Язык SQL подразделяется на несколько частей, здесь я рассмотрю 2 наиболее важные его части:

**DDL – Data Definition Language (язык описания данных)**

В частности, к этому типу мы можем отнести следующие команды:

* **CREATE**: создает объекты базы данных (саму базу данных, таблицы, индексы и т.д.)
* **ALTER**: изменяет объекты базы данных
* **DROP**: удаляет объекты базы данных
* **TRUNCATE**: удаляет все данные из таблиц

**DML – Data Manipulation Language (язык манипулирования данными),** который содержит следующие конструкции:

* **SELECT**: извлекает данные из БД
* **UPDATE**: обновляет данные
* **INSERT**: добавляет новые данные
* **DELETE**: удаляет данные

**MERGE** – слияние данных

* **DCL** (**Data Control Language** / Язык управления доступа к данным). К этому типу относят команды, которые управляют правами по доступу к данным. В частности, это следующие команды:
  + **GRANT**: предоставляет права для доступа к данным
  + **REVOKE**: отзывает права на доступ к данным

**SQL Server Management Studio (SSMS)** — утилита для Microsoft SQL Server для конфигурирования, управления и администрирования компонентов базы данных.

Данная утилита содержит редактор скриптов (который в основном и будет нами использоваться) и графическую программу, которая работает с объектами и

настройками сервера. Главным инструментом SQL Server Management Studio является Object Explorer, который позволяет пользователю просматривать,

извлекать объекты сервера, а также управлять ими.

**SQL** — язык позволяющий осуществлять запросы в БД посредством СУБД. В конкретной СУБД, язык SQL может иметь специфичную реализацию (свой диалект).

**DDL и DML** — подмножество языка SQL:

Язык DDL служит для создания и модификации структуры БД, т.е. для создания/изменения/удаления таблиц и связей.

Язык DML позволяет осуществлять манипуляции с данными таблиц, т.е. с ее строками. Он позволяет делать выборку данных из таблиц, добавлять новые данные в

таблицы, а так же обновлять и удалять существующие данные.

**-- однострочный комментарий**

**и**

**/\***

**многострочный**

**комментарий**

**\*/**

**DDL – Data Definition Language** (язык описания данных)

Для организации баз данных MS SQL Server использует **реляционную модель.**

Реляционная модель предполагает хранение данных в виде таблиц, каждая из которых состоит из строк и столбцов. Каждая строка хранит отдельный объект, а в столбцах размещаются атрибуты этого объекта.

Для идентификации каждой строки в рамках таблицы применяется первичный ключ (primary key)

Через ключи одна таблица может быть связана с другой, то есть между двумя таблицами могут быть организованы связи. А сама таблица может быть представлена в виде отношения ("relation").

API расшифровывается как «Application Programming Interface» (интерфейс программирования приложений, программный интерфейс приложения)

## Работа с MySQL через phpMyAdmin

Еще помимо страшной черной консоли есть удобный веб-интерфейс, который предоставляется программой phpMyAdmin. С запущенным OpenServer перейдите по пути:

<http://127.0.0.1/openserver/phpmyadmin/index.php>

Базу данных часто отождествляют с набором таблиц, которые хранят данные. Но это не совсем так. Лучше сказать, что база данных представляет хранилище объектов. Основные из них:

* **Таблицы**: хранят собственно данные
* **Представления** (Views): выражения языка SQL, которые возвращают набор данных в виде таблицы
* **Хранимые процедуры**: выполняют код на языке SQL по отношению к данным к БД (например, получает данные или изменяет их)
* **Функции**: также код SQL, который выполняет определенную задачу

### Системные базы данных

В MS SQL Server по умолчанию создается четыре системных баз данных:

* **master**: эта главная база данных сервера, в случае ее отсутствия или повреждения сервер не сможет работать. Она хранит все используемые логины пользователей сервера, их роли, различные конфигурационные настройки, имена и информацию о базах данных, которые хранятся на сервере, а также ряд другой информации.
* **model**: эта база данных представляет шаблон, на основе которого создаются другие базы данных. То есть когда мы создаем через SSMS свою бд, она создается как копия базы model.
* **msdb**: хранит информацию о работе, выполняемой таким компонентом как планировщик SQL. Также она хранит информацию о бекапах баз данных.
* **tempdb**: эта база данных используется как хранилище для временных объектов. Она заново пересоздается при каждом запуске сервера.

