесь также указывается, что столбцы Email и Phone не могут иметь пустую строку в качестве значения (пустая строка **не** эквивалентна значению NULL).

Для соединения условий используется ключевое слово **AND**. Условия можно задать в виде операций сравнения больше (>), меньше (<), не равно (!=).

Также с помощью CHECK можно создать ограничение в целом для таблицы:

```

1 CREATE TABLE Customers
2 (
3     Id INT PRIMARY KEY IDENTITY,
4     Age INT DEFAULT 18,
5     FirstName NVARCHAR(20) NOT NULL,
6     LastName NVARCHAR(20) NOT NULL,
7     Email VARCHAR(30) UNIQUE,
8     Phone VARCHAR(20) UNIQUE,
9     CHECK((Age >0 AND Age<100) AND (Email != '') AND (Phone != ''))
10 )

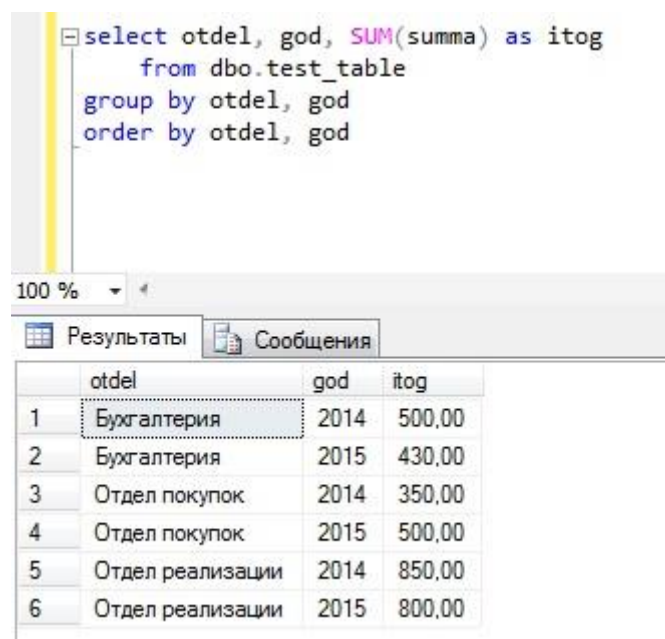
```

7.

ROLLUP

ROLLUP – оператор Transact-SQL, который формирует промежуточные итоги для каждого указанного элемента и общий итог.

Для того чтобы понять, как работает данный оператор, предлагаю сразу перейти к примерам, и допустим, что нам необходимо получить сумму расхода на оплату труда по отделам и по годам, и сначала давайте попробуем написать запрос с группировкой без использования оператора ROLLUP.



The screenshot shows a SQL query window with the following text:

```
select otдел, god, SUM(summa) as itog
from dbo.test_table
group by otдел, god
order by otдел, god
```

Below the query window, the 'Results' tab is active, displaying a table with 6 rows and 4 columns: 'id', 'otдел', 'god', and 'itog'.

	otдел	god	itog
1	Бухгалтерия	2014	500,00
2	Бухгалтерия	2015	430,00
3	Отдел покупок	2014	350,00
4	Отдел покупок	2015	500,00
5	Отдел реализации	2014	850,00
6	Отдел реализации	2015	800,00

```
SELECT otдел, god, SUM(summa) AS itog
FROM dbo.test_table
GROUP BY otдел, god
ORDER BY otдел, god
```

Как видите, группировка у нас получилась и в принципе мы видим что, например, в бухгалтерии в 2014 был такой расход, а 2015 такой, но иногда руководство хочет видеть и общую информацию, например, общий расход по каждому отделу. Для этих целей мы можем использовать оператор ROLLUP.

```
select otdel, god, SUM(summa) as itog
  from dbo.test_table
 group by
  rollup (otdel,god)
```

100 %

Результаты

Сообщения

	otdel	god	itog
1	Бухгалтерия	2014	500,00
2	Бухгалтерия	2015	430,00
3	Бухгалтерия	NULL	930,00
4	Отдел покупок	2014	350,00
5	Отдел покупок	2015	500,00
6	Отдел покупок	NULL	850,00
7	Отдел реализации	2014	850,00
8	Отдел реализации	2015	800,00
9	Отдел реализации	NULL	1650,00
10	NULL	NULL	3430,00

Текст запроса

```
SELECT otdel, god, SUM(summa) AS itog
FROM dbo.test_table
GROUP BY
ROLLUP (otdel,god)
```

Строки со значением NULL и есть промежуточные итоги по отделам, а самая последняя строка общий итог. Согласитесь, что это уже более наглядно.

Можно также использовать rollup и с группировкой по одному полю, например:

Группировка по отделам с общим итогом

Курс по SQL



47 занятий



65 задач



156 вопросов
в тестах



Поддержка
ментора



Экзамен



Сертификат

```
select otdel, SUM(summa) as itog
  from dbo.test_table
 group by
  rollup (otdel)
```

100 %

Результаты Сообщения

	otdel	itog
1	Бухгалтерия	930,00
2	Отдел покупок	850,00
3	Отдел реализации	1650,00
4	NULL	3430,00

Группировка по годам с общим итогом

```
select god, SUM(summa) as itog
  from dbo.test_table
 group by
  rollup (god)
```

100 %

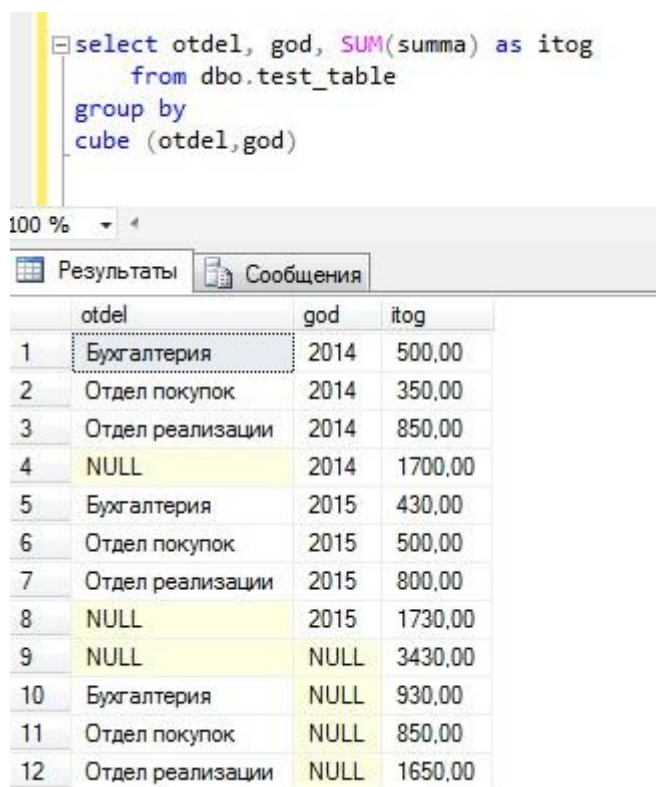
Результаты Сообщения

	god	itog
1	2014	1700,00
2	2015	1730,00
3	NULL	3430,00

CUBE

CUBE — оператор Transact-SQL, который формирует результаты для всех возможных перекрестных вычислений.

Давайте напишем практически такой же SQL запрос, только вместо rollup укажем cube и посмотрим на полученный результат.



The screenshot shows a SQL query window with the following text:

```
select otdel, god, SUM(summa) as itog
  from dbo.test_table
 group by
  cube (otdel,god)
```

Below the query window, the 'Results' tab is active, displaying a table with 12 rows and 4 columns: 'otdel', 'god', and 'itog'. The 'itog' column represents the sum of 'summa' for each group.

	otdel	god	itog
1	Бухгалтерия	2014	500,00
2	Отдел покупок	2014	350,00
3	Отдел реализации	2014	850,00
4	NULL	2014	1700,00
5	Бухгалтерия	2015	430,00
6	Отдел покупок	2015	500,00
7	Отдел реализации	2015	800,00
8	NULL	2015	1730,00
9	NULL	NULL	3430,00
10	Бухгалтерия	NULL	930,00
11	Отдел покупок	NULL	850,00
12	Отдел реализации	NULL	1650,00

Текст запроса

```
SELECT otдел, god, SUM(summa) AS itog
FROM dbo.test_table
GROUP BY
CUBE (otдел,god)
```

В данном случае отличие от rollup заключается в том, что группировка и промежуточные итоги выполнены как для otдел, так и для god.

GROUPING SETS

GROUPING SETS – оператор Transact-SQL, который формирует результаты нескольких группировок в один набор данных, другими словами, он эквивалентен [конструкции UNION ALL](#) к указанным группам.

Пример GROUPING SETS

<pre> select otdel, god, SUM(summa) as itog from dbo.test_table group by grouping sets (otdel,god) </pre>			
100 %			
<div>Результаты</div> <div>Сообщения</div>			
	otdel	god	itog
1	NULL	2014	1700,00
2	NULL	2015	1730,00
3	Бухгалтерия	NULL	930,00
4	Отдел покупок	NULL	850,00
5	Отдел реализации	NULL	1650,00

Текст запроса

```

SELECT otdel, god, SUM(summa) AS itog
FROM dbo.test_table
GROUP BY
GROUPING SETS (otdel,god)

```

тот же результат, но с использованием UNION ALL

<pre> select null as otdel, god, SUM(summa) as itog from dbo.test_table group by god union all select otdel, null as god, SUM(summa) as itog from dbo.test_table group by otdel </pre>			
100 %			
<div>Результаты</div> <div>Сообщения</div>			
	otdel	god	itog
1	NULL	2014	1700,00
2	NULL	2015	1730,00
3	Бухгалтерия	NULL	930,00
4	Отдел покупок	NULL	850,00
5	Отдел реализации	NULL	1650,00

Текст запроса

```

SELECT null AS otdel, god, SUM(summa) AS itog
FROM dbo.test_table
GROUP BY god

```



```
UNION ALL
```

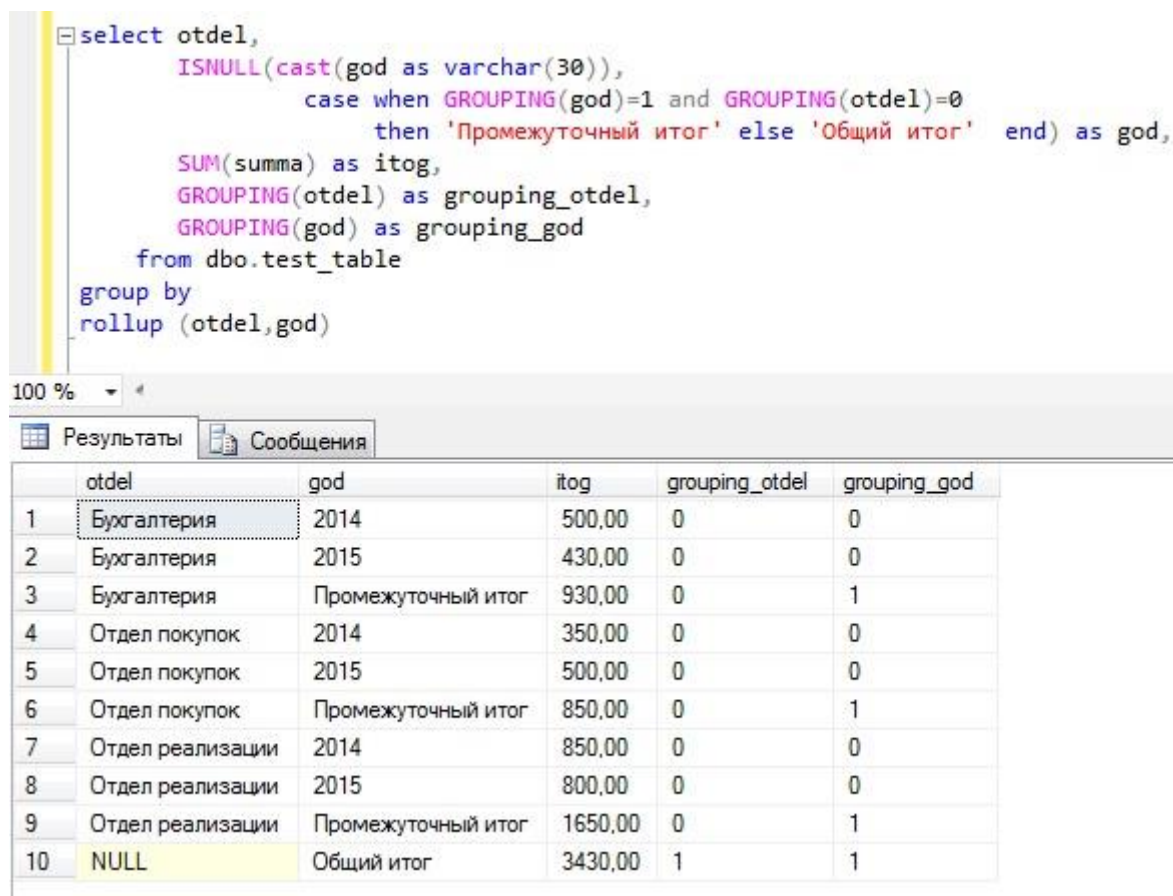
```
SELECT otdel, null AS god, SUM(summa) AS itog  
FROM dbo.test_table  
GROUP BY otdel
```

А сейчас предлагаю поговорить еще об одной полезной возможности, которая напрямую относится к перечисленным выше операторам, а именно о функции GROUPING.

GROUPING – функция Transact-SQL, которая возвращает истину, если указанное выражение является статистическим, и ложь, если выражение нестатистическое.

Данная функция создана для того, чтобы отличить статистические строки, которые добавил SQL сервер, от строк, которые и есть сами данные, так как когда используешь много группировок, запутаться в строках очень легко.

Пример GROUPING



```
select otdel,  
       ISNULL(cast(god as varchar(30)),  
              case when GROUPING(god)=1 and GROUPING(otdel)=0  
                    then 'Промежуточный итог' else 'Общий итог' end) as god,  
       SUM(summa) as itog,  
       GROUPING(otdel) as grouping_otdel,  
       GROUPING(god) as grouping_god  
from dbo.test_table  
group by  
rollup (otdel,god)
```

	otdel	god	itog	grouping_otdel	grouping_god
1	Бухгалтерия	2014	500,00	0	0
2	Бухгалтерия	2015	430,00	0	0
3	Бухгалтерия	Промежуточный итог	930,00	0	1
4	Отдел покупок	2014	350,00	0	0
5	Отдел покупок	2015	500,00	0	0
6	Отдел покупок	Промежуточный итог	850,00	0	1
7	Отдел реализации	2014	850,00	0	0
8	Отдел реализации	2015	800,00	0	0
9	Отдел реализации	Промежуточный итог	1650,00	0	1
10	NULL	Общий итог	3430,00	1	1

Текст запроса

```
SELECT otdel,  
       ISNULL(CAST(god AS VARCHAR(30)),
```



```
        CASE WHEN GROUPING(god)=1 AND GROUPING(otdel)=0
              THEN 'Промежуточный итог'
              ELSE 'Общий итог' END) AS god,
SUM(summa) AS itog,
GROUPING(otdel) AS grouping_otdel,
GROUPING(god) AS grouping_god
FROM dbo.test_table
GROUP BY
ROLLUP (otdel,god)
```

На этом все, надеюсь, что возможность формировать итоги и промежуточные итоги в Microsoft SQL Server Вам будет полезна. Также рекомендую Вам почитать мою книгу «[SQL код](#)», в ней язык SQL рассматривается как стандарт, чтобы после прочтения данной книги можно было работать с языком SQL в любой системе управления базами данных. Удачи!

8)

То есть SQL Server автоматически может преобразовать число 100.0 (float) в дату и время (datetime).

В тех случаях, когда необходимо выполнить преобразования от типов с высшим приоритетом к типам с низшим приоритетом, то надо выполнять явное приведение типов. Для этого в T-SQL определены две функции: **CONVERT** и **CAST**.

Функция **CAST** преобразует выражение одного типа к другому. Она имеет следующую форму:

```
1 CAST(выражение AS тип_данных)
```

Для примера возьмем следующие таблицы:

```
1 CREATE TABLE Products
2 (
3     Id INT IDENTITY PRIMARY KEY,
4     ProductName NVARCHAR(30) NOT NULL,
5     Manufacturer NVARCHAR(20) NOT NULL,
6     ProductCount INT DEFAULT 0,
7     Price MONEY NOT NULL
8 );
9 CREATE TABLE Customers
10 (
11     Id INT IDENTITY PRIMARY KEY,
12     FirstName NVARCHAR(30) NOT NULL
13 );
14 CREATE TABLE Orders
15 (
16     Id INT IDENTITY PRIMARY KEY,
17     ProductId INT NOT NULL REFERENCES Products(Id) ON DELETE CASCADE,
18     CustomerId INT NOT NULL REFERENCES Customers(Id) ON DELETE CASCADE,
19     CreatedAt DATE NOT NULL,
20     ProductCount INT DEFAULT 1,
21     Price MONEY NOT NULL
```

22);

Например, при выводе информации о заказах преобразует числовое значение и дату в строку:

```
1 SELECT Id, CAST(CreatedAt AS nvarchar) + '; total: ' + CAST(Price * ProductCount AS  
   nvarchar)  
2 FROM Orders
```

Convert

Большую часть преобразований охватывает функция CAST. Если же необходимо какое-то дополнительное форматирование, то можно использовать функцию **CONVERT**. Она имеет следующую форму:

```
1 CONVERT(тип_данных, выражение [, стиль])
```

Третий необязательный параметр задает стиль форматирования данных. Этот параметр представляет числовое значение, которое для разных типов данных имеет разную интерпретацию. Например, некоторые значения для форматирования дат и времени:

- 0 или 100 - формат даты "Mon dd yyyy hh:miAM/PM" (значение по умолчанию)
- 1 или 101 - формат даты "mm/dd/yyyy"
- 3 или 103 - формат даты "dd/mm/yyyy"
- 7 или 107 - формат даты "Mon dd, yyyy hh:miAM/PM"
- 8 или 108 - формат даты "hh:mi:ss"
- 10 или 110 - формат даты "mm-dd-yyyy"
- 14 или 114 - формат даты "hh:mi:ss:mmmm" (24-часовой формат времени)

Некоторые значения для форматирования данных типа money в строку:

- 0 - в дробной части числа остаются только две цифры (по умолчанию)
- 1 - в дробной части числа остаются только две цифры, а для разделения разрядов применяется запятая
- 2 - в дробной части числа остаются только четыре цифры

Например, выведем дату и стоимость заказов с форматированием:

```
1 SELECT CONVERT(nvarchar, CreatedAt, 3),  
2         CONVERT(nvarchar, Price * ProductCount, 1)  
3 FROM Orders
```

TRY_CONVERT

При использовании функций CAST и CONVERT SQL Server выбрасывает исключение, если данные нельзя привести к определенному типу. Например:

```
1  SELECT CONVERT(int, 'sql')
```

Чтобы избежать генерации исключения можно использовать функцию **TRY_CONVERT**. Ее использование аналогично функции CONVERT за тем исключением, что если выражение не удастся преобразовать к нужному типу, то функция возвращает NULL:

```
1  SELECT TRY_CONVERT(int, 'sql')      -- NULL
2  SELECT TRY_CONVERT(int, '22')      -- 22
```

Дополнительные функции

Кроме CAST, CONVERT, TRY_CONVERT есть еще ряд функций, которые могут использоваться для преобразования в ряд типов:

- **STR(float [, length [,decimal]])**: преобразует число в строку. Второй параметр указывает на длину строки, а третий - сколько знаков в дробной части числа надо оставлять
- **CHAR(int)**: преобразует числовой код ASCII в символ. Нередко используется для тех ситуаций, когда необходим символ, который нельзя ввести с клавиатуры
- **ASCII(char)**: преобразует символ в числовой код ASCII
- **NCHAR(int)**: преобразует числовой код UNICODE в символ
- **UNICODE(char)**: преобразует символ в числовой код UNICODE

```
1  SELECT STR(123.4567, 6,2)    -- 123.46
2  SELECT CHAR(219)             --  Ъ
3  SELECT ASCII('Ъ')           -- 219
4  SELECT NCHAR(1067)           --  Ъ
5  SELECT UNICODE('Ъ')         -- 1067
```

9)

Функции даты и времени

В следующих таблицах приводятся функции даты и времени Transact-SQL. Дополнительные сведения о детерминизме см. в статье [Детерминированные и недетерминированные функции](#).

Функции, возвращающие значения системной даты и времени

Transact-SQL наследует все значения системной даты и времени от операционной системы компьютера, на котором работает экземпляр SQL Server.

Высокоточные функции системной даты и времени

SQL Server 2019 (15.x) получает значения даты и времени с помощью функции GetSystemTimeAsFileTime() Windows API. Точность зависит от физического оборудования и версии Windows, в которой запущен экземпляр SQL Server. Точность возвращаемых значений этого API-интерфейса задана равной 100 нс. Точность может быть определена с помощью метода GetSystemTimeAdjustment() API-интерфейса Windows.

ВЫСОКОТОЧНЫЕ ФУНКЦИИ СИСТЕМНОЙ ДАТЫ И ВРЕМЕНИ				
Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминированность
SYSDATETIME	SYSDATETIME ()	Возвращает значение типа datetime2(7) , которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Возвращаемое значение не содержит смещение часового пояса.	datetime2(7)	Недетерминированная
SYSDATETIMEOFFSET	SYSDATETIMEOFFSET ()	Возвращает значение типа datetimeoffset(7) , которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Возвращаемое значение содержит смещение часового пояса.	datetimeoffset(7)	Недетерминированная
SYSUTCDATETIME	SYSUTCDATETIME ()	Возвращает значение типа datetime2(7) , которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Функция	datetime2(7)	Недетерминированная

ВЫСОКОТОЧНЫЕ ФУНКЦИИ СИСТЕМНОЙ ДАТЫ И ВРЕМЕНИ				
Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминизм
		возвращает значения даты и времени в формате UTC.		

Функции системной даты и времени меньшей точности

ФУНКЦИИ СИСТЕМНОЙ ДАТЫ И ВРЕМЕНИ МЕНЬШЕЙ ТОЧНОСТИ				
Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминизм
CURRENT_TIMESTAMP	CURRENT_TIMESTAMP	Возвращает значение типа datetime , которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Возвращаемое значение не содержит смещение часового пояса.	datetime	Недетерминированная
GETDATE	GETDATE ()	Возвращает значение типа datetime , которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Возвращаемое значение не содержит смещение часового пояса.	datetime	Недетерминированная
GETUTCDATE	GETUTCDATE ()	Возвращает значение типа datetime , которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Функция возвращает значения даты и времени в формате UTC.	datetime	Недетерминированная

Функции, возвращающие компоненты даты и времени

ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ КОМПОНЕНТЫ ДАТЫ И ВРЕМЕНИ				
Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминизм
DATENAME	DATENAME (<i>datepart</i> , <i>date</i>)	Возвращает строку символов, представляющую указанную часть <i>datepart</i> заданного типа <i>date</i> .	nvarchar	Недетерминированная

ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ КОМПОНЕНТЫ ДАТЫ И ВРЕМЕНИ

Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминизм
DATEPART	DATEPART (<i>datepart</i> , <i>date</i>)	Возвращает целое число, представляющее указанную часть <i>datepart</i> заданного типа <i>date</i> .	int	Недетерминирован
DAY	DAY (<i>date</i>)	Возвращает целое число, представляющее часть дня указанного типа <i>date</i> .	int	Детерминирован
MONTH	MONTH (<i>date</i>)	Возвращает целое число, представляющее часть месяца указанного типа <i>date</i> .	int	Детерминирован
YEAR	YEAR (<i>date</i>)	Возвращает целое число, представляющее часть года указанного типа <i>date</i> .	int	Детерминирован

Функции, возвращающие значения даты и времени из их компонентов

ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ ДАТУ И ВРЕМЯ

Компонент	Синтаксис
DATEFROMPARTS	DATEFROMPARTS (<i>year</i> , <i>month</i> , <i>day</i>)
DATETIME2FROMPARTS	DATETIME2FROMPARTS (<i>year</i> , <i>month</i> , <i>day</i> , <i>hour</i> , <i>minute</i> , <i>seconds</i> , <i>fractions</i> , <i>precision</i>)
DATETIMEFROMPARTS	DATETIMEFROMPARTS (<i>year</i> , <i>month</i> , <i>day</i> , <i>hour</i> , <i>minute</i> , <i>seconds</i> , <i>milliseconds</i>)
DATETIMEOFFSETFROMPARTS	DATETIMEOFFSETFROMPARTS (<i>year</i> , <i>month</i> , <i>day</i> , <i>hour</i> , <i>minute</i> , <i>seconds</i> , <i>fractions</i> , <i>hour_offset</i> , <i>minute_offset</i> , <i>precision</i>)
SMALLDATETIMEFROMPARTS	SMALLDATETIMEFROMPARTS (<i>year</i> , <i>month</i> , <i>day</i> , <i>hour</i> , <i>minute</i>)

Компонент	Синтаксис
TIMEFROMPARTS	TIMEFROMPARTS (<i>hour, minute, seconds, fractions, precision</i>)

Функции, возвращающие значения разности даты и времени

ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ ЗНАЧЕНИЯ РАЗНОСТИ ДАТЫ И ВРЕМ

Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминизм
DATEDIFF	DATEDIFF (<i>datepart, startdate, enddate</i>)	Возвращает количество границ даты или времени <i>datepart</i> , пересекающихся между двумя указанными датами.	int	Детерминирован
DATEDIFF_BIG	DATEDIFF_BIG (<i>datepart, startdate, enddate</i>)	Возвращает количество границ даты или времени <i>datepart</i> , пересекающихся между двумя указанными датами.	bigint	Детерминирован

Функции, изменяющие значения даты и времени

ФУНКЦИИ, ИЗМЕНЯЮЩИЕ ЗНА

Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных
DATEADD	DATEADD (<i>datepart, number, date</i>)	Возвращает новое значение datetime , добавляя интервал к указанной части <i>datepart</i> заданной даты <i>date</i> .	Тип данных аргумента
EOMONTH	EOMONTH (<i>start_date</i> [, <i>month_to_add</i>])	Возвращает последний день месяца, содержащего указанную дату, с необязательным смещением.	Тип возвращаемого значения — это тип данных <i>start_date</i>
SWITCHOFFSET	SWITCHOFFSET (<i>DATETIMEOFFSET, time_zone</i>)	Функция SWITCHOFFSET изменяет смещение часового пояса для значения DATETIMEOFFSET и сохраняет значение UTC.	Значение datetime точно в дол. заданной в аргументе <i>DATETIMEOFFSET</i>

ФУНКЦИИ, ИЗМЕНЯЮЩИЕ ЗНА

Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных
TODATETIMEOFFSET	TODATETIMEOFFSET (<i>expression</i> , <i>time_zone</i>)	TODATETIMEOFFSET преобразует значение типа datetime2 в значение типа datetimeoffset. Функция TODATETIMEOFFSET преобразует значение datetime2 в местное время для указанного time_zone.	Значение datetimeoffset с точностью в доли секунды, заданной в аргументе <i>time_zone</i> .

Функции, устанавливающие или возвращающие функции формата сеанса

ФУНКЦИИ, УСТАНОВЛИВАЮЩИЕ ИЛИ ВОЗВРАЩАЮЩИЕ ФУНКЦИИ ФОРМАТА

Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминированность
@@DATEFIRST	@@DATEFIRST	Возвращает текущее значение параметра SET DATEFIRST для сеанса.	tinyint	Недетерминированная
SET DATEFIRST	SET DATEFIRST { <i>number</i> @ <i>number_var</i> }	Устанавливает первый день недели в виде числа от 1 до 7.	Неприменимо	Неприменимо
SET DATEFORMAT	SET DATEFORMAT { <i>format</i> @ <i>format_var</i> }	Задаёт порядок составляющих даты (месяц/день/год) для ввода данных типа datetime или smalldatetime .	Неприменимо	Неприменимо
@@LANGUAGE	@@LANGUAGE	Возвращает название используемого в настоящий момент языка. Функция @@LANGUAGE не является функцией даты или времени. Однако на данные, выводимые функциями даты, могут повлиять настройки языка.	Неприменимо	Неприменимо
SET LANGUAGE	SET LANGUAGE { [N] ' <i>language</i> ' @ <i>language_var</i> }	Устанавливает языковую среду сеанса и системных сообщений. SET LANGUAGE не является функцией даты или времени. Однако на данные, выводимые функциями даты, влияет параметр языка.	Неприменимо	Неприменимо
sp_helplanguage	sp_helplanguage [[@ <i>language</i> =] ' <i>language</i> ']	Возвращает сведения о формате даты всех поддерживаемых языков. sp_helplanguage не	Неприменимо	Неприменимо

ФУНКЦИИ, УСТАНОВЛИВАЮЩИЕ ИЛИ ВОЗВРАЩАЮЩИЕ ФУНКЦИИ ФОРМАТ

Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминизм
		является хранимой процедурой даты или времени. Однако на данные, выводимые функциями даты, влияет параметр языка.		

Функции, проверяющие значения даты и времени

ФУНКЦИИ, ПРОВЕРЯЮЩИЕ ЗНАЧЕНИЯ ДАТЫ И ВРЕМЕНИ

Компонент	Синтаксис	Возвращаемое значение	Тип возвращаемых данных	Детерминизм
ISDATE	ISDATE (<i>expression</i>)	Определяет, является ли входное выражение типа datetime или smalldatetime допустимым значением даты или времени.	int	Функция ISDATE детерминирована только если используется совместно с функциями CONVERT и если заданный параметр стиля CONVERT не равен 0, 100, 9 или

Дата и время — см. также

ДАТА И ВРЕМЯ — СМ. ТАКЖЕ

Раздел	Описание
FORMAT	Возвращает значение в указанных формате и культуре (не обязательно). Для выполнения форматирования значения даты, времени и чисел с учетом локали в виде строк используется функция FORMAT.
Функции CAST и CONVERT (Transact-SQL)	Предоставляет сведения о преобразовании значений даты и времени в строковые литералы и обратно, а также в другие форматы даты и времени.
Написание инструкций Transact-SQL, адаптированных к международному использованию	Предоставляет рекомендации относительно переносимости баз данных и приложений баз данных, использующих инструкции Transact-SQL, одного языка на другой или в многоязычную среду.
Скалярные функции ODBC (Transact-SQL)	Предоставляет сведения о скалярных функциях ODBC, которые могут использоваться в инструкциях Transact-SQL. К ним относятся функции даты и времени ODBC.
AT TIME ZONE (Transact-SQL)	Обеспечивает преобразование часовых поясов.

10)

[CHARINDEX](#)

[CONCAT](#)

[CONCAT_WS](#)

[DIFFERENCE](#)

[FORMAT](#)

[LEFT](#)

[LEN](#)

[LOWER](#)

[LTRIM](#)

[NCHAR](#)

[PATINDEX](#)

[QUOTENAME](#)

[REPLACE](#)

[REPLICATE](#)

[REVERSE](#)

[RIGHT](#)

[RTRIM](#)

[SOUNDEX](#)

[SPACE](#)

[STR](#)

[STRING_AGG](#)

[STRING_ESCAPE](#)

[STRING_SPLIT](#)

[STUFF](#)

[SUBSTRING](#)

[TRANSLATE](#)

[TRIM](#)

[UNICODE](#)

[UPPER](#)

11)

Синтаксис

syntaxsqlКопировать

```
IF Boolean_expression
    { sql_statement | statement_block }
[ ELSE
    { sql_statement | statement_block } ]
```

Примечание

Ссылки на описание синтаксиса Transact-SQL для SQL Server 2014 и более ранних версий, см. в статье [Документация по предыдущим версиям](#).

