Программирование на Transact-SQL

Содержание

Введение

SQL (Structured Query Language) - это универсальный язык программирования, применяемый для создания, модификации и управления данными в реляционных базах данных (язык структурированных запросов).

SQL в его исходном виде является языком декларативного типа, но вместе SQL предусматривает возможность его процедурных расширений. В настоящее время широко распространены следующие спецификации SQL:

Базы данных и спецификации SQL	
Тип базы данных	Спецификация SQL
Microsoft SQL	Transact-SQL
Microsoft Jet/Access	Jet SQL
MySQL	SQL/PSM (SQL/Persistent Stored Module)
Oracle	PL/SQL (Procedural Language/SQL)
IBM DB2	SQL PL (SQL Procedural Language)
InterBase/Firebird	PSQL (Procedural SQL)

Далее будет рассмотрена спецификация Transact-SQL, которая используется СУБД Microsoft SQL Server. А так как база у всех спецификаций SQL одинаковая, то большинство команд и сценариев с легкостью переносятся на другие типы SQL.

Итак, Transact-SQL – это процедурное расширение языка SQL компаний Microsoft. SQL был расширен такими дополнительными возможностями как:

- управляющие операторы,
- локальные и глобальные переменные,
- различные дополнительные функции для обработки строк, дат, математики и т.п.,
- поддержка аутентификации Microsoft Windows

Язык Transact-SQL является ключом к использованию SQL Server. Все приложения, взаимодействующие с экземпляром SQL Server, независимо от их реализации и пользовательского интерфейса, отправляют серверу инструкции Transact-SQL.

к содержанию

Пример создания базы данных и таблиц

Для того, чтобы усвоить теоретический материал, его, конечно же, нужно применить на практике. Для практических занятий создадим базу данных и заполним ее небольшим количеством значений.

Итак, чтобы создать базу данных и заполнить ее значениями, необходимо открыть консоль выполнения команд и запросов SQL сервера и выполнить следующий сценарий:

```
-- Создание базы данных
USE master
CREATE DATABASE TestDatabase
GO
-- Создание таблиц
USE TestDatabase
CREATE TABLE Users
     (UserID int PRIMARY KEY,
      UserName nvarchar(40),
      UserSurname nvarchar(40),
      DepartmentID int,
      PositionID int)
CREATE TABLE Departments
     (DepartmentID int PRIMARY KEY,
      DepartmentName nvarchar(40))
CREATE TABLE Positions
     (PositionID int PRIMARY KEY,
      PositionName nvarchar(40),
      BaseSalary money)
CREATE TABLE [Local Customers]
     (CustomerID int PRIMARY KEY,
      CustomerName nvarchar(40),
      CustomerAddress nvarchar(255))
CREATE TABLE [Local Orders]
     (OrderID int PRIMARY KEY,
      CustomerID int,
      UserID int,
     [Description] text)
GO
-- Заполнение таблиц
USE TestDatabase
INSERT Users VALUES (1, 'Ivan', 'Petrov', 1, 1)
INSERT Users VALUES (2, 'Ivan', 'Sidorov', 1, 2)
INSERT Users VALUES (3, 'Petr', 'Ivanov', 1, 2)
INSERT Users VALUES (4, 'Nikolay', 'Petrov', 1, 3)
INSERT Users VALUES (5, 'Nikolay', 'Ivanov', 2, 1)
INSERT Users VALUES (6, 'Sergey', 'Sidorov', 2, 3)
INSERT Users VALUES (7, 'Andrey', 'Bukin', 2, 2)
INSERT Users VALUES (8, 'Viktor', 'Rybakov', 4, 1)
```

```
INSERT Departments VALUES (1, 'Production')
INSERT Departments VALUES (2, 'Distribution')
INSERT Departments VALUES (3, 'Purchasing')
INSERT Positions VALUES (1, 'Manager', 1000)
INSERT Positions VALUES (2, 'Senior analyst', 650)
INSERT [Local Customers] VALUES (1, 'Alex Company', '606443, Russia, Bor,
Lenina str., 15')
INSERT [Local Customers] VALUES (2, 'Potrovka', '115516, Moscow,
Promyshlennaya str., 1')
INSERT [Local Orders] VALUES (1, 1, 1, 'Special parts')
GO
```

Примечание. В Microsoft SQL Server 2000 запросы выполняются в приложении Query Analyzer. В Microsoft SQL Server 2005 запросы выполняются в SQL Server Management Studio.

В результате работы сценария на SQL сервере будет создана база данных TestDatabase с пятью пользовательскими таблицами: Users, Departments, Positions, Local Customers, Local Orders.

Users				
UserID	**UserName**	**UserSurname**	**DepartmentID**	**PositionID**
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
3	Petr	Ivanov	1	3
4	Nikolay	Petrov	1	3
5	Nikolay	Ivanov	2	1
6	Sergey	Sidorov	2	3
7	Andrey	Bukin	2	3
8	Viktor	Rybakov	4	1

Positions		
PositionID	**PositionName**	**BaseSalary**
1	Manager	1000
2	Senior analyst	650
3	Analyst	400

Local Orders			
OrderID	**CustomerID**	**UserID**	**Description**
1	1	1	Special parts

Departments	
DepartmentID	DepartmentName

1	Production
2	Distribution
3	Purchasing

Local Customers		
CustomerID	CustomerName	CustomerAddress
1	Alex Company	606443, Russia, Bor, Lenina str., 15
2	Potrovka	115516, Moscow, Promyshlennaya str., 1

к содержанию

Элементы синтаксиса

Здесь рассмотрим основные элементы синтаксиса языка Transact-SQL, которыми являются: директивы сценария, комментарии, типы данных, идентификаторы, переменные, операторы, системные функции и команды.

Директивы сценария

Директивы сценария – это специфические команды, которые используются только в MS SQL. Эти команды помогают серверу определять правила работы со скриптом и транзакциями. Типичные представители: GO – сигнализирует SQL-серверу об окончании сценария, EXEC (или EXECUTE) – выполняет процедуру или скалярную функцию.