Мы можем заметить, что название таблицы на самом деле начинается с префикса **dbo**. Этот префикс представляет схему. Схема определяет контейнер, который хранит объекты. То есть схема логически разграничивает базы данных. Если схема явным образом не указывается при создании объекта, то объект принадлежит схеме по умолчанию - схеме **dbo**.

В качестве каталога для базы данных лучше использовать каталог, где хранятся остальные базы данных сервера. На Windows 10 по умолчанию это каталог

C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\DATA

Одна база данных может содержать до 2 миллиардов таблиц.

Общий синтаксис создания таблицы выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE TABLE название\_таблицы  (название\_столбца1 тип\_данных атрибуты\_столбца1,   название\_столбца2 тип\_данных атрибуты\_столбца2,   ................................................   название\_столбцаN тип\_данных атрибуты\_столбцаN,   атрибуты\_таблицы  ) |

Имя объекта не может включать пробелы и не может представлять одно из ключевых слов языка Transact-SQL. Если идентификатор все же содержит пробельные символы, то его следует заключать в кавычки. Если необходимо в качестве имени использовать ключевые слова, то эти слова помещаются в квадратные скобки.

**nchar** [(n)]  
Строковые данные фиксированного размера. n определяет размер строки в парах байтов и должен быть значением от 1 до 4000. Размер хранилища составляет два раза n байтов.

**nvarchar** [(n | **max** )]  
Строковые данные переменного размера. n определяет размер строки в парах байтов и может принимать значения от 1 до 4000. **max** указывает, что максимальный размер хранилища составляет 2 ^ 30-1 символов (2 ГБ). Размер хранилища составляет два раза n байтов + 2 байта.

Если n не указано в операторе определения данных или объявления переменной, длина по умолчанию равна 1. Если n не указано с функцией CAST, длина по умолчанию равна 30.

### Переименование таблицы

Для переименования таблиц применяется системная хранимая процедура "sp\_rename". Например, переименование таблицы Users в UserAccounts в базе данных usersdb:

|  |  |
| --- | --- |
| 1  2 | USE usersdb;  EXEC sp\_rename 'Users', 'UserAccounts'; |

#### 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 c 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 (мантиссы). По умолчанию n = 53.

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

#### 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.

#### 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 столце 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.

#### инарные 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.

## Атрибуты и ограничения столбцов и таблиц

### PRIMARY KEY

С помощью выражения **PRIMARY KEY** столбец можно сделать первичным ключом.

### IDENTITY

Атрибут **IDENTITY** позволяет сделать столбец идентификатором. Этот атрибут может назначаться для столбцов числовых типов INT, SMALLINT, BIGINT, TYNIINT, DECIMAL и NUMERIC.

Также можно использовать полную форму атрибута:

|  |  |
| --- | --- |
| 1 | IDENTITY(seed, increment) |

### UNIQUE

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

### 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.

### 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.

### 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:

### Оператор CONSTRAINT. Установка имени ограничений.

С помощью ключевого слова **CONSTRAINT** можно задать имя для ограничений. В качестве ограничений могут использоваться PRIMARY KEY, UNIQUE, DEFAULT, CHECK.

Имена ограничений можно задать на уровне столбцов. Они указываются после CONSTRAINT перед атрибутами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | CREATE TABLE Customers  (      Id INT CONSTRAINT PK\_Customer\_Id PRIMARY KEY IDENTITY,      Age INT          CONSTRAINT DF\_Customer\_Age DEFAULT 18          CONSTRAINT CK\_Customer\_Age CHECK(Age >0 AND Age < 100),      FirstName NVARCHAR(20) NOT NULL,      LastName NVARCHAR(20) NOT NULL,      Email VARCHAR(30) CONSTRAINT UQ\_Customer\_Email UNIQUE,      Phone VARCHAR(20) CONSTRAINT UQ\_Customer\_Phone UNIQUE  ) |