Аргументы

Boolean_expression

Выражение, возвращающее значение TRUE или FALSE. Если логическое выражение содержит инструкцию SELECT, инструкция SELECT должна быть заключена в скобки.

{ *sql_statement* | *statement_block* }

Любая инструкция или группа инструкций языка Transact-SQL, указанная с помощью блока инструкций. Без использования блока инструкций условия IF и ELSE могут повлиять на выполнение только одной инструкции языка Transact-SQL.

Для определения блока инструкций используйте ключевые слова потока управления BEGIN и END.

Remarks

Конструкция IF...ELSE может быть использована в пакетах, хранимых процедурах и нерегламентированных запросах. При использовании в хранимой процедуре эта конструкция часто применяется для проверки существования некоторого параметра.

Проверки IF могут находиться внутри другого IF или следующего ELSE. Ограничение количества вложенных уровней зависит от свободной памяти.

Пример

SQLКопировать

```
IF DATENAME(weekday, GETDATE()) IN (N'Saturday', N'Sunday')
    SELECT 'Weekend';
ELSE
    SELECT 'Weekday';
```

Дополнительные сведения см. в разделе [ELSE \(IF...ELSE\) \(Transact-SQL\)](#).

Оценка списка условий и возвращение одного из нескольких возможных выражений результатов.

Выражение CASE имеет два формата:

- простое выражение CASE для определения результата сравнивает выражение с набором простых выражений;
- поисковое выражение CASE для определения результата вычисляет набор логических выражений.

Оба формата поддерживают дополнительный аргумент ELSE.

Выражение CASE может использоваться в любой инструкции или предложении, которые допускают допустимые выражения. Например, выражение CASE можно использовать в таких инструкциях, как SELECT, UPDATE, DELETE и SET, а также в таких предложениях, как select_list, IN, WHERE, ORDER BY и HAVING.

[Синтаксические обозначения в Transact-SQL](#)

Синтаксис

syntasqlКопировать

-- Syntax for SQL Server, Azure SQL Database and Azure Synapse Analytics

--Simple CASE expression:

```
CASE input_expression
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

--Searched CASE expression:

```
CASE
    WHEN Boolean_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

syntasqlКопировать

-- Syntax for Parallel Data Warehouse

```
CASE
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Примечание

Ссылки на описание синтаксиса Transact-SQL для SQL Server 2014 и более ранних версий, см. в статье [Документация по предыдущим версиям](#).

Аргументы

input_expression

Выражение, полученное при использовании простого формата функции CASE. *input_expression* — это любое допустимое [выражение](#).

WHEN *when_expression*

Простое выражение, с которым сравнивается *input_expression* при использовании простого формата CASE. *when_expression* — это любое допустимое выражение.

Типы данных аргумента *input_expression* и каждого из выражений *when_expression* должны быть одинаковыми или неявно приводимыми друг к другу.

THEN *result_expression*

Выражение, возвращаемое, когда равенство *input_expression* и *when_expression* имеет значение TRUE или *Boolean_expression* имеет значение TRUE. *result expression* — это любое допустимое [выражение](#).

ELSE *else_result_expression*

Это выражение, возвращаемое, если ни одна из операций сравнения не дает в результате TRUE. Если этот аргумент опущен и ни одна из операций сравнения не дает в результате TRUE, функция CASE возвращает NULL. *else_result_expression* — это любое допустимое выражение. Типы данных аргумента *else_result_expression* и каждого из выражений *result_expression* должны быть одинаковыми или неявно приводимыми друг к другу.

WHEN *Boolean_expression*

Логическое выражение, полученное при использовании поискового формата функции CASE. *Boolean_expression* — это любое допустимое логическое выражение.

Примечание

Ссылки на описание синтаксиса Transact-SQL для SQL Server 2014 и более ранних версий, см. в статье [Документация по предыдущим версиям](#).

Типы возвращаемых данных

Возвращает тип с наивысшим приоритетом из набора типов в выражении *result_expressions* и необязательном выражении *else_result_expression*. Дополнительные сведения см. в разделе [Приоритет типов данных \(Transact-SQL\)](#).

Возвращаемые значения

Простое выражение CASE

Простое выражение CASE сравнивает первое выражение с выражением в каждом предложении WHEN. Если эти выражения эквивалентны, то возвращается выражение в предложении THEN.

- Допускается только проверка равенства.
- В указанном порядке сравнивает значения выражений *input_expression* и *when_expression* для каждого предложения WHEN.
- Возвращает выражение *result_expression*, соответствующее первой операции *input_expression = when_expression*, равной TRUE.
- Если ни одна из операций *input_expression = when_expression* не дает значения TRUE, Компонент SQL Server Database Engine возвращает выражение *else_result_expression*, если указано предложение ELSE, или значение NULL, если предложение ELSE не указано.

Поисковое выражение CASE

- Вычисляет в указанном порядке выражения *Boolean_expression* для каждого предложения WHEN.
- Возвращает выражение *result_expression*, соответствующее первому выражению *Boolean_expression*, которое имеет значение TRUE.
- Если ни одно выражение *Boolean_expression* не равно TRUE, Компонент Database Engine возвращает выражение *else_result_expression*, если указано предложение ELSE, или значение NULL, если предложение ELSE не указано.

Remarks

SQL Server допускает применение в выражениях CASE не более 10 уровней вложенности.

Выражение CASE нельзя использовать для управления потоком выполнения инструкций Transact-SQL, блоков инструкций, определяемых пользователем функций и хранимых процедур. Список методов управления потоком см. в статье [Язык управления потоком \(Transact-SQL\)](#).

Выражение CASE последовательно оценивает свои условия и останавливается, когда находит первое выполнимое условие. В некоторых ситуациях выражение оценивается до того, как выражение CASE получает результаты выражения в качестве входных данных. При оценке этих выражений возможны ошибки. Агрегатные выражения в аргументах WHEN выражения CASE вначале оцениваются, после чего передаются выражению CASE. Например в следующем запросе создается ошибка деления на ноль при вычислении значения агрегата MAX. Это происходит до оценки выражения CASE.

SQLКопировать

```
WITH Data (value) AS  
(
```

```

SELECT 0
UNION ALL
SELECT 1
)
SELECT
CASE
    WHEN MIN(value) <= 0 THEN 0
    WHEN MAX(1/value) >= 100 THEN 1
END
FROM Data ;

```

Следует создавать зависимости только от порядка оценки условий WHEN для скалярных выражений (в том числе нескоррелированных вложенных запросов, возвращающих скалярные значения), а не для агрегатных выражений.

Примеры

A. Использование инструкции SELECT с простым выражением CASE

При использовании в инструкции SELECT простое выражение CASE позволяет выполнить только проверку на равенство. Другие проверки не выполняются. В следующем примере выражение CASE используется для изменения способа отображения категорий линейки продуктов с целью сделать их более понятными.

SQLКопировать

```

USE AdventureWorks2012;
GO
SELECT ProductNumber, Category =
CASE ProductLine
    WHEN 'R' THEN 'Road'
    WHEN 'M' THEN 'Mountain'
    WHEN 'T' THEN 'Touring'
    WHEN 'S' THEN 'Other sale items'
    ELSE 'Not for sale'
END,
Name
FROM Production.Product
ORDER BY ProductNumber;
GO

```

Б. Использование инструкции SELECT с поисковым выражением CASE

При использовании в инструкции SELECT поисковое выражение CASE позволяет заменять значения в результирующем наборе в зависимости от результатов сравнения. В следующем примере отображается список цен в виде текстового комментария, основанного на диапазоне цен для продукта.

SQLКопировать

```

USE AdventureWorks2012;
GO
SELECT ProductNumber, Name, "Price Range" =
CASE
    WHEN ListPrice = 0 THEN 'Mfg item - not for resale'
    WHEN ListPrice < 50 THEN 'Under $50'
    WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'

```

```

        WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'
        ELSE 'Over $1000'
    END
FROM Production.Product
ORDER BY ProductNumber ;
GO

```

В. Использование выражения CASE в предложении ORDER BY

В следующем примере выражение CASE используется в предложении ORDER BY, чтобы определить порядок сортировки строк на основе значения заданного столбца таблицы. В первом примере вычисляется значение столбца SalariedFlag таблицы HumanResources.Employee. Сотрудники, для которых столбец SalariedFlag имеет значение 1, возвращаются в порядке BusinessEntityID (по убыванию). Сотрудники, для которых столбец SalariedFlag имеет значение 0, возвращаются в порядке BusinessEntityID (по возрастанию). Во втором примере результирующий набор упорядочивается по столбцу TerritoryName, если столбец CountryRegionName содержит значение «США», и по столбцу CountryRegionName в остальных строках.

SQLКопировать

```

SELECT BusinessEntityID, SalariedFlag
FROM HumanResources.Employee
ORDER BY CASE SalariedFlag WHEN 1 THEN BusinessEntityID END DESC
        ,CASE WHEN SalariedFlag = 0 THEN BusinessEntityID END;
GO

```

SQLКопировать

```

SELECT BusinessEntityID, LastName, TerritoryName, CountryRegionName
FROM Sales.vSalesPerson
WHERE TerritoryName IS NOT NULL
ORDER BY CASE CountryRegionName WHEN 'United States' THEN TerritoryName
        ELSE CountryRegionName END;

```

Г. Использование выражения CASE в инструкции UPDATE

В следующем примере выражение CASE используется в инструкции UPDATE, чтобы определить значение, установленное в столбце VacationHours для сотрудников, у которых столбец SalariedFlag имеет значение 0. Если при вычитании 10 часов из VacationHours получается отрицательное значение, VacationHours увеличивается на 40 часов. В противном случае значение VacationHours увеличивается на 20 часов. С помощью предложения OUTPUT отображаются исходная и обновленная продолжительности отпуска.

SQLКопировать

```

USE AdventureWorks2012;
GO
UPDATE HumanResources.Employee
SET VacationHours =
    ( CASE
        WHEN ((VacationHours - 10.00) < 0) THEN VacationHours + 40
        ELSE (VacationHours + 20.00)
    )

```

```

        END
    )
    OUTPUT Deleted.BusinessEntityID, Deleted.VacationHours AS BeforeValue,
           Inserted.VacationHours AS AfterValue
    WHERE SalariedFlag = 0;

```

Д. Использование выражения CASE в инструкции SET

В следующем примере выражение CASE используется в инструкции SET для функции `dbo.GetContactInfo` с табличным значением. В базе данных **AdventureWorks2012** все данные, связанные с людьми, хранятся в таблице `Person.Person`. Например, человек может быть сотрудником, представителем поставщика или заказчиком. Функция возвращает имя и фамилию человека с заданным `BusinessEntityID` и соответствующий тип контакта для этого пользователя. Выражение CASE в инструкции SET определяет отображаемое значение для столбца `ContactType` в зависимости от наличия столбца `BusinessEntityID` в таблицах `Employee`, `Vendor` или `Customer`.

SQLКопировать

```

USE AdventureWorks2012;
GO
CREATE FUNCTION dbo.GetContactInformation(@BusinessEntityID INT)
    RETURNS @retContactInformation TABLE
(
    BusinessEntityID INT NOT NULL,
    FirstName NVARCHAR(50) NULL,
    LastName NVARCHAR(50) NULL,
    ContactType NVARCHAR(50) NULL,
    PRIMARY KEY CLUSTERED (BusinessEntityID ASC)
)
AS
-- Returns the first name, last name and contact type for the specified contact.
BEGIN
    DECLARE
        @FirstName NVARCHAR(50),
        @LastName NVARCHAR(50),
        @ContactType NVARCHAR(50);

    -- Get common contact information
    SELECT
        @BusinessEntityID = BusinessEntityID,
        @FirstName = FirstName,
        @LastName = LastName
    FROM Person.Person
    WHERE BusinessEntityID = @BusinessEntityID;

    SET @ContactType =
        CASE
            -- Check for employee
            WHEN EXISTS(SELECT * FROM HumanResources.Employee AS e
                        WHERE e.BusinessEntityID = @BusinessEntityID)
            THEN 'Employee'

            -- Check for vendor
            WHEN EXISTS(SELECT * FROM Person.BusinessEntityContact AS bec
                        WHERE bec.BusinessEntityID = @BusinessEntityID)
            THEN 'Vendor'

```

```

-- Check for store
WHEN EXISTS(SELECT * FROM Purchasing.Vendor AS v
            WHERE v.BusinessEntityID = @BusinessEntityID)
            THEN 'Store Contact'

-- Check for individual consumer
WHEN EXISTS(SELECT * FROM Sales.Customer AS c
            WHERE c.PersonID = @BusinessEntityID)
            THEN 'Consumer'

END;

-- Return the information to the caller
IF @BusinessEntityID IS NOT NULL
BEGIN
    INSERT @retContactInformation
    SELECT @BusinessEntityID, @FirstName, @LastName, @ContactType;
END;

RETURN;
END;
GO

SELECT BusinessEntityID, FirstName, LastName, ContactType
FROM dbo.GetContactInformation(2200);
GO
SELECT BusinessEntityID, FirstName, LastName, ContactType
FROM dbo.GetContactInformation(5);

```

E. Использование выражения CASE в предложении HAVING

В следующем примере выражение CASE используется в предложении HAVING, чтобы ограничить строки, возвращаемые инструкцией SELECT. Инструкция возвращает максимальную почасовую ставку для каждой должности в таблице HumanResources.Employee. Предложение HAVING ограничивает должности, оставляя только те, которые заняты мужчинами с максимальной почасовой ставкой более 40 долларов или женщинами с максимальной почасовой ставкой более 42 долларов.

SQLКопировать

```

USE AdventureWorks2012;
GO
SELECT JobTitle, MAX(ph1.Rate)AS MaximumRate
FROM HumanResources.Employee AS e
JOIN HumanResources.EmployeePayHistory AS ph1 ON e.BusinessEntityID =
ph1.BusinessEntityID
GROUP BY JobTitle
HAVING (MAX(CASE WHEN Gender = 'M'
                THEN ph1.Rate
                ELSE NULL END) > 40.00
        OR MAX(CASE WHEN Gender = 'F'
                THEN ph1.Rate
                ELSE NULL END) > 42.00)
ORDER BY MaximumRate DESC;

```

Примеры: Azure Synapse Analytics и Параллельное хранилище данных Ж. Использование инструкции SELECT с выражением CASE

При использовании в инструкции SELECT выражение CASE позволяет заменять значения в результирующем наборе в зависимости от результатов сравнения. В приведенном ниже примере выражение CASE используется для изменения способа отображения категорий линейки продуктов с целью сделать их более понятными. Если значение отсутствует, выводится текст "Not for sale".

SQLКопировать

```
-- Uses AdventureWorks

SELECT ProductAlternateKey, Category =
    CASE ProductLine
        WHEN 'R' THEN 'Road'
        WHEN 'M' THEN 'Mountain'
        WHEN 'T' THEN 'Touring'
        WHEN 'S' THEN 'Other sale items'
        ELSE 'Not for sale'
    END,
    EnglishProductName
FROM dbo.DimProduct
ORDER BY ProductKey;
```

3. Использование выражения CASE в инструкции UPDATE

В следующем примере выражение CASE используется в инструкции UPDATE, чтобы определить значение, установленное в столбце VacationHours для сотрудников, у которых столбец SalariedFlag имеет значение 0. Если при вычитании 10 часов из VacationHours получается отрицательное значение, VacationHours увеличивается на 40 часов. В противном случае значение VacationHours увеличивается на 20 часов.

SQLКопировать

```
-- Uses AdventureWorks

UPDATE dbo.DimEmployee
SET VacationHours =
    ( CASE
        WHEN ((VacationHours - 10.00) < 0) THEN VacationHours + 40
        ELSE (VacationHours + 20.00)
    END
    )
WHERE SalariedFlag = 0;
```

12)

Для фильтрации в команде SELECT применяется оператор **WHERE**. После этого оператора ставится условие, которому должна соответствовать строка:

```
1 WHERE условие
```

Если условие истинно, то строка попадает в результирующую выборку. В качестве можно использовать операции сравнения. Эти операции сравнивают два выражения. В T-SQL можно применять следующие операции сравнения:

- **=**: сравнение на равенство (в отличие от си-подобных языков в T-SQL для сравнения на равенство используется один знак равно)
- **<>**: сравнение на неравенство
- **<**: меньше чем
- **>**: больше чем
- **!<**: не меньше чем
- **!>**: не больше чем
- **<=**: меньше чем или равно
- **>=**: больше чем или равно

Например, найдем всех товары, производителем которых является компания Samsung:

```
1 SELECT * FROM Products
2 WHERE Manufacturer = 'Samsung'
```

Стоит отметить, что в данном случае регистр не имеет значение, и мы могли бы использовать для поиска и строку "Samsung", и "SAMSUNG", и "samsung". Все эти варианты давали бы эквивалентный результат выборки.

Другой пример - найдем все товары, у которых цена больше 45000:

```
1 SELECT * FROM Products
2 WHERE Price > 45000
```

В качестве условия могут использоваться и более сложные выражения. Например, найдем все товары, у которых совокупная стоимость больше 200 000:

```
1 SELECT * FROM Products
2 WHERE Price * ProductCount > 200000
```

Логические операторы

Для объединения нескольких условий в одно могут использоваться логические операторы. В T-SQL имеются следующие логические операторы:

- **AND**: операция логического И. Она объединяет два выражения:

```
1  выражение1 AND выражение2
```

- Только если оба этих выражения одновременно истинны, то и общее условие оператора AND также будет истинно. То есть если и первое условие истинно, и второе.
- **OR**: операция логического ИЛИ. Она также объединяет два выражения:

```
1  выражение1 OR выражение2
```

- Если хотя бы одно из этих выражений истинно, то общее условие оператора OR также будет истинно. То есть если или первое условие истинно, или второе.
- **NOT**: операция логического отрицания. Если выражение в этой операции ложно, то общее условие истинно.

```
1  NOT выражение
```

Если эти операторы встречаются в одном выражении, то сначала выполняется NOT, потом AND и в конце OR.

Например, выберем все товары, у которых производитель Samsung и одновременно цена больше 50000:

```
1  SELECT * FROM Products
2  WHERE Manufacturer = 'Samsung' AND Price > 50000
```

Теперь изменим оператор на OR. То есть выберем все товары, у которых либо производитель Samsung, либо цена больше 50000:

```
1  SELECT * FROM Products
2  WHERE Manufacturer = 'Samsung' OR Price > 50000
```

Применение оператора NOT - выберем все товары, у которых производитель не Samsung:


```
1 SELECT * FROM Products
2 WHERE NOT Manufacturer = 'Samsung'
```

Но в большинстве случаев вполне можно обойтись без оператора NOT. Так, в предыдущий пример мы можем переписать следующим образом:

```
1 SELECT * FROM Products
2 WHERE Manufacturer <> 'Samsung'
```

Также в одной команде SELECT можно использовать сразу несколько операторов:

```
1 SELECT * FROM Products
2 WHERE Manufacturer = 'Samsung' OR Price > 30000 AND ProductCount > 2
```

Так как оператор AND имеет более высокий приоритет, то сначала будет выполняться подвыражение `Price > 30000 AND ProductCount > 2`, и только потом оператор OR. То есть здесь выбираются товары, которых на складе больше 2 и у которых одновременно цена больше 30000, либо те товары, производителем которых является Samsung.

С помощью скобок мы также можем переопределить порядок операций:

```
1 SELECT * FROM Products
2 WHERE (Manufacturer = 'Samsung' OR Price > 30000) AND ProductCount > 2
```

IS NULL

Ряд столбцов может допускать значение NULL. Это значение не эквивалентно пустой строке ". NULL представляет полное отсутствие какого-либо значения. И для проверки на наличие подобного значения применяется оператор **IS NULL**.

Например, выберем все товары, у которых не установлено поле ProductCount:

```
1 SELECT * FROM Products
2 WHERE ProductCount IS NULL
```

Если, наоборот, необходимо получить строки, у которых поле ProductCount не равно NULL, то можно использовать оператор NOT:

```
1  SELECT * FROM Products
2  WHERE ProductCount IS NOT NULL
```

Описание OFFSET-FETCH

OFFSET-FETCH – это конструкция языка Transact-SQL, которая является частью ORDER BY, и позволяет применять фильтр к результирующему, уже отсортированному, набору данных.

OFFSET-FETCH предназначена как раз для разбиения результирующего набора на части (страницы), а также для ограничения количества строк.

В языке T-SQL есть оператор TOP, с помощью которого мы можем примерно так же ограничивать количество строк, возвращаемых запросом. Однако TOP не позволяет пропускать строки, т.е. мы всегда получаем строки, начиная с первой, учитывая сортировку, т.е. с помощью сортировки мы можем влиять на результат.

Используя конструкцию OFFSET-FETCH, мы уже можем пропускать определённое количество строк.

Заметка! Всем тем, кто только начинает свое знакомство с языком SQL, рекомендую прочитать книгу «*SQL код*» – это самоучитель по языку SQL, которую написал я, и в которой я подробно, и в то же время простым языком, рассказываю о языке SQL.

Упрощённый синтаксис OFFSET-FETCH

ORDER BY *Выражение для сортировки*

OFFSET *Целое число* ROWS FETCH NEXT *Целое число* ROWS ONLY

Где,

- ORDER BY – инструкция для сортировки данных, возвращаемых запросом;
- OFFSET *Целое число* ROWS – инструкция задает количество строк, которые необходимо пропустить. Вместо ROWS можно использовать ключевое слово ROW, они эквиваленты. Однако ROWS и ROW могут сделать код более читабельным, например, ROWS использовать для пропуска нескольких строк, а ROW — для пропуска одной строки (т.е. единственное и множественное число, но снова повторюсь, они взаимозаменяемы);
- FETCH NEXT *Целое число* ROWS ONLY – инструкция задает количество строк, которые необходимо вернуть, после обработки инструкции OFFSET. Вместо ROWS можно использовать ключевое слово ROW, они эквиваленты. Также вместо NEXT можно использовать ключевое слово FIRST.

Важные замечания по использованию OFFSET-FETCH

- OFFSET-FETCH – это часть ORDER BY, без сортировки использовать конструкцию OFFSET-FETCH не получится;

- Инструкцию OFFSET можно использовать без указания FETCH, а вот FETCH использовать без указания OFFSET нельзя, т.е. FETCH требует обязательного наличия OFFSET;
- Не поддерживается совместная работа операторов TOP и OFFSET-FETCH в одном запросе SELECT.

Примеры использования OFFSET-FETCH в T-SQL

Сейчас давайте рассмотрим несколько примеров использования конструкции OFFSET-FETCH в языке T-SQL, но сначала давайте определимся с исходными данными.



Исходные данные для примеров

Допустим, у нас есть таблица TestTable, и она содержит следующие данные. В качестве сервера у меня выступает Microsoft SQL Server 2016 Express.

```
--Создание таблицы
CREATE TABLE TestTable(
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Price] [Money] NULL
)
GO
--Добавление строк в таблицу
INSERT INTO TestTable(ProductName, Price)
VALUES ('Системный блок', 300),
       ('Монитор', 200),
       ('Клавиатура', 100),
       ('Мышь', 50),
       ('Принтер', 200),
       ('Сканер', 150),
       ('Телефон', 250),
       ('Планшет', 300)
GO
--Выборка данных
```

```
SELECT * FROM TestTable
```

```
--Создание таблицы
CREATE TABLE TestTable(
    [ProductId]      [INT] IDENTITY(1,1) NOT NULL,
    [ProductName]    [VARCHAR](100) NOT NULL,
    [Price]          [Money] NULL
)
GO
--Добавление строк в таблицу
INSERT INTO TestTable(ProductName, Price)
VALUES ('Системный блок', 300),
       ('Монитор', 200),
       ('Клавиатура', 100),
       ('Мышь', 50),
       ('Принтер', 200),
       ('Сканер', 150),
       ('Телефон', 250),
       ('Планшет', 300)
GO
--Выборка данных
SELECT * FROM TestTable
```

100 %

Результаты		Сообщения	
	ProductId	ProductName	Price
1	1	Системный блок	300,00
2	2	Монитор	200,00
3	3	Клавиатура	100,00
4	4	Мышь	50,00
5	5	Принтер	200,00
6	6	Сканер	150,00
7	7	Телефон	250,00
8	8	Планшет	300,00

Заметка! [Обзор инструментов для работы с Microsoft SQL Server.](#)

OFFSET-FETCH – пропуск первых 3 строк

В этом примере мы пропустим первые три строки результирующего набора и вернем все последующие строки. Для этого мы просто напишем OFFSET 3 ROWS после определения инструкции ORDER BY.

```
--Пропуск первых 3 строк
SELECT * FROM TestTable
ORDER BY ProductId
OFFSET 3 ROWS
```

```
--Пропуск первых 3 строк
SELECT * FROM TestTable
ORDER BY ProductId
OFFSET 3 ROWS
```

	ProductId	ProductName	Price
1	4	Мышь	50,00
2	5	Принтер	200,00
3	6	Сканер	150,00
4	7	Телефон	250,00
5	8	Планшет	300,00

OFFSET-FETCH – пропуск первых 3 строк и возвращение следующих 3

В данном случае мы также пропустим первые три строки, только дополнительно мы еще укажем инструкцию `FETCH NEXT 3 ROWS ONLY`, которая будет говорить SQL серверу о том, что нужно вернуть не все последующие строки, а только 3 следующие.

Как Вы понимаете, значение 3 в обоих случаях можно изменять на то значение, которое нужно Вам, также вместо константы (*т.е. цифры 3*) можно подставлять и переменные, и выражения, которые возвращают целое значение.

```
--Пропуск первых 3 строк и возвращение следующих 3
SELECT * FROM TestTable
ORDER BY ProductId
OFFSET 3 ROWS FETCH NEXT 3 ROWS ONLY
```

```
--Пропуск первых 3 строк и возвращение следующих 3
SELECT * FROM TestTable
ORDER BY ProductId
OFFSET 3 ROWS FETCH NEXT 3 ROWS ONLY
```

	ProductId	ProductName	Price
1	4	Мышь	50,00
2	5	Принтер	200,00
3	6	Сканер	150,00

Заметка! Все возможности языка SQL и T-SQL очень подробно рассматриваются на моих [курсах по T-SQL](#), с помощью которых Вы «с нуля» научитесь работать с SQL и программировать на T-SQL в Microsoft SQL Server.

Вот мы с Вами и рассмотрели конструкцию OFFSET-FETCH языка T-SQL, надеюсь, всё было понятно, удачи!

SQL SELECT TOP

Инструкция SELECT TOP используется для указания количества возвращаемых записей.

Инструкция SELECT TOP полезно для больших таблиц с тысячами записей. Возврат большого количества записей может повлиять на производительность.

Примечание: Не все базы данных поддерживают SELECT TOP. MySQL поддерживает предложение LIMIT для выбора ограниченного числа записей, в то время как Oracle использует ROWNUM.

Синтаксис SQL Server / MS Access:

```
SELECT TOP number | percent column_name(s)  
FROM table_name  
WHERE condition;
```

14)



Схемы используются в модели безопасности компонента Database Engine для упрощения взаимоотношений между пользователями и объектами, и, следовательно, схемы имеют очень большое влияние на взаимодействие пользователя с компонентом Database Engine. В этом разделе рассматривается роль схем в безопасности компонента Database Engine. В первом подразделе описывается взаимодействие между схемами и пользователями, а во втором обсуждаются все три инструкции языка Transact-SQL, применяемые для создания и модификации схем.

Разделение пользователей и схем

Схема - это коллекция объектов базы данных, имеющая одного владельца и формирующая одно пространство имен. (Две таблицы в одной и той же схеме не могут иметь одно и то же имя.) Компонент Database Engine поддерживает именованные схемы с использованием понятия принципала (principal). Как уже упоминалось, принципом может быть индивидуальный принципал и групповой принципал.

Индивидуальный принципал представляет одного пользователя, например, в виде регистрационного имени или учетной записи пользователя Windows. Групповым принципом может быть группа пользователей, например, роль или группа Windows. Принципы владеют схемами, но владение схемой может быть с легкостью передано другому принципу без изменения имени схемы.

Отделение пользователей базы данных от схем дает значительные преимущества, такие как:

- один принципал может быть владельцем нескольких схем;
- несколько индивидуальных принципов могут владеть одной схемой посредством членства в ролях или группах Windows;
- удаление пользователя базы данных не требует переименования объектов, содержащихся в схеме этого пользователя.

Каждая база данных имеет схему по умолчанию, которая используется для определения имен объектов, ссылки на которые делаются без указания их полных уточненных имен. В схеме по умолчанию указывается первая схема, в которой сервер базы данных будет выполнять поиск для разрешения имен объектов. Для настройки и изменения схемы по умолчанию применяется параметр **DEFAULT_SCHEMA** инструкции CREATE USER или ALTER USER. Если схема по умолчанию DEFAULT_SCHEMA не определена, в качестве схемы по умолчанию пользователю базы данных назначается **схема dbo**.

Инструкция CREATE SCHEMA

В примере ниже показано создание схемы и ее использование для управления безопасностью базы данных. Прежде чем выполнять этот пример, необходимо создать пользователей базы данных Alex и Vasya, как будет описано в следующей статье (вы можете вернуться к этим примерам позже).