Комментарии

Комментарии используются для создания пояснений для блоков сценариев, а также для временного отключения команд при отладке скрипта. Комментарии бывают как строковыми, так и блоковыми:

- -- - строковый комментарий исключает из выполнения только одну строку, перед которой стоят два минуса.
- /* */ - блоковый комментарий исключает из выполнения целый блок команд, заключенный в указанную конструкцию.

Типы данных

Как и в языках программирования, в SQL существуют различные типы данных для хранения переменных:

- Числа - для хранения числовых переменных (int, tinyint, smallint, bigint, numeric, decimal, money, smallmoney, float, real).
- Даты - для хранения даты и времени (datetime, smalldatetime).
- Символы - для хранения символьных данных (char, nchar, varchar, nvarchar).
- Двоичные - для хранения бинарных данных (binary, varbinary, bit).
- Большеобъемные - типы данных для хранения больших бинарных данных (text, ntext, image).
- Специальные - указатели (cursor), 16-байтовое шестнадцатиричное число, которое используется для GUID (uniqueidentifier), штамп изменения строки (timestamp), версия строки (rowversion), таблицы (table).

Примечание. Для использования русских символов (не ASCII кодировки) испольюзуются типы данных с приставкой "n" (nchar, nvarchar, ntext), которые кодируют символы двумя байтами. Иначе говоря, для работы с Unicode используются типы данных с "n".

Примечание. Для данных переменной длины используются типы данных с приставкой "var". Типы данных без приставки "var" имеют фиксированную длину области памяти, неиспользованная часть которой заполняется пробелами или нулями.

Идентификаторы

Идентификаторы - это специальные символы, которые используются с переменными для идентифицирования их типа или для группировки слов в переменную. Типы идентификаторов:

- @ - идентификатор локальной переменной (пользовательской).
- @@ - идентификатор глобальной переменной (встроенной).
- # - идентификатор локальной таблицы или процедуры.
- ## - идентификатор глобальной таблицы или процедуры.
- [ ] - идентификатор группировки слов в переменную.

Переменные

Переменные используются в сценариях и для хранения временных данных. Чтобы работать с переменной, ее нужно объявить, притом объявление должно быть осуществлено в той транзакции, в которой выполняется команда, использующая эту переменную. Иначе говоря, после завершения транзакции, то есть после команды GO, переменная уничтожается.

Объявление переменной выполняется командой DECLARE, задание значения переменной осуществляется либо командой SET, либо SELECT:

```
USE TestDatabase
-- Объявление переменных
DECLARE @EmpID int, @EmpName varchar(40)
-- Задание значения переменной @EmpID
SET @EmpID = 1
-- Задание значения переменной @EmpName
SELECT @EmpName = UserName FROM Users WHERE UserID = @EmpID
-- Вывод переменной @EmpName в результат запроса
SELECT @EmpName AS [Employee Name]
GO
```

Примечание. В этом примере используется группировка слов в переменную - конструкция [Employee Name] воспринимается как одна переменная, так как слова заключены в квадратные скобки.

Операторы

Операторы - это специальные команды, предназначенные для выполнения простых операций над переменными:

- Арифметические операторы: "*" – умножить, "/" – делить, "%" – модуль от деления, "+" – сложить, "-" – вычесть, "()" – скобки.
- Операторы сравнения: "=" – равно, ">" – больше, "<" – меньше, ">=" – больше или равно, "<=" – меньше или равно, "<>" – не равно.
- Операторы соединения: "+" – соединение строк.
- Логические операторы: "AND" – и, "OR" – или, "NOT" – не.

Системные функции

Спецификация Transact-SQl значительно расширяет стандартные возможности SQL благодаря системным или встроенным функциям, которые делятся на агрегатные, скалярные и функции-указатели:

- Агрегатные функции – функции, которые работают с коллекциями значений и выдают одно значение. Типичные представители: AVG – среднее значение колонки, SUM – сумма колонки, MAX – максимальное значение колонки, COUNT - количество элементов колонки.
- Скалярные функции – это функции, которые возвращают одно значение, работая со скалярными данными или вообще без входных данных. Типичные представители: DATEDIFF - разница между датами, ABS - модуль числа, DB_NAME - имя базы данных, USER_NAME - имя текущего пользователя, LEFT – часть строки слева.
- Функции-указатели – функции, которые используются как ссылки на другие данные. Типичные представители: OPENXML – указатель на источник данных в виде XML-структуры, OPENQUERY – указатель на источник данных в виде другого запроса.

Примечание 1. Полный список функций можно найти в справке к SQL серверу.

Примечание 2. К скалярным функциям можно также отнести и глобальные переменные, которые в тексте сценария вызываются двойной собакой "@@".

Пример:

```
USE TestDatabase
-- Использование агрегатной функции для подсчета средней зарплаты
SELECT AVG(BaseSalary) AS [Average salary] FROM Positions
GO
-- Использование скалярной функции для получения имени базы данных
SELECT DB_NAME() AS [Database name]
GO
-- Использование скалярной функции для получения имени текущего пользователя
```

```
DECLARE @MyUser char(30)
SET @MyUser = USER_NAME()
SELECT 'The current user''s database username is: '+ @MyUser
GO
-- Использование функции-указателя для получения данных с другого сервера
SELECT * FROM OPENQUERY(OracleSvr, 'SELECT name, id FROM owner.titles')
GO
```

Команды

Команда – это комбинация символов и операторов, которая получает на вход скалярную величину, а на выходе дает другую величину или исполняет какое-то действие. В Transact-SQL команды делятся на 3 типа: DDL, DCL и DML.