Ограничения могут носить произвольные названия, но, как правило, для применяются следующие префиксы:

* "PK\_" - для PRIMARY KEY
* "FK\_" - для FOREIGN KEY
* "CK\_" - для CHECK
* "UQ\_" - для UNIQUE
* "DF\_" - для DEFAULT

**Внешние ключи** применяются для установки связи между таблицами. Внешний ключ устанавливается для столбцов из зависимой, подчиненной таблицы, и указывает на один из столбцов из главной таблицы. Хотя, как правило, внешний ключ указывает на первичный ключ из связанной главной таблицы, но это необязательно должно быть непременным условием. Внешний ключ также может указывать на какой-то другой столбец, который имеет уникальное значение.

Общий синтаксис установки внешнего ключа на уровне столбца:

|  |  |
| --- | --- |
| 1  2  3 | [FOREIGN KEY] REFERENCES главная\_таблица (столбец\_главной\_таблицы)      [ON DELETE {CASCADE|NO ACTION}]      [ON UPDATE {CASCADE|NO ACTION}] |

С помощью оператора **CONSTRAINT** можно задать имя для ограничения внешнего ключа. Обычно это имя начинается с префикса "FK\_"

### ON DELETE и ON UPDATE

С помощью выражений **ON DELETE** и **ON UPDATE** можно установить действия, которые выполняться соответственно при удалении и изменении связанной строки из главной таблицы. И для определения действия мы можем использовать следующие опции:

* **CASCADE**: автоматически удаляет или изменяет строки из зависимой таблицы при удалении или изменении связанных строк в главной таблице.
* **NO ACTION**: предотвращает какие-либо действия в зависимой таблице при удалении или изменении связанных строк в главной таблице. То есть фактически какие-либо действия отсутствуют.
* **SET NULL**: при удалении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение NULL.
* **SET DEFAULT**: при удалении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение по умолчанию, которое задается с помощью атрибуты DEFAULT. Если для столбца не задано значение по умолчанию, то в качестве него применяется значение NULL.
* CREATE TABLE Orders
* (
* Id INT PRIMARY KEY IDENTITY,
* CustomerId INT,
* CreatedAt Date,
* FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE SET DEFAULT
* )

Для изменения таблиц используется выражение **ALTER TABLE**.

Общий формальный синтаксис команды выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ALTER TABLE название\_таблицы [WITH CHECK | WITH NOCHECK]  { ADD название\_столбца тип\_данных\_столбца [атрибуты\_столбца] |    DROP COLUMN название\_столбца |    ALTER COLUMN название\_столбца тип\_данных\_столбца [NULL|NOT NULL] |    ADD [CONSTRAINT] определение\_ограничения |    DROP [CONSTRAINT] имя\_ограничения} |

### Добавление нового столбца

Добавим в таблицу Customers новый столбец Address:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Customers  ADD Address NVARCHAR(50) NULL; |

В данном случае столбец Address имеет тип NVARCHAR и для него определен атрибут NULL. Но что если нам надо добавить столбец, который не должен принимать значения NULL? Если в таблице есть данные, то следующая команда не будет выполнена:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Customers  ADD Address NVARCHAR(50) NOT NULL; |

Поэтому в данном случае решение состоит в установке значения по умолчанию через атрибут DEFAULT:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Customers  ADD Address NVARCHAR(50) NOT NULL DEFAULT 'Неизвестно'; |

В этом случае, если в таблице уже есть данные, то для них для столбца Address будет добавлено значение "Неизвестно".

### Удаление столбца

Удалим столбец Address из таблицы Customers:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Customers  DROP COLUMN Address; |

### Изменение типа столбца

Изменим в таблице Customers тип данных у столбца FirstName на NVARCHAR(200):

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Customers  ALTER COLUMN FirstName NVARCHAR(200); |

### Добавление ограничения CHECK

При добавлении ограничений SQL Server автоматически проверяет имеющиеся данные на соответствие добавляемым ограничениям. Если данные не соответствуют ограничениям, то такие ограничения не будут добавлены. Например, установим для столбца Age в таблице Customers ограничение Age > 21.