```
USE SampleDb;

GO

CREATE SCHEMA poco AUTHORIZATION Vasya

GO

CREATE TABLE Product (

    Number CHAR(10) NOT NULL UNIQUE,

    Name CHAR(20) NULL,

    Price MONEY NULL);

GO

CREATE VIEW view_Product

    AS SELECT Number, Name

    FROM Product;

GO

GRANT SELECT TO Alex;

DENY UPDATE TO Alex;
```

В этом примере создается схема poco, содержащая таблицу Product и представление view_Product. Пользователь базы данных Vasya является принципалом уровня базы данных, а также владельцем схемы. (Владелец схемы указывается посредством *параметра AUTHORIZATION*. Принципал может быть

владельцем других схем и не может использовать текущую схему в качестве схемы по умолчанию.)

Две другие инструкции, применяемые для работы с разрешениями для объектов базы данных, GRANT и DENY, подробно рассматриваются позже. В этом примере инструкция GRANT предоставляет инструкции SELECT разрешения для всех создаваемых в схеме объектов, тогда как инструкция DENY запрещает инструкции UPDATE разрешения для всех объектов схемы.

С помощью инструкции **CREATE SCHEMA** можно создать схему, сформировать содержащиеся в этой схеме таблицы и представления, а также предоставить, запретить или удалить разрешения на защищаемый объект. Как упоминалось ранее, защищаемые объекты - это ресурсы, доступ к которым регулируется системой авторизации SQL Server. Существует три основные области защищаемых объектов: сервер, база данных и схема, которые содержат другие защищаемые объекты, такие как регистрационные имена, пользователи базы данных, таблицы и хранимые процедуры.

Инструкция CREATE SCHEMA является атомарной. Иными словами, если в процессе выполнения этой инструкции происходит ошибка, не выполняется ни одна из содержащихся в ней подинструкций.

Порядок указания создаваемых в инструкции CREATE SCHEMA объектов базы данных может быть произвольным, с одним исключением: представление, которое ссылается на другое представление, должно быть указано после представления, на которое оно ссылается.

Принципалом уровня базы данных может быть пользователь базы данных, роль или роль приложения. (Роли и роли приложения рассматриваются в одной из следующих статей.) Принципал, указанный в предложении AUTHORIZATION инструкции CREATE SCHEMA, является владельцем всех объектов, созданных в этой схеме. Владение содержащихся в схеме объектов можно передавать любому принципалу уровня базы данных посредством инструкции **ALTER AUTHORIZATION**.

Для исполнения инструкции CREATE SCHEMA пользователь должен обладать правами базы данных CREATE SCHEMA. Кроме этого, для создания объектов, указанных в инструкции CREATE SCHEMA, пользователь должен иметь соответствующие разрешения CREATE.

Инструкция ALTER SCHEMA

Инструкция **ALTER SCHEMA** перемещает объекты между разными схемами одной и той же базы данных. Инструкция ALTER SCHEMA имеет следующий синтаксис:

Использование инструкции ALTER SCHEMA показано в примере ниже:

```
USE AdventureWorks2012;
```

```
ALTER SCHEMA HumanResources TRANSFER Person.ContactType;
```

Здесь изменяется схема HumanResources базы данных AdventureWorks2012, перемещая в нее таблицу ContactType из схемы Person этой же базы данных. Инструкцию ALTER SCHEMA можно использовать для перемещения объектов между разными схемами только одной и той же базы данных. (Отдельные объекты в схеме можно изменить посредством инструкции ALTER TABLE или ALTER VIEW.)

Инструкция DROP SCHEMA

Для удаления схемы из базы данных применяется **инструкция DROP SCHEMA**. Схему можно удалить только при условии, что она не содержит никаких объектов. Если схема содержит объекты, попытка выполнить инструкцию DROP SCHEMA будет unsuccessful.

Как указывалось ранее, владельца схемы можно изменить посредством инструкции ALTER AUTHORIZATION, которая изменяет владение сущностью. Язык Transact-SQL не поддерживает инструкции CREATE AUTHORIZATION и DROP AUTHORIZATION. Владелец схемы указывается с помощью инструкции CREATE SCHEMA

Вычисляемые столбцы в таблицах

Вычисляемый столбец – это виртуальный столбец таблицы, который вычисляется на основе выражения, в этих выражениях могут участвовать другие столбцы этой же таблицы. Такие столбцы физически не хранятся, их значения рассчитываются каждый раз при обращении к ним. Это поведение по умолчанию, но можно сделать так, чтобы вычисляемые столбцы физически хранились, для этого нужно указать ключевое слово **PERSISTED** при создании подобного столбца. В данном случае значения данного столбца будут обновляться, когда будут вноситься любые изменения в столбцы, входящие в вычисляемое выражение.

Вычисляемые столбцы нужны для того, чтобы было проще и надежнее получить результат каких-то постоянных вычислений. Например, при обращении к таблице, Вы всегда в SQL запросе применяете какую-нибудь формулу (*один столбец перемножаете с другим или что-то в этом роде, хотя формула может быть и сложнее*), так вот, если в таблице определить вычисляемый столбец, указав в его определении нужную формулу, Вам больше не нужно будет каждый раз писать эту формулу в SQL запросе в инструкции **SELECT**. Вам достаточно обратиться к определенному столбцу (*вычисляемому столбцу*), который автоматически при выводе значений применяет эту формулу. При этом этот столбец можно использовать в запросах также как обычный столбец, например, в секциях **WHERE** (*в условии*) или в **ORDER BY** (*в сортировке*).

Также важно понимать, что вычисляемый столбец не может быть указан в инструкциях **INSERT** или **UPDATE** в качестве целевого столбца.

Создание вычисляемого столбца при создании новой таблицы в Microsoft SQL Server

Давайте представим, что нам нужно создать таблицу, в которой будут храниться товары с указанием их количества и цены, при этом мы хотим постоянно знать, на какую сумму у нас того или иного товара, т.е. нам нужно умножить количество на цену. Чтобы этого не делать каждый раз в запросе, мы и создаем вычисляемый столбец, в определении которого указываем соответствующую формулу. Также в данном примере мы укажем ключевое слово **PERSISTED**, для того чтобы наш столбец хранился физически, а обновлялся, т.е. заново вычислялся, только если изменятся значения в столбцах с количеством или ценой.

Примечание! Все примеры выполнены в [Microsoft SQL Server 2016 Express](#).

```
--Создание таблицы с вычисляемым столбцом
CREATE TABLE TestTable (
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Quantity] [SMALLINT] NULL,
    [Price] [Money] NULL,
    [Summa] AS ([Quantity] * [Price]) PERSISTED --
Вычисляемый столбец
```

```

)

--Добавление данных в таблицу
INSERT INTO TestTable
VALUES ('Портфель', 1, 500),
      ('Карандаш', 5, 20),
      ('Тетрадь', 10, 50)

--Выборка данных
SELECT * FROM TestTable

```

```

--Создание таблицы с вычисляемым столбцом
CREATE TABLE TestTable (
    [ProductId] [INT] IDENTITY(1,1) NOT NULL,
    [ProductName] [VARCHAR](100) NOT NULL,
    [Quantity] [SMALLINT] NULL,
    [Price] [Money] NULL,
    [Summa] AS ([Quantity] * [Price]) PERSISTED --Вычисляемый столбец
)

--Добавление данных в таблицу
INSERT INTO TestTable
VALUES ('Портфель', 1, 500),
      ('Карандаш', 5, 20),
      ('Тетрадь', 10, 50)

--Выборка данных
SELECT * FROM TestTable

```

100 %

	ProductId	ProductName	Quantity	Price	Summa
1	1	Портфель	1	500,00	500,00
2	2	Карандаш	5	20,00	100,00
3	3	Тетрадь	10	50,00	500,00

Добавление вычисляемого столбца в существующую таблицу в Microsoft SQL Server

А сейчас давайте допустим, что возникла необходимость знать еще и сумму с учетом некоего статического коэффициента (например, 1,7). Но нам об этом сказали уже после того, как мы создали таблицу, иными словами, нам нужно добавить вычисляемый столбец в существующую таблицу.

```

--Добавление вычисляемого столбца в таблицу
ALTER TABLE TestTable ADD SummaALL AS ([Quantity] * [Price] *
1.7);

--Выборка данных
SELECT * FROM TestTable

```

--Добавление вычисляемого столбца в таблицу

```
ALTER TABLE TestTable ADD SummaALL AS ([Quantity] * [Price] * 1.7);
```

--Выборка данных

```
SELECT * FROM TestTable
```

100 %

Результаты

Сообщения

	ProductId	ProductName	Quantity	Price	Summa	SummaALL
1	1	Портфель	1	500,00	500,00	850.00000
2	2	Карандаш	5	20,00	100,00	170.00000
3	3	Тетрадь	10	50,00	500,00	850.00000

Обеспечение целостности данных

Во время проектирования базы данных вы должны заботиться о целостности данных. Правильная структура таблиц позволяет защитить данные от нарушения связей и внесения неверных значений. Вы должны определить наилучший путь обеспечения целостности данных. Целостность данных основывается на стойкости и точности данных, которые хранит база данных.

Существуют различные типы целостности данных:

- Целостность полей - указывает набор значений данных, которые являются правильными для поля, и определяет, возможно ли использование нулевого значения. Например, поле для хранения пола человека может содержать одно из двух значений - М или Ж. Во-первых, этого достаточно, во-вторых, других значений пола просто не бывает и мы должны запретить ввод других букв в данное поле. Целостность полей часто всего (и лучше) обеспечивается с помощью ограничения CHECK, формата (с помощью шаблона) или региона возможных значений для поля.
- Целостность таблицы - требуют, чтобы все строки в таблице имели уникальный идентификатор, называемый первичным ключом. Может ли первичный ключ изменяться, или может ли строка удаляться, зависит от уровня целостности. Например, в некоторых случаях можно разрешить удаление записей, но чаще всего оно должно быть запрещено. Не желательно терять данные, потому что мы в последствии не сможем узнать историю изменений в таблице.
- Целостность ссылок - подразумевает отношения между первичным ключом (таблицы, на которую ссылаются) и внешним ключом (таблицы, которая ссылается на другую) всегда защищенными. Строка основной таблицы, на которую ссылаются, не может быть удалена и первичный ключ не может быть изменен, если вторичный ключ ссылается на строку, пока не будет уничтожена связь. Иначе связь нарушается и восстановить ее потом становится проблематичным. Вы можете назначить отношения внутри таблицы или между несколькими отдельными таблицами с помощью встроенных в SQL Server средств, не надеясь на возможности языка программирования, который вы используете для доступа к данным. Конечно же, связь между таблицами можно навести и без внешних ключей, но в этом случае сервер не гарантирует целостность. Вся ответственность ложиться на программиста.

Все операторы, необходимые для реализации всех уровней целостности нам уже знакомы. Я специально вынес рассмотрение теории обеспечения целостности после того, как мы узнали средства. Знание операторов упростит понимание теоретических данных.

Как мы можем гарантировать целостность данных? Для этого существует два способа: описанная целостность данных и предшествующая целостность данных. Пока эти понятия не понятны, но после того, как вы увидите, какими средствами достигается тот, или иной способ, все встанет на свои места.

Описанная целостность данных - вы объявляете критерии, которые данные должны содержать как часть описания объекта и после этого SQL Server автоматически гарантирует, что данные соответствуют критериям. Уже можно догадаться, что такая целостность обеспечивается с помощью ограничений CHECK, DEFAULT и внешнего ключа.

Описанная целостность является частью объявления базы данных, и объявляется с помощью ограничений, которые вы можете назначить колонкам и таблицам напрямую.

Предшествующая целостность данных - это программа, которая определяет критерии, которым должны соответствовать данные. Этот метод обеспечивается с помощью процедур и триггеров (о них мы поговорим в главе 3), которые могут выполняться на сервере или с помощью кода программ в клиентском приложении.

Вы должны минимизировать использование этого метода для упрощения бизнес логики и ошибок, но иногда без триггера не возможно гарантировать, что таблицы будут содержать нужные или разрешенные значения.

В определении столбца обратим внимание на параметр **IDENTITY**, который указывает, что соответствующий столбец будет *столбцом-счетчиком*. Для таблицы может быть определен только один столбец с таким свойством. Можно дополнительно указать начальное значение и шаг приращения. Если эти значения не указываются, то по умолчанию они оба равны 1. Если с ключевым словом **IDENTITY** указано **NOT FOR REPLICATION**, то сервер не будет выполнять автоматического генерирования значений для этого столбца, а разрешит вставку в столбец произвольных значений.

В качестве ограничений используются *ограничения столбца* и *ограничения таблицы*. Различие между ними в том, что **ограничение столбца** применяется только к определенному полю, а **ограничение таблицы** - к группам из одного или более полей.

```
<ограничение_столбца> ::=
[ CONSTRAINT имя_ограничения ]
{ [ NULL | NOT NULL ]
| [ {PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ]
[ WITH FILLFACTOR=фактор_заполнения ]
[ ON {имя_группы_файлов | DEFAULT } ] ] ]
| [ [ FOREIGN KEY ]
REFERENCES имя_род_таблицы
[ (имя_столбца_род_таблицы) ]
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]
[ NOT FOR REPLICATION ] ]
| CHECK [ NOT FOR REPLICATION ] (<лог_выражение>) }
```

```
<ограничение_таблицы> ::=
[ CONSTRAINT имя_ограничения ]
{ [ {PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ]
{ (имя_столбца [ASC | DESC] [, ...n]) }
[ WITH FILLFACTOR=фактор_заполнения ]
[ ON {имя_группы_файлов | DEFAULT } ] ]
| FOREIGN KEY [ (имя_столбца [, ...n]) ]
REFERENCES имя_род_таблицы
[ (имя_столбца_род_таблицы [, ...n]) ]
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]
| NOT FOR REPLICATION ]
| CHECK [ NOT FOR REPLICATION ] (лог_выражение) }
```

Рассмотрим отдельные параметры представленных конструкций, связанные с ограничениями *целостности данных*. *Ограничения целостности* имеют приоритет над триггерами, *правилами* и значениями по умолчанию. К **ограничениям целостности** относятся *ограничение первичного ключа* **PRIMARY KEY**, *ограничение внешнего ключа* **FOREIGN KEY**, *ограничение уникальности* **UNIQUE**, *ограничение значения* **NULL**, *ограничение на проверку* **CHECK**.

Ограничение первичного ключа (PRIMARY KEY)

Таблица обычно имеет столбец или комбинацию столбцов, значения которых уникально идентифицируют каждую строку в таблице. Этот столбец (или столбцы) называется **первичным ключом** таблицы и нужен для обеспечения ее целостности. Если в *первичный ключ* входит более одного столбца, то значения в пределах одного столбца могут дублироваться, но любая комбинация значений всех столбцов *первичного ключа* должна быть уникальна.

При создании *первичного ключа* SQL Server автоматически создает уникальный индекс для столбцов, входящих в *первичный ключ*. Он ускоряет доступ к данным этих столбцов при использовании *первичного ключа* в запросах.

Таблица может иметь только одно ограничение **PRIMARY KEY**, причем ни один из включенных в *первичный ключ* столбцов не может принимать значение **NULL**. При попытке использовать в качестве *первичного ключа* столбец (или группу столбцов), для которого *ограничения первичного ключа* не выполняются, *первичный ключ* создан не будет, а система выдаст сообщение об ошибке.

Поскольку ограничение **PRIMARY KEY** гарантирует уникальность данных, оно часто определяется для *столбцов-счетчиков*. Создание *ограничения целостности PRIMARY KEY* возможно как при создании, так и при *изменении таблицы*. Одним из назначений *первичного ключа* является обеспечение *ссылочной целостности* данных нескольких таблиц. Естественно, это может быть реализовано только при определении соответствующих *внешних ключей* в других таблицах.

Ограничение внешнего ключа (FOREIGN KEY)

Ограничение внешнего ключа - это основной механизм для поддержания *ссылочной целостности* между таблицами реляционной базы данных. Столбец дочерней таблицы, определенный в качестве *внешнего ключа* в параметре **FOREIGN KEY**, применяется для ссылки на столбец родительской таблицы, являющийся в ней *первичным ключом*. Имя родительской таблицы и столбца ее *первичного ключа* указываются в предложении **REFERENCES**. Данные в столбцах, определенных в качестве *внешнего ключа*, могут принимать только такие же значения, какие находятся в связанных с ним столбцах *первичного ключа* родительской таблицы. Совпадение имен столбцов для связи дочерней и родительской таблиц необязательно. *Первичный ключ* может быть определен для столбца с одним именем, в то время как столбец, на который наложено ограничение **FOREIGN KEY**, может иметь совершенно другое имя. Единственным требованием остается соответствие столбцов по типу и размеру данных.

На *первичный ключ* могут ссылаться не только столбцы других таблиц, но и столбцы, расположенные в той же таблице, что и собственно *первичный ключ*; это позволяет создавать рекурсивные структуры.

Внешний ключ может быть связан не только с *первичным ключом* другой таблицы. Он может быть определен и для столбцов с ограничением **UNIQUE** второй таблицы или любых других столбцов, но таблицы должны находиться в одной базе данных.

Столбцы *внешнего ключа* могут содержать значение **NULL**, однако проверка на ограничение **FOREIGN KEY** игнорируется. *Внешний ключ* может быть проиндексирован, тогда сервер будет быстрее отыскивать нужные данные. *Внешний ключ* определяется как при создании, так и при *изменении таблиц*.

Ограничение *ссылочной целостности* задает требование, согласно которому для каждой записи в дочерней таблице должна иметься запись в родительской таблице. При этом изменение значения столбца связи в записи родительской таблицы при наличии дочерней записи блокируется, равно как и удаление родительской записи (запрет каскадного изменения и удаления), что гарантируется параметрами **ON DELETE NO ACTION** и **ON UPDATE NO ACTION**, принятыми по умолчанию. Для разрешения каскадного воздействия следует использовать параметры **ON DELETE CASCADE** и **ON UPDATE CASCADE**.

Ограничение уникального ключа (UNIQUE)

Это ограничение задает требование уникальности значения поля (столбца) или группы полей (столбцов), входящих в *уникальный ключ*, по отношению к другим записям. Ограничение **UNIQUE** для столбца таблицы похоже на *первичный ключ*: для каждой строки данных в нем должны содержаться уникальные значения. Установив для некоторого столбца *ограничение первичного ключа*, можно одновременно установить для другого столбца ограничение **UNIQUE**. Отличие в *ограничении первичного* и *уникального ключа* заключается в том, что *первичный ключ* служит как для упорядочения данных в таблице, так и для соединения связанных между собой таблиц. Кроме того, при использовании ограничения **UNIQUE** допускается существование значения **NULL**, но лишь единственный раз.

Ограничение на значение (NOT NULL)

Для каждого столбца таблицы можно установить ограничение **NOT NULL**, запрещающее ввод в этот столбец нулевого значения.

Ограничение проверочное (CHECK) и правила

Данное ограничение используется для проверки допустимости данных, вводимых в конкретный столбец таблицы, т.е. ограничение **CHECK** обеспечивает еще один уровень защиты данных.

*Ограничения целостности **CHECK** задают диапазон возможных значений для столбца или столбцов. В основе ограничений целостности **CHECK** лежит использование логических выражений.*

Допускается применение нескольких ограничений **CHECK** к одному и тому же столбцу. В этом случае они будут применимы в той последовательности, в которой происходило их создание. Возможно применение одного и того же ограничения к разным столбцам и использование в логических выражениях значений других столбцов. Указание параметра **NOT FOR REPLICATION** предписывает не выполнять проверочных действий, если они выполняются подсистемой репликации.

*Проверочные ограничения могут быть реализованы и с помощью правил. **Правило** представляет собой самостоятельный объект базы данных, который связывается со столбцом таблицы или пользовательским типом данных. Причем одно и то же *правило* может быть одновременно связано с несколькими столбцами и пользовательскими типами данных, что является неоспоримым преимуществом. Однако существенный недостаток заключается в том, что с каждым столбцом или типом данных может быть связано только одно *правило*. Разрешается комбинирование *ограничений целостности **CHECK*** с *правилами*. В этом случае выполняется проверка соответствия вводимого значения как *ограничениям целостности*, так и *правилам*.*

Правило может быть создано командой:

```
CREATE RULE имя_правила AS выражение
```

Чтобы связать *правило* с тем или иным столбцом какой-либо таблицы, необходимо использовать системную хранимую процедуру:

```
sp_bindrule [@rulename=] 'rule'  
[@objname=] 'object_name'  
[,[@futureonly='futureonly_flag']]
```

Чтобы отменить *правила*, следует выполнить следующую процедуру:

```
sp_unbindrule [@objname=] 'object_name'  
[,[@futureonly='futureonly_flag']]
```

Удаление *правила* производится командой

```
DROP RULE {имя_правила} [,...n].
```

Ограничение по умолчанию (DEFAULT)

Столбцу может быть присвоено значение по умолчанию. Оно будет актуальным в том случае, если пользователь не введет в столбец никакого иного значения.

Отдельно необходимо отметить пользу от использования значений по умолчанию при добавлении нового столбца в таблицу. Если для добавляемого столбца не разрешено хранение значений **NULL** и не определено значение по умолчанию, то операция добавления столбца закончится неудачей.

При определении в таблице столбца с параметром **ROWGUIDCOL** сервер автоматически определяет для него значение по умолчанию в виде функции **NEWID()**. Таким образом, для каждой новой строки будет автоматически генерироваться глобальный уникальный идентификатор.

Дополнительным механизмом использования значений по умолчанию являются объекты базы данных, созданные командой:

```
CREATE DEFAULT имя_умолчания AS константа
```

Умолчание связывается с тем или иным столбцом какой-либо таблицы с помощью процедуры:

```
sp_bindefault [@defname=] 'default',  
[@objname=] 'object_name'  
[,[@futureonly=] 'futureonly_flag'],
```

где

```
'object_name'
```

может быть представлен как

```
'имя_таблицы.имя_столбца'
```

Удаление *ограничения по умолчанию* выполняется командой

```
DROP DEFAULT {имя_умолчания} [,...n]
```

если предварительно это ограничение было удалено из всех таблиц процедурой

```
sp_unbindefault [@objname=] 'object_name'  
[,[@futureonly=] 'futureonly_flag']
```

При *создании таблицы*, кроме рассмотренных приемов, можно указать необязательное ключевое слово **CONSTRAINT**, чтобы присвоить ограничению имя, уникальное в пределах базы данных.

Ключевые слова **CLUSTERED** и **NONCLUSTERED** позволяют создать для столбца *кластерный* или *некластерный индекс*. Для ограничения **PRIMARY KEY** по умолчанию создается *кластерный индекс*, а для ограничения **UNIQUE** - *некластерный*. В каждой таблице может быть создан лишь один *кластерный индекс*, отличительной особенностью которого является то, что в соответствии с ним изменяется физический порядок строк в таблице. **ASC** и **DESC** определяют метод упорядочения данных в индексе.

С помощью параметра **WITH FILLFACTOR=фактор_заполнения** задается степень заполнения индексных страниц при создании индекса. Значение фактора заполнения указывается в процентах и может изменяться в промежутке от 0 до 100.

Параметр **ON имя_группы_файлов** обозначает группу, в которой предполагается хранить таблицу.

Объединения JOIN — это объединение двух или более объектов базы данных по средствам определенного ключа или ключей или в случае cross join и вовсе без ключа. Под объектами здесь подразумевается различные таблицы, представления (views), табличные функции или просто подзапросы sql, т.е. все, что возвращает табличные данные.

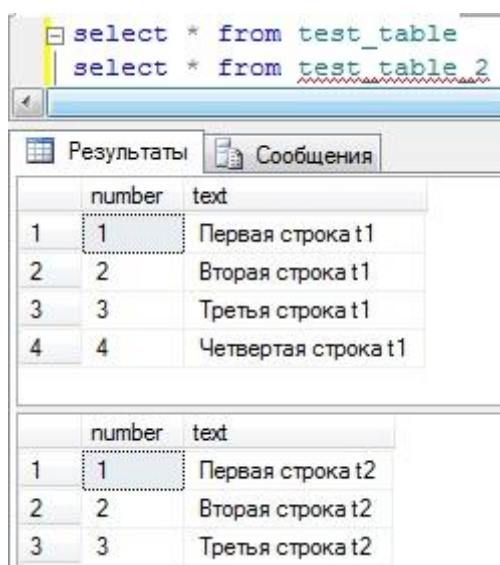
Объединение SQL LEFT и RIGHT JOIN

LEFT JOIN – это объединение данных по левому ключу, т.е. допустим, мы объединяем две таблицы по left join, и это значит что все данные из второй таблицы подтянутся к первой, а в случае отсутствия ключа выведется NULL значения, другими словами выведутся все данные из левой таблицы и все данные по ключу из правой таблицы.

RIGHT JOIN – это такое же объединение как и Left join только будут выводиться все данные из правой таблицы и только те данные из левой таблицы в которых есть ключ объединения.

Теперь давайте рассматривать примеры, и для начала создадим две таблицы:

Вот такие простенькие таблицы, И я для примера заполнил их вот такими данными:



```
select * from test_table
select * from test_table 2
```

	number	text
1	1	Первая строка t1
2	2	Вторая строка t1
3	3	Третья строка t1
4	4	Четвертая строка t1

	number	text
1	1	Первая строка t2
2	2	Вторая строка t2
3	3	Третья строка t2

Теперь давайте напишем запрос с объединением этих таблиц по ключу number, для начала по LEFT:

```

--t1 это псевдоним таблицы test_table
--t2 это псевдоним таблицы test_table_2
--t1.number=t2.number это приравнивание по ключу
select t1.number as t1_number, t1.text as t1_text, t2.number as t2_number, t2.text as t2_text
from test_table t1
left join test_table_2 t2 on t1.number=t2.number

```

	t1_number	t1_text	t2_number	t2_text
1	1	Первая строка t1	1	Первая строка t2
2	2	Вторая строка t1	2	Вторая строка t2
3	3	Третья строка t1	3	Третья строка t2
4	4	Четвертая строка t1	NULL	NULL

Как видите, здесь данные из таблицы t1 вывелись все, а данные из таблицы t2 не все, так как строки с number = 4 там нет, поэтому и вывелись NULL значения.

А что будет, если бы мы объединяли по средствам right join, а было бы вот это:

```

--t1 это псевдоним таблицы test_table
--t2 это псевдоним таблицы test_table_2
--t1.number=t2.number это приравнивание по ключу
select t1.number as t1_number, t1.text as t1_text, t2.number as t2_number, t2.text as t2_text
from test_table t1
right join test_table_2 t2 on t1.number=t2.number

```

	t1_number	t1_text	t2_number	t2_text
1	1	Первая строка t1	1	Первая строка t2
2	2	Вторая строка t1	2	Вторая строка t2
3	3	Третья строка t1	3	Третья строка t2

Другими словами, вывелись все строки из таблицы t2 и соответствующие записи из таблицы t1, так как все те ключи, которые есть в таблице t2, есть и в таблице t1, и поэтому у нас нет NULL значений.

Объединение SQL INNER JOIN

Inner join – это объединение когда выводятся все записи из одной таблицы и все соответствующие записи из другой таблице, а те записи которых нет в одной или в другой таблице выводиться не будут, т.е. только те записи которые соответствуют ключу. Кстати сразу скажу, что inner join это то же самое, что и просто join без Inner. Пример:


```
select t1.number as t1_number, t1.text as t1_text, t2.number as t2_number, t2.text as t2_text
from test_table t1
inner join test_table_2 t2 on t1.number=t2.number
```

	t1_number	t1_text	t2_number	t2_text
1	1	Первая строка t1	1	Первая строка t2
2	2	Вторая строка t1	2	Вторая строка t2
3	3	Третья строка t1	3	Третья строка t2

А теперь давайте попробуем объединить наши таблицы по двум ключам, для этого немного вспомним, как добавлять колонку в таблицу и как обновить данные через update, так как в наших таблицах всего две колонки, и объединять по текстовому полю как-то не хорошо. Для этого добавим колонки:

Обновим наши данные, просто проставим в колонку number2 значение 1:

И давайте напишем запрос с объединением по двум ключам:

И результат будет таким же, как и в предыдущем примере:

	t1_number	t1_text	t2_number	t2_text
1	1	Первая строка t1	1	Первая строка t2
2	2	Вторая строка t1	2	Вторая строка t2
3	3	Третья строка t1	3	Третья строка t2

Но если мы, допустим во второй таблице в одной строке изменим, поле number2 на значение скажем 2, то результат будет уже совсем другой.

Запрос тот же самый, а вот результат:

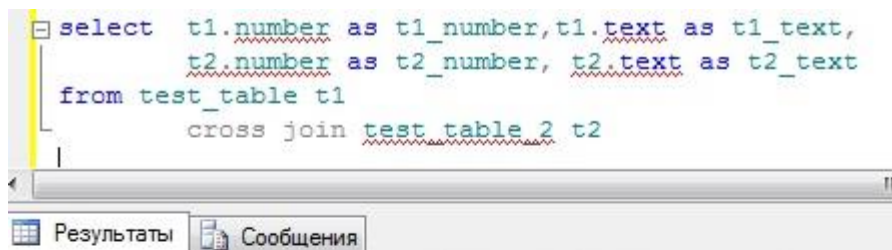
```
select t1.number as t1_number, t1.text as t1_text, t2.number as t2_number, t2.text as t2_text
from test_table t1
inner join test_table_2 t2 on t1.number=t2.number and t1.number2=t2.number2
```

	t1_number	t1_text	t2_number	t2_text
1	2	Вторая строка t1	2	Вторая строка t2
2	3	Третья строка t1	3	Третья строка t2

Как видите, по второму ключу у нас одна строка не вывелась.

Объединение SQL CROSS JOIN

CROSS JOIN – это объединение SQL по которым каждая строка одной таблицы объединяется с каждой строкой другой таблицы. Лично у меня это объединение редко требуется, но все равно иногда требуется, поэтому Вы также должны уметь его использовать. Например, в нашем случае получится, конечно, не понятно что, но все равно давайте попробуем, тем более синтаксис немного отличается:



```
select t1.number as t1_number, t1.text as t1_text,
       t2.number as t2_number, t2.text as t2_text
from test_table t1
     cross join test_table_2 t2
```

Результаты

	t1_number	t1_text	t2_number	t2_text
1	1	Первая строка t1	1	Первая строка t2
2	2	Вторая строка t1	1	Первая строка t2
3	3	Третья строка t1	1	Первая строка t2
4	4	Четвертая строка t1	1	Первая строка t2
5	1	Первая строка t1	2	Вторая строка t2
6	2	Вторая строка t1	2	Вторая строка t2
7	3	Третья строка t1	2	Вторая строка t2
8	4	Четвертая строка t1	2	Вторая строка t2
9	1	Первая строка t1	3	Третья строка t2
10	2	Вторая строка t1	3	Третья строка t2
11	3	Третья строка t1	3	Третья строка t2
12	4	Четвертая строка t1	3	Третья строка t2

Здесь у нас каждой строке таблицы test_table соответствует каждая строка из таблицы test_table_2, т.е. в таблице test_table у нас 4 строки, а в таблице test_table_2 3 строки 4 умножить 3 и будет 12, как и у нас вывелось 12 строк.

И напоследок, давайте покажу, как можно объединять несколько таблиц, для этого я, просто для примера, несколько раз объединю нашу первую таблицу со второй, смысла в объединение в данном случае, конечно, нет но, Вы увидите, как можно это делать и так приступим:

<pre> select t1.number as t1_number, t1.text as t1_text, t2.number as t2_number, t2.text as t2_text, t3.number as t3_number, t3.text as t3_text, t4.number as t4_number, t4.text as t4_text from test_table t1 left join test_table 2 t2 on t1.number=t2.number right join test_table 2 t3 on t1.number=t3.number inner join test_table 2 t4 on t1.number=t4.number </pre>								
<div> <div>Результаты</div> <div>Сообщения</div> </div>								
	t1_number	t1_text	t2_number	t2_text	t3_number	t3_text	t4_number	t4_text
1	1	Первая строка t1	1	Первая строка t2	1	Первая строка t2	1	Первая строка t2
2	2	Вторая строка t1	2	Вторая строка t2	2	Вторая строка t2	2	Вторая строка t2
3	3	Третья строка t1	3	Третья строка t2	3	Третья строка t2	3	Третья строка t2

Как видите, я здесь объединяю и по left и по right и по inner просто, для того чтобы это было наглядно.

С объединениями я думаю достаточно, тем более ничего сложного в них нет. Но на этом изучение SQL не закончено в следующих статьях мы продолжим, а пока тренируйтесь и пишите свои запросы. Удачи!

Outer join - все входит

23)

Существует два типа подзапросов: независимые и связанные. В независимых подзапросах вложенный запрос логически выполняется ровно один раз. Связанный запрос отличается от независимого тем, что его значение зависит от переменной, получаемой от внешнего запроса. Таким образом, вложенный запрос связанного подзапроса выполняется каждый раз, когда система получает новую строку от внешнего запроса. В этом разделе приводятся несколько примеров независимых подзапросов. Связанные подзапросы рассматриваются далее в следующей статье совместно с оператором соединения JOIN.

Независимый подзапрос может применяться со следующими операторами:

- операторами сравнения;
- оператором IN;
- операторами ANY и ALL.

Что такое - Коррелированный Подзапрос?

Коррелированный подзапрос - это оператор SELECT, вложенный в другой оператор T-SQL, и ссылающийся на один или несколько столбцов внешнего запроса. Поэтому можно сказать, что коррелированный подзапрос зависит от внешнего запроса. Это - главное различие между коррелированным и простым подзапросом. Простой подзапрос не ссылается на внешний запрос, он может быть выполнен независимо от него. После того как коррелированный подзапрос будет связан с внешним запросом, он будет возвращать сообщение о синтаксической ошибке, если попытается вызвать самого себя. Коррелированный подзапрос может быть исполнен несколько раз в процессе обработки оператора T-SQL, содержащего такой подзапрос. Он будет исполняться для каждой строки, отобранной во внешнем запросе. На каждом из этих шагов поля внешнего запроса, на которые ссылается коррелированный подзапрос, будут сравниваться с результатами выборки коррелированного подзапроса. Результат выполнения коррелированного подзапроса определит, попадет ли строка внешнего запроса в результирующую выборку.

[\[В начало\]](#)

Применение коррелированного подзапроса в условии WHERE

Предположим, что Вы хотите получить список всех OrderID, для которых покупатели приобрели не больше 10% от среднего объема продаж каждого из товаров. Подобный анализ покажет тех покупателей, с которыми нужно связаться, чтобы выяснить причину столь низкого интереса к приобретённому товару. Для этих целей можно использовать коррелированный подзапрос, который будет помещён в предложении WHERE. Вот запрос, который вернет интересующий нас список товаров:

```
select distinct OrderId

from Northwind.dbo.[Order Details] OD

where

Quantity <= (select avg(Quantity) * .1

              from Northwind.dbo.[Order Details]

              where OD.ProductID = ProductID)
```

В приведённом выше запросе коррелированный подзапрос располагается в круглых скобках. Как Вы могли заметить, этот коррелированный подзапрос содержит ссылку на "OD.ProductID". Эта ссылка участвует в сравнении "ProductID" внешнего запроса с "ProductID" внутреннего запроса. Движок SQL Server будет исполнять внутренний запрос (коррелированный подзапрос) для каждой записи "[Order Details]". Этот внутренний запрос подсчитает среднее количество (Quantity) для записей каждого товара (ProductID), отобранных во внешнем запросе. Средствами коррелированного подзапроса будет определено, возвращает ли внутренний запрос значение, удовлетворяющее условию WHERE. Если да, то строка, возвращенная внешним запросом, будет включена в итоговую выборку всего запроса T-SQL.

В следующем примере, также использующем коррелированный подзапрос в операторе WHERE, отбираются по два лучших по сумме покупок в долларах США покупателя для каждого региона. Подобный запрос может быть полезен при необходимости поощрения лучших в своих регионах покупателей.