- DDL (Data Definition Language) – используются для создания объектов в базе данных. Основные представители данного класса: CREATE – создание объектов, ALTER – изменение объектов, DROP – удаление объектов.
- DCL (Data Control Language) – предназначены для назначения прав на объекты базы данных. Основные представители данного класса: GRANT – разрешение на объект, DENY – запрет на объект, REVOKE – отмена разрешений и запретов на объект.
- DML (Data Manipulation Language) – используются для запросов и изменения данных. Основные представители данного класса: SELECT – выборка данных, INSERT – вставка данных, UPDATE – изменение данных, DELETE – удаление данных.

Пример:

```
USE TestDatabase
-- Использование DDL
CREATE TABLE TempUsers (UserID int, UserName nvarchar(40), DepartmentID int)
GO
-- Использование DCL
GRANT SELECT ON Users TO public
GO
-- Использование DML
SELECT UserID, UserName + ' ' + UserSurname AS [User Full Name] FROM Users
GO
-- Использование DDL
DROP TABLE TempUsers
GO
```

Управление выполнением сценария

В Transact-SQL существуют специальные команды, которые позволяют управлять потоком выполнения сценария, прерывая его или направляя в нужную логику.

- Блок группировки - структура, объединяющая список выражений в один логический блок (BEGIN … END).
- Блок условия - структура, проверяющая выполнения определенного условия (IF … ELSE).
- Блок цикла - структура, организующая повторение выполнения логического блока (WHILE … BREAK … CONTINUE).
- Переход - команда, выполняющая переход потока выполнения сценария на указанную метку (GOTO).
- Задержка - команда, задерживающая выполнение сценария (WAITFOR)
- Вызов ошибки - команда, генерирующая ошибку выполнения сценария (RAISERROR)

к содержанию

Динамическое конструирование команд

Итак, поняв основы Transact-SQL и попрактиковавшись на простых примерах, можно перейти к более сложным структурам. Обычно базы данных создаются и заполняются с помощью сценариев (скриптов) – хотя визуальный редактор прост в обращении, но им никогда быстро и без недочетов не создашь большую базу данных и не заполнишь ее данными. Если вспомнить начало, то опытная база данных как раз создавалась и заполнялась с помощью сценария. Сценарий – это одно или более команд, объединенных в логический блок, которые автоматизируют работу администратора.

Обычно сценарии пишутся как универсальное средство для выполнения стандартных задач, поэтому в них применяется динамическое конструирование логики – в запросы и команды вставляются переменные, а не конкретные названия объектов, что позволяет быстро изменять параметры скрипта.

Пример:

```
USE master
-- Задание динамических данных
DECLARE @dbname varchar(30), @tablename varchar(30), @column varchar(30)
SET @dbname = 'TestDatabase'
SET @tablename = 'Positions'
SET @column = 'BaseSalary'
-- Использование динамических данных
EXECUTE ('USE ' + @dbname + ' SELECT AVG(' + @column + ')
 AS [Average salary] FROM ' + @tablename)
GO
```

к содержанию

Выборка данных

В языках SQL выборка данных из таблиц осуществляется с помощью команды SELECT:

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <название таблицы>
```

По умолчанию в команде SELECT используется параметр ALL, который можно не указывать. Если в команде указать параметр DISTINCT, то в результат попадут только уникальные (неповторяющиеся) записи из выборки.

Для того, чтобы изменить имена объектов в командах к SQL-серверу, используется команда AS. Использование этой команды помогает сокращать длину строки запроса, а так же получать результат в более удобочитаемом виде.

Пример:

```
-- Выбрать все записи из таблицы Local Customers
SELECT * FROM [Local Customers]
```

CustomerID	CustomerName	CustomerAddress
1	Alex Company	606443, Russia, Bor, Lenina str., 15')
2	Potrovka	115516, Moscow, Promyshlennaya str., 1

```
-- Выбрать записи колонки DepartmentName из таблицы Departments
-- и назвать результирующую колонку Department Name
SELECT DepartmentName AS 'Department Name' FROM Departments
```

Department Name
Production
Distribution
Purchasing

```
-- Выбрать уникальные записи колонки UserName из таблицы Users
SELECT DISTINCT UserName FROM Users
```

UserName
Andrey
Ivan
Nikolay
Petr
Sergey
Viktor

Фильтрация данных осуществляется с помощью команды WHERE, в которой используются следующие операторы и команды сравнения: =, <, >, <=, >=, <>, LIKE, NOT

LIKE, AND, OR, NOT, BETWEEN, NOT BETWEEN, IN, NOT IN, IS NULL, IS NOT NULL.
В общем виде команда SELECT с фильтром выглядит так:

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <название таблицы>
 WHERE <условие>
```

В строке сравнения разрешается использовать подстановочные символы:

- % - любое количество символов;
- _ - один символ;
- [] - любой символ, указанный в скобках;
- [^] - любой символ, не указанный в скобках.

```
-- Выбрать все записи из таблицы Users, где DepartmentID = 1
SELECT * FROM Users WHERE DepartmentID = 1
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
3	Petr	Ivanov	1	2
4	Nikolay	Petrov	1	3

```
-- Выбрать все записи из таблицы Users, у кого в имени есть буква A
SELECT * FROM Users WHERE UserName LIKE '%a%'
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
4	Nikolay	Petrov	1	3
5	Nikolay	Ivanov	2	1
7	Andrey	Bukin	2	2

```
-- Выбрать все записи из таблицы Users, у кого в имени вторая буква не V
SELECT * FROM Users WHERE UserName LIKE '_[^v]%'
```

UserID	UserName	UserSurname	DepartmentID	PositionID
3	Petr	Ivanov	1	2
4	Nikolay	Petrov	1	3
5	Nikolay	Ivanov	2	1
6	Sergey	Sidorov	2	3
7	Andrey	Bukin	2	2
8	Viktor	Rybakov	4	1

```
-- Выбрать записи колонок UserName и UserSurname из таблицы Users, у кого
-- PositionID между 2 и 3, результирующую колонку назвать Full name.
SELECT [UserName] + ' ' + [UserSurname] AS 'Full name' FROM Users
 WHERE PositionID BETWEEN 2 AND 3
```