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Customers  ADD CHECK (Age > 21); |

Если в таблице есть строки, в которых в столбце Age есть значения, несоответствующие этому ограничению, то sql-команда завершится с ошибкой. Чтобы избежать подобной проверки на соответствие и все таки добавить ограничение, несмотря на наличие несоответствующих ему данных, используется выражение **WITH NOCHECK**:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Customers WITH NOCHECK  ADD CHECK (Age > 21); |

По умолчанию используется значение **WITH CHECK**, которое проверяет на соответствие ограничениям.

### Добавление внешнего ключа

Пусть изначально в базе данных будут добавлены две таблицы, никак не связанные:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | CREATE TABLE Customers  (      Id INT PRIMARY KEY IDENTITY,      Age INT DEFAULT 18,      FirstName NVARCHAR(20) NOT NULL,      LastName NVARCHAR(20) NOT NULL,      Email VARCHAR(30) UNIQUE,      Phone VARCHAR(20) UNIQUE  );  CREATE TABLE Orders  (      Id INT IDENTITY,      CustomerId INT,      CreatedAt Date  ); |

Добавим ограничение внешнего ключа к столбцу CustomerId таблицы Orders:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Orders  ADD FOREIGN KEY(CustomerId) REFERENCES Customers(Id); |

### Добавление первичного ключа

Используя выше определенную таблицу Orders, добавим к ней первичный ключ для столбца Id:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Orders  ADD PRIMARY KEY (Id); |

### Добавление ограничений с именами

При добавлении ограничений мы можем указать для них имя, используя оператор **CONSTRAINT**, после которого указывается имя ограничения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ALTER TABLE Orders  ADD CONSTRAINT PK\_Orders\_Id PRIMARY KEY (Id),      CONSTRAINT FK\_Orders\_To\_Customers FOREIGN KEY(CustomerId) REFERENCES Customers(Id);    ALTER TABLE Customers  ADD CONSTRAINT CK\_Age\_Greater\_Than\_Zero CHECK (Age > 0); |

### Удаление ограничений

Для удаления ограничений необходимо знать их имя. Если мы точно не знаем имя ограничения, то его можно узнать через SQL Server Management Studio:

Раскрыв узел таблиц в подузле Keys можно увидеть названия ограничений первичного и внешних ключей. Названия ограничений внешних ключей начинаются с "FK". А в подузле Constraints можно найти все ограничения CHECK и DEFAULT. Названия ограничений CHECK начинаются с "CK", а ограничений DEFAULT - с "DF".

Например, как видно на скриншоте в моем случае имя ограничения внешнего ключа в таблице Orders называется "FK\_Orders\_To\_Customers". Поэтому для удаления внешнего ключа я могу использовать следующее выражение:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE Orders  DROP FK\_Orders\_To\_Customers; |

## Пакеты. Команда GO

В предыдущих случаях сначала создавалась база данных, а затем в эту БД добавлялась таблица с помощью отдельных команд SQL. Но можно сразу совместить в одном скрипте несколько команд. В этом случае отдельные наборы команд называются **пакетами** (batch).

Каждый пакет состоит из одного или нескольких SQL-выражений, которые выполняются как оно целое. В качестве сигнала завершения пакета и выполнения его выражений служит команда **GO**.

Смысл разделения SQL-выражений на пакеты состоит в том, что одни выражения должны успешно выполниться до запуска других выражений. Например, при добавлении таблиц мы должны бы уверены, что была создана база данных, в которой мы собираемся создать таблицы.

Например, определим следующий скрипт:

|  |
| --- |
| CREATE DATABASE internetstore;  GO    USE internetstore;    CREATE TABLE Customers  (      Id INT PRIMARY KEY IDENTITY,      Age INT DEFAULT 18,      FirstName NVARCHAR(20) NOT NULL,      LastName NVARCHAR(20) NOT NULL,      Email VARCHAR(30) UNIQUE,      Phone VARCHAR(20) UNIQUE  );    CREATE TABLE Orders  (      Id INT PRIMARY KEY IDENTITY,      CustomerId INT,      CreatedAt DATE,      FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE CASCADE  ); |

Вначале создается бд internetstore. Затем идет команда GO, которая сигнализирует, что можно выполнять следующий пакет выражений. И далее выполняется второй пакет, который добавляет в нее две таблицы - Customers и Orders.