```
select CompanyName, ContactName, Address,
```

```

        City, Country, PostalCode from Northwind.dbo.Customers OuterC

where CustomerID in (

select top 2 InnerC.CustomerId

        from Northwind.dbo.[Order Details] OD join Northwind.dbo.Orders O

                on OD.OrderId = O.OrderID

        join Northwind.dbo.Customers InnerC

                on O.CustomerID = InnerC.CustomerId

Where Region = OuterC.Region

group by Region, InnerC.CustomerId

order by sum(UnitPrice * Quantity * (1-Discount)) desc

)

order by Region

```

Как видно из примера, внутренний запрос - коррелированный, потому что он ссылается на "OuterC", псевдоним использующейся во внешнем запросе таблицы "Northwind.DBO.Customer". Внутренний запрос использует значения поля "Region" для определения двух лучших покупателей в каждом регионе, ассоциированных со строкой внешнего запроса. В результирующую выборку попадут записи о тех двух "CustomerID" из внешнего запроса, которые попадут в число лучших покупателей.

[\[В начало\]](#)

Коррелированный подзапрос в разделе HAVING

Допустим, в целях увеличения своего дохода ваша организация решила в течение года проводить акцию по стимуляции потребительского спроса. Для этого покупатели извещаются о том, что если каждый сделанный ими в течение года заказ будет превышать сумму 750\$, то в конце года на каждый их заказ вы сделаете скидку в 75\$. Ниже приведен пример вычисления размера скидки. В этом примере для определения подпадающих под условие получения скидки покупателей, используется коррелированный подзапрос, помещённый в раздел HAVING.

```

select C.CustomerID, Count(*)*75 Rebate

        from Northwind.DBO.Customers C

                join

        Northwind.DBO.Orders O

                on c.CustomerID = O.CustomerID

where Datepart(yy,OrderDate) = '1998'

group by C.CustomerId

```

```

having 750 < ALL(select sum(UnitPrice * Quantity * (1-Discount))

from Northwind.DBO.Orders O

join

Northwind.DBO.[Order Details] OD

on O.OrderID = OD.OrderID

where CustomerID = C.CustomerId

and Datepart(yy,OrderDate) = '1998'

group by O.OrderId

)

```

Как Вы можете заметить, имеющийся в разделе HAVING коррелированный запрос используется для того, чтобы вычислить сумму заказа для каждого заказа клиента. Из внешнего запроса выбираются "CustomerID" и "Datepart (yy, OrderDate)" - год, когда был сделан заказ - которые нужны для того, что бы отобрать те заказы клиента, которые были сделаны в 1998 году. Для отобранных таким образом записей вычисляется сумма покупки по каждому заказу, для чего суммируются все записи "[Order Details]" по такой формуле: $\text{sum}(\text{UnitPrice} * \text{Quantity} * (1 - \text{Discount}))$. Если каждая сделанная в 1998 покупка клиента превышает в сумме 750\$, тогда во внешнем запросе вычисляется размер скидки по формуле: $\text{"Count"} * 75$. Процессор запросов SQL Server выполнит помещённый в разделе HAVING внутренний коррелированный подзапрос для всех отобранных внешним запросом покупателей, которые делали заказы в 1998 году.

[\[В начало\]](#)

Применение коррелированного подзапроса в операторе Update

Коррелированный подзапрос также может использоваться и в операторе Update:

```

create table A(A int, S int)

create table B(A int, B int)


set nocount on

insert into A(A) values(1)

insert into A(A) values(2)

insert into A(A) values(3)

insert into B values(1,1)

insert into B values(2,1)

```

```

insert into B values(2,1)

insert into B values(3,1)

insert into B values(3,1)

insert into B values(3,1)


update A

    set S = (select sum(B)

                from B

                where A.A = A group by A)


select * from A


drop table A,B

```

В результате мы получим:

A	S
1	1
2	2
3	3

В приведенном выше запросе коррелированный подзапрос используется для замены значений в столбце S таблицы A на сумму столбца B таблицы B , для тех строк, которые имеют одинаковые значения столбцов A, как в используемых для суммирования, так и в обновляемых строках.

[\[В начало\]](#)

Заключение

Давайте теперь подведём небольшой итог в этой статье. Подзапрос и коррелированный подзапрос - это операторы SELECT, используемые в другом запросе, называемом внешним. Коррелированный подзапрос и простой подзапрос очень полезны в оформлении выборки данных. Подзапрос, когда он выполняется независимо от внешнего запроса, возвращает выборку, которая также будет независимой от внешнего запроса. В свою очередь, коррелированный подзапрос не может исполняться независимо от внешнего запроса, потому что он ссылается на один или несколько столбцов внешнего запроса. Надеюсь, что теперь вы понимаете разницу между обычным и коррелированным подзапросом, и как их можно применять в T-SQL

24)



Табличные выражения

[Работа с базами данных в .NET Framework --- SQL Server 2012 --- Табличные выражения](#)
[Исходники баз данных](#)

Табличными выражениями называются подзапросы, которые используются там, где ожидается наличие таблицы. Существует два типа табличных выражений:

- производные таблицы;
- обобщенные табличные выражения.

Эти две формы табличных выражений рассматриваются в следующих подразделах.

Производные таблицы

Производная таблица (derived table) - это табличное выражение, входящее в предложение FROM запроса. Производные таблицы можно применять в тех случаях, когда использование псевдонимов столбцов не представляется возможным, поскольку транслятор SQL обрабатывает другое предложение до того, как псевдоним станет известным. В примере ниже показана попытка использовать псевдоним столбца в ситуации, когда другое предложение обрабатывается до того, как станет известным псевдоним:

```
USE SampleDb;
```

```
SELECT MONTH(EnterDate) as enter_month
```

```
FROM Works_on
```

```
GROUP BY enter_month;
```

Попытка выполнить этот запрос выдаст следующее сообщение об ошибке:

Причиной ошибки является то обстоятельство, что предложение GROUP BY обрабатывается до обработки соответствующего списка инструкции SELECT, и при обработке этой группы псевдоним столбца enter_month неизвестен.

Эту проблему можно решить, используя производную таблицу, содержащую предшествующий запрос (без предложения GROUP BY), поскольку предложение FROM выполняется перед предложением GROUP BY:

```
USE SampleDb;
```

```
SELECT enter_month
```

```
FROM (
```

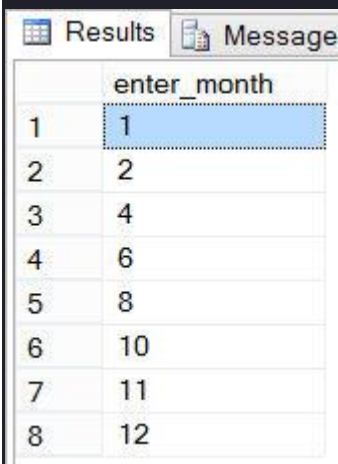
```
    SELECT MONTH(EnterDate) as enter_month
```

```
    FROM Works_on)
```

```
AS m
```

```
GROUP BY enter_month;
```

Результат выполнения этого запроса будет таким:



	enter_month
1	1
2	2
3	4
4	6
5	8
6	10
7	11
8	12

Обычно табличное выражение можно разместить в любом месте инструкции SELECT, где может появиться имя таблицы. (Результатом табличного выражения всегда является таблица или, в особых случаях, выражение.) В примере ниже показывается использование табличного выражения в списке выбора инструкции SELECT:

```
USE SampleDb;
```

```
SELECT w.Job AS 'Работа', (SELECT e.LastName
```

```
FROM Employee e WHERE e.Id = w.EmpId) AS 'Фамилия'
```

```
FROM Works_on w
```

```
WHERE w.Job IN ('Аналитик', 'Менеджер')
```

Результат выполнения этого запроса:

	Работа	Фамилия
1	Аналитик	Иванова
2	Менеджер	Иванова
3	Аналитик	Фролов
4	Менеджер	Лебедеенко

Обобщенные табличные выражения

Обобщенным табличным выражением (ОТВ) (Common Table Expression - сокращенно CTE) называется именованное табличное выражение, поддерживаемое языком Transact-SQL. Обобщенные табличные выражения используются в следующих двух типах запросов:

- нерекурсивных;
- рекурсивных.

Эти два типа запросов рассматриваются в следующих далее разделах.

ОТВ и нерекурсивные запросы

Нерекурсивную формулу ОТВ можно использовать в качестве альтернативы производным таблицам и представлениям. Обычно ОТВ определяется посредством **предложения WITH** и дополнительного запроса, который ссылается на имя, используемое в предложении WITH. В языке Transact-SQL значение ключевого слова WITH неоднозначно. Чтобы избежать неопределенности, инструкцию, предшествующую оператору WITH, следует завершать точкой с запятой.

далее показано использование ОТВ в нерекурсивных запросах. В примере ниже используется стандартное решение (здесь используется тестовая база данных AdventureWorks2012 из исходников):

```
USE AdventureWorks2012;
```

```

SELECT SalesOrderID

FROM Sales.SalesOrderHeader

WHERE TotalDue > (SELECT AVG(TotalDue)

                  FROM Sales.SalesOrderHeader

                  WHERE YEAR(OrderDate) = '2005')

AND Freight > (SELECT AVG(TotalDue)

              FROM Sales.SalesOrderHeader

              WHERE YEAR(OrderDate) = '2005')/2.5;

```

Запрос в этом примере выбирает заказы, чьи общие суммы налогов (TotalDue) большие, чем среднее значение по всем налогам, и плата за перевозку (Freight) которых больше чем 40% среднего значения налогов. Основным свойством этого запроса является его объемистость, поскольку вложенный запрос требуется писать дважды. Одним из возможных способов уменьшить объем конструкции запроса будет создать представление, содержащее вложенный запрос. Но это решение несколько сложно, поскольку требует создания представления, а потом его удаления после окончания выполнения запроса. Лучшим подходом будет создать ОТВ. В примере ниже показывается использование нерекурсивного ОТВ, которое сокращает определение запроса, приведенного выше:

```

USE AdventureWorks2012;

WITH price_calc(year_2005) AS

    (SELECT AVG(TotalDue)

     FROM Sales.SalesOrderHeader

     WHERE YEAR(OrderDate) = '2005')

SELECT SalesOrderID

FROM Sales.SalesOrderHeader

WHERE TotalDue > (SELECT year_2005 FROM price_calc)

```

```
AND Freight > (SELECT year_2005 FROM price_calc)/2.5;
```

Синтаксис предложения WITH в нерекурсивных запросах имеет следующий вид:

Соглашения по синтаксису

Параметр `cte_name` представляет имя ОТВ, которое определяет результирующую таблицу, а параметр `column_list` - список столбцов табличного выражения. (В примере выше ОТВ называется `price_calc` и имеет один столбец - `year_2005`.) Параметр `inner_query` представляет инструкцию `SELECT`, которая определяет результирующий набор соответствующего табличного выражения. После этого определенное табличное выражение можно использовать во внешнем запросе `outer_query`. (Внешний запрос в примере выше использует ОТВ `price_calc` и ее столбец `year_2005`, чтобы упростить употребляющийся дважды вложенный запрос.)

ОТВ и рекурсивные запросы

В этом разделе представляется материал повышенной сложности. Поэтому при первом его чтении рекомендуется его пропустить и вернуться к нему позже. Посредством ОТВ можно реализовывать рекурсии, поскольку ОТВ могут содержать ссылки на самих себя. Основной синтаксис ОТВ для рекурсивного запроса выглядит таким образом:

Соглашения по синтаксису

Параметры `cte_name` и `column_list` имеют такое же значение, как и в ОТВ для нерекурсивных запросов. Тело предложения `WITH` состоит из двух запросов, объединенных оператором **UNION ALL**. Первый запрос вызывается только один раз, и он начинает накапливать результат рекурсии. Первый операнд оператора `UNION ALL` не ссылается на ОТВ. Этот запрос называется опорным запросом или источником.

Второй запрос содержит ссылку на ОТВ и представляет ее рекурсивную часть. Вследствие этого он называется рекурсивным членом. В первом вызове рекурсивной части ссылка на ОТВ представляет результат опорного запроса. Рекурсивный член использует результат первого вызова запроса. После этого система снова вызывает рекурсивную часть. Вызов рекурсивного члена прекращается, когда предыдущий его вызов возвращает пустой результирующий набор.

Оператор UNION ALL соединяет накопившиеся на данный момент строки, а также дополнительные строки, добавленные текущим вызовом рекурсивного члена. (Наличие оператора UNION ALL означает, что повторяющиеся строки не будут удалены из результата.)

Наконец, параметр outer_query определяет внешний запрос, который использует ОТВ для получения всех вызовов объединения обеих членов.

Для демонстрации рекурсивной формы ОТВ мы используем таблицу Airplane, определенную и заполненную кодом, показанным в примере ниже:

```
USE SampleDb;

CREATE TABLE Airplane (

    ContainingAssembly VARCHAR(10),

    ContainedAssembly VARCHAR(10),

    QuantityContained INT,

    UnitCost DECIMAL (6,2)

);

INSERT INTO Airplane VALUES ( 'Самолет', 'Фюзеляж',1, 10);

INSERT INTO Airplane VALUES ( 'Самолет', 'Крылья', 1, 11);

INSERT INTO Airplane VALUES ( 'Самолет', 'Хвост',1, 12);

INSERT INTO Airplane VALUES ( 'Фюзеляж', 'Салон', 1, 13);

INSERT INTO Airplane VALUES ( 'Фюзеляж', 'Кабина', 1, 14);

INSERT INTO Airplane VALUES ( 'Фюзеляж', 'Нос',1, 15);

INSERT INTO Airplane VALUES ( 'Салон', NULL, 1,13);

INSERT INTO Airplane VALUES ( 'Кабина', NULL, 1, 14);
```

```
INSERT INTO Airplane VALUES ( 'Нос', NULL, 1, 15);
```

```
INSERT INTO Airplane VALUES ( 'Крылья', NULL, 2, 11);
```

```
INSERT INTO Airplane VALUES ( 'Хвост', NULL, 1, 12);
```

Таблица Airplane состоит из четырех столбцов. Столбец ContainingAssembly определяет сборку, а столбец ContainedAssembly - части (одна за другой), которые составляют соответствующую сборку. На рисунке ниже приведена графическая иллюстрация возможного вида самолета и его составляющих частей:

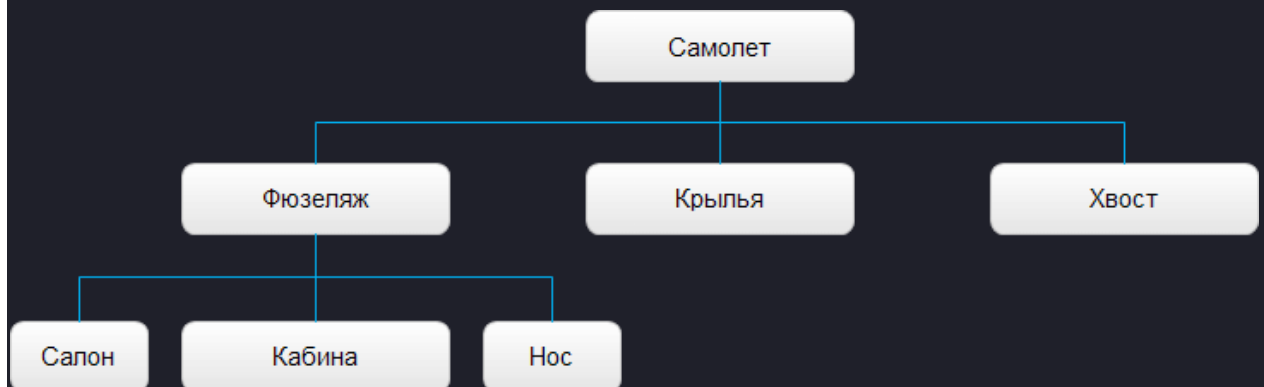


Таблица Airplane состоит из следующих 11 строк:

	ContainingAssembly	ContainedAssembly	QuantityContained	UnitCost
1	Самолет	Фюзеляж	1	10.00
2	Самолет	Крылья	1	11.00
3	Самолет	Хвост	1	12.00
4	Фюзеляж	Салон	1	13.00
5	Фюзеляж	Кабина	1	14.00
6	Фюзеляж	Нос	1	15.00
7	Салон	NULL	1	13.00
8	Кабина	NULL	1	14.00
9	Нос	NULL	1	15.00
10	Крылья	NULL	2	11.00
11	Хвост	NULL	1	12.00

В примере ниже показано применение предложения WITH для определения запроса, который вычисляет общую стоимость каждой сборки:

```
USE SampleDb;
```

```
WITH list_of_parts(assembly1, quantity, cost) AS
```

```

(SELECT ContainingAssembly, QuantityContained, UnitCost

FROM Airplane

WHERE ContainedAssembly IS NULL

UNION ALL

SELECT a.ContainingAssembly, a.QuantityContained,

CAST(l.quantity * l.cost AS DECIMAL(6,2))

FROM list_of_parts l, Airplane a

WHERE l.assembly1 = a.ContainedAssembly)

SELECT assembly1 'Деталь', quantity 'Кол-во', cost 'Цена'

FROM list_of_parts;

```

Предложение WITH определяет список ОТВ с именем list_of_parts, состоящий из трех столбцов: assembly1, quantity и cost. Первая инструкция SELECT в примере вызывается только один раз, чтобы сохранить результаты первого шага процесса рекурсии. Инструкция SELECT в последней строке примера отображает следующий результат:

	Деталь	Кол-во	Цена
2	Кабина	1	14.00
3	Нос	1	15.00
4	Крылья	2	11.00
5	Хвост	1	12.00
6	Самолет	1	12.00
7	Самолет	1	22.00
8	Фюзеляж	1	15.00
9	Самолет	1	15.00
10	Фюзеляж	1	14.00
11	Самолет	1	14.00
12	Фюзеляж	1	13.00
13	Самолет	1	13.00

Первые пять строчек этого результата являются результирующим набором первого вызова опорного члена запроса, а все остальные строчки - результатом рекурсивного члена (вторая часть) запроса в этом примере. Рекурсивный член

запроса вызывается дважды: первый раз для сборки фюзеляжа (fuselage), а второй раз для всего самолета (Airplane).

Запрос в примере ниже вычисляет стоимость каждой сборки со всеми ее составляющими:

```
USE SampleDb;

WITH list_of_parts(assembly1, quantity, cost) AS

    (SELECT ContainingAssembly, QuantityContained, UnitCost

    FROM Airplane

    WHERE ContainedAssembly IS NULL

    UNION ALL

    SELECT a.ContainingAssembly, a.QuantityContained,

    CAST(l.quantity * l.cost AS DECIMAL(6,2))

    FROM list_of_parts l, Airplane a

    WHERE l.assembly1 = a.ContainedAssembly)

SELECT assembly1 'Деталь', SUM(quantity) 'Кол-во частей', SUM(cost) 'Цена'

FROM list_of_parts

GROUP BY assembly1;
```

Результат выполнения этого запроса дает следующий результат:

Results		Messages	
	Деталь	Кол-во частей	Цена
1	Кабина	1	14.00
2	Крылья	2	11.00
3	Нос	1	15.00
4	Салон	1	13.00
5	Самолет	5	76.00
6	Фюзеляж	3	42.00
7	Хвост	1	12.00

В рекурсивных запросах на ОТВ налагается несколько следующих ограничений:

- определение ОТВ должно содержать, по крайней мере, две инструкции SELECT (опорный член и рекурсивный член), объединенные оператором UNION ALL;
- опорный член и рекурсивный член должны иметь одинаковое количество столбцов (это является прямым следствием использования оператора UNION ALL);
- столбцы в рекурсивном члене должны иметь такой же тип данных, как и соответствующие столбцы в опорном члене;
- предложение FROM рекурсивного члена должно ссылаться на имя ОТВ только один раз;
- определение рекурсивного члена не может содержать следующие параметры: SELECT DISTINCT, GROUP BY, HAVING, агрегатные функции, TOP и подзапросы. Кроме этого, единственным типом операции соединения, разрешенной в определении запроса, является внутреннее соединение.

25)

Табличные функции в Transact-SQL – описание и примеры создания

Раньше мы уже знакомились с функциями, которые возвращают таблицу, правда, на языке PL/pgSQL для сервера PostgreSQL ([Написание табличной функции на PL/pgSQL](#)). Теперь пришло время поговорить о такой реализации на Transact-SQL.

Вспомним, а для чего нам вообще нужны такие функции. Самый простой ответ на этот вопрос это то, что в таких функциях можно программировать (*объявлять переменные, выполнять какие-то расчеты*) и передавать параметры внутрь этой функции, как в обычных скалярных функциях, а результат получать в виде таблицы. И это хорошо, ведь в представлениях (вьюхах) этого делать нельзя, а процедура

ничего не возвращает (можно сделать, чтобы возвращала, но это в большинстве случаев не очень удобно).

Пример создания простой табличной функции

Итак, приступим, для начала приведем самый простой вариант реализации такой функции. Допустим, нам нужно выбрать несколько полей из таблицы по определенному критерию.

Примечание! Данный пример можно реализовать и с помощью представления. Но мы пока только учимся писать такие функции.

```
--название нашей функции
CREATE FUNCTION [dbo].[fun_test_tabl]
(
  --входящие параметры и их тип
  @id INT
)
--возвращающее значение, т.е. таблица
RETURNS TABLE
AS
--сразу возвращаем результат
RETURN
(
  --сам запрос
  SELECT * FROM table WHERE id = @id
)
GO
```

В итоге мы создали функцию, в которую будем передавать один параметр id, его мы используем в условии исходного SQL запроса.

Получить данные из этой функции можно следующим образом:

```
SELECT * FROM dbo.fun_test_tabl (1)
```

Как видите все проще простого. Теперь давайте создадим функцию уже с использованием программирования в этой функции.

Курс по SQL



47 занятий



65 задач



156 вопросов
в тестах



Поддержка
ментора



Экзамен



Сертификат

Пример создания табличной функции, в которой можно программировать

```
--название нашей функции
CREATE FUNCTION [dbo].[fun_test_tabl_new]
(
  --входящие параметры
  @number INT
)
--возвращающее значение, т.е. таблица с перечислением полей и их типов
RETURNS @tabl TABLE (id INT, number INT, summa MONEY)
AS
BEGIN
  --объявляем переменные
  DECLARE @var MONEY
  --выполняем какие-то действия на Transact-SQL
  IF @number >=0
  BEGIN
    SET @var=1000
  END
  ELSE
    SET @var=0
  --вставляем данные в возвращающий результат
  INSERT @tabl
    SELECT id, number, summa
    FROM tabl
    WHERE summa > @var
  --возвращаем результат
  RETURN
END
```

Здесь мы уже программируем и можем выполнять любые действия как в обычных функциях и процедурах, при этом получая результат в виде таблицы. В этом примере мы передаем один параметр внутрь нашей функции (*их может быть*

несколько!), внутри функции мы уже смотрим, что за параметр к нам пришел, и на основе этого уже формируем условие для запроса. Как Вы понимаете это тоже простой пример, но можно писать очень и очень сложные алгоритмы как в процедурах, именно поэтому, и созданы эти табличные функции.

Теперь давайте обратимся к нашей функции, например, вот так

```
SELECT * FROM dbo.fun_test_tabl_new (1)
```

26) CROSS APPLY в T-SQL

CROSS APPLY – это тип оператора APPLY, который позволяет вызывать табличную функцию для каждой строки внешнего табличного выражения. Вместо табличной функции можно также использовать и [вложенный запрос](#), который возвращает производную таблицу.

Есть еще и другой тип OUTER APPLY, он в отличие от CROSS APPLY возвращает и строки, которые формируют результирующий набор, и те, которые этого не делают, т.е. со значениями NULL в столбцах. Например, табличная функция может не возвращать никаких данных для определенных значений, CROSS APPLY в таких случаях подобные строки не выводит, а OUTER APPLY выводит.

Примеры использования CROSS APPLY в T-SQL

Давайте рассмотрим несколько примеров использования операторов CROSS APPLY и OUTER APPLY в языке T-SQL.

Заметка! Все примеры в данной статье рассмотрены в [Microsoft SQL Server 2019 Express](#).

Исходные данные для примеров

Для начала давайте определимся с исходными данными, которые мы будем использовать в примерах, допустим, у нас есть таблица товаров (Goods) и таблица продаж (Sales).

SQL инструкция ниже создаёт эти таблицы и наполняет их тестовыми данными.

```
--Таблица товаров
CREATE TABLE Goods (
    ProductId INT IDENTITY(1,1) NOT NULL,
    ProductName VARCHAR(100) NOT NULL
);

INSERT INTO Goods(ProductName)
VALUES ('Системный блок'),
       ('Принтер'),
       ('Монитор'),
       ('Клавиатура');

--Таблица продаж
CREATE TABLE Sales (
    SaleId INT IDENTITY(1,1) NOT NULL,
    ProductId INT NOT NULL,
    SaleDate DATETIME NOT NULL
);

INSERT INTO Sales (ProductId, SaleDate)
VALUES (1, '01.02.2020'),
```

```

(1, '10.03.2020'),
(1, '25.04.2020'),
(1, '15.05.2020'),
(2, '25.02.2020'),
(2, '15.06.2020'),
(2, '01.07.2020'),
(3, '01.04.2020'),
(3, '05.05.2020');

```

```

SELECT * FROM Goods;
SELECT * FROM Sales;

```

```

--Таблица товаров
CREATE TABLE Goods (
    ProductId INT IDENTITY(1,1) NOT NULL,
    ProductName VARCHAR(100) NOT NULL
);
INSERT INTO Goods(ProductName)
VALUES ('Системный блок'),
       ('Принтер'),
       ('Монитор'),
       ('Клавиатура');

--Таблица продаж
CREATE TABLE Sales (
    SaleId INT IDENTITY(1,1) NOT NULL,
    ProductId INT NOT NULL,
    SaleDate DATETIME NOT NULL
);
INSERT INTO Sales (ProductId, SaleDate)
VALUES (1, '01.02.2020'),
       (1, '10.03.2020'),
       (1, '25.04.2020'),
       (1, '15.05.2020'),
       (2, '25.02.2020'),
       (2, '15.06.2020'),
       (2, '01.07.2020'),
       (3, '01.04.2020'),
       (3, '05.05.2020');

SELECT * FROM Goods;
SELECT * FROM Sales;

```

100 %

Результаты Сообщения

	ProductId	ProductName
1	1	Системный блок
2	2	Принтер
3	3	Монитор
4	4	Клавиатура

	SaleId	ProductId	SaleDate
1	1	1	2020-02-01 00:00:00.000
2	2	1	2020-03-10 00:00:00.000
3	3	1	2020-04-25 00:00:00.000
4	4	1	2020-05-15 00:00:00.000
5	5	2	2020-02-25 00:00:00.000
6	6	2	2020-06-15 00:00:00.000
7	7	2	2020-07-01 00:00:00.000
8	8	3	2020-04-01 00:00:00.000
9	9	3	2020-05-05 00:00:00.000

Запрос успешно выполнен.

А также у нас есть табличная функция SaleGoods, которая просто выводит список продаж по идентификатору товара.