Full name
Ivan Sidorov
Petr Ivanov
Nikolay Petrov
Sergey Sidorov
Andrey Bukin

Фильтрация позволяет использовать подзапросы, то есть конструировать запрос из нескольких подзапросов:

```
-- Выбрать записи колонки PositionID из таблицы Positions, где BaseSalary < 600
SELECT PositionID FROM Positions WHERE BaseSalary < 600
```

PositionID
3

```
-- Выбрать все записи из таблицы Users, у кого оклад не меньше 600
SELECT * FROM Users WHERE PositionID NOT IN
 (SELECT PositionID FROM Positions WHERE BaseSalary < 600)
```

UserID	UserName	UserSurname	DepartmentID	PositionID
4	Nikolay	Petrov	1	3
6	Sergey	Sidorov	2	3

```
-- Выбрать все записи из таблицы Users, у кого имя Ivan или Andrey
SELECT * FROM Users WHERE UserName IN ('Ivan', 'Andrey')
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
7	Andrey	Bukin	2	2

```
-- Вычислить суммарную зарплату отдела с идентификатором 1
DECLARE @DepID int
SET @DepID = 1
SELECT DepartmentName AS 'Department name',
   (SELECT SUM(Positions.BaseSalary) FROM Positions
      INNER JOIN Users ON Users.PositionID = Positions.PositionID
      WHERE Users.DepartmentID = @DepID
   ) AS 'Summary salary'
 FROM Departments
 WHERE DepartmentID = @DepID
```

Department name	Summary salary
Production	2700.0000

Для сортировки данных в выборке используется команда ORDER BY, но следует учесть, что эта команда не сортирует данные типа text, ntext и image. По умолчанию сортировка производится по возрастанию, поэтому параметр ASC в этом случае можно не указывать:

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <название таблицы>
 WHERE <условие> ORDER BY <названия колонок> [ASC или DESC]
```

Для того, чтобы ограничить количество строк в результате запроса, используется команда TOP:

```
SELECT [ALL или DISTINCT] TOP [количество строк] <названия колонок или *>
 FROM <название таблицы> WHERE <условие>
 ORDER BY <названия колонок> [ASC или DESC]
```

Внутри запроса можно проводить вычисления над полученными данными. Для этого используются функции агрегирования:

- AVG(колонка) - среднее значение колонки;
- COUNT(колонка) - количество не NULL элементов колонки;
- COUNT(*) - количество элементов запроса;
- MAX(колонка) - максимальное значение в колонке;
- MIN(колонка) - минимальное значение в колонке;
- SUM(колонка) - сумма значений в колонке.

Примеры использования команд ORDER, TOP и функций агрегирования:

```
-- Выбрать 3 первые уникальные записи колонки UserName из таблицы Users,
-- отсортированных по возрастанию UserName
SELECT DISTINCT TOP 3 UserName FROM Users ORDER BY UserName
```

UserName
Andrey
Ivan
Nikolay

```
-- Выбрать 15 процентов строк из таблицы Users
SELECT TOP 15 PERCENT * FROM Users
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2

```
-- Найти величину максимального оклада в организации
SELECT MAX(BaseSalary) FROM Positions
```

(No column name)
1000.0000

```
-- Найти должности, у которых максимальный оклад в организации
        SELECT * FROM Positions
         WHERE BaseSalary IN (SELECT MAX(BaseSalary) FROM Positions)
```

PositionID	PositionName	BaseSalary
1	Manager	1000.0000

```
-- Найти сотрудников, у кого максимальный оклад в организации
SELECT * FROM Users
 WHERE PositionID IN (SELECT PositionID FROM Positions
 WHERE BaseSalary IN (SELECT MAX(BaseSalary) FROM Positions))
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
5	Nikolay	Ivanov	2	1
8	Viktor	Rybakov	4	1

```
-- Найти количество сотрудников, у кого максимальный оклад в организации
SELECT COUNT(*) FROM Users
 WHERE PositionID IN (SELECT PositionID FROM Positions
 WHERE BaseSalary IN (SELECT MAX(BaseSalary) FROM Positions))
```

(No column name)
3

к содержанию

Группировка данных

SQL позволяет производить группировку данных по определенным полям таблицы. Чтобы сгруппировать данные по какому-нибудь параметру, в SQL-запросе необходимо написать команду GROUP BY, в которой указать имя колонки, по которой производится группировка. Колонки, упомянутые в команде GROUP BY, должны присутствовать в команде SELECT, а так же команда SELECT должна содержать функцию агрегирования, которая будет применена к сгруппированным данным.

```
-- Найти количество работников в каждом отделе (сгруппировать работников по
-- идентификатору отделов и сосчитать количество записей в каждой группе)
SELECT DepartmentID, COUNT(UserID) AS 'Number of users'
 FROM Users GROUP BY DepartmentID
```

DepartmentID	Number of users
1	4
2	3
4	1

Чтобы отфильтровать строки в запросе с группировкой применяется специальная команда HAVING, в которой указывается условие фильтрации. Колонки, по которым производится фильтрация, должны присутствовать в команде GROUP BY. Команда HAVING может использоваться и без GROUP BY, в этом случае она работает аналогично команде WHERE, но она разрешает применять в условиях фильтрации только функции агрегирования.

```
-- Найти количество работников в первом отделе (сгруппировать работников по
-- идентификатору отделов, сосчитать количество записей в каждой группе и
-- вывести в результат только отдел с идентификатором равным 1)
SELECT DepartmentID, COUNT(UserID) AS 'Number of users'
 FROM Users GROUP BY DepartmentID HAVING DepartmentID = 1
```