# Основы T-SQL. DML

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.

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

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

Для получения данных применяется команда **SELECT**. В упрощенном виде она имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1 | SELECT список\_столбцов FROM имя\_таблицы |

С помощью оператора **AS** можно изменить название выходного столбца или определить его псевдоним:

|  |
| --- |
| SELECT  ProductName + ' (' + Manufacturer + ')' AS ModelName,  Price,  Price \* ProductCount AS TotalSum  FROM Products |

### DISTINCT

Оператор **DISTINCT** позволяет выбрать уникальные строки. Например, в нашем случае в таблице может быть по несколько товаров от одних и тех же производителей. Выберем всех производителей:

SELECT DISTINCT Manufacturer

FROM Products

### Выборка с добавлением

#### SELECT INTO

Выражение **SELECT INTO** позволяет выбрать из одной таблицы некоторые данные в другую таблицу, при этом вторая таблица создается автоматически.

INSERT INTO ProductSummary

SELECT ProductName + ' (' + Manufacturer + ')' AS ModelName, Price

FROM Products

## Сортировка. ORDER BY

Оператор **ORDER BY** позволяет отсортировать извлекаемые значения по определенному столбцу:

|  |
| --- |
| SELECT \*  FROM Products  ORDER BY ProductName |

В данном случае строки сортируются по возрастанию значения столбца ProductName

Сортировку также можно проводить по псевдониму столбца, который определяется с помощью оператора AS:

|  |  |
| --- | --- |
| 1  2  3 | SELECT ProductName, ProductCount \* Price AS TotalSum  FROM Products  ORDER BY TotalSum |

По умолчанию применяется сортировка по возрастанию. С помощью дополнительного оператора **DESC** можно задать сортировку по убыванию. По умолчанию вместо DESC используется оператор **ASC**

Если необходимо отсортировать сразу по нескольким столбцам, то все они перечисляются после оператора ORDER BY:

|  |  |
| --- | --- |
| 1  2  3 | SELECT ProductName, Price, Manufacturer  FROM Products  ORDER BY Manufacturer, ProductName |

### Оператор TOP

Оператор **TOP** позволяет выбрать определенное количество строк из таблицы:

|  |  |
| --- | --- |
| 1  2 | SELECT TOP 4 ProductName  FROM Products |

Дополнительный оператор **PERCENT** позволяет выбрать процентное количество строк из таблицы. Например, выберем 75% строк:

|  |  |
| --- | --- |
| 1  2 | SELECT TOP 75 PERCENT ProductName  FROM Products |

### OFFSET и FETCH

Оператор TOP позволяет извлечь определенное количество строк, начиная с начала таблицы. Для извлечения набора строк из любого места, применяются операторы **OFFSET** и **FETCH**. Важно, что эти операторы применяются только в отсортированном наборе данных после выражения ORDER BY.

SELECT \* FROM Products

ORDER BY Id

    OFFSET 2 ROWS;

FETCH NEXT 3 ROWS ONLY;

## Фильтрация. WHERE

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

### 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.

### 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**.

## Операторы фильтрации

### Оператор IN

Оператор **IN** позволяет определить набор значений, которые должны иметь столбцы:

|  |  |
| --- | --- |
| 1 | WHERE выражение [NOT] IN (выражение) |

Выражение в скобках после IN определяет набор значений. Этот набор может вычисляться динамически на основании, например, еще одного запроса, либо это могут быть константные значения.