```
CREATE FUNCTION SaleGoods
(
    @ProductID INT
)
RETURNS TABLE
AS
RETURN
(
    SELECT S.SaleId, S.SaleDate, G.ProductName
    FROM Sales S
    INNER JOIN Goods G ON S.ProductID = G.ProductID
    WHERE S.ProductID = @ProductID
);

GO

SELECT * FROM SaleGoods (1);
```

```
CREATE FUNCTION SaleGoods
(
    @ProductID INT
)
RETURNS TABLE
AS
RETURN
(
    SELECT S.SaleId, S.SaleDate, G.ProductName
    FROM Sales S
    INNER JOIN Goods G ON S.ProductID = G.ProductID
    WHERE S.ProductID = @ProductID
);

GO

SELECT * FROM SaleGoods (1);
```

100 %

Результаты		Сообщения	
	SaleId	SaleDate	ProductName
1	1	2020-02-01 00:00:00.000	Системный блок
2	2	2020-03-10 00:00:00.000	Системный блок
3	3	2020-04-25 00:00:00.000	Системный блок
4	4	2020-05-15 00:00:00.000	Системный блок

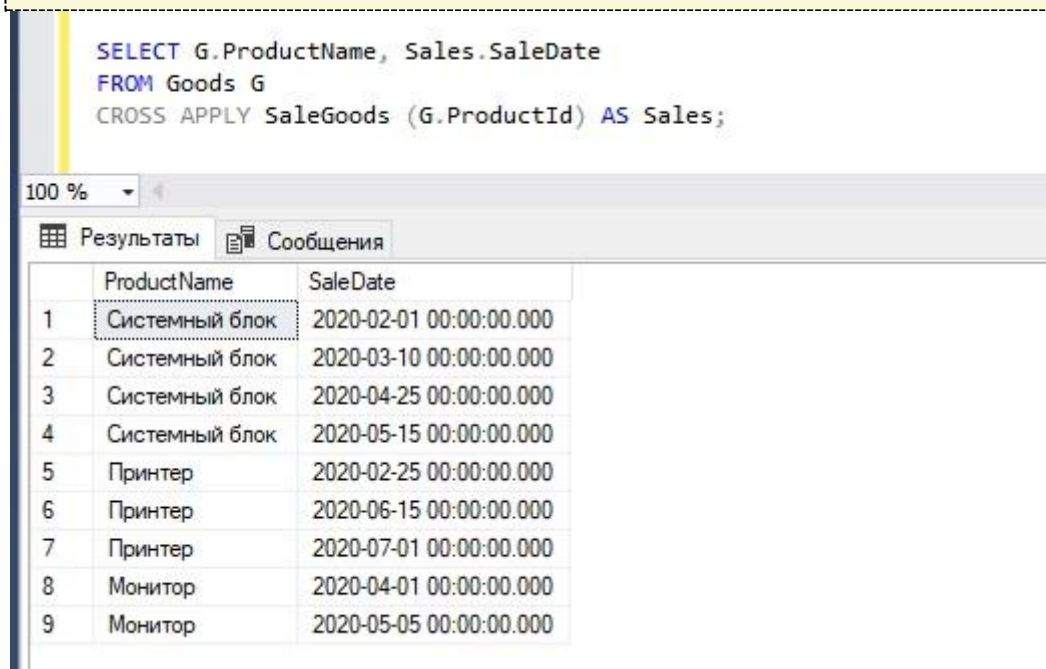
Заметка! Если Вы не знаете, что делает вышеуказанная инструкция, рекомендую посмотреть мой видеокурс [«T-SQL. Путь программиста от новичка к профессионалу. Уровень 1 – Новичок»](#), который предназначен для начинающих и

в нем подробно рассмотрены все базовые конструкции языка SQL, включая все вышеперечисленные.

Пример использования CROSS APPLY с табличной функцией

Теперь допустим, нам необходимо получить продажи не только одного товара, но и других, для этого мы можем использовать оператор CROSS APPLY.

```
SELECT G.ProductName, Sales.SaleDate
FROM Goods G
CROSS APPLY SaleGoods (G.ProductId) AS Sales;
```

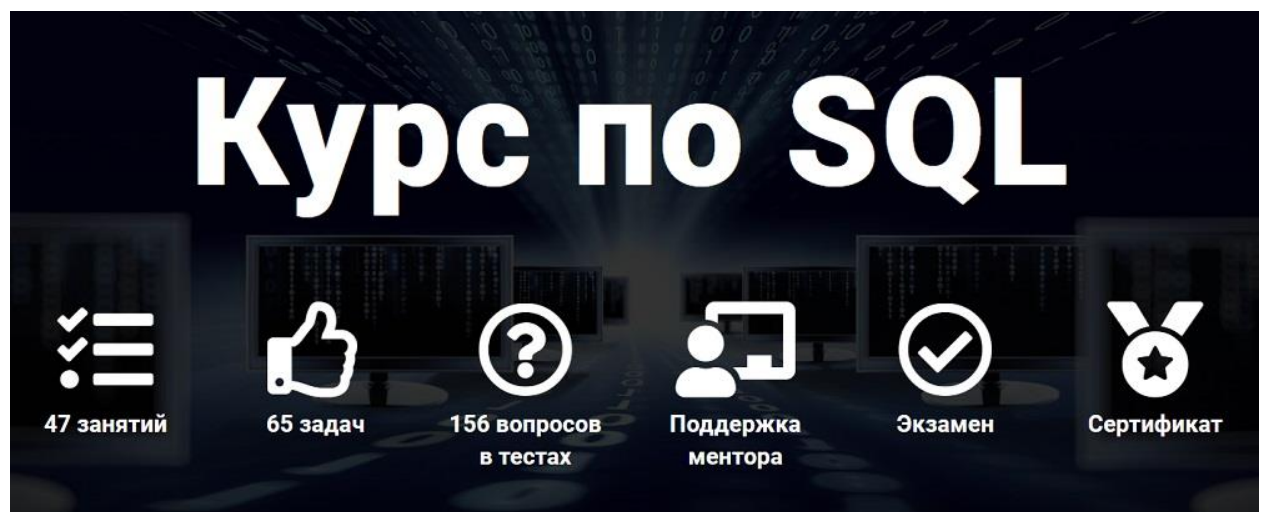


The screenshot shows a SQL query execution window with the following query:

```
SELECT G.ProductName, Sales.SaleDate
FROM Goods G
CROSS APPLY SaleGoods (G.ProductId) AS Sales;
```

The results are displayed in a table with two columns: ProductName and SaleDate. The table contains 9 rows of data.

	ProductName	SaleDate
1	Системный блок	2020-02-01 00:00:00.000
2	Системный блок	2020-03-10 00:00:00.000
3	Системный блок	2020-04-25 00:00:00.000
4	Системный блок	2020-05-15 00:00:00.000
5	Принтер	2020-02-25 00:00:00.000
6	Принтер	2020-06-15 00:00:00.000
7	Принтер	2020-07-01 00:00:00.000
8	Монитор	2020-04-01 00:00:00.000
9	Монитор	2020-05-05 00:00:00.000



Курс по SQL

- 47 занятий
- 65 задач
- 156 вопросов в тестах
- Поддержка ментора
- Экзамен
- Сертификат

Где, как Вы понимаете, табличная функция SaleGoods была вызвана для каждой строки таблицы Goods.

Пример использования CROSS APPLY с подзапросом

Как уже было сказано выше, CROSS APPLY можно использовать и с подзапросом, для примера давайте представим, что нам нужно получить последнюю дату продажи каждого товара.

```
SELECT G.ProductName, Sales.SaleDate
FROM Goods G
CROSS APPLY (
    SELECT TOP 1 S.SaleDate
    FROM SaleGoods(G.ProductId) AS S
    ORDER BY S.SaleDate DESC
) AS Sales;
```

```
SELECT G.ProductName, Sales.SaleDate
FROM Goods G
CROSS APPLY (
    SELECT TOP 1 S.SaleDate
    FROM SaleGoods(G.ProductId) AS S
    ORDER BY S.SaleDate DESC
) AS Sales;
```

100 %

Результаты Сообщения

	ProductName	SaleDate
1	Системный блок	2020-05-15 00:00:00.000
2	Принтер	2020-07-01 00:00:00.000
3	Монитор	2020-05-05 00:00:00.000

Как видим, после CROSS APPLY у нас идет вложенный запрос, формирующий производную таблицу.

Заметка! [ТОП 20 статей для изучения языка T-SQL – Уровень «Продвинутый».](#)

Пример использования OUTER APPLY

Если Вы заметили, у нас в исходных данных есть позиция «Клавиатура», по которой не было продаж. И в случае возникновения необходимости получить последнюю дату продажи, включая товары, по которым не было продаж, т.е. выводить NULL значения, чтобы мы видели, какие товары не продавались вообще, оператор CROSS APPLY можно заменить на OUTER APPLY.

```
SELECT G.ProductName, Sales.SaleDate
FROM Goods G
OUTER APPLY (
    SELECT TOP 1 S.SaleDate
    FROM SaleGoods(G.ProductId) AS S
    ORDER BY S.SaleDate DESC
)
```

```
) AS Sales;
```

```
SELECT G.ProductName, Sales.SaleDate
FROM Goods G
OUTER APPLY (
    SELECT TOP 1 S.SaleDate
    FROM SaleGoods(G.ProductId) AS S
    ORDER BY S.SaleDate DESC
) AS Sales;
```

100 %

Результаты Сообщения

	ProductName	SaleDate
1	Системный блок	2020-05-15 00:00:00.000
2	Принтер	2020-07-01 00:00:00.000
3	Монитор	2020-05-05 00:00:00.000
4	Клавиатура	NULL

Теперь видно, что некоторые товары продавались и последняя дата их продажи такая-то, а некоторые товары не продавались, т.е. у них отсутствует дата продажи.

Интересные материалы по теме:

- [SQL код – самоучитель по SQL для начинающих программистов](#)
- [Как выполнить код Python в Microsoft SQL Server на T-SQL](#)
- [Объект SEQUENCE \(последовательность\) в Microsoft SQL Server](#)
- [Что такое DDL, DML, DCL и TCL в языке SQL](#)

27)

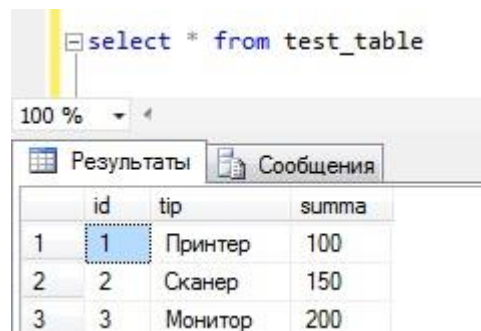
Исходные данные для примеров

Для того чтобы мы могли попробовать эти операторы в действии, нам потребуются какие-то данные, предлагаю создать две таблицы и заполнить их тестовыми данными.

Таблица 1

```
CREATE TABLE [dbo].[test_table] (  
    [id] [int] NOT NULL,  
    [tip] [varchar](50) NULL,  
    [summa] [varchar](50) NULL  
) ON [PRIMARY]  
GO
```

Ее данные



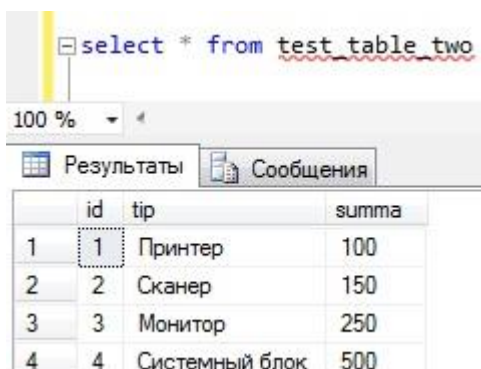
The screenshot shows a SQL query window with the command `select * from test_table`. Below the query, the 'Results' tab is active, displaying a table with four columns: 'id', 'tip', and 'summa'. The table contains three rows of data. The first row has '1' for id, 'Принтер' for tip, and '100' for summa. The second row has '2' for id, 'Сканер' for tip, and '150' for summa. The third row has '3' for id, 'Монитор' for tip, and '200' for summa.

	id	tip	summa
1	1	Принтер	100
2	2	Сканер	150
3	3	Монитор	200

Таблица 2

```
CREATE TABLE [dbo].[test_table_two] (  
    [id] [int] NOT NULL,  
    [tip] [varchar](50) NULL,  
    [summa] [varchar](50) NULL  
) ON [PRIMARY]  
GO
```

И ее данные



The screenshot shows a SQL query window with the command `select * from test_table_two`. Below the query, the 'Results' tab is active, displaying a table with four columns: 'id', 'tip', and 'summa'. The table contains four rows of data. The first row has '1' for id, 'Принтер' for tip, and '100' for summa. The second row has '2' for id, 'Сканер' for tip, and '150' for summa. The third row has '3' for id, 'Монитор' for tip, and '250' for summa. The fourth row has '4' for id, 'Системный блок' for tip, and '500' for summa.

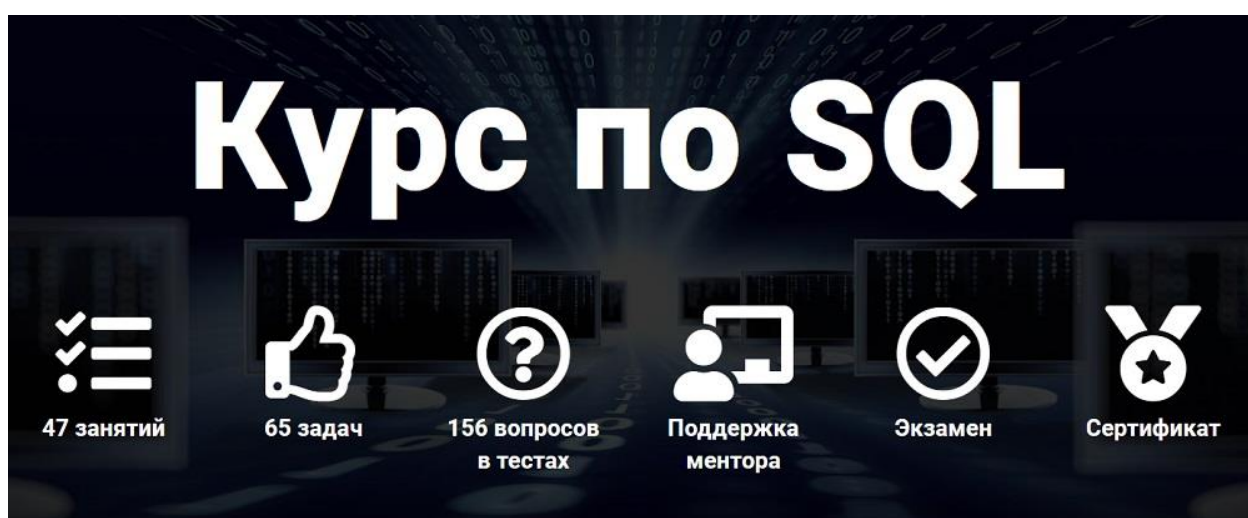
	id	tip	summa
1	1	Принтер	100
2	2	Сканер	150
3	3	Монитор	250
4	4	Системный блок	500

Оператор INTERSECT

INTERSECT (*пересечение*) – это оператор Transact-SQL, который выводит одинаковые строки из первого, второго и последующих наборов данных. Другими словами, он выведет только те строки, которые есть как в первом результирующем наборе, так и во втором (*третьем и так далее*), т.е. происходит пересечение этих строк.

Данный оператор очень полезен, например, тогда, когда необходимо узнать какие строки есть и в первой таблице и во второй (*к примеру, повтор данных*).

Как и у оператора UNION, у INTERSECT есть правила, например, то, что количество полей во всех результирующих наборах должно быть одинаковым, также как и их тип данных.



Пример

Давайте узнаем, какие данные у нас есть и в таблице test_table и в таблице test_table_two, т.е. совпадения, для этого пишем простой [SQL запрос](#):

```
SELECT tip, summa FROM test_table
      intersect
SELECT tip, summa FROM test_table_two
```

The screenshot shows a SQL query window with the following text:

```
select tip, summa from test_table
intersect
select tip, summa from test table two
```

Below the query, the 'Результаты' (Results) tab is active, displaying a table with two columns: 'tip' and 'summa'. The table contains two rows of data.

	tip	summa
1	Принтер	100
2	Сканер	150

Как видите, у нас в обеих таблицах есть «Принтер» с суммой 100 и «Сканер» с суммой 150.

Оператор EXCEPT

EXCEPT (*разность*) — это оператор Transact-SQL, который выводит только те данные из первого набора строк, которых нет во втором наборе.

Здесь те же правила, что и у оператора INTERSECT, т.е. количество столбцов (*и их тип*) должно быть одинаковым.

EXCEPT полезен тогда, когда необходимо сравнить две таблицы и вывести только те строки первой таблице, которых нет в другой таблице.

Пример

Давайте посмотрим, какие строки есть только в первой таблице

```
SELECT tip, summa FROM test_table
except
SELECT tip, summa FROM test_table_two
```

The screenshot shows a SQL query window with the following text:

```
select tip, summa from test_table
except
select tip, summa from test table two
```

Below the query, the 'Результаты' (Results) tab is active, displaying a table with two columns: 'tip' and 'summa'. The table contains one row of data.

	tip	summa
1	Монитор	200

Как видите, во второй таблице нет строки, у которой tip «Монитор», а сумма 200, если помните, то у нас во второй таблице монитор с суммой 250.

А теперь давайте поменяем наши таблицы местами и посмотрим на результат.

```
SELECT tip, summa FROM test_table_two  
except  
SELECT tip, summa FROM test_table
```

	tip	summa
1	Монитор	250
2	Системный блок	500

Здесь результат уже другой, так как за основу взята другая таблица, и в результате у нас вывелись те строки, которых нет в таблице test_table.

Синтаксис

С удалением дублей:

```
SELECT * FROM имя_таблицы1 WHERE условие  
UNION SELECT * FROM имя_таблицы2 WHERE условие
```

Без удаления дублей:

```
SELECT * FROM имя_таблицы1 WHERE условие  
UNION ALL SELECT * FROM имя_таблицы2 WHERE условие
```

Можно объединять не две таблицы, а три или более:

```
SELECT * FROM имя_таблицы1 WHERE условие  
UNION SELECT * FROM имя_таблицы2 WHERE условие  
UNION SELECT * FROM имя_таблицы3 WHERE условие  
UNION SELECT * FROM имя_таблицы4 WHERE условие
```

Примеры

Все примеры будут по таблицам **countries** и **cities**, если не сказано иное.

Таблица **countries**:

id айди	name название
1	Беларусь
2	Россия
3	Украина

Таблица **cities**:

id айди	name название	country_id айди страны
1	Минск	1
2	Минск	1
3	Москва	2
4	Киев	3

Пример

В данном примере отсутствует ключевое слово ALL, однако дубли не будут удалены, так как дублиями считается полное совпадение строк:

```
SELECT id, name FROM countries UNION SELECT id, name FROM cities
```

SQL запрос выберет следующие строки:

id айди	name название
1	Беларусь
2	Россия

3	Украина
1	Минск
2	Минск
3	Москва
4	Киев

28)

Не использовал ваш sql выше, но вот пример обновления таблицы на основе инструкции объединения.

```

UPDATE p

SET   p.category = c.category

FROM   products p

      INNER JOIN prodductcatagories pg

      ON   p.productid = pg.productid

      INNER JOIN categories c

      ON   pg.categoryid = c.cateogryid

WHERE  c.categories LIKE 'whole%'

```

UPDATE – инструкция SQL, с помощью которой происходит изменение существующих данных в таблицах.

Важные моменты:

- Если инструкция UPDATE, т.е. обновление строк, нарушает какое-нибудь ограничение или правило, или новое значение имеет несовместимый тип данных (*хотя бы для одной строки*), то возникнет ошибка и все изменения отменяются, никакие строки не обновляются;
- По умолчанию инструкция UPDATE получает монопольную блокировку на целевую таблицу, которую она изменяет, это означает, что пока одна инструкция UPDATE выполняется, т.е. изменяет данные в таблице, другие инструкции не могут изменять данные в этой таблице;
- Чтобы использовать инструкцию UPDATE, нужны соответствующие разрешения на изменение данных, а также на чтение данных, если инструкция содержит условие WHERE;
- Если Вам нужно узнать количество строк, которые Вы обновили инструкцией UPDATE, например, для возврата в клиентское приложение или для любых других целей, то для этого Вы можете использовать [функцию @@ROWCOUNT](#).

Упрощенный синтаксис UPDATE

Синтаксис UPDATE достаточно большой, и начинающим понять его сложно, поэтому, для того чтобы было проще понять логику формирования инструкции UPDATE, я приведу упрощенный синтаксис.

UPDATE *Целевая таблица* SET *Имя столбца* = *Значение*

FROM *Таблица источник*

WHERE *Условие*

Где

- UPDATE – инструкция обновления;
- Целевая таблица – таблица, данные в которой необходимо изменить;
- SET – команда, которая задает список обновляемых столбцов. Каждый следующий столбец указывается через запятую;
- Имя столбца – столбец, в котором расположены данные, которые необходимо изменить;
- Значение – новое значение, на которое необходимо изменить значение столбца. Можно указывать как конкретное значение, так и расчётное выражение, функцию или подзапрос. Также можно указать ключевое слово DEFAULT, что будет означать, что столбцу необходимо присвоить значение по умолчанию;
- FROM – секция, которая указывает таблицу, из которой необходимо взять новое значение столбца. Секция может содержать объединение JOIN;

- Таблица источник – таблица, в которой расположено новое значение столбца;
- WHERE – условие отбора строк, подлежащих обновлению.

Примеры использования инструкции UPDATE

Сейчас давайте рассмотрим несколько примеров SQL инструкций, которые будут обновлять данные в таблице инструкцией UPDATE.

Исходные данные для примеров

Для начала давайте определимся с исходными данными, чтобы Вы понимали, какие именно данные у нас есть, и что мы будем обновлять.

Также сразу скажу, что в качестве SQL сервера у меня выступает версия [Microsoft SQL Server 2017 Express](#).

Следующая инструкция создает таблицы, которые мы будем использовать в примерах, и добавляет в них данные.



```
--Создание таблицы Goods
CREATE TABLE Goods (
    ProductId      INT IDENTITY(1,1) NOT NULL CONSTRAINT
PK_ProductId PRIMARY KEY,
    Category       INT NOT NULL,
    ProductName    VARCHAR(100) NOT NULL,
    ProductDescription VARCHAR(300) NULL,
    Price          MONEY NULL,
);
GO

--Создание таблицы Categories
CREATE TABLE Categories (
    CategoryId     INT IDENTITY(1,1) NOT NULL CONSTRAINT
PK_CategoryId PRIMARY KEY,
```

```
        CategoryName VARCHAR(100) NOT NULL
    );

--Добавление строк в таблицу Categories
INSERT INTO Categories(CategoryName)
VALUES ('Комплекующие ПК'),
       ('Мобильные устройства');

GO

--Добавление строк в таблицу Goods
INSERT INTO Goods(Category, ProductName, ProductDescription,
Price)
VALUES (1, 'Системный блок', 'Товар 1', 300),
       (1, 'Монитор', 'Товар 2', 200),
       (2, 'Смартфон', 'Товар 3', 100);

GO

--Выборка данных
SELECT * FROM Goods;
SELECT * FROM Categories;
```

```
--Создание таблицы Goods
CREATE TABLE Goods (
    ProductId INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_ProductId PRIMARY KEY,
    Category INT NOT NULL,
    ProductName VARCHAR(100) NOT NULL,
    ProductDescription VARCHAR(300) NULL,
    Price MONEY NULL,
);
GO

--Создание таблицы Categories
CREATE TABLE Categories (
    CategoryId INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_CategoryId PRIMARY KEY,
    CategoryName VARCHAR(100) NOT NULL
);
GO

--Добавление строк в таблицу Categories
INSERT INTO Categories(CategoryName)
VALUES ('Комплектующие ПК'),
('Мобильные устройства');
GO

--Добавление строк в таблицу Goods
INSERT INTO Goods(Category, ProductName, ProductDescription, Price)
VALUES (1, 'Системный блок', 'Товар 1', 300),
(1, 'Монитор', 'Товар 2', 200),
(2, 'Смартфон', 'Товар 3', 100);
GO

--Выборка данных
SELECT * FROM Goods;
SELECT * FROM Categories;
```

100 %

Результаты Сообщения

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар 1	300,00
2	2	1	Монитор	Товар 2	200,00
3	3	2	Смартфон	Товар 3	100,00

	CategoryId	CategoryName
1	1	Комплектующие ПК
2	2	Мобильные устройства

Пример обновления одного столбца всех строк таблицы

В этом примере мы обновим значения одного столбца, при этом никаких условий мы делать не будем, т.е. обновим все строки в таблице.

Для наглядности и удобства отслеживания внесенных изменений я буду во всех примерах перед и после UPDATE посылать простой запрос SELECT, чтобы видеть, какие данные были и какие стали.

```
--Выборка данных
SELECT * FROM Goods;
```

```
--Обновление
UPDATE Goods SET ProductDescription = 'Товар';

--Выборка данных
SELECT * FROM Goods;
```

--Выборка данных
 SELECT * FROM Goods;

--Обновление
 UPDATE Goods SET ProductDescription = 'Товар';

--Выборка данных
 SELECT * FROM Goods;

100 %

Результаты Сообщения

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар 1	300,00
2	2	1	Монитор	Товар 2	200,00
3	3	2	Смартфон	Товар 3	100,00

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар	300,00
2	2	1	Монитор	Товар	200,00
3	3	2	Смартфон	Товар	100,00

Как видите, в итоге получился очень простой запрос на обновление, мы обновили значения в столбце ProductDescription у всех строк на «Товар».

Пример обновления двух столбцов и только некоторых строк таблицы

Теперь давайте обновим два столбца, и конкретизируем строки для обновления, т.е. мы обновим не все строки в таблице, как в предыдущем примере, а только те, которые подходят под указанное нами условие (для примера Category с идентификатором 1).

```
--Выборка данных
SELECT * FROM Goods;

--Обновление
UPDATE Goods SET ProductDescription = 'Товар NEW',
                Price = 400
WHERE Category = 1;
```

```
--Выборка данных
SELECT * FROM Goods;
```

```
--Выборка данных
SELECT * FROM Goods;

--Обновление
UPDATE Goods SET ProductDescription = 'Товар NEW',
               Price = 400
WHERE Category = 1;

--Выборка данных
SELECT * FROM Goods;
```

100 %

Результаты

Сообщения

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар	300,00
2	2	1	Монитор	Товар	200,00
3	3	2	Смартфон	Товар	100,00

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар NEW	400,00
2	2	1	Монитор	Товар NEW	400,00
3	3	2	Смартфон	Товар	100,00

В этом случае изменились значения столбцов ProductDescription и Price в строках, в которых Category = 1.

Пример использования выражений в инструкции UPDATE

Как я уже отмечал, в качестве нового значения может выступать не только какое-то конкретное значение, но и целое выражение, в котором могут использоваться как другие столбцы таблицы, так и столбец, который в данный момент обновляется.

В следующем примере в столбец ProductDescription мы добавим дополнительный текст (*просто цифру 3*), а значение Price мы увеличим в полтора раза. Все это мы сделаем для строки с Category = 2.

```
--Выборка данных
SELECT * FROM Goods;

--Обновление
UPDATE Goods SET ProductDescription = ProductDescription + '
3',
               Price = Price * 1.5
WHERE Category = 2;
```



```
--Выборка данных
SELECT * FROM Goods;
```

```
--Выборка данных
SELECT * FROM Goods;

--Обновление
UPDATE Goods SET ProductDescription = ProductDescription + ' 3',
              Price = Price * 1.5
WHERE Category = 2;

--Выборка данных
SELECT * FROM Goods;
```

100 %					
Результаты					
Сообщения					
	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар NEW	400,00
2	2	1	Монитор	Товар NEW	400,00
3	3	2	Смартфон	Товар	100,00
	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар NEW	400,00
2	2	1	Монитор	Товар NEW	400,00
3	3	2	Смартфон	Товар 3	150,00

Пример обновления данных таблицы на основе данных другой таблицы

Достаточно часто требуется обновить данные одной таблицы на основе данных другой, например, просто скопировать данные. Это можно сделать за счет объединения нужных таблиц в инструкции UPDATE. При этом существует несколько способов объединения, я покажу два.

Для примера здесь мы скопируем название категорий из таблицы Categories, и вставим их в столбец ProductDescription таблицы Goods, объединять будем по идентификатору категории.

```
--Выборка данных
SELECT * FROM Goods;

--Обновление
--Способ 1
UPDATE G SET ProductDescription = C.CategoryName
FROM Goods G
INNER JOIN Categories C ON G.Category = C.CategoryId;

--Способ 2 (эквивалент)
```

```
UPDATE Goods SET ProductDescription = C.CategoryName
FROM Categories C
WHERE Goods.Category = C.CategoryId;
```

```
--Выборка данных
SELECT * FROM Goods;
```

```
--Выборка данных
SELECT * FROM Goods;

--Обновление
--Способ 1
UPDATE G SET ProductDescription = C.CategoryName
FROM Goods G
INNER JOIN Categories C ON G.Category = C.CategoryId;

--Способ 2 (эквивалент)
UPDATE Goods SET ProductDescription = C.CategoryName
FROM Categories C
WHERE Goods.Category = C.CategoryId;

--Выборка данных
SELECT * FROM Goods;
```

100 %

Результаты Сообщения

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Товар NEW	400,00
2	2	1	Монитор	Товар NEW	400,00
3	3	2	Смартфон	Товар 3	150,00

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Комплектующие ПК	400,00
2	2	1	Монитор	Комплектующие ПК	400,00
3	3	2	Смартфон	Мобильные устройства	150,00

Пример обновления данных с использованием подзапроса

В этом примере я покажу, как можно использовать подзапрос в инструкции UPDATE. Для примера мы подсчитаем количество товаров в каждой категории и присвоим полученное значение столбцу ProductDescription.

Для того чтобы узнать количество товаров, мы будем использовать встроенную функцию COUNT, а для преобразования числа в строку — функцию CAST. Фильтровать строки в подзапросе мы будем по идентификатору категории, значение для сравнения будем получать из основного запроса.