DepartmentID	Number of users
1	4

Команда группировки может дополняться оператором WITH ROLLUP, который дополняет результат группировки сводной строкой с суммой значений колонок.

```
-- Найти количество работников в каждом отделе (сгруппировать работников по
-- идентификатору отделов и сосчитать количество записей в каждой группе),
-- а также сосчитать общее количество работников
SELECT DepartmentID, COUNT(UserID) AS 'Number of users'
 FROM Users GROUP BY DepartmentID WITH ROLLUP
```

DepartmentID	Number of users
1	4
2	3
4	1
NULL	8

```
-- Найти количество работников с определенной должностью в каждом отделе
-- (сгруппировать работников по идентификатору должностей и отделов и
-- сосчитать количество записей в каждой группе), а также сосчитать
-- количество работников в каждом отделе и общее количество работников
SELECT DepartmentID, PositionID, COUNT(UserID) AS 'Number of users'
 FROM Users GROUP BY DepartmentID, PositionID WITH ROLLUP
```

DepartmentID	PositionID	Number of users
1	1	1
1	2	2
1	3	1
1	NULL	4
2	1	1
2	2	1
2	3	1
2	NULL	3
4	1	1
4	NULL	1
NULL	NULL	8

Команда группировки также может дополняться оператором WITH CUBE, который формирует всевозможные комбинации из группируемых колонок: если есть N колонок, то получится 2^N комбинаций.

```
-- Найти количество работников с определенной должностью в каждом отделе
-- (сгруппировать работников по идентификатору должностей и отделов и
-- сосчитать количество записей в каждой группе), а также сосчитать
-- количество работников по каждой должности, по каждому отделу и
-- общее количество работников
SELECT DepartmentID, PositionID, COUNT(UserID) AS 'Number of users'
 FROM Users GROUP BY DepartmentID, PositionID WITH CUBE
```

DepartmentID	PositionID	Number of users
1	1	1
1	2	2
1	3	1
1	NULL	4
2	1	1
2	2	1
2	3	1
2	NULL	3
4	1	1
4	NULL	1

NULL	NULL	8
NULL	1	3
NULL	2	3
NULL	3	2

Функция агрегирования GROUPING позволяет определить, была ли запись добавлена командами ROLLUP и CUBE, или это запись получена из источника данных.

```
-- Найти количество работников в каждом отделе (сгруппировать работников по
-- идентификатору отделов и сосчитать количество записей в каждой группе)
-- а так же пометить дополнительные строки, несуществующие в источнике данных
SELECT DepartmentID, COUNT(UserID) AS 'Number of users',
 GROUPING(DepartmentID) AS 'Added row'
 FROM Users GROUP BY DepartmentID WITH ROLLUP
```

DepartmentID	Number of users	Added row
1	4	0
2	3	0
4	1	0
NULL	8	1

Еще одна команда группировки COMPUTE позволяет группировать данные и выводить по ним отчет в разные таблицы. То есть команда GROUP BY с операторами ROLLUP и CUBE группирует данные и дописывает в таблицу дополнительные строки с отчетом, а команда COMPUTE группирует данные, разрывая исходную таблицу на несколько подтаблиц, а также формирует подтаблицы с отчетами. Команда COMPUTE может использоваться в двух режимах:

- как простая функция агрегирования, выводящая результат в отдельную таблицу;
- с параметром BY как команда группировки, разрезающая таблицу на несколько подтаблиц

Команда COMPUTE с параметром BY может использоваться только совместно с командой ORDER BY, причем столбцы сортировки должны совпадать со столбцами группировки.

```
-- Вывести таблицу пользователей компании, а также посчитать их количество
        SELECT * FROM Users COMPUTE COUNT(UserID)
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
3	Petr	Ivanov	1	2
4	Nikolay	Petrov	1	3
5	Nikolay	Ivanov	2	1
6	Sergey	Sidorov	2	3
7	Andrey	Bukin	2	2

8	Viktor	Rybakov	4	1

cnt
8

```
-- Найти количество работников в каждом отделе (сгруппировать работников по
-- идентификатору отделов и сосчитать количество записей в каждой группе)
SELECT * FROM Users ORDER BY DepartmentID
     COMPUTE COUNT(UserID) BY DepartmentID
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
3	Petr	Ivanov	1	2
4	Nikolay	Petrov	1	3

cnt
4

UserID	UserName	UserSurname	DepartmentID	PositionID
5	Nikolay	Ivanov	2	1
6	Sergey	Sidorov	2	3
7	Andrey	Bukin	2	2

cnt
3

UserID	UserName	UserSurname	DepartmentID	PositionID
8	Viktor	Rybakov	4	1

cnt
1

к содержанию

Соединение и объединение таблиц

Самые важные и нужные запросы в SQL - это с запросы с соединением таблиц, когда выборка осуществляется сразу из нескольких источников. Такие запросы более сложны в написании, но и более удобны в обработке, так как часто выдают в программу уже готовый результат, который остается только вывести на экран. Соединять таблицы в SQL можно двумя способами: вертикально и горизонтально.

Вертикальное соединение называется *объединением таблиц* и осуществляется командой UNION, которая в конец первой таблицы допишет вторую таблицу. При объединении количество колонок объединяемых таблиц должно быть одинаковым, а сами колонки должны иметь одинаковые названия и типы данных. При этом одинаковые строки, встречающиеся в обеих таблицах, будут удалены, если в команде не указан параметр ALL.

```
-- Найти всех пользователей с именем Ivan и соединить результат с
-- результатом от запроса "Найти всех пользователей с фамилией Petrov"
-- дублирующие записи исключить
SELECT * FROM Users WHERE UserName = 'Ivan'
UNION
SELECT * FROM Users WHERE UserSurname = 'Petrov'
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
4	Nikolay	Petrov	1	3