### Оператор BETWEEN

Оператор **BETWEEN** определяет диапазон значений с помощью начального и конечного значения, которому должно соответствовать выражение:

|  |  |
| --- | --- |
| 1 | WHERE выражение [NOT] BETWEEN начальное\_значение AND конечное\_значение |

Например, получим все товары, у которых цена от 20 000 до 40 000 (начальное и конечное значения также включаются в диапазон)

### Оператор LIKE

Оператор **LIKE** принимает шаблон строки, которому должно соответствовать выражение.

|  |  |
| --- | --- |
| 1 | WHERE выражение [NOT] LIKE шаблон\_строки |

Для определения шаблона могут применяться ряд специальных символов подстановки:

* **%**: соответствует любой подстроке, которая может иметь любое количество символов, при этом подстрока может и не содержать ни одного символа
* **\_**: соответствует любому одиночному символу
* **[ ]**: соответствует одному символу, который указан в квадратных скобках
* **[ - ]**: соответствует одному символу из определенного диапазона
* **[ ^ ]**: соответствует одному символу, который **не** указан после символа ^

Некоторые примеры использования подстановок:

* WHERE ProductName LIKE 'Galaxy%'

Соответствует таким значениям как "Galaxy Ace 2" или "Galaxy S7"

* WHERE ProductName LIKE 'Galaxy S\_'

Соответствует таким значениям как "Galaxy S7" или "Galaxy S8"

* WHERE ProductName LIKE 'iPhone [78]'

Соответствует таким значениям как "iPhone 7" или "iPhone8"

* WHERE ProductName LIKE 'iPhone [6-8]'

Соответствует таким значениям как "iPhone 6", "iPhone 7" или "iPhone8"

* WHERE ProductName LIKE 'iPhone [^7]%'

Соответствует таким значениям как "iPhone 6", "iPhone 6S" или "iPhone8". Но не соответствует значениям "iPhone 7" и "iPhone 7S"

* WHERE ProductName LIKE 'iPhone [^1-6]%'

Соответствует таким значениям как "iPhone 7", "iPhone 7S" и "iPhone 8". Но не соответствует значениям "iPhone 5", "iPhone 6" и "iPhone 6S"

## Обновление данных. Команда UPDATE

Для изменения уже имеющихся строк в таблице применяется команда **UPDATE**. Она имеет следующий формальный синтаксис:

|  |  |
| --- | --- |
| 1  2  3  4 | UPDATE имя\_таблицы  SET столбец1 = значение1, столбец2 = значение2, ... столбецN = значениеN  [FROM выборка AS псевдоним\_выборки]  [WHERE условие\_обновления] |

Например, увеличим у всех товаров цену на 5000:

|  |  |
| --- | --- |
| 1  2 | UPDATE Products  SET Price = Price + 5000 |

## Удаление данных. Команда DELETE

Для удаления применяется команда **DELETE**:

|  |  |
| --- | --- |
| 1  2 | DELETE [FROM] имя\_таблицы  WHERE условие\_удаления |

Например, удалим строки, у которых id равен 9:

|  |  |
| --- | --- |
| 1  2 | DELETE Products  WHERE Id=9 |

Если необходимо вовсе удалить все строки вне зависимости от условия, то условие можно не указывать:

|  |  |
| --- | --- |
| 1 | DELETE Products |

# Группировка

## Агрегатные функции

Агрегатные функции выполняют вычисления над значениями в наборе строк. В T-SQL имеются следующие агрегатные функции:

* **AVG**: находит среднее значение
* **SUM**: находит сумму значений
* **MIN**: находит наименьшее значение
* **MAX**: находит наибольшее значение
* **COUNT**: находит количество строк в запросе
* В качестве аргумента все агрегатные функции принимают выражение, которое представляет критерий дя определения значений. Зачастую, в качестве выражения выступает название столбца, над значениями которого надо проводить вычисления.
* Выражения в функциях **AVG** и **SUM** должно представлять числовое значение. Выражение в функциях **MIN**, **MAX** и **COUNT** может представлять числовое или строковое значение или дату.
* Все агрегатные функции за исключением COUNT(\*) игнорируют значения NULL.

### All и Distinct

По умолчанию все вышеперечисленных пять функций учитывают все строки выборки для вычисления результата. Но выборка может содержать повторяющие значения. Если необходимо выполнить вычисления только над уникальными значениями, исключив из набора значений повторяющиеся данные, то для этого применяется оператор **DISTINCT**.