```
--Выборка данных
SELECT * FROM Goods;
```

```

--Обновление
UPDATE Goods SET ProductDescription = 'Всего товаров: ' +
(SELECT CAST(COUNT(*) AS VARCHAR(10))
FROM
Goods G
WHERE
G.Category = Goods.Category);

--Выборка данных
SELECT * FROM Goods;

```

--Выборка данных
 SELECT * FROM Goods;

--Обновление
 UPDATE Goods SET ProductDescription = 'Всего товаров: ' + (SELECT CAST(COUNT(*) AS VARCHAR(10))
 FROM Goods G
 WHERE G.Category = Goods.Category);

--Выборка данных
 SELECT * FROM Goods;

100 %

Результаты Сообщения

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Комплектующие ПК	400,00
2	2	1	Монитор	Комплектующие ПК	400,00
3	3	2	Смартфон	Мобильные устройства	150,00

	ProductId	Category	ProductName	ProductDescription	Price
1	1	1	Системный блок	Всего товаров: 2	400,00
2	2	1	Монитор	Всего товаров: 2	400,00
3	3	2	Смартфон	Всего товаров: 1	150,00

Как видите, все отработало так, как мы задумали

ТАБЛИЧНОЕ ВЫРАЖЕНИЕ?

30)

31)

32)

Сведение (pivoting) является методикой агрегирования и поворота данных из строк в столбцы. При сведении нужно определить три элемента: элемент, который нужно видеть в строках (элемент группировки), элемент, который нужно видеть в столбцах (элемент распределения) и элемент, который нужно видеть в разделе данных (элемент агрегирования).

В качестве примера представьте, что нужно получить представление Sales.OrderValues и вернуть по строке на каждый год, по строке на каждый месяц и общую сумму всех сумм заказов для пересечений годов и месяцев. В этом запросе элементом группировки по строкам является YEAR(orderdate), элементом распределения по столбцам является MONTH(orderdate), уникальными значениями распределения являются 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 и 12, а элемент данных, или агрегации, является SUM (val).

Для получения сведения нужно прежде всего подготовить табличное выражение, такое как CTE, которое возвращает только три элемента, задействованные в задаче сведения. Затем во внешней инструкции запрашивается табличное выражение и используется оператор для обработки логики сведения (выходные данные выровнены):

```
WITH C AS
(
    SELECT YEAR(orderdate) AS orderyear, MONTH(orderdate) AS ordermonth, val
    FROM Sales.OrderValues
)
SELECT *
FROM C
PIVOT(SUM(val)
    FOR ordermonth IN ([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12])) AS P;
```

orderyear	1	2	3	4	5	6
2007	61258.08	38483.64	38547.23	53032.95	53781.30	36362.82
2008	94222.12	99415.29	104854.18	123798.70	18333.64	NULL
2006	NULL	NULL	NULL	NULL	NULL	NULL
orderyear	7	8	9	10	11	12
2007	51020.86	47287.68	55629.27	66749.23	43533.80	71398.44
2008	NULL	NULL	NULL	NULL	NULL	NULL
2006	27861.90	25485.28	26381.40	37515.73	45600.05	45239.63

В данном случае все три элемента сведения известны, в том числе уникальные значения в элементе распределения (месяцы). Но есть случаи сведения, когда элемент распределения заранее не существует и его требуется вычислить. Например, представьте запрос, который должен вернуть для каждого клиента идентификаторы самих последних пяти заказов. Хочется видеть идентификаторы клиентов в строках и идентификаторы заказов в виде данных, но у идентификаторов заказов нет ничего общего с клиентами, которых можно было бы использовать в качестве элемента распределения.

Решение заключается в использовании функции ROW_NUMBER, которая присваивает порядковые номера в секциях клиентов с использованием нужного упорядочения — в нашем случае по orderdate DESC и orderid DESC. Затем атрибут, содержащий номер строки, может использоваться как элемент распределения, и порядковые номера могут вычисляться как значения распределения.

Поэтому для начала приведу код, который генерирует номера строк клиентских заказов с самого последнего до самого старого:

```
SELECT custid, val,

ROW_NUMBER() OVER(PARTITION BY custid

ORDER BY orderdate DESC, orderid DESC) AS rownum

FROM Sales.OrderValues;
```


custid	val	rownum
1	933.50	1
1	471.20	2
1	845.80	3
1	330.00	4
1	878.00	5
1	814.50	6
2	514.40	1
2	320.00	2
2	479.75	3
2	88.80	4
3	660.00	1
3	375.50	2

Теперь можно определить выражение CTE на основе этого запроса, а затем во внешнем запросе организовать логику сведения с использованием rownum в качестве элемента распределения:

```
WITH C AS
(
    SELECT custid, val,
           ROW_NUMBER() OVER(PARTITION BY custid
                              ORDER BY orderdate DESC, orderid DESC) AS rownum
    FROM Sales.OrderValues
)
SELECT *
FROM C
PIVOT(MAX(val) FOR rownum IN ([1],[2],[3],[4],[5])) AS P;
```

custid	1	2	3	4	5
1	933.50	471.20	845.80	330.00	878.00
2	514.40	320.00	479.75	88.80	NULL
3	660.00	375.50	813.37	2082.00	1940.85
4	491.50	4441.25	390.00	282.00	191.10
5	1835.70	709.55	1096.20	2048.21	1064.50
6	858.00	677.00	625.00	464.00	330.00
7	730.00	660.00	450.00	593.75	1761.00
8	224.00	3026.85	982.00	NULL	NULL
9	792.75	360.00	1788.63	917.00	1979.23
10	525.00	1309.50	877.73	1014.00	717.50
11	1500.00	220.00	711.00	493.00	477.00
12	305.00	644.80	150.00	477.00	12.50

Если для каждого клиента нужно конкатенировать в одну строку идентификаторы последних пяти заказов, можно воспользоваться появившейся в SQL Server 2012 функцией *CONCAT*:

```
WITH C AS
```

```
(
```

```
    SELECT custid, CAST(orderid AS VARCHAR(11)) AS sorderid,
```

```
           ROW_NUMBER() OVER(PARTITION BY custid
```

```
                               ORDER BY orderdate DESC, orderid DESC) AS rownum
```

```
    FROM Sales.OrderValues
```

```
)
```

```
SELECT custid, CONCAT([1], ','+[2], ','+[3], ','+[4], ','+[5]) AS orderids
```

```
FROM C
```

```
    PIVOT(MAX(sorderid) FOR rownum IN ([1],[2],[3],[4],[5])) AS P;
```

custid	orderids
1	11011,10952,10835,10702,10692
2	10926,10759,10625,10308
3	10856,10682,10677,10573,10535
4	11016,10953,10920,10864,10793
5	10924,10875,10866,10857,10837
6	11058,10956,10853,10614,10582
7	10826,10679,10628,10584,10566
8	10970,10801,10326
9	11076,10940,10932,10876,10871
10	11048,11045,11027,10982,10975
11	11023,10947,10943,10599,10578
12	11054,10937,10881,10819,10782

Функция CONCAT автоматически заменяет NULL на пустую строку. Для получения такой же результата в более ранних версиях SQL Server придется воспользоваться оператором конкатенации "+" и функцией COALESCE, которая заменит значения NULL на пустую строку:

```
WITH C AS
```

```
(
```

```
    SELECT custid, CAST(orderid AS VARCHAR(11)) AS sorderid,
```

```
        ROW_NUMBER() OVER(PARTITION BY custid
```

```
                            ORDER BY orderdate DESC, orderid DESC) AS rownum
```

```
    FROM Sales.OrderValues
```

```
)
```

```
SELECT custid,
```

```
    [1] + COALESCE(',',+[2], '')
```

```
        + COALESCE(',',+[3], '')
```

```
        + COALESCE(',',+[4], '')
```

```
        + COALESCE(',',+[5], '') AS orderids
```

```
FROM C
```

```
PIVOT(MAX(sorderid) FOR rownum IN ([1],[2],[3],[4],[5])) AS P;
```

Оконные функции

Оконные функции определены в стандарте *ISO SQL*. В СУБД семейства *MS SQL Server* предоставляются ранжирующие и статистические *оконные функции*. **Окно — это набор строк, определяемый пользователем. Оконная функция вычисляет значение для каждой строки в результирующем наборе, полученном из окна.**

Оконные функции могут использоваться для вычисления кумулятивных, скользящих и центрированных агрегатов. Они возвращают *значение* для каждой строки в таблице, которое зависит от других строк соответствующего окна. Они могут быть использованы только в предложениях *SELECT* и *ORDER BY* запроса. Как правило, *оконные функции* обеспечивают *доступ* к более чем одной строке таблицы без самосоединения.

Предложение OVER

Предложение *OVER* определяет *секционирование* и упорядочение набора строк до применения соответствующей *оконной функции*. В качестве *оконных функций* используются агрегатные и статистические (*SUM*, *AVG*, *MAX*, *MIN*, *COUNT*), ранжирующие функции. Каждая из ранжирующих функций *ROW_NUMBER*, *DENSE_RANK*, *RANK* и *NTILE* задействует *предложение OVER* (см. сл. раздел настоящей лекции).

Синтаксис:

- Для ранжирующих *оконных функций*
`< OVER_CLAUSE > ::= OVER ([PARTITION BY value_expression, ... [n]] <ORDER BY expression>)`
- Для агрегатных функций
`< OVER_CLAUSE > ::= OVER ([PARTITION BY value_expression, ... [n]]`

PARTITION BY разделяет результирующий набор на секции. *Оконная функция* применяется к каждой секции отдельно, и вычисление начинается заново для каждой секции. Если это предложение опущено, функция интерпретирует все результирующее множество как одну группу.

value_expression указывает столбец, по которому секционируется набор строк, произведенный соответствующим предложением *FROM*. Аргумент *value_expression* может ссылаться только на столбцы, доступные через предложение *FROM*. Аргумент *value_expression* не может ссылаться на выражения или псевдонимы в списке выбора. Выражение *value_expression* может быть выражением столбца, скалярным вложенным запросом, скалярной функцией или пользовательской переменной.

Предложение *ORDER BY* задает порядок для ранжирующей оконной функции. Если предложение *ORDER BY* используется в контексте ранжирующей оконной функции, оно может ссылаться только на столбцы, доступные через предложение *FROM*. Указывать положение имени или псевдонима столбца в списке выборки с помощью целого числа нельзя. Предложение *ORDER BY* не может работать со статистическими оконными функциями.

В одном запросе с одним предложением *FROM* может использоваться несколько статистических или ранжирующих *оконных функций*. Однако *предложение OVER* для каждой функции может применять свое *секционирование* и упорядочение. *Предложение OVER* не может работать со *статистической функцией CHECKSUM*.

Семантика *NULL-значений оконных функций* соответствует семантике *NULL-значений агрегатных функций SQL*.

Статистические оконные функции

Покажем на примере, как используются статистические *оконные функции*.

Пример. 16.4. Статистические *оконные функции*.

Пусть в ХД имеется таблица фактов "Позиции счетов" (OrderDetail), содержащая номер позиции (OrderID), идентификатор товара (ProductID), количество товара (OrderQty) и стоимость товара (Price). Физическая структура таблицы приведена на [рис. 23.4](#).

OrderDetail			
OrderID	integer	<pk>	not null
ProductID	integer	<pk>	not null
OrderQty	integer		not null
Price	decimal(82,0)		not null

Рис. 23.4. Физическая структура таблицы фактов "Финансы" (Finance)

Пусть необходимо для двух позиций счетов 43659 и 43664 посчитать для каждого проданного товара общее количество проданного товара, среднее количество каждого проданного товара, минимальное и максимальное количество проданного товара.

Следующий запрос решает поставленную задачу с использованием *оконных функций*.

```
SELECT OrderID, ProductID, OrderQty
      ,SUM(OrderQty) OVER(PARTITION BY OrderID) AS 'Итого'
      ,AVG(OrderQty) OVER(PARTITION BY OrderID) AS 'Среднее'
      ,COUNT(OrderQty) OVER(PARTITION BY OrderID) AS 'Кол-во'
      ,MIN(OrderQty) OVER(PARTITION BY OrderID) AS 'Min'
      ,MAX(OrderQty) OVER(PARTITION BY OrderID) AS 'Max'
FROM OrderDetail
WHERE OrderID IN(43659,43664);
GO
```

Результат выполнения запроса приведен ниже.

Вывод 3.

OrderIDProductIDOrderQtyИтогоСреднееКол-воMinMax

43659	776	1	26	2	12	1	6
43659	777	3	26	2	12	1	6
43659	778	1	26	2	12	1	6
43659	771	1	26	2	12	1	6
43659	772	1	26	2	12	1	6
43664	772	1	14	1	8	1	4
43664	775	4	14	1	8	1	4
43664	714	1	14	1	8	1	4

Пусть руководство организации требует подсчитать процент проданного товара по позиции 43659. Следующий запрос с использованием *оконных функций* решает поставленную задачу.

```
SELECT OrderID, ProductID, OrderQty
      ,SUM(OrderQty) OVER(PARTITION BY OrderID) AS 'Итого'
      ,CAST(1. * OrderQty / SUM(OrderQty) OVER(PARTITION BY OrderID)
*100 AS DECIMAL(5,2))AS 'Процент проданного товара'
FROM OrderDetail
WHERE OrderID = 43659;
```

Результат выполнения запроса приведен ниже.

Вывод 4.

OrderID ProductID OrderQty Итого Процент проданного товара

43659	776	1	26	3.85
43659	777	3	26	11.54
43659	778	1	26	3.85
43659	771	1	26	3.85
43659	772	1	26	3.85

Как видно из рассмотренных примеров, использование *предложения OVER* является более эффективным, чем использование вложенных запросов. Применение оконных ранжирующих функций будет рассмотрено в следующем разделе.

тандарт SQL поддерживает четыре оконные функции, которые служат для ранжирования. Это ROW_NUMBER, NTILE, RANK и DENSE_RANK. В стандарте первые две считаются относящимися к одной категории, а вторые две — ко второй. Это связано с различиями в отношении детерминизма. Подробнее я расскажу в процессе рассказа об отдельных функциях.

Функции ранжирования появились еще в SQL Server 2005. Тем не менее я покажу альтернативные, основанные на наборах методы для получения того же результата. Я сделаю это по двум причинам: во-первых, это может быть интересным, а во-вторых, я верю, что это помогает лучше понять нюансы работы функций. Тем не менее имейте в виду, что на практике я настоятельно рекомендую придерживаться оконных функций, потому что они и проще, и намного эффективнее, чем альтернативные решения. Подробнее об оптимизации мы поговорим позже.

Все четыре функции ранжирования поддерживают необязательное предложение секционирования и обязательное предложение упорядочения окна. Если предложение секционирования окна отсутствует, весь результирующий набор базового запроса (вспомните о входных данных этапа SELECT) считается одной секцией. Что касается предложения упорядочения окна, то оно обеспечивает упорядочение при вычислениях. Понятно, что ранжирование строк без определения упорядочения вряд ли имеет смысл. В ранжирующих оконных функциях упорядочение служит другой цели по сравнению с функциями, поддерживающими кадрирование, такими как агрегирующие оконные функции. В первом случае упорядочение имеет логический смысл для самих вычислений, а во втором — упорядочение связано с кадрированием, то есть служит целям фильтрации.

Функция ROW_NUMBER

Эта функция вычисляет последовательные номера строк, начиная с 1, в соответствующей секции окна и в соответствии с заданным упорядочением окна. Посмотрите на пример запроса:

```
SELECTorderid, val,  
  
       ROW_NUMBER() OVER(ORDER BYorderid) AS rownum  
  
FROM Sales.OrderValues;
```

Вот сокращенный результат выполнения этого запроса:

Results		
orderid	val	rownum

10248	440.00	1
10249	1863.40	2
10250	1552.60	3
10251	654.06	4
10252	3597.90	5
10253	1444.80	6
10254	556.62	7
10255	2490.50	8
10256	517.80	9
10257	1119.90	10
10258	1614.88	11
10259	100.80	12

Этот запрос кажется тривиальным, но здесь есть несколько моментов, о которых стоит сказать.

Так как в запросе нет предложения представления ORDER BY, упорядочение представления не гарантируется. Поэтому упорядочение представления здесь нужно считать произвольным. На практике SQL Server оптимизирует запрос с учетом отсутствия предложения ORDER BY, поэтому строки могут возвращаться в любом порядке. Если нужно гарантировать упорядочение представления, нужно не забыть добавить предложение представления ORDER BY. Если нужно, чтобы упорядочение представления выполнялось на основе номера строки, можно использовать псевдоним, назначенный в процессе вычисления предложения представления ORDER BY примерно так:

```
SELECT orderid, val,

ROW_NUMBER() OVER(ORDER BY orderid) AS rownum

FROM Sales.OrderValues

ORDER BY rownum;
```

Считайте вычисление номеров строк генерацией еще одного атрибута в результирующем наборе запроса. Естественно, если хочется, можно получить упорядочение представления, которое отличается от упорядочения окна, как в следующем запросе:

```
SELECT orderid, val,

ROW_NUMBER() OVER(ORDER BY orderid) AS rownum

FROM Sales.OrderValues
```

```
ORDER BY val DESC;
```

orderid	val	rownum
10865	16387.50	618
10981	15810.00	734
11030	12615.05	783
10889	11380.00	642
10417	11188.40	170
10817	10952.85	570
10897	10835.24	650
10479	10495.60	232
10540	10191.70	293
10691	10164.80	444
10515	9921.30	268
10372	9210.90	125

Можно использовать *оконный агрегат COUNT* для создания операции которая логически эквивалентна функции ROW_NUMBER. Допустим, WPO - определение секционирования и упорядочения окна, примененное в функции ROW_NUMBER. Тогда ROW_NUMBER OVER WPO эквивалентно COUNT(*) OVER(WPO ROWS UNBOUNDED PRECEDING). Например, следующее эквивалентно запросу из листинга предыдущего примера:

```
SELECT orderid, val,

COUNT(*) OVER(ORDER BY orderid

                ROWS UNBOUNDED PRECEDING) AS rownum

FROM Sales.OrderValues;
```

Как я говорил, это хорошая тренировка — попытаться и создать альтернативные варианты вместо использования оконных функций, и не важно, что эти варианты сложнее и менее эффективные. Если уж мы говорим о функции ROW_NUMBER, то вот основанная на наборах стандартная альтернатива запросу, в которой оконные функции не используются:

```
SELECT orderid, val,

(SELECT COUNT(*)

 FROM Sales.OrderValues AS O2

 WHERE O2.orderid <= O1.orderid) AS rownum
```

```
FROM Sales.OrderValues AS 01;
```

В этом решении используется агрегат COUNT во вложенном запросе для определения, у скольких строк значение упорядочения (в нашем случаеorderid) меньше или равно текущему. Это просто, если у вас уникальное упорядочение, основанное на одном атрибуте. Но все сильно усложняется, если упорядочение неуникально, что я и продемонстрирую при обсуждении детерминизма.

Детерминизм

Если упорядочение окна уникально, вычисление ROW_NUMBER является детерминистическим. Это означает, что у запроса есть только один правильный результат, то есть если не менять входные данные, вы гарантировано будете получать повторяющиеся результаты. Но если упорядочение окна не уникально, вычисление становится недетерминистическим. Функция ROW_NUMBER генерирует уникальные номера строк в рамках секции, даже для строк с одинаковыми значениями в атрибутах упорядочения окна. В качестве примера посмотрите на следующий запрос:

```
SELECT orderid, orderdate, val,  
  
       ROW_NUMBER() OVER(ORDER BY orderdate DESC) AS rownum  
  
FROM Sales.OrderValues;
```

Results			
orderid	orderdate	val	rownum
11074	2008-05-06 00:00:00.000	232.09	1
11075	2008-05-06 00:00:00.000	498.10	2
11076	2008-05-06 00:00:00.000	792.75	3
11077	2008-05-06 00:00:00.000	1255.72	4
11070	2008-05-05 00:00:00.000	1629.98	5
11071	2008-05-05 00:00:00.000	484.50	6
11072	2008-05-05 00:00:00.000	5218.00	7
11073	2008-05-05 00:00:00.000	300.00	8
11067	2008-05-04 00:00:00.000	86.85	9
11068	2008-05-04 00:00:00.000	2027.08	10

Так как атрибут orderdate не уникальный, упорядочение строк с одинаковым значением orderdate следует считать произвольным. В принципе существует более одного корректного результата этого запроса. В качестве примера возьмем четыре строки с датой заказа 2008-05-06. Любой порядок строк с номерами от 1 до 4 считается правильным. Поэтому если вы выполните запрос снова, то в принципе можете получить другой порядок — сейчас не будем оценивать вероятность такого события, обусловленную особенностями реализации SQL Server.

Если нужно гарантировать повторяемость результатов, нужно сделать запрос детерминистическим. Это можно сделать, добавив дополнительный параметр в определение упорядочения окна, чтобы обеспечить уникальность в рамках секции. К примеру в следующем запросе уникальность упорядочения в окне достигается за счет добавления в список orderid DESC:

```
SELECT orderid, orderdate, val,  
  
       ROW_NUMBER() OVER(ORDER BY orderdate DESC, orderid DESC) AS rownum  
  
FROM Sales.OrderValues;
```

Results			
orderid	orderdate	val	rownum
11077	2008-05-06 00:00:00.000	1255.72	1
11076	2008-05-06 00:00:00.000	792.75	2
11075	2008-05-06 00:00:00.000	498.10	3
11074	2008-05-06 00:00:00.000	232.09	4
11073	2008-05-05 00:00:00.000	300.00	5
11072	2008-05-05 00:00:00.000	5218.00	6
11071	2008-05-05 00:00:00.000	484.50	7
11070	2008-05-05 00:00:00.000	1629.98	8
11069	2008-05-04 00:00:00.000	360.00	9
11068	2008-05-04 00:00:00.000	2027.08	10

При использовании оконных функций детерминизм вычисления номеров страниц реализуется просто. Сделать то же, не прибегая к оконным функциям сложнее, но вполне реально:

```
SELECT orderdate, orderid, val,  
  
       (SELECT COUNT(*)  
  
        FROM Sales.OrderValues AS O2  
  
        WHERE O2.orderdate >= O1.orderdate  
  
              AND (O2.orderdate > O1.orderdate  
  
                   OR O2.orderid >= O1.orderid)) AS rownum  
  
FROM Sales.OrderValues AS O1;
```

Но вернемся к функции ROW_NUMBER: как мы выдели, ее можно использовать для создания недетерминистических вычислений при использовании

неуникального упорядочения. Таким образом, недетерминизм разрешен, но странно то, что он не разрешен полностью. Я имею в виду то, что предложение ORDER BY обязательно. Но что, если вы хотите просто получить в секции уникальные номера строк не обязательно в каком-то определенном порядке? Можно создать такой запрос:

```
SELECTorderid, orderdate, val,  
  
    ROW_NUMBER() OVER() AS rownum  
  
FROM Sales.OrderValues;
```

Но, как уже говорилось, предложение ORDER BY в функциях ранжирования является обязательным:

Вы можете поумничать и определить константу в списке ORDER BY:

```
SELECTorderid, orderdate, val,  
  
    ROW_NUMBER() OVER(ORDER BY NULL) AS rownum  
  
FROM Sales.OrderValues;
```

Но SQL Server вернет такую ошибку:

Однако решение есть, и я вскоре его приведу.

Функции RANK и DENSE_RANK

Функции RANK и DENSE_RANK похожи на ROW_NUMBER, но в отличие от нее они не создают уникальные значения в оконной секции. При упорядочении окна по возрастанию «Обычный ранг» RANK вычисляется как единица плюс число строк со значением, по которому выполняется упорядочение, меньшим, чем текущее значение в секции. «Плотный ранг» DENSE_RANK вычисляется как единица плюс число уникальных строк со значением, по которому выполняется упорядочение, меньшим, чем текущее значение в секции. При упорядочении окна по убыванию RANK вычисляется как единица плюс число строк со значением, по которому выполняется упорядочение, большим, чем текущее значение в секции. DENSE_RANK вычисляется как единица плюс число уникальных строк со значением, по которому выполняется упорядочение, большим, чем текущее значение в секции.

Вот пример запроса, вычисляющего номера строк, ранги и «уплотненные» ранги. При этом используется секционирование окна по умолчанию и упорядочение по orderdate DESC:

```

SELECT orderid, orderdate, val,

ROW_NUMBER() OVER(ORDER BY orderdate DESC) AS rownum,

RANK()          OVER(ORDER BY orderdate DESC) AS rnk,

DENSE_RANK() OVER(ORDER BY orderdate DESC) AS drnk

FROM Sales.OrderValues;

```

orderid	orderdate	val	rownum	rnk	drnk
11074	2008-05-06 00:00:00.000	232.09	1	1	1
11075	2008-05-06 00:00:00.000	498.10	2	1	1
11076	2008-05-06 00:00:00.000	792.75	3	1	1
11077	2008-05-06 00:00:00.000	1255.72	4	1	1
11070	2008-05-05 00:00:00.000	1629.98	5	5	2
11071	2008-05-05 00:00:00.000	484.50	6	5	2
11072	2008-05-05 00:00:00.000	5218.00	7	5	2
11073	2008-05-05 00:00:00.000	300.00	8	5	2
11067	2008-05-04 00:00:00.000	86.85	9	9	3
11068	2008-05-04 00:00:00.000	2027.08	10	9	3
11069	2008-05-04 00:00:00.000	360.00	11	9	3
11064	2008-05-01 00:00:00.000	4330.40	12	12	4

Атрибут orderdate не уникален. Но заметьте, что при этом номера строк уникальны. Значения RANK и DENSE_RANK не уникальны. Все строки с одной датой заказа, например 2008-05-05, получили одинаковый «неплотный» ранг 5 и «плотный» ранг 2. Ранг 5 означает, что есть четыре строки с большими (более поздними) датами заказов (упорядочение ведется по убыванию), а «плотный» ранг 2 означает, что есть одна более поздняя уникальная дата.

Альтернативное решение, заменяющее RANK и DENSE_RANK и не использующие оконные функции, создается просто:

```

SELECT orderid, orderdate, val,

(SELECT COUNT(*)

FROM Sales.OrderValues AS O2

WHERE O2.orderdate > O1.orderdate) + 1 AS rnk,

(SELECT COUNT(DISTINCT orderdate)

FROM Sales.OrderValues AS O2

```

```
WHERE 02.orderdate > 01.orderdate) + 1 AS drnk  
  
FROM Sales.OrderValues AS 01;
```

Для вычисления ранга определяется число строк с большим значением, по которому ведется упорядочение, (как вы помните, упорядочение ведется по убыванию) и к нему добавляется единица. Для вычисления плотного ранга нужно пересчитать уникальные большие значения, по которым ведется упорядочение, и добавить к полученному числу единицу.

Детерминизм

Как вы уже сами, наверное, поняли, что как RANK, так и DENSE_RANK детерминистичны по определению. При одном значении упорядочения — независимо от его уникальности — возвращается одно и то же значение ранга. Вообще говоря, эти две функции обычно интересны, если упорядочение неуникально. Если упорядочение уникально, они дают те же результаты, что и Функция ROW_NUMBER.

Функция	Возвращаемое значение
RANK	Возвращает ранг каждой строки в секции результирующего набора. Ранг строки вычисляется как единица плюс количество рангов, находящихся до этой строки. Возвращаемый тип данных — <code>bigint</code>
DENSE_RANK	Возвращает ранг строк в секции результирующего набора без промежутков в ранжировании. Ранг строки равен количеству различных значений рангов, которые предшествуют строке, увеличенному на единицу. Возвращаемый тип данных — <code>bigint</code>
NTILE	Распределяет строки упорядоченной секции в заданное количество групп. Группы нумеруются, начиная с единицы. Для каждой строки функция <code>NTILE</code> возвращает номер группы, которой принадлежит строка
ROW_NUMBER	Возвращает последовательный номер строки в секции результирующего набора, 1 соответствует первой строке в каждой из секций. Возвращаемый тип данных — <code>bigint</code>

Оконные функции смещения делятся на две категории. Первая категория - функции, смещение которых указывается по отношению к текущей строке. Это LAG и LEAD. В функциях второй категории смещение указывается по отношению к началу или концу оконного кадра. Сюда относятся функции FIRST_VALUE, LAST_VALUE и NTH_VALUE. SQL Server 2012 поддерживает LAG, LEAD, FIRST_VALUE и LAST_VALUE и не поддерживает NTH_VALUE.

Функции первой категории (LAG и LEAD) поддерживают предложение секционирования, а также упорядочения окна. Ясно, что вторая часть вносит смысл в смещение. Функции из второй категории (FIRST_VALUE, LAST_VALUE и NTH_VALUE) помимо предложения секционирования и упорядочения окна поддерживают предложение оконного кадра.

Функции LAG и LEAD

Функции LAG и LEAD позволяют возвращать выражение значения из строки в секции окна, которая находится на заданном смещении перед (LAG) или после (LEAD) текущей строки. Смещение по умолчанию — «1», оно применяется, если смещение не указать.<./

Например, следующий запрос возвращает текущую стоимость для каждого клиентского заказа, а также стоимости предыдущего и последующего заказов этого же клиента:

```
SELECT custid, orderdate,orderid, val,
       LAG(val) OVER(PARTITION BY custid
                    ORDER BY orderdate,orderid) AS prevval,
       LEAD(val) OVER(PARTITION BY custid
                     ORDER BY orderdate,orderid) AS nextval
FROM Sales.OrderValues;
```


custid	orderdate	orderid	val	prevval	nextval
1	2007-08-25 00:00:00.000	10643	814.50	NULL	878.00
1	2007-10-03 00:00:00.000	10692	878.00	814.50	330.00
1	2007-10-13 00:00:00.000	10702	330.00	878.00	845.80
1	2008-01-15 00:00:00.000	10835	845.80	330.00	471.20
1	2008-03-16 00:00:00.000	10952	471.20	845.80	933.50
1	2008-04-09 00:00:00.000	11011	933.50	471.20	NULL
2	2006-09-18 00:00:00.000	10308	88.80	NULL	479.75
2	2007-08-08 00:00:00.000	10625	479.75	88.80	320.00
2	2007-11-28 00:00:00.000	10759	320.00	479.75	514.40
2	2008-03-04 00:00:00.000	10926	514.40	320.00	NULL
3	2006-11-27 00:00:00.000	10365	403.20	NULL	749.06
3	2007-04-15 00:00:00.000	10507	749.06	403.20	1940.85
3	2007-05-13 00:00:00.000	10535	1940.85	749.06	2082.00
3	2007-06-19 00:00:00.000	10573	2082.00	1940.85	813.37

Так как мы явно не задали смещение, по умолчанию предполагается смещение в единицу. Так как данные в функции секционируются по custid, поиск строк выполняется только в рамках той же секции, содержащей данные одного клиента. Что касается упорядочения окон, то понятия «предыдущий» и «следующий» определяются упорядочением по orderdate и orderid в качестве дополнительного параметра. Заметьте, что в результатах запроса LAG возвращает NULL для первой строки оконной секции, потому что перед первой строкой других строк нет; аналогично LEAD возвращает NULL для последней строки.

Если нужно смещение, отличное от единицы, нужно указать его после входного выражения значения, как в этом запросе:

```
SELECT custid, orderdate, orderid,

LAG(val, 3) OVER(PARTITION BY custid

ORDER BY orderdate, orderid) AS prev3val

FROM Sales.OrderValues;
```

custid	orderdate	orderid	prev3val
1	2007-08-25 00:00:00.000	10643	NULL
1	2007-10-03 00:00:00.000	10692	NULL
1	2007-10-13 00:00:00.000	10702	NULL
1	2008-01-15 00:00:00.000	10835	814.50
1	2008-03-16 00:00:00.000	10952	878.00
1	2008-04-09 00:00:00.000	11011	330.00
2	2006-09-18 00:00:00.000	10308	NULL
2	2007-08-08 00:00:00.000	10625	NULL
2	2007-11-28 00:00:00.000	10759	NULL
2	2008-03-04 00:00:00.000	10926	88.80
3	2006-11-27 00:00:00.000	10365	NULL
3	2007-04-15 00:00:00.000	10507	NULL
3	2007-05-13 00:00:00.000	10535	NULL
3	2007-06-19 00:00:00.000	10573	403.20

Как говорилось, LAG и LEAD по умолчанию возвращают NULL, если по заданному смещению нет строки. Если нужно возвращать другое значение, можно указать его в качестве третьего аргумента функции. Например, LAG(val, 3, 0.00) возвращает «0.00», если по смещению 3 перед текущей строкой строки вообще нет.

Для реализации подобного поведения в LAG и LEAD на версии SQL Server, предшествующей SQL Server 2012, можно применить следующий подход:

1. Напишите запрос, который возвращает номера строк с требуемыми параметрами секционирования и упорядочения, и создайте на его основе табличное выражение.
2. Соедините множественные табличные выражения так, чтобы они представляли текущую, предыдущую и следующую строки.
3. В предикате соединения сопоставьте столбцы секционирования различных экземпляров (текущего с предыдущим или последующим). Также в предикате соединения вычислите разницу между числом строк текущего и предыдущего или следующего экземпляра, а затем отфильтруйте на основе значения смещения, которое требуется в ваших вычислениях.

Вот запрос, реализующий этот подход и возвращающий для каждого заказа значения текущего, предыдущего и следующего заказа клиента:

```
WITH OrdersRN AS
```

```
(
```

```
SELECT custid, orderdate, orderid, val,
```

```

    ROW_NUMBER() OVER(ORDER BY custid, orderdate, orderid) AS rn

FROM Sales.OrderValues

)