```
-- Найти всех пользователей с именем Ivan и соединить результат с
-- результатом от запроса "Найти всех пользователей с фамилией Petrov"
-- дублирующие записи сохранить
SELECT * FROM Users WHERE UserName = 'Ivan'
UNION ALL
SELECT * FROM Users WHERE UserSurname = 'Petrov'
```

UserID	UserName	UserSurname	DepartmentID	PositionID
1	Ivan	Petrov	1	1
2	Ivan	Sidorov	1	2
1	Ivan	Petrov	1	1
4	Nikolay	Petrov	1	3

Горизонтальное соединение или просто *соединение* производится путем сцепки нескольких таблиц по ключевым колонкам. Самое простое соединение выполняется с помощью команды INNER JOIN, которая сцепляет таблицы, выбирая строки по ключевому полю, которое встречается в обеих таблицах. Такое соединение называется *внутренним*.

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <таблица_1>
         INNER JOIN таблица_2 ON таблица_1.ключевое_поле =
таблица_2.ключевое_поле
```

Чтобы выполнить сцепление по всем полям левой таблицы, независимо, есть ли такие записи в правой таблице, необходимо использовать команду LEFT JOIN. Эта команда соединяет таблицы, выбирая все строки из левой таблицы, а отсутствующие данные правой таблицы заполняются значением NULL. Такое соединение называется *левым внешним соединением*.

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <таблица_1>
 LEFT JOIN таблица_2 ON таблица_1.ключевое_поле = таблица_2.ключевое_поле
```

Команда RIGHT JOIN аналогична предыдущей, разница заключается лишь в том, что она соединяет таблицы, выбирая все строки из правой таблицы, а отсутствующие данные левой таблицы заполняются значением NULL. Такое соединение называется *правым внешним соединением*.

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <таблица_1>
 RIGHT JOIN таблица_2 ON таблица_1.ключевое_поле = таблица_2.ключевое_поле
```

Команда FULL JOIN объединяет в себе левое и правое соединение, то есть она соединяет таблицы, выбирая строки из обеих таблиц, а отсутствующие данные заполняются значением NULL. Такое соединение называется *полным внешним соединением*.

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <таблица_1>
 FULL JOIN таблица_2 ON таблица_1.ключевое_поле = таблица_2.ключевое_поле
```

Последняя и редко используемая команда соединения таблиц – это CROSS JOIN. Эта команда сцепляет таблицы без использования ключевого поля, а результат - это комбинация из всевозможных строк исходных таблиц. Такое соединение называется *перекрестным соединением* или *декартовым произведением таблиц*.

```
SELECT [ALL или DISTINCT] <названия колонок или *> FROM <таблица_1>
 CROSS JOIN таблица_2
```

Сцепление не ограничивается только двумя таблицами, запрос может содержать несколько команда JOIN, что очень удобно при формировании конечных отчетов. Ниже приведены примеры для всех команд соединения таблиц.

```
SELECT * FROM Users INNER JOIN Departments
     ON Users.DepartmentID = Departments.DepartmentID
```

UserID	UserName	UserSurname	DepartmentID	PositionID	DepartmentID	DepartmentName
1	Ivan	Petrov	1	1	1	Production
2	Ivan	Sidorov	1	2	1	Production
3	Petr	Ivanov	1	2	1	Production

4	Nikolay	Petrov	1	3	1	Production
5	Nikolay	Ivanov	2	1	2	Distribution
6	Sergey	Sidorov	2	3	2	Distribution
7	Andrey	Bukin	2	2	2	Distribution

```
SELECT * FROM Users LEFT JOIN Departments ON Users.DepartmentID =
Departments.DepartmentID
```

UserID	UserName	UserSurname	DepartmentID	PositionID	DepartmentID	DepartmentName
1	Ivan	Petrov	1	1	1	Production
2	Ivan	Sidorov	1	2	1	Production
3	Petr	Ivanov	1	2	1	Production
4	Nikolay	Petrov	1	3	1	Production
5	Nikolay	Ivanov	2	1	2	Distribution
6	Sergey	Sidorov	2	3	2	Distribution
7	Andrey	Bukin	2	2	2	Distribution
8	Viktor	Rybakov	4	1	NULL	NULL

```
SELECT * FROM Users RIGHT JOIN Departments ON Users.DepartmentID =
Departments.DepartmentID
```

UserID	UserName	UserSurname	DepartmentID	PositionID	DepartmentID	DepartmentName
1	Ivan	Petrov	1	1	1	Production
2	Ivan	Sidorov	1	2	1	Production
3	Petr	Ivanov	1	2	1	Production
4	Nikolay	Petrov	1	3	1	Production
5	Nikolay	Ivanov	2	1	2	Distribution
6	Sergey	Sidorov	2	3	2	Distribution
7	Andrey	Bukin	2	2	2	Distribution
NULL	NULL	NULL	NULL	NULL	3	Purchasing

```
SELECT * FROM Users FULL JOIN Departments ON Users.DepartmentID =
Departments.DepartmentID
```

UserID	UserName	UserSurname	DepartmentID	PositionID	DepartmentID	DepartmentName
1	Ivan	Petrov	1	1	1	Production
2	Ivan	Sidorov	1	2	1	Production
3	Petr	Ivanov	1	2	1	Production
4	Nikolay	Petrov	1	3	1	Production
5	Nikolay	Ivanov	2	1	2	Distribution
6	Sergey	Sidorov	2	3	2	Distribution
7	Andrey	Bukin	2	2	2	Distribution
NULL	NULL	NULL	NULL	NULL	3	Purchasing
8	Viktor	Rybakov	4	1	NULL	NULL