## Операторы GROUP BY и HAVING

Для группировки данных в T-SQL применяются операторы **GROUP BY** и **HAVING**, для использования которых применяется следующий формальный синтаксис:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT столбцы  FROM таблица  [WHERE условие\_фильтрации\_строк]  [GROUP BY столбцы\_для\_группировки]  [HAVING условие\_фильтрации\_групп]  [ORDER BY столбцы\_для\_сортировки] |

### GROUP BY

Оператор **GROUP BY** определяет, как строки будут группироваться.

Например, сгруппируем товары по производителю

|  |  |
| --- | --- |
| 1  2  3 | SELECT Manufacturer, COUNT(\*) AS ModelsCount  FROM Products  GROUP BY Manufacturer |

Следует учитывать, что выражение GROUP BY должно идти после выражения WHERE, но до выражения ORDER BY

SELECT Manufacturer, COUNT(\*) AS ModelsCount

FROM Products

WHERE Price > 30000

GROUP BY Manufacturer

ORDER BY ModelsCount DESC

### Фильтрация групп. HAVING

Оператор **HAVING определяет, какие группы будут включены в выходной результат, то есть выполняет фильтрацию групп.**

Применение HAVING во многом аналогично применению WHERE. Только есть WHERE применяется к фильтрации строк, то HAVING используется для фильтрации групп.

Например, найдем все группы товаров по производителям, для которых определено более 1 модели:

|  |
| --- |
| SELECT Manufacturer, COUNT(\*) AS ModelsCount  FROM Products  GROUP BY Manufacturer  HAVING COUNT(\*) > 1 |

Если при этом необходимо провести сортировку, то выражение ORDER BY идет после выражения HAVING

## Расширения SQL Server для группировки

Дополнительно к стандартным операторам GROUP BY и HAVING SQL Server поддерживает еще четыре специальных расширения для группировки данных: **ROLLUP**, **CUBE**, **GROUPING SETS** и **OVER**.

### ROLLUP

Оператор **ROLLUP** добавляет суммирующую строку в результирующий набор:

|  |
| --- |
| SELECT Manufacturer, COUNT(\*) AS Models, SUM(ProductCount) AS Units  FROM Products  GROUP BY Manufacturer WITH ROLLUP |

### CUBE

**CUBE** похож на ROLLUP за тем исключением, что CUBE добавляет суммирующие строки для каждой комбинации групп.

|  |  |
| --- | --- |
| 1  2  3 | SELECT Manufacturer, COUNT(\*) AS Models, SUM(ProductCount) AS Units  FROM Products  GROUP BY Manufacturer, ProductCount WITH CUBE |

### GROUPING SETS

Оператор **GROUPING SETS** аналогично ROLLUP и CUBE добавляет суммирующую строку для групп. Но при этом он не включает сами группам:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SELECT Manufacturer, COUNT(\*) AS Models, ProductCount  FROM Products  GROUP BY GROUPING SETS(Manufacturer, ProductCount)  С помощью скобок можно определить более сложные сценарии группировки:   |  |  |  |  | | --- | --- | --- | --- | | 1  2  3  4 | SELECT Manufacturer, COUNT(\*) AS Models,          ProductCount, SUM(ProductCount) AS Units  FROM Products  GROUP BY GROUPING SETS((Manufacturer, ProductCount), ProductCount) OVER Выражение **OVER** позволяет суммировать данные, при этому возвращая те строки, которые использовались для получения суммированных данных. Например, найдем количество моделей и общее количество товаров этих моделей по производителю:   |  |  | | --- | --- | | 1  2  3  4 | SELECT ProductName, Manufacturer, ProductCount,          COUNT(\*) OVER (PARTITION BY Manufacturer) AS Models,          SUM(ProductCount) OVER (PARTITION BY Manufacturer) AS Units  FROM Products | | |

Выражение **OVER** ставится после агрегатной функции, затем в скобках идет выражение **PARTITION BY** и столбец, по которому выполняется группировка.