SELECT C.custid, C.orderdate, C.orderid, C.val,

    P.val AS prevval,

    N.val AS nextval

FROM OrdersRN AS C

    LEFT OUTER JOIN OrdersRN AS P

        ON C.custid = P.custid

        AND C.rn = P.rn + 1

    LEFT OUTER JOIN OrdersRN AS N

        ON C.custid = N.custid

        AND C.rn = N.rn - 1;

```

Ясно, что решить эту задачу можно также с помощью простых вложенных запросов.

Функции FIRST_VALUE, LAST_VALUE и NTH_VALUE

В предыдущем разделе я рассказал о функциях смещения LAG и LEAD, которые позволяют задавать смещение относительно текущей строки. Этот раздел посвящен функциям, которые позволяют определять смещение относительно начала или конца оконного кадра. Это функции FIRST_VALUE, LAST_VALUE и NTH_VALUE, причем последняя не реализована в SQL Server 2012.

Напомню, что LAG и LEAD поддерживают предложения секционирования и упорядочение, но не поддерживают предложение кадрирования окна. Это разумно, если смещение указывается относительно текущей строки. В функциях, в которых смещение указывается по отношению к началу или концу окна, кадрирование имеет смысл. Функции FIRST_VALUE и LAST_VALUE возвращают запрошенное выражение значения соответственно из первой и последней строки в кадре. Вот запрос, демонстрирующий, как возвращать с каждым заказом

клиента текущее значение этого заказа, а также значения первого и последнего заказа:

```
SELECT custid, orderdate, orderid, val,  
  
       FIRST_VALUE(val) OVER(PARTITION BY custid  
  
                               ORDER BY orderdate, orderid) AS val_firstorder,  
  
       LAST_VALUE(val)  OVER(PARTITION BY custid  
  
                               ORDER BY orderdate, orderid  
  
                               ROWS BETWEEN CURRENT ROW  
  
                               AND UNBOUNDED FOLLOWING) AS val_lastorder  
  
FROM Sales.OrderValues;
```

Results					
custid	orderdate	orderid	val	val_firstorder	val_lastorder
1	2008-04-09 00:00:00.000	11011	933.50	814.50	933.50
1	2008-03-16 00:00:00.000	10952	471.20	814.50	933.50
1	2008-01-15 00:00:00.000	10835	845.80	814.50	933.50
1	2007-10-13 00:00:00.000	10702	330.00	814.50	933.50
1	2007-10-03 00:00:00.000	10692	878.00	814.50	933.50
1	2007-08-25 00:00:00.000	10643	814.50	814.50	933.50
2	2008-03-04 00:00:00.000	10926	514.40	88.80	514.40
2	2007-11-28 00:00:00.000	10759	320.00	88.80	514.40
2	2007-08-08 00:00:00.000	10625	479.75	88.80	514.40
2	2006-09-18 00:00:00.000	10308	88.80	88.80	514.40
3	2008-01-28 00:00:00.000	10856	660.00	403.20	660.00
3	2007-09-25 00:00:00.000	10682	375.50	403.20	660.00
3	2007-09-22 00:00:00.000	10677	813.37	403.20	660.00
3	2007-06-19 00:00:00.000	10573	2082.00	403.20	660.00

С технической точки зрения нам нужны значения из первой и последней строки секции. С `FIRST_VALUE` просто, потому что можно использовать кадрирование по умолчанию. Как вы помните, если поддерживается кадрирование и не указать предложение кадрирования окна, по умолчанию будет применяться `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Но с `LAST_VALUE` кадрирование по умолчанию бесполезно, потому что последней является текущая строка. Поэтому в этом примере используется явное определение кадра с `UNBOUNDED FOLLOWING` в качестве нижней границы кадра.

Обычно не возвращают первое или последнее значение вместе со всеми подробностями строк, как в предыдущем примере — в вычислениях обычно работают с одной цифрой и значением, возвращенным оконной функцией. В следующем примере запрос возвращает, вместе с каждым клиентским заказом, стоимость текущего заказа, а также разницу между ней и стоимостью первого и последнего заказа клиента:

```

SELECT custid, orderdate, orderid, val,

val - FIRST_VALUE(val) OVER(PARTITION BY custid

ORDER BY orderdate, orderid) AS difffirst,

val - LAST_VALUE(val) OVER(PARTITION BY custid

ORDER BY orderdate, orderid

ROWS BETWEEN CURRENT ROW

AND UNBOUNDED FOLLOWING) AS diffblast

FROM Sales.OrderValues;

```

Results						
custid	orderdate	orderid	val	difffirst	diffblast	
1	2008-04-09 00:00:00.000	11011	933.50	119.00	0.00	
1	2008-03-16 00:00:00.000	10952	471.20	-343.30	-462.30	
1	2008-01-15 00:00:00.000	10835	845.80	31.30	-87.70	
1	2007-10-13 00:00:00.000	10702	330.00	-484.50	-603.50	
1	2007-10-03 00:00:00.000	10692	878.00	63.50	-55.50	
1	2007-08-25 00:00:00.000	10643	814.50	0.00	-119.00	
2	2008-03-04 00:00:00.000	10926	514.40	425.60	0.00	
2	2007-11-28 00:00:00.000	10759	320.00	231.20	-194.40	
2	2007-08-08 00:00:00.000	10625	479.75	390.95	-34.65	
2	2006-09-18 00:00:00.000	10308	88.80	0.00	-425.60	
3	2008-01-28 00:00:00.000	10856	660.00	256.80	0.00	
3	2007-09-25 00:00:00.000	10682	375.50	-27.70	-284.50	
3	2007-09-22 00:00:00.000	10677	813.37	410.17	153.37	
3	2007-06-19 00:00:00.000	10573	2082.00	1678.80	1422.00	

Как я говорил, стандартная функция NTH_VALUE не реализована в SQL Server 2012. Эта функция позволяет запрашивать выражение значения, которое находится на заданном смещении, выраженном в числе строк, от первой или последней строки в оконном кадре. Смещение задается во втором входном значении после выражения значения и ключевого слова FROM_FIRST или FROM_LAST, которое указывает, от какой строки отсчитывать смещение — от первой или последней. Например, следующее выражение возвращает значение из третьей строки, если считать от самой нижней в секции:

```

NTH_VALUE(val, 3) FROM LAST OVER(ROWS BETWEEN CURRENT ROW

AND UNBOUNDED FOLLOWING)

```

Представим, что нам надо обеспечить функциональность, которую реализуют функции FIRST_VALUE, LAST_VALUE и NTH_VALUE, в версии, предшествующей SQL Server 2012. Для этого можно использовать такие конструкции, как

обобщенные табличные выражения (CTE), функцию ROW_NUMBER и выражение CASE, группировку и соединение, следующим образом:

```
WITH OrdersRN AS

(

    SELECT custid, val,

        ROW_NUMBER() OVER(PARTITION BY custid

                            ORDER BY orderdate, orderid) AS rna,

        ROW_NUMBER() OVER(PARTITION BY custid

                            ORDER BY orderdate DESC, orderid DESC) AS rnd

    FROM Sales.OrderValues

),

Agg AS

(

    SELECT custid,

        MAX(CASE WHEN rna = 1 THEN val END) AS firstorderval,

        MAX(CASE WHEN rnd = 1 THEN val END) AS lastorderval,

        MAX(CASE WHEN rna = 3 THEN val END) AS thirddorderval

    FROM OrdersRN

    GROUP BY custid

)

SELECT O.custid, O.orderdate, O.orderid, O.val,

    A.firstorderval, A.lastorderval, A.thirddorderval
```

```

FROM Sales.OrderValues AS O

JOIN Agg AS A

ON O.custid = A.custid

ORDER BY custid, orderdate, orderid;

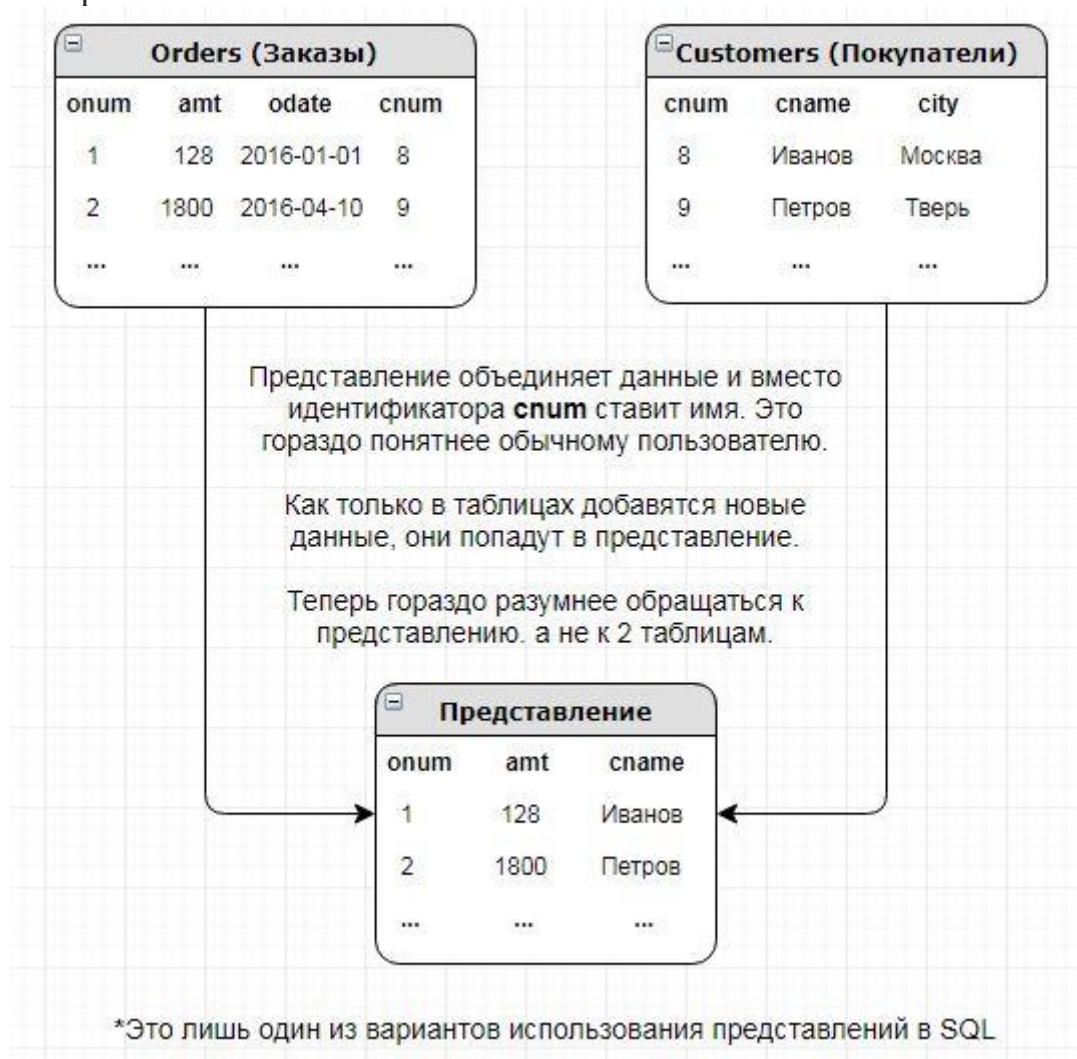
```

Results						
custid	orderdate	orderid	val	firstorderval	lastorderval	thirdorderval
1	2007-08-25 00:00:00.000	10643	814.50	814.50	933.50	330.00
1	2007-10-03 00:00:00.000	10692	878.00	814.50	933.50	330.00
1	2007-10-13 00:00:00.000	10702	330.00	814.50	933.50	330.00
1	2008-01-15 00:00:00.000	10835	845.80	814.50	933.50	330.00
1	2008-03-16 00:00:00.000	10952	471.20	814.50	933.50	330.00
1	2008-04-09 00:00:00.000	11011	933.50	814.50	933.50	330.00
2	2006-09-18 00:00:00.000	10308	88.80	88.80	514.40	320.00
2	2007-08-08 00:00:00.000	10625	479.75	88.80	514.40	320.00
2	2007-11-28 00:00:00.000	10759	320.00	88.80	514.40	320.00
2	2008-03-04 00:00:00.000	10926	514.40	88.80	514.40	320.00
3	2006-11-27 00:00:00.000	10365	403.20	403.20	660.00	1940.85
3	2007-04-15 00:00:00.000	10507	749.06	403.20	660.00	1940.85
3	2007-05-13 00:00:00.000	10535	1940.85	403.20	660.00	1940.85
3	2007-06-19 00:00:00.000	10573	2082.00	403.20	660.00	1940.85
3	2007-09-22 00:00:00.000	10677	813.37	403.20	660.00	1940.85
3	2007-09-25 00:00:00.000	10682	375.50	403.20	660.00	1940.85
3	2008-01-28 00:00:00.000	10856	660.00	403.20	660.00	1940.85

В первом CTE по имени OrdersRN определяются номера строк как возрастающем, так и убывающем порядке для отметки позиций строк по отношению к первой и последней строке в секции. Во втором CTE по имени Agg используется выражение CASE для фильтрации только нужных номеров строк, группировки данных по элементу секционирования {custid) и применения агрегата к результату выражения CASE, чтобы вернуть запрошенное значение для каждой группы. Наконец во внешнем запросе результат группового запроса соединяется с исходной таблицей для сопоставления детализованной информации с агрегатами.

37) Итак, представления в SQL являются особым объектом, который содержит данные, полученные запросом SELECT из обычных таблиц. Это виртуальная таблица, к которой можно обратиться как к обычным таблицам и получить хранимые данные. Представление в SQL может содержать в себе как данные из одной единственной таблицы, так и из нескольких таблиц.

Представления нужны для того, чтобы упростить работу с базой данных и ускорить время ответа сервера. Так как представление — это уже результат некой выборки данных с помощью SELECT, то, очевидно, в следующий раз вместо запроса к нескольким таблицам достаточно просто обратиться к уже созданному представлению. Работу этого объекта характеризует следующее изображение:



На изображении — простой вариант использования представления, когда объединяются данные по идентификатору. Но, помимо этого, в представлениях могут быть разные виды условий и ограничений, также вложенные запросы и группировки по каким либо полям. Об этом будет сказано чуть позже.

Создание представления в SQL

Создание представления осуществляется следующей командой:

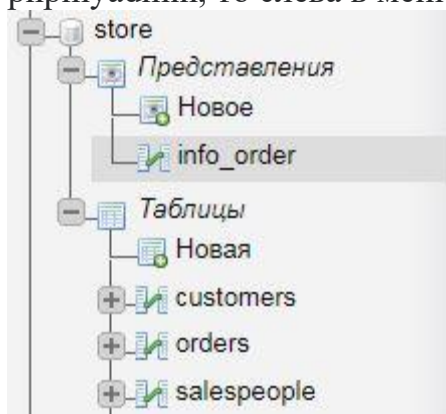
```
CREATE VIEW name_view  
as
```

Где **name_view** — имя, которое задает пользователь при создании. После ключевого слова **as** идет код запроса **SELECT**, данные которого и поместятся в представление. Чтобы легче понять разберем простой пример, иллюстрация которого была представлена выше.

```
CREATE VIEW info_order  
AS SELECT onum, amt, cname  
FROM orders, customers  
WHERE orders.cnum = customers.cnum
```

С помощью **SELECT** выбираются данные и помещаются в представление. Еще раз повторим, что когда данные в исходных таблицах изменятся, то они поменяются и в представлении.

Теперь представление практически является таблицей, если вы используете `phpmyadmin`, то слева в меню можно увидеть такую картину:



38)

На мой взгляд, возможности объектов просмотра слишком малы и далеки от идеала. Главная их проблема - статичность - статичность. Чтобы получить новый результат (добавить или изменить критерий поиска) приходится изменять саму хранимую процедуру, что достаточно проблематично и большая часть преимуществ объектов просмотра просто теряется.

Хранимые процедуры - это именованный набор операторов Transact-SQL хранящийся на сервере. Хранимые процедуры - это метод выполнения повторяющихся задач и при этом обладают большими возможностями, чем объекты просмотра.

Сервер SQL поддерживает 5 типов встроенных процедур:

- системные хранимые процедуры - хранятся в базе данных master. Система хранит процедуры (определяющиеся по префиксу sp_) предоставляющие эффективные методы получения информации из системных таблиц. Они позволяют системному администратору выполнять администраторские задачи над базой данных, которые обновляют необходимые таблицы напрямую. Системные встроенные процедуры могут быть выполнены из любой базы данных;
- локальные хранимые процедуры - создаются в определенных пользовательских таблицах;
- временные хранимые процедуры - могут быть локальными с именами, начинающимися с единичного знака # или глобальными начинающимися со знака ## (как и локальные/глобальные временные таблицы). Локальные временные процедуры доступны только в единственной пользовательской сессии. Глобальные - доступны всем пользователям. Как и для таблиц, так и для процедур я не рекомендую использовать временные процедуры. Я еще не встречался с такой задачей, которую нельзя было решить без временных процедур;
- удаленные хранимые процедуры - устаревшая технология MS SQL Server. На данный момент эту задачу решают распределенные запросы;
- расширенные встроенные процедуры (содержат в имени префикс xp_) - разрабатываются в виде DLL (Dynamic Link Library, динамически подгружаемая библиотека) и выполняются вне окружения SQL Server. Обычно такие процедуры идентифицируются по префиксу xp_.

Хранимые процедуры в MS SQL Server похожи на процедуры в других языках программирования. Если вы имели опыт программирования на каком-либо языке и не понаслышке знаете о таком понятии как процедуры, то материал этой главы покажется вам слишком простым. Но уровень подготовки читателей может быть разным, поэтому я постараюсь описать все максимально простым и доступным языком

На этом теорию на время остановим и посмотрим, как на практике создаются хранимые процедуры. Для этого используется оператор CREATE PROCEDURE, который выглядит следующим образом:

```
CREATE PROC [ EDURE ] procedure_name [ ; number ]
    [ { @parameter data_type }
        [ VARYING ] [ = default ] [ OUTPUT ]
    ] [ ,...n ]

[ WITH
    { RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION } ]

[ FOR REPLICATION ]

AS sql_statement [ ...n ]
```

Для создания процедуры, вы должны иметь соответствующие права, например, быть владельцем базы данных или администратором сервера базы данных.

Вы можете создать процедуры только в текущей базе данных, исключая временные процедуры, которые всегда создаются в базе данных tempdb. Создание процедуры похоже на создание объекта просмотра. Первым делом напишите и протестируйте операторы Transact-SQL. После этого, если вы получили результат, который ожидали, создавайте процедуру.

Для имен процедур лучше всего выбрать префикс, который будет указывать, что эта процедура создана именно вами. Только не используйте для этого префикс sp_, чтобы не было конфликтов с системными процедурами.

Давайте попробуем создать процедуру, которая будет получать из таблицы данные о работниках телефонах, т.е. следующий запрос:

```
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
       dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

Это простейшая процедура, которая не будет использовать переменных, поэтому для ее создания необходимо написать:

```
CREATE PROCEDURE GetPhones
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
       dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

Как видите, в начало запроса всего лишь добавляется две строки (хотя, можно написать и в одну). В первой мы пишем операторы **CREATE PROCEDURE** и имя процедуры, а во второй строке ключевое слово **AS**. После этого идет простой запрос **SELECT**, который выбирает данные.

В общем виде команда выглядит достаточно страшно, но к концу главы мы рассмотрим достаточно примеров, и вы увидите, что ничего страшного тут нет. Для процедуры **GetPhones**, которую мы создали ранее, необходимо указать только:

```
EXEC GetPhones
```

или

```
EXECUTE GetPhones
```

Триггер - это механизм, который вызывается, когда в указанной таблице происходит определенное действие. Каждый триггер имеет следующие основные составляющие: имя, действие и исполнение. Имя триггера может содержать максимум 128 символов. Действием триггера может быть или инструкция DML (INSERT, UPDATE или DELETE), или инструкция DDL. Таким образом, существует два типа триггеров: триггеры DML и триггеры DDL. Исполнительная составляющая триггера обычно состоит из хранимой процедуры или пакета.

Компонент Database Engine позволяет создавать триггеры, используя или язык Transact-SQL, или один из языков среды CLR, такой как C# или Visual Basic.

Создание триггера DML

Триггер создается с помощью инструкции **CREATE TRIGGER**, которая имеет следующий синтаксис:

Соглашения по синтаксису

Предшествующий синтаксис относится только к триггерам DML. Триггеры DDL имеют несколько иную форму синтаксиса, которая будет показана позже.

Здесь в параметре `schema_name` указывается имя схемы, к которой принадлежит триггер, а в параметре `trigger_name` - имя триггера. В параметре `table_name` задается имя таблицы, для которой создается триггер. (Также поддерживаются триггеры для представлений, на что указывает наличие параметра `view_name`.)

Также можно задать тип триггера с помощью двух дополнительных параметров: AFTER и INSTEAD OF. (Параметр FOR является синонимом параметра AFTER.) Триггеры типа **AFTER** вызываются после выполнения действия, запускающего триггер, а триггеры типа **INSTEAD OF** выполняются вместо действия, запускающего триггер. Триггеры AFTER можно создавать только для таблиц, а триггеры INSTEAD OF - как для таблиц, так и для представлений.

Параметры INSERT, UPDATE и DELETE задают действие триггера. Под действием триггера имеется в виду инструкция Transact-SQL, которая запускает триггер. Допускается любая комбинация этих трех инструкций. Инструкция DELETE не разрешается, если используется параметр IF UPDATE.

Как можно видеть в синтаксисе инструкции CREATE TRIGGER, действие (или действия) триггера указывается в спецификации AS `sql_statement`.

Компонент Database Engine позволяет создавать несколько триггеров для каждой таблицы и для каждого действия (INSERT, UPDATE и DELETE). По

умолчанию определенного порядка исполнения нескольких триггеров для данного модифицирующего действия не имеется.

Только владелец базы данных, администраторы DDL и владелец таблицы, для которой определяется триггер, имеют право создавать триггеры для текущей базы данных. (В отличие от разрешений для других типов инструкции CREATE это разрешение не может передаваться.)

Изменение структуры триггера

Язык Transact-SQL также поддерживает инструкцию **ALTER TRIGGER**, которая модифицирует структуру триггера. Эта инструкция обычно применяется для изменения тела триггера. Все предложения и параметры инструкции ALTER TRIGGER имеют такое же значение, как и одноименные предложения и параметры инструкции CREATE TRIGGER.

Для удаления триггеров в текущей базе данных применяется инструкция **DROP TRIGGER**.

Использование виртуальных таблиц **deleted** и **inserted**

При создании действия триггера обычно требуется указать, ссылается ли он на значение столбца до или после его изменения действием, запускающим триггер. По этой причине, для тестирования следствия инструкции, запускающей триггер, используются две специально именованные виртуальные таблицы:

- **deleted** - содержит копии строк, удаленных из таблицы;
- **inserted** - содержит копии строк, вставленных в таблицу.

Структура этих таблиц эквивалентна структуре таблицы, для которой определен триггер.

Таблица deleted используется в том случае, если в инструкции CREATE TRIGGER указывается предложение DELETE или UPDATE, а если в этой инструкции указывается предложение INSERT или UPDATE, то используется *таблица inserted*. Это означает, что для каждой инструкции DELETE, выполненной в действии триггера, создается таблица deleted. Подобным образом для каждой инструкции INSERT, выполненной в действии триггера, создается таблица inserted.

Инструкция UPDATE рассматривается, как инструкция DELETE, за которой следует инструкция INSERT. Поэтому для каждой инструкции UPDATE, выполненной в действии триггера, создается как таблица deleted, так и таблица inserted (в указанной последовательности).

Таблицы inserted и deleted реализуются, используя управление версиями строк, которое рассматривалось в предыдущей статье. Когда для таблицы с

соответствующими триггерами выполняется инструкция DML (INSERT, UPDATE или DELETE), для всех изменений в этой таблице всегда создаются версии строк. Когда триггеру требуется информация из таблицы deleted, он обращается к данным в хранилище версий строк. В случае таблицы inserted, триггер обращается к самым последним версиям строк.

В качестве хранилища версий строк механизм управления версиями строк использует системную базу данных tempdb. По этой причине, если база данных содержит большое число часто используемых триггеров, следует ожидать значительного увеличения объема этой системной базы данных.

Области применения DML-триггеров

Такие триггеры применяются для решения разнообразных задач. В этом разделе мы рассмотрим несколько областей применения триггеров DML, в частности триггеров AFTER и INSTEAD OF.

Триггеры AFTER

Как вы уже знаете, триггеры AFTER вызываются после того, как выполняется действие, запускающее триггер. Триггер AFTER задается с помощью ключевого слова AFTER или FOR. Триггеры AFTER можно создавать только для базовых таблиц. Триггеры этого типа можно использовать для выполнения, среди прочих, следующих операций:

- создания журнала логов действий в таблицах базы данных;
- реализации бизнес-правил;
- принудительного обеспечения ссылочной целостности.

Создание журнала логов

В SQL Server можно выполнять отслеживание изменения данных, используя систему перехвата изменения данных CDC (change data capture). Эту задачу можно также решить с помощью триггеров DML. В примере ниже показывается, как с помощью триггеров можно создать журнал логов действий в таблицах базы данных:

```
USE SampleDb;
```

```
/* Таблица AuditBudget используется в качестве
```

```
журнала логов действий в таблице Project */
```

GO

```
CREATE TABLE AuditBudget (  
  
    ProjectNumber CHAR(4) NULL,  
  
    UserName CHAR(16) NULL,  
  
    Date DATETIME NULL,  
  
    BudgetOld FLOAT NULL,  
  
    BudgetNew FLOAT NULL  
  
);
```

GO

```
CREATE TRIGGER trigger_ModifyBudget  
  
    ON Project AFTER UPDATE  
  
    AS IF UPDATE(budget)  
  
BEGIN  
  
    DECLARE @budgetOld FLOAT  
  
    DECLARE @budgetNew FLOAT  
  
    DECLARE @projectNumber CHAR(4)  
  
  
    SELECT @budgetOld = (SELECT Budget FROM deleted)  
  
    SELECT @budgetNew = (SELECT Budget FROM inserted)  
  
    SELECT @projectNumber = (SELECT Number FROM deleted)
```



```
INSERT INTO AuditBudget VALUES
```

```
(@projectNumber, USER_NAME(), GETDATE(), @budgetOld, @budgetNew)
```

```
END
```

В этом примере создается таблица AuditBudget, в которой сохраняются все изменения столбца Budget таблицы Project. Изменения этого столбца будут записываться в эту таблицу посредством триггера trigger_ModifyBudget.

Этот триггер активируется для каждого изменения столбца Budget с помощью инструкции UPDATE. При выполнении этого триггера значения строк таблиц deleted и inserted присваиваются соответствующим переменным @budgetOld, @budgetNew и @projectNumber. Эти присвоенные значения, совместно с именем пользователя и текущей датой, будут затем вставлены в таблицу AuditBudget.

В этом примере предполагается, что за один раз будет обновление только одной строки. Поэтому этот пример является упрощением общего случая, когда триггер обрабатывает многострочные обновления. Если выполнить следующие инструкции Transact-SQL:

```
USE SampleDb;
```

```
UPDATE Project
```

```
SET Budget = 200000
```

```
WHERE Number = 'p2';
```

то содержимое таблицы AuditBudget будет таким:

Results		Messages			
	ProjectNumber	UserName	Date	BudgetOld	BudgetNew
1	p2	dbo	2015-05-20 15:28:36.623	95000	200000

Реализация бизнес-правил

С помощью триггеров можно создавать бизнес-правила для приложений. Создание такого триггера показано в примере ниже:

```
USE SampleDb;
```

```
-- Триггер trigger_TotalBudget является примером использования
```

```
-- триггера для реализации бизнес-правила
```

```
GO
```

```
CREATE TRIGGER trigger_TotalBudget
```

```
    ON Project AFTER UPDATE
```

```
    AS IF UPDATE (Budget)
```

```
BEGIN
```

```
    DECLARE @sum_old1
```

```
    FLOAT DECLARE @sum_old2
```

```
    FLOAT DECLARE @sum_new FLOAT
```

```
    SELECT @sum_new = (SELECT SUM(Budget) FROM inserted)
```

```
    SELECT @sum_old1 = (SELECT SUM(p.Budget)
```

```
        FROM project p WHERE p.Number
```

```
        NOT IN (SELECT d.Number FROM deleted d))
```

```
    SELECT @sum_old2 = (SELECT SUM(Budget) FROM deleted)
```

```
    IF @sum_new > (@sum_old1 + @sum_old2) * 1.5
```

```
BEGIN
```

```
    PRINT 'Бюджет не изменился'
```

```

        ROLLBACK TRANSACTION

    END

    ELSE

        PRINT 'Изменение бюджета выполнено'

    END

```

Здесь создается правило для управления модификацией бюджетов проектов. Триггер trigger_TotalBudget проверяет каждое изменение бюджетов и выполняет только такие инструкции UPDATE, которые увеличивают сумму всех бюджетов не более чем на 50%. В противном случае для инструкции UPDATE выполняется откат посредством инструкции ROLLBACK TRANSACTION.

Принудительное обеспечение ограничений целостности

В системах управления базами данных применяются два типа ограничений для обеспечения целостности данных: декларативные ограничения, которые определяются с помощью инструкций языка CREATE TABLE и ALTER TABLE; процедурные ограничения целостности, которые реализуются посредством триггеров.

В обычных ситуациях следует использовать декларативные ограничения для обеспечения целостности, поскольку они поддерживаются системой и не требуют реализации пользователем. Применение триггеров рекомендуется только в тех случаях, для которых декларативные ограничения для обеспечения целостности отсутствуют.

В примере ниже показано принудительное обеспечение ссылочной целостности посредством триггеров для таблиц Employee и Works_on:

```

USE SampleDb;

GO

CREATE TRIGGER trigger_WorksonIntegrity

    ON Works_on AFTER INSERT, UPDATE

    AS IF UPDATE(EmpId)

    BEGIN

```

```

IF (SELECT Employee.Id

      FROM Employee, inserted

      WHERE Employee.Id = inserted.EmpId) IS NULL

BEGIN

    ROLLBACK TRANSACTION

    PRINT 'Строка не была вставлена/модифицирована'

END

ELSE

    PRINT 'Строка была вставлена/модифицирована'

END

```

Триггер trigger_WorksonIntegrity в этом примере проверяет ссылочную целостность для таблиц Employee и Works_on. Это означает, что проверяется каждое изменение столбца Id в ссылочной таблице Works_on, и при любом нарушении этого ограничения выполнение этой операции не допускается. (То же самое относится и к вставке в столбец Id новых значений.) Инструкция ROLLBACK TRANSACTION во втором блоке BEGIN выполняет откат инструкции INSERT или UPDATE в случае нарушения ограничения для обеспечения ссылочной целостности.

В этом примере триггер выполняет проверку на проблемы ссылочной целостности первого и второго случая между таблицами Employee и Works_on. А в примере ниже показан триггер, который выполняет проверку на проблемы ссылочной целостности третьего и четвертого случая между этими же таблицами (эти случаи обсуждались в статье ["Transact-SQL - создание таблиц"](#)):

```

USE SampleDb;