```
SELECT * FROM Users CROSS JOIN Departments
```

UserID	UserName	UserSurname	DepartmentID	PositionID	DepartmentID	DepartmentName
1	Ivan	Petrov	1	1	1	Production

2	Ivan	Sidorov	1	2	1	Production
3	Petr	Ivanov	1	2	1	Production
4	Nikolay	Petrov	1	3	1	Production
5	Nikolay	Ivanov	2	1	1	Production
6	Sergey	Sidorov	2	3	1	Production
7	Andrey	Bukin	2	2	1	Production
8	Viktor	Rybakov	4	1	1	Production
1	Ivan	Petrov	1	1	2	Distribution
2	Ivan	Sidorov	1	2	2	Distribution
3	Petr	Ivanov	1	2	2	Distribution
4	Nikolay	Petrov	1	3	2	Distribution
5	Nikolay	Ivanov	2	1	2	Distribution
6	Sergey	Sidorov	2	3	2	Distribution
7	Andrey	Bukin	2	2	2	Distribution
8	Viktor	Rybakov	4	1	2	Distribution
1	Ivan	Petrov	1	1	3	Purchasing
2	Ivan	Sidorov	1	2	3	Purchasing
3	Petr	Ivanov	1	2	3	Purchasing
4	Nikolay	Petrov	1	3	3	Purchasing
5	Nikolay	Ivanov	2	1	3	Purchasing
6	Sergey	Sidorov	2	3	3	Purchasing
7	Andrey	Bukin	2	2	3	Purchasing
8	Viktor	Rybakov	4	1	3	Purchasing

```
SELECT dpt.DepartmentName AS 'Department',
 usr.UserName + ' ' + usr.UserSurname AS 'User name',
 pos.PositionName AS 'Position'
 FROM Users AS usr
 LEFT JOIN Departments AS dpt ON usr.DepartmentID = dpt.DepartmentID
 LEFT JOIN Positions AS pos ON usr.PositionID = pos.PositionID
 ORDER BY dpt.DepartmentID, pos.PositionID
```

Department	User name	Position
NULL	Viktor Rybakov	Manager
Production	Ivan Petrov	Manager
Production	Ivan Sidorov	Senior analyst
Production	Petr Ivanov	Senior analyst
Production	Nikolay Petrov	Analyst
Distribution	Nikolay Ivanov	Manager
Distribution	Andrey Bukin	Senior analyst
Distribution	Sergey Sidorov	Analyst

к содержанию

Изменение данных и транзакции

Прежде, чем рассказывать о командах изменения данных, нужно пояснить особенность диалекта Transact-SQL. Как видно из самого названия, этот механизм основан на транзакциях, то есть на последовательности операций, объединенных в один логический модуль, будь то запрос на выборку данных, изменения данных или структуры таблиц. На время транзакции все используемые в сценарии данные блокируются, что позволяет избежать несоответствия данных во время начала работы с таблицей и завершением сценария.

За транзакции в Transact-SQL отвечает структура BEGIN TRANSACTION ... COMMIT TRANSACTION. Эту структуру использовать необязательно, но тогда все команды сценария являются необратимыми, то есть нельзя сделать "откат" к предыдущему состоянию. Полная структура блока транзакций:

```
BEGIN TRANSACTION [имя транзакции]
 [операции]
COMMIT TRANSACTION [имя транзакции] или ROLLBACK TRANSACTION [имя транзакции]
```

Ниже приведен пример использования этого блока:

```
-- Установить всем сотрудникам новый оклад
BEGIN TRANSACTION TR1
 UPDATE Positions SET BaseSalary = 2500000000000000
 IF @@ERROR <> 0
   BEGIN
     RAISERROR('Error, transaction not completed!',16,-1)
     ROLLBACK TRANSACTION TR1
   END
 ELSE
   COMMIT TRANSACTION TR1
```

Для вставки данных в таблицы SQL-сервера используется команда INSERT INTO:

```
INSERT INTO [название таблицы] (колонки) VALUES ([значения колонок])
```

Вторая часть команды является необязательной для MS SQL Server 2003, но MS JET SQL без этого слова будет выдавать ошибку синтаксиса. Вставка обычно производиться целой строкой, то есть в команде указываются все колонки таблицы и значения, которые нужно в них занести. Если же колонка имеет значение по умолчанию или разрешает пустое значения, то в команде вставки эту колонку можно не указывать. Команда INSERT INTO также разрешает указывать вносимые данные не по порядку следования колонок, но в этом случае нужно обозначить используемый порядок колонок.

```
-- В таблицу Users вставить строку с данными UserID = 9,
-- UserName = 'Nikolay',
-- UserSurname = 'Gryzlov', DepartmentID = 4, PositionID = 2.
INSERT INTO Users VALUES (9, 'Nikolay', 'Gryzlov', 4, 2)
-- В таблицу Users вставить строку с данными UserID = 10,
-- UserName = 'Nikolay', UserSurname = 'Kozin',
-- DepartmentID - значение по умолчанию, PositionID - не указано.
INSERT Users VALUES (10, 'Nikolay', 'Kozin', DEFAULT, NULL)
-- В таблицу Users вставить строку с данными UserName = 'Angrey',
-- UserSurname = 'Medvedev', UserID = 11, остальные значения по умолчанию
INSERT INTO Users (UserName, UserSurname, UserID)
VALUES ('Angrey', 'Medvedev', 11)
```

Для того, чтобы изменить значение ячейки таблицы, используется команда UPDATE:

```
UPDATE [название таблицы] SET [имя колонки]=[значение колонок]
WHERE [условие]
```

Обновление (изменение) значений в таблице можно производить безусловно, с условием или с выборкой данных из другой таблицы.

```
-- Установить всем должностям зарплату в 2000 единиц.
UPDATE Positions SET BaseSalary = 2000
-- Должностям с идентификатором 1 установить зарплату в 2500 единиц.
UPDATE Positions SET BaseSalary = 2500 WHERE PositionID = 1
-- Должностям с идентификатором 2 уменьшить зарплату на 30%.
UPDATE Positions SET BaseSalary = BaseSalary * 0.7 WHERE PositionID = 2
-- Установить всем должностям зарплату, равную (30 000 разделить на
количество
-- сотрудников в организации)
UPDATE Positions SET BaseSalary = 30000 / (SELECT COUNT(UserID) FROM Users)
```

Удаление данных производится командой DELETE:

```
DELETE FROM [название таблицы] WHERE [условие]
```

Удаление данных обычно производится по какому-то критерию. Так как удаление данных – это достаточно опасная операция, то перед выполнением такой команды лучше всего произвести тестовую выборку командой SELECT, которая выведет в результат те данные, которые будут стерты. Если это то, что требуется, тогда можно смело заменять SELECT на DELETE и выполнять удаление данных.