**Для преобразования полей можно использовать функцию CAST или CONVERT.** Для примера воспользуемся полем ID, оно у нас типа int:  
  
**SELECT**

**ID**,

**ID**/100, -- здесь произойдет целочисленное деление

**CAST**(**ID** **AS** float)/100, -- используем функцию CAST для преобразования в тип float

**CONVERT**(float,**ID**)/100, -- используем функцию CONVERT для преобразования в тип float

**ID**/100. -- используем преобразование за счет указания что знаменатель вещественное число

**FROM** Employees

Схема – это логическая единица БД, которая имеет свое наименование и позволяет сгруппировать внутри себя объекты БД такие как таблицы, представления и т.д.  
  
Определение схемы в разных БД может отличатся, где-то схема непосредственно связанна с пользователем БД, т.е. в данном случае можно сказать, что схема и пользователь – это синонимы и все создаваемые в схеме объекты по сути являются объектами данного пользователя. В MS SQL схема – это независимая логическая единица, которая может быть создана сама по себе (см. CREATE SCHEMA).  
  
По умолчанию в базе MS SQL создается одна схема с именем **dbo (Database Owner)** и все создаваемые объекты по умолчанию создаются именно в данной схеме. Соответственно, если мы в запросе указываем просто имя таблицы, то она будет искаться в схеме dbo текущей БД. Если мы хотим создать объект в конкретной схеме, мы должны будем так же предварить имя объекта именем схемы, например, «CREATE TABLE имя\_схемы.имя\_таблицы(…)».  
  
В случае MS SQL имя схемы может еще предваряться именем БД, в которой находится данная схема:  
**SELECT** \*

**FROM** Test.dbo.Employees -- имя\_базы.имя\_схемы.таблица

В MS SQL псевдонимы еще можно задавать при помощи знака равенства:

**SELECT**

'Дата приема'=HireDate, -- помимо "…" и […] можно использовать '…'

[Дата рождения]=Birthday,

ZP=Salary

**FROM** Employees

Использовать для задания псевдонима ключевое слово AS или же знак равенства, наверное, больше дело вкуса. Но при разборе чужих запросов, данные знания могут пригодиться.

И еще раз повторюсь, что любая операция с NULL дает NULL, например: 10+NULL, NULL\*15/3, 100/NULL – все это даст в результате NULL. Т.е. говоря просто неопределенное значение не может дать определенный результат. Учитывайте это при составлении запроса и при необходимости делайте обработку NULL значений функциями **ISNULL, COALESCE**

**SELECT**

**ID**,**Name**,

Salary/100\*BonusPercent **AS** Result1, -- без обработки NULL значений

Salary/100\***ISNULL**(BonusPercent,0) **AS** Result2, -- используем функцию ISNULL

Salary/100\***COALESCE**(BonusPercent,0) **AS** Result3 -- используем функцию COALESCE

**FROM** Employees

COALESCE (expr1, expr2, ..., exprn) - Возвращает первое не NULL значение из списка значений.

Так же с TOP можно использовать опцию WITH TIES, которая поможет вернуть все строки в случае неоднозначной сортировки, т.е. это предложение вернет все строки, которые равны по составу строкам, которые попадают в выборку TOP N, в итоге строк может быть выбрано больше чем N.

При помощи ESCAPE можно задать отменяющий символ, который отменяет проверяющее действие специальных символов «\_» и «%». Данное предложение используется, когда в строке нужно непосредственно проверить наличие знака процента или знака подчеркивания.  
  
Для демонстрации ESCAPE давайте занесем в одну запись мусор:

**SELECT** \*

**FROM** Employees

**WHERE** FirstName **LIKE** '%!%%' ESCAPE '!' -- строка содержит знак "%"

**SELECT** \*

**FROM** Employees

**WHERE** FirstName **LIKE** '%!\_%' ESCAPE '!' -- строка содержит знак "\_"

## Немного о датах

При проверке на дату, вы можете использовать, как и со строками одинарные кавычки '…'.  
Вне зависимости от региональных настроек в MS SQL можно использовать следующий синтаксис дат 'YYYYMMDD' (год, месяц, день слитно без пробелов). Такой формат даты MS SQL поймет всегда:  
**SELECT** **ID**,**Name