GO

CREATE TRIGGER trigger_RefintWorkson2

    ON Employee AFTER DELETE, UPDATE

    AS IF UPDATE (Id)

```

```

BEGIN

    IF (SELECT COUNT(*)

        FROM Works_on, deleted

        WHERE Works_on.EmpId = deleted.Id) > 0

    BEGIN

        ROLLBACK TRANSACTION

        PRINT 'Строка не была вставлена/модифицирована'

    END

    ELSE

        PRINT 'Строка была вставлена/модифицирована'

    END

```

Триггеры INSTEAD OF

Триггер с предложением **INSTEAD OF** заменяет соответствующее действие, которое запустило его. Этот триггер выполняется после создания соответствующих таблиц inserted и deleted, но перед выполнением проверки ограничений целостности или каких-либо других действий.

Триггеры INSTEAD OF можно создавать как для таблиц, так и для представлений. Когда инструкция Transact-SQL ссылается на представление, для которого определен триггер INSTEAD OF, система баз данных выполняет этот триггер вместо выполнения любых действий с любой таблицей. Данный тип триггера всегда использует информацию в таблицах inserted и deleted, созданных для представления, чтобы создать любые инструкции, требуемые для создания запрошенного события.

Значения столбцов, предоставляемые триггером INSTEAD OF, должны удовлетворять определенным требованиям:

- значения не могут задаваться для вычисляемых столбцов;
- значения не могут задаваться для столбцов с типом данных timestamp;
- значения не могут задаваться для столбцов со свойством IDENTITY, если только параметру IDENTITY_INSERT не присвоено значение ON.

Эти требования действительны только для инструкций INSERT и UPDATE, которые ссылаются на базовые таблицы. Инструкция INSERT, которая ссылается на представления с триггером INSTEAD OF, должна предоставлять значения для всех столбцов этого представления, не допускающих пустые значения NULL. (То же самое относится и к инструкции UPDATE. Инструкция UPDATE, ссылающаяся на представление с триггером INSTEAD OF, должна предоставить значения для всех столбцов представления, которое не допускает пустых значений и на которое осуществляется ссылка в предложении SET.)

В примере ниже показана разница в поведении при вставке значений в вычисляемые столбцы, используя таблицу и ее соответствующее представление:

```
USE SampleDb;
```

```
CREATE TABLE Orders (  
  
    OrderId INT NOT NULL,  
  
    Price MONEY NOT NULL,  
  
    Quantity INT NOT NULL,  
  
    OrderDate DATETIME NOT NULL,  
  
    Total AS Price * Quantity,  
  
    ShippedDate AS DATEADD (DAY, 7, orderdate)  
  
);
```

```
GO
```

```
CREATE VIEW view_AllOrders  
  
    AS SELECT *  
  
    FROM Orders;
```

```
GO
```

```
CREATE TRIGGER trigger_orders

ON view_AllOrders INSTEAD OF INSERT

AS BEGIN

    INSERT INTO Orders

    SELECT OrderId, Price, Quantity, OrderDate

    FROM inserted

END
```

В этом примере используется таблица Orders, содержащая два вычисляемых столбца. Представление view_AllOrders содержит все строки этой таблицы. Это представление используется для задания значения в его столбце, которое соотносится с вычисляемым столбцом в базовой таблице, на которой создано представление. Это позволяет использовать триггер INSTEAD OF, который в случае инструкции INSERT заменяется пакетом, который вставляет значения в базовую таблицу посредством представления view_AllOrders. (Инструкция INSERT, обращающаяся непосредственно к базовой таблице, не может задавать значение вычисляемому столбцу.)

Триггеры first и last

Компонент Database Engine позволяет создавать несколько триггеров для каждой таблицы или представления и для каждой операции (INSERT, UPDATE и DELETE) с ними. Кроме этого, можно указать порядок выполнения для нескольких триггеров, определенных для конкретной операции. С помощью системной процедуры **sp_settriggerorder** можно указать, что один из определенных для таблицы триггеров AFTER будет выполняться первым или последним для каждого обрабатываемого действия. Эта системная процедура имеет параметр @order, которому можно присвоить одно из трех значений:

- first - указывает, что триггер является первым триггером AFTER, выполняющимся для модифицирования действия;
- last - указывает, что данный триггер является последним триггером AFTER, выполняющимся для инициирования действия;
- none - указывает, что для триггера отсутствует какой-либо определенный порядок выполнения. (Это значение обычно используется для того, чтобы выполнить сброс ранее установленного порядка выполнения триггера как первого или последнего.)

Изменение структуры триггера посредством инструкции ALTER TRIGGER отменяет порядок выполнения триггера (первый или последний). Применение системной процедуры sp_settriggerorder показано в примере ниже:

```
USE SampleDb;
```

```
EXEC sp_settriggerorder @triggername = 'trigger_ModifyBudget',
```

```
@order = 'first', @stmttype='update'
```

Для таблицы разрешается определить только один первый и только один последний триггер AFTER. Остальные триггеры AFTER выполняются в неопределенном порядке. Узнать порядок выполнения триггера можно с помощью системной процедуры **sp_helptrigger** или функции OBJECTPROPERTY.

Возвращаемый системной процедурой sp_helptrigger результирующий набор содержит столбец order, в котором указывается порядок выполнения указанного триггера. При вызове функции objectproperty в ее втором параметре указывается значение ExeclsFirstTrigger или ExeclsLastTrigger, а в первом параметре всегда указывается идентификационный номер объекта базы данных. Если указанное во втором параметре свойство имеет значение true, функция возвращает значение 1.

Поскольку триггер INSTEAD OF исполняется перед тем, как выполняются изменения в его таблице, для триггеров этого типа нельзя указать порядок выполнения "первым" или "последним".

Триггеры DDL и области их применения

Ранее мы рассмотрели триггеры DML, которые задают действие, предпринимаемое сервером при изменении таблицы инструкциями INSERT, UPDATE или DELETE. Компонент Database Engine также позволяет определять триггеры для инструкций DDL, таких как CREATE DATABASE, DROP TABLE и ALTER TABLE. Триггеры для инструкций DDL имеют следующий синтаксис:

Соглашения по синтаксису

Как можно видеть по их синтаксису, триггеры DDL создаются таким же способом, как и триггеры DML. А для изменения и удаления этих триггеров используются те же инструкции ALTER TRIGGER и DROP TRIGGER, что и для триггеров DML. Поэтому в этом разделе рассматриваются только те параметры инструкции CREATE TRIGGER, которые новые для синтаксиса триггеров DDL.

Первым делом при определении триггера DDL нужно указать его область действия. *Предложение DATABASE* указывает в качестве области действия триггера DDL текущую базу данных, а *предложение ALL SERVER* - текущий сервер.

После указания области действия триггера DDL нужно в ответ на выполнение одной или нескольких инструкций DDL указать способ запуска триггера. В параметре `event_type` указывается инструкция DDL, выполнение которой запускает триггер, а в альтернативном параметре `event_group` указывается группа событий языка Transact-SQL. Триггер DDL запускается после выполнения любого события языка Transact-SQL, указанного в параметре `event_group`. Ключевое слово **LOGON** указывает триггер входа.

Кроме сходства триггеров DML и DDL, между ними также есть несколько различий. Основным различием между этими двумя видами триггеров является то, что для триггера DDL можно задать в качестве его области действия всю базу данных или даже весь сервер, а не всего лишь отдельный объект. Кроме этого, триггеры DDL не поддерживают триггеров **INSTEAD OF**. Как вы, возможно, уже догадались, для триггеров DDL не требуются таблицы `inserted` и `deleted`, поскольку эти триггеры не изменяют содержимого таблиц.

В следующих подразделах подробно рассматриваются две формы триггеров DDL: триггеры уровня базы данных и триггеры уровня сервера.

Триггеры DDL уровня базы данных

В примере ниже показано, как можно реализовать триггер DDL, чья область действия распространяется на текущую базу данных:

```
USE SampleDb;

GO

CREATE TRIGGER trigger_PreventDrop

    ON DATABASE FOR DROP_TRIGGER

    AS PRINT 'Перед тем, как удалить триггер, вы должны отключить "trigger_PreventDrop"'

ROLLBACK
```

Триггер в этом примере предотвращает удаление любого триггера для базы данных `SampleDb` любым пользователем. Предложение `DATABASE` указывает, что триггер `trigger_PreventDrop` является триггером уровня базы данных.

Ключевое слово **DROP_TRIGGER** указывает predetermined тип события, запрещающий удаление любого триггера.

Триггеры DDL уровня сервера

Триггеры уровня сервера реагируют на серверные события. Триггер уровня сервера создается посредством использования предложения ALL SERVER в инструкции CREATE TRIGGER. В зависимости от выполняемого триггером действия, существует два разных типа триггеров уровня сервера: обычные триггеры DDL и триггеры входа. Запуск обычных триггеров DDL основан на событиях инструкций DDL, а запуск триггеров входа - на событиях входа.

В примере ниже демонстрируется создание триггера уровня сервера, который является триггером входа:

```
USE master;

GO

CREATE LOGIN loginTest WITH PASSWORD = '12345!',

CHECK_EXPIRATION = ON;

GO

GRANT VIEW SERVER STATE TO loginTest;

GO

CREATE TRIGGER trigger_ConnectionLimit

ON ALL SERVER WITH EXECUTE AS 'loginTest'

FOR LOGON AS

BEGIN

    IF ORIGINAL_LOGIN()= 'loginTest' AND
```

```

        (SELECT COUNT(*) FROM sys.dm_exec_sessions

        WHERE is_user_process = 1 AND

        original_login_name = 'loginTest') > 1

    ROLLBACK;

END;

```

Здесь сначала создается имя входа SQL Server loginTest, которое потом используется в триггере уровня сервера. По этой причине, для этого имени входа требуется разрешение VIEW SERVER STATE, которое и предоставляется ему посредством инструкции GRANT. После этого создается триггер trigger_ConnectionLimit. Этот триггер является триггером входа, что указывается ключевым словом LOGON.

С помощью представления **sys.dm_exec_sessions** выполняется проверка, был ли уже установлен сеанс с использованием имени входа loginTest. Если сеанс уже был установлен, выполняется инструкция ROLLBACK. Таким образом имя входа loginTest может одновременно установить только один сеанс.

Триггеры и среда CLR

Подобно хранимым процедурам и определяемым пользователем функциям, триггеры можно реализовать, используя общезыковую среду выполнения (CLR - Common Language Runtime). Триггеры в среде CLR создаются в три этапа:

1. Создается исходный код триггера на языке C# или Visual Basic, который затем компилируется, используя соответствующий компилятор в объектный код.
2. Объектный код обрабатывается инструкцией CREATE ASSEMBLY, создавая соответствующий выполняемый файл.
3. Посредством инструкции CREATE TRIGGER создается триггер.

Выполнение всех этих трех этапов создания триггера CLR демонстрируется в последующих примерах. Ниже приводится пример исходного кода программы на языке C# для триггера из первого примера в статье. Прежде чем создавать триггер CLR в последующих примерах, сначала нужно удалить триггер trigger_PreventDrop, а затем удалить триггер trigger_ModifyBudget, используя в обоих случаях инструкцию DROP TRIGGER.

```

using System;

using System.Data.SqlClient;

using Microsoft.SqlServer.Server;

```

```
public class Triggers
{
    public static void ModifyBudget()
    {
        SqlTriggerContext context = SqlContext.TriggerContext;

        if (context.IsUpdatedColumn(2)) // Столбец Budget
        {
            float budget_old;

            float budget_new;

            string project_number;

            SqlConnection conn = new SqlConnection("context connection=true");

            conn.Open();

            SqlCommand cmd = conn.CreateCommand();

            cmd.CommandText = "SELECT Budget FROM DELETED";

            budget_old = (float)Convert.ToDouble(cmd.ExecuteScalar());

            cmd.CommandText = "SELECT Budget FROM INSERTED";
```

```

        budget_new = (float)Convert.ToDouble(cmd.ExecuteScalar());

        cmd.CommandText = "SELECT Number FROM DELETED";

        project_number = Convert.ToString(cmd.ExecuteScalar());

        cmd.CommandText = @"INSERT INTO AuditBudget
                                (@projectNumber, USER_NAME(), GETDATE(), @budgetOld,
                                @budgetNew)";

        cmd.Parameters.AddWithValue("@projectNumber", project_number);

        cmd.Parameters.AddWithValue("@budgetOld", budget_old);

        cmd.Parameters.AddWithValue("@budgetNew", budget_new);

        cmd.ExecuteNonQuery();

    }

}

}

```

Пространство имен Microsoft.SqlServer.Server содержит все классы клиентов, которые могут потребоваться программе C#.

Классы **SqlTriggerContext** и **SqlFunction** являются членами этого пространства имен. Кроме этого, пространство имен System.Data.SqlClient содержит классы SqlConnection и SqlCommand, которые используются для установления соединения и взаимодействия между клиентом и сервером базы данных. Соединение устанавливается, используя строку соединения "context connection = true".

Затем определяется класс Triggers, который применяется для реализации триггеров. Метод ModifyBudget() реализует одноименный триггер. Экземпляр context класса SqlTriggerContext позволяет программе получить доступ к

виртуальной таблице, создаваемой при выполнении триггера. В этой таблице сохраняются данные, вызвавшие срабатывание триггера. Метод `IsUpdatedColumn()` класса `SqlTriggerContext` позволяет узнать, был ли модифицирован указанный столбец таблицы.

Данная программа содержит два других важных класса: `SqlConnection` и `SqlCommand`. Экземпляр класса `SqlConnection` обычно применяется для установления соединения с базой данных, а экземпляр класса `SqlCommand` позволяет исполнять SQL-инструкции.

Программу из этого примера можно скомпилировать с помощью компилятора `csc`, который встроен в Visual Studio. Следующий шаг состоит в добавлении ссылки на скомпилированную сборку в базе данных:

```
USE SampleDb;

GO

CREATE ASSEMBLY CLRStoredProcedures

    FROM 'D:\Projects\CLRStoredProcedures\bin\Debug\CLRStoredProcedures.dll'

    WITH PERMISSION_SET = SAFE
```

Инструкция `CREATE ASSEMBLY` принимает в качестве ввода управляемый код и создает соответствующий объект, на основе которого создается триггер CLR. Предложение `WITH PERMISSION_SET` в примере указывает, что разрешениям доступа присвоено значение `SAFE`.

Наконец, в примере ниже посредством инструкции `CREATE TRIGGER` создается триггер `trigger_modify_budget`:

```
USE SampleDb;

GO

CREATE TRIGGER trigger_modify_budget ON Project

    AFTER UPDATE AS

    EXTERNAL NAME CLRStoredProcedures.Triggers.ModifyBudget
```

Инструкция CREATE TRIGGER в примере отличается от такой же инструкции в примерах ранее тем, что она содержит **параметр EXTERNAL NAME**. Этот параметр указывает, что код создается средой CLR. Имя в этом параметре состоит из трех частей. В первой части указывается имя соответствующей сборки (CLRStoredProcedures), во второй - имя открытого класса, определенного в примере выше (Triggers), а в третьей указывается имя метода, определенного в этом классе (ModifyBudget).

40)

В языках программирования обычно имеется два типа подпрограмм:

- хранимые процедуры;
- определяемые пользователем функции (UDF).

Как уже было рассмотрено в предыдущей статье, хранимые процедуры состоят из нескольких инструкций и имеют от нуля до нескольких входных параметров, но обычно не возвращают никаких параметров. В отличие от хранимых процедур, функции всегда возвращают одно значение. В этом разделе мы рассмотрим создание и использование *определяемых пользователем функций (User Defined Functions - UDF)*.

Создание и выполнение определяемых пользователем функций

Определяемые пользователем функции создаются посредством инструкции **CREATE FUNCTION**, которая имеет следующий синтаксис:

Соглашения по синтаксису

Параметр `schema_name` определяет имя схемы, которая назначается владельцем создаваемой UDF, а параметр `function_name` определяет имя этой функции. Параметр `@param` является входным параметром функции (формальным аргументом), чей тип данных определяется параметром `type`. Параметры функции - это значения, которые передаются вызывающим объектом определяемой пользователем функции для использования в ней. Параметр `default` определяет значение по умолчанию для соответствующего параметра функции. (Значением по умолчанию также может быть NULL.)

Предложение *RETURNS* определяет тип данных значения, возвращаемого UDF. Это может быть почти любой стандартный тип данных, поддерживаемый системой баз данных, включая тип данных `TABLE`. Единственным типом данных, который нельзя указывать, является тип данных `timestamp`.

Определяемые пользователем функции могут быть либо скалярными, либо табличными. Скалярные функции возвращают атомарное (скалярное) значение. Это означает, что в предложении *RETURNS* скалярной функции указывается один из стандартных типов данных. Функция является табличной, если предложение *RETURNS* возвращает набор строк.

Параметр *WITH ENCRYPTION* в системном каталоге кодирует информацию, содержащую текст инструкции **CREATE FUNCTION**. Таким образом, предотвращается несанкционированный просмотр текста, который был использован для создания функции. Данная опция позволяет повысить безопасность системы баз данных.

Альтернативное предложение *WITH SCHEMABINDING* привязывает UDF к объектам базы данных, к которым эта функция обращается. После этого любая попытка модифицировать объект базы данных, к которому обращается функция, претерпевает неудачу. (Привязка функции к объектам базы данных, к которым она обращается, удаляется только при изменении функции, после чего параметр SCHEMABINDING больше не задан.)

Для того чтобы во время создания функции использовать предложение SCHEMABINDING, объекты базы данных, к которым обращается функция, должны удовлетворять следующим условиям:

- все представления и другие UDF, к которым обращается определяемая функция, должны быть привязаны к схеме;
- все объекты базы данных (таблицы, представления и UDF) должны быть в той же самой базе данных, что и определяемая функция.

Параметр block определяет блок BEGIN/END, содержащий реализацию функции. Последней инструкцией блока должна быть инструкция RETURN с аргументом. (Значением аргумента является возвращаемое функцией значение.) Внутри блока BEGIN/END разрешаются только следующие инструкции:

- инструкции присвоения, такие как SET;
- инструкции для управления ходом выполнения, такие как WHILE и IF;
- инструкции DECLARE, объявляющие локальные переменные;
- инструкции SELECT, содержащие списки столбцов выборки с выражениями, значения которых присваиваются переменным, являющимися локальными для данной функции;
- инструкции INSERT, UPDATE и DELETE, которые изменяют переменные с типом данных TABLE, являющиеся локальными для данной функции.

По умолчанию инструкцию CREATE FUNCTION могут использовать только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db_owner или db_ddladmin. Но члены этих ролей могут присвоить это право другим пользователям с помощью инструкции GRANT CREATE FUNCTION.

В примере ниже показано создание функции ComputeCosts:

```
USE SampleDb;

-- Эта функция вычисляет возникающие дополнительные общие затраты,
-- при увеличении бюджетов проектов

GO
```

```

CREATE FUNCTION ComputeCosts (@percent INT = 10)

    RETURNS DECIMAL(16, 2)

    BEGIN

        DECLARE @addCosts DEC (14,2), @sumBudget DEC(16,2)

        SELECT @sumBudget = SUM (Budget) FROM Project

        SET @addCosts = @sumBudget * @percent/100

        RETURN @addCosts

    END;

```

Функция ComputeCosts вычисляет дополнительные расходы, возникающие при увеличении бюджетов проектов. Единственный входной параметр, @percent, определяет процентное значение увеличения бюджетов. В блоке BEGIN/END сначала объявляются две локальные переменные: @addCosts и @sumBudget, а затем с помощью инструкции SELECT переменной @sumBudget присваивается общая сумма всех бюджетов. После этого функция вычисляет общие дополнительные расходы и посредством инструкции RETURN возвращает это значение.

Вызов определяемой пользователем функции

Определенную пользователем функцию можно вызывать с помощью инструкций Transact-SQL, таких как SELECT, INSERT, UPDATE или DELETE. Вызов функции осуществляется, указывая ее имя с парой круглых скобок в конце, в которых можно задать один или несколько аргументов. Аргументы - это значения или выражения, которые передаются входным параметрам, определяемым сразу же после имени функции. При вызове функции, когда для ее параметров не определены значения по умолчанию, для всех этих параметров необходимо предоставить аргументы в том же самом порядке, в каком эти параметры определены в инструкции CREATE FUNCTION.

В примере ниже показан вызов функции ComputeCosts в инструкции SELECT:

```

USE SampleDb;

-- Вернет проект "p2 - Gemini"

SELECT Number, ProjectName

```

```
FROM Project
```

```
WHERE Budget < dbo.ComputeCosts(25);
```

Инструкция SELECT в примере отображает названия и номера всех проектов, бюджеты которых меньше, чем общие дополнительные расходы по всем проектам при заданном значении процентного увеличения.

В инструкциях Transact-SQL имена функций необходимо задавать, используя имена, состоящие из двух частей: schema name и function name, поэтому в примере мы использовали префикс схемы dbo.

Возвращающие табличное значение функции

Как уже упоминалось ранее, функция является возвращающей табличное значение, если ее предложение RETURNS возвращает набор строк. В зависимости от того, каким образом определено тело функции, возвращающие табличное значение функции классифицируются как *встраиваемые (inline)* и *многоинструкционные (multistatement)*. Если в предложении RETURNS ключевое слово TABLE указывается без сопровождающего списка столбцов, такая функция является встроенной. Инструкция SELECT встраиваемой функции возвращает результирующий набор в виде переменной с типом данных TABLE.

Многоинструкционная возвращающая табличное значение функция содержит имя, определяющее внутреннюю переменную с типом данных TABLE. Этот тип данных указывается **ключевым словом TABLE**, которое следует за именем переменной. В эту переменную вставляются выбранные строки, и она служит возвращаемым значением функции.

Создание возвращающей табличное значение функции показано в примере ниже:

```
USE SampleDb;
```

```
GO
```

```
CREATE FUNCTION EmployeesInProject (@projectNumber CHAR(4))
```

```
RETURNS TABLE
```

```
AS RETURN (SELECT FirstName, LastName
```

```
FROM Works_on, Employee
```

```
WHERE Employee.Id = Works_on.EmpId
```

```
AND ProjectNumber = @projectNumber)
```

Функция `EmployeesInProject` отображает имена всех сотрудников, работающих над определенным проектом, номер которого задается входным параметром `@projectNumber`. Тогда как функция в общем случае возвращает набор строк, предложение `RETURNS` в определении данной функции содержит ключевое слово `TABLE`, указывающее, что функция возвращает табличное значение. (Обратите внимание на то, что в примере блок `BEGIN/END` необходимо опустить, а предложение `RETURN` содержит инструкцию `SELECT`.)

Использование функции `Employees_in_Project` приведено в примере ниже:

```
USE SampleDb;

SELECT *

FROM EmployeesInProject('p3')
```

Результат выполнения:

Results		Messages	
	FirstName	LastName	
1	Анна	Иванова	
2	Василий	Фролов	
3	Елена	Лебедев	

Возвращающие табличное значение функции и инструкция `APPLY`

Реляционная инструкция `APPLY` позволяет вызывать возвращающую табличное значение функцию для каждой строки табличного выражения. Эта инструкция задается в предложении `FROM` соответствующей инструкции `SELECT` таким же образом, как и инструкция `JOIN`. Инструкция `APPLY` может быть объединена с табличной функцией для получения результата, похожего на результирующий набор операции соединения двух таблиц. Существует две формы инструкции `APPLY`:

- `CROSS APPLY`
- `OUTER APPLY`

Инструкция `CROSS APPLY` возвращает те строки из внутреннего (левого) табличного выражения, которые совпадают с внешним (правым) табличным выражением. Таким образом, логически, инструкция `CROSS APPLY` функционирует так же, как и инструкция `INNER JOIN`.

Инструкция *OUTER APPLY* возвращает все строки из внутреннего (левого) табличного выражения. (Для тех строк, для которых нет совпадений во внешнем табличном выражении, он содержит значения NULL в столбцах внешнего табличного выражения.) Логически, инструкция *OUTER APPLY* эквивалентна инструкции *LEFT OUTER JOIN*.

Применение инструкции *APPLY* показано в примерах ниже:

```
USE SampleDb;

GO

-- Создать функцию

CREATE FUNCTION GetJob(@empid AS INT)

    RETURNS TABLE AS

    RETURN

        SELECT Job

        FROM Works_on

        WHERE EmpId = @empid

        AND Job IS NOT NULL

        AND ProjectNumber = 'p1';
```

Функция *GetJob()* возвращает набор строк с таблицы *Works_on*. В примере ниже этот результирующий набор "соединяется" предложением *APPLY* с содержимым таблицы *Employee*:

```
USE SampleDb;

-- Используется CROSS APPLY

SELECT E.Id, FirstName, LastName, Job

    FROM Employee as E
```

```

CROSS APPLY GetJob(E.Id) AS A

-- Используется OUTER APPLY

SELECT E.Id, FirstName, LastName, Job

FROM Employee as E

OUTER APPLY GetJob(E.Id) AS A

```

Результатом выполнения этих двух функций будут следующие две таблицы (отображаются после выполнения второй функции):

Results		Messages		
	Id	FirstNa...	LastName	Job
1	10102	Анна	Иванова	Аналитик
2	9031	Елена	Лебеде	Менеджер
3	29346	Олег	Маменко	Консультант
	Id	FirstNa...	LastName	Job
1	2581	Василий	Фролов	NULL
2	9031	Елена	Лебеде	Менеджер
3	10102	Анна	Иванова	Аналитик
4	18316	Игорь	Соловьев	NULL
5	25348	Дмитрий	Волков	NULL
6	28559	Наталья	Вершини...	NULL
7	29346	Олег	Маменко	Консульт...

В первом запросе примера результирующий набор табличной функции GetJob() "соединяется" с содержимым таблицы Employee посредством инструкции CROSS APPLY. Функция GetJob() играет роль правого ввода, а таблица Employee - левого. Выражение правого ввода вычисляется для каждой строки левого ввода, а полученные строки комбинируются, создавая конечный результат.

Второй запрос похожий на первый (но в нем используется инструкция OUTER APPLY), который логически соответствует операции внешнего соединения двух таблиц.

Возвращающие табличное значение параметры

Во всех версиях сервера, предшествующих SQL Server 2008, задача передачи подпрограмме множественных параметров была сопряжена со значительными сложностями. Для этого сначала нужно было создать временную таблицу, вставить в нее передаваемые значения, и только затем можно было вызывать подпрограмму. Начиная с версии SQL Server 2008, эта задача упрощена, благодаря возможности использования

возвращающих табличное значение параметров, посредством которых результирующий набор может быть передан соответствующей подпрограмме.

Использование возвращающего табличное значение параметра показано в примере ниже:

```
USE SampleDb;

CREATE TYPE departmentType AS TABLE

    (Number CHAR(4), DepartmentName CHAR(40), Location CHAR(40));

GO

CREATE TABLE #moscowTable

    (Number CHAR(4), DepartmentName CHAR(40), Location CHAR(40));

GO

CREATE PROCEDURE InsertProc

    @Moscow departmentType READONLY

AS SET NOCOUNT ON

    INSERT INTO #moscowTable (Number, DepartmentName, Location)

    SELECT * FROM @Moscow

GO

DECLARE @Moscow AS departmentType;
```



```

INSERT INTO @Moscow (Number, DepartmentName, Location)

SELECT * FROM department

WHERE location = 'Москва';

EXEC InsertProc @Moscow;

```

В этом примере сначала определяется табличный тип departmentType. Это означает, что данный тип является типом данных TABLE, вследствие чего он разрешает вставку строк. В процедуре InsertProc объявляется переменная @Moscow с типом данных departmentType. (Предложение READONLY указывает, что содержимое этой таблицы нельзя изменять.) В последующем пакете в эту табличную переменную вставляются данные, после чего процедура запускается на выполнение. В процессе исполнения процедура вставляет строки из табличной переменной во временную таблицу #moscowTable. Вставленное содержимое временной таблицы выглядит следующим образом:

Results		Messages	
	Number	DepartmentName	Location
1	d1	Исследования	Москва
2	d3	Маркетинг	Москва

Использование возвращающих табличное значение параметров предоставляет следующие преимущества:

- упрощается модель программирования подпрограмм;
- уменьшается количество обращений к серверу и получений соответствующих ответов;
- таблица результата может иметь произвольное количество строк.

Изменение структуры определяемых пользователями инструкций

Язык Transact-SQL также поддерживает инструкцию **ALTER FUNCTION**, которая модифицирует структуру определяемых пользователями инструкций (UDF). Эта инструкция обычно используется для удаления привязки функции к схеме. Все параметры инструкции ALTER FUNCTION имеют такое же значение, как и одноименные параметры инструкции CREATE FUNCTION.

Для удаления UDF применяется инструкция **DROP FUNCTION**. Удалить функцию может только ее владелец или член предопределенной роли db_owner или sysadmin.

Определяемые пользователем функции и среда CLR

В предыдущей статье мы рассмотрели способ создания хранимых процедур из управляемого кода среды CLR на языке C#. Этот подход можно использовать и для определяемых пользователем функций (UDF), с одним только различием, что для сохранения UDF в виде объекта базы данных используется инструкция CREATE FUNCTION, а не CREATE PROCEDURE. Кроме этого, определяемые пользователем функции также применяются в другом контексте, чем хранимые процедуры, поскольку UDF всегда возвращают значение.

В примере ниже показан исходный код определяемых пользователем функций (UDF), реализованный на языке C#:

```
using System.Data.SqlTypes;

public class BudgetPercent
{
    private const float percent = 12;

    public static SqlDouble ComputeBudget(float budget)
    {
        return budget * percent;
    }
}
```

В исходном коде определяемых пользователем функций в примере вычисляется новый бюджет проекта, увеличивая старый бюджет на определенное количество процентов. Вы можете использовать инструкцию CREATE ASSEMBLY для создания сборки CLR в базе данных, как это было показано ранее. Если вы прорабатывали примеры из предыдущей статьи и уже добавили сборку CLRStoredProcedures в базу данных, то вы можете обновить эту сборку, после ее перекомпиляции с новым классом (CLRStoredProcedures это имя моего проекта классов C#, в котором я добавлял определение хранимых процедур и функций, у вас сборка может называться иначе):

```
USE SampleDb;
```

```
GO
```

```
ALTER ASSEMBLY CLRStoredProcedures
```

```
FROM 'D:\Projects\CLRStoredProcedures\bin\Debug\CLRStoredProcedures.dll'
```

```
WITH PERMISSION_SET = SAFE
```

Инструкция CREATE FUNCTION в примере ниже сохраняет метод ComputeBudget в виде объекта базы данных, который в дальнейшем можно использовать в инструкциях для манипулирования данными.

```
USE SampleDb;
```

```
GO
```

```
CREATE FUNCTION RecomputeBudget (@budget Real)
```

```
RETURNS FLOAT
```

```
AS EXTERNAL NAME CLRStoredProcedures.BudgetPercent.ComputeBudget
```

Использование одной из таких инструкций, инструкции SELECT, показано в примере ниже:

```
USE SampleDb;
```

```
-- Вернет 4098
```

```
SELECT dbo.RecomputeBudget (341.5);
```

Определяемую пользователем функцию можно поместить в разных местах инструкции SELECT. В примерах выше она вызывалась в предложениях WHERE, FROM и в списке выбора оператора SELECT.