```
-- Удалить пользователя с идентификатором 10
-- В режиме отладки рекомендуется использовать команду SELECT,
-- чтобы знать, какие данные будут удалены:
-- SELECT UserID FROM Users WHERE UserID = 10
DELETE FROM Users WHERE UserID = 10
```

```
-- Удалить всех пользователей отдела Production
DELETE Users FROM Users INNER JOIN Departments
ON Users.DepartmentID = Departments.DepartmentID
WHERE Departments.DepartmentName = 'Production'
-- Удалить всех пользователей
DELETE FROM Users
```

Примечание! В примере для фильтрации данных применено сцепление таблиц. Хотя в команде перечисляются несколько таблиц, удаление данных будет произведено только из той таблицы, которая указана после слова DELETE.

Более быстрая команда для очистки таблицы – это TRUNCATE TABLE.

```
TRUNCATE TABLE [название таблицы]
```

Пример удаления всех данных:

```
-- Очистить таблицу Users
TRUNCATE TABLE Users
```

Transact-SQL позволяет использовать временные таблицы, то есть таблицы, которые создаются в памяти сервера на время работы пользователя с базой данных. Временные таблицы могут иметь любое имя, но начинаться обязаны с символа #.

```
-- Создать временную таблицу #TempTable, в которую скопировать содержание
-- колонки UserName таблицы Users
SELECT UserName INTO #TempTable FROM Users
-- Выбрать все записи временной таблицы #TempTable
SELECT * FROM #TempTable
```

к содержанию

Хранимые процедуры и функции

Хранимые процедуры и функции представляют собой набор SQL-операторов, которые можно сохранять на сервере. Если сценарий сохранен на сервере, то клиентам не придется повторно задавать одни и те же отдельные операторы, вместо этого они смогут обращаться к хранимой процедуре. Вот несколько ситуаций, когда хранимые процедуры особенно полезны:

- Многочисленные клиентские приложения написаны на разных языках или работают на различных платформах, но должны выполнять одинаковые операции с базами данных.
- Безопасность играет первостепенную роль. Хранимые процедуры используются для всех стандартных операций, что обеспечивает совместимость и безопасность среды, а процедуры гарантируют надлежащую регистрацию каждой операции. При таком типе установки приложения и пользователи не получают непосредственный доступ к таблицам базы данных и могут выполнять только конкретные хранимые процедуры.
- Необходимо снизить сетевой трафик между клиентом и сервером. Объем пересылаемой информации между сервером и клиентом существенно снижается, но увеличивается нагрузка на систему сервера баз данных, так как в этом случае на стороне сервера выполняется большая часть работы по обработке данных.

Пример создания хранимой процедуры и хранимой функции:

```
-- Создание функции обновления зарплат
CREATE PROCEDURE usp_UpdateSalary AS
   UPDATE Positions SET BaseSalary = 2000
GO

-- Создание функции получения имени пользователя
CREATE FUNCTION usf_GetName (@UserID int) RETURNS varchar(255)
BEGIN
 IF @UserID IS NULL
   SET @UserID = 1
 RETURN (SELECT UserName + ' ' + UserSurname FROM Users
         WHERE UserID = @UserID)
END
GO

-- Обновление зарплат
EXEC TestDatabase.dbo.usp_UpdateSalary

-- Получение имени пользователя с идентификатором 2
SELECT TestDatabase.dbo.usf_GetName(2)
```

Итак, хранимые процедуры и функции дают следующие преимущества:

- производительность;
- общая логика для всех запросов;
- уменьшение трафика;
- безопасность – доступ пользователю дается не к таблице, а к процедуре.

Триггеры

Триггеры – это специальный вид хранимых процедур, вызов которых производится автоматически при наступлении некоторого события в базе данных, например вставки, удаления или обновления строк в определенной таблице. Триггеры обычно используют, чтобы реализовать сложную бизнес логику, которую трудно или невозможно реализовать при использовании других механизмов поддержки целостности данных, например, таких как ограничения.

Далее демонстрируется создание триггера **insrtWorkOrder**, срабатывающего после вставки строк в таблицу **Production.WorkOrder** базы данных **AdventureWorks**. Заметим, что таблица **inserted** – это временная таблица, создаваемая сервером автоматически, которая хранит вставляемые строки. Этот триггер сохраняет только что добавленные строки в другой таблице **Production.TransactionHistory**.

```
CREATE TRIGGER [insrtWorkOrder] ON [Production].[WorkOrder]
AFTER INSERT AS
BEGIN
  SET NOCOUNT ON;
  INSERT INTO [Production].[TransactionHistory](
     [ProductID],[ReferenceOrderID],[TransactionType]
    ,[TransactionDate],[Quantity],[ActualCost])
     SELECT inserted.[ProductID],inserted.[WorkOrderID]
     ,'W',GETDATE(),inserted.[OrderQty],0
     FROM inserted;
END;
```

к содержанию

Производительность

Для увеличения производительности, то есть для быстрого выполнения запросов, следует помнить некоторые правила составления строк запросов:

- Избегать NOT – команды отрицания выполняются в несколько этапов, что увеличивает нагрузку на сервер.
- Избегать LIKE – этот оператор сравнения применяет более мягкие шаблоны сравнения, чем оператор =, что увеличивает необходимое число этапов фильтрации.
- Применять точные шаблоны поиска – применение подстановочных символов увеличивает время выполнения запроса, так как для проверки всех вариантов подстановки требуется дополнительные ресурсы сервера.
- Избегать ORDER BY – команда сортировки требует упорядочивания строк таблицы вывода, что задерживает получение результата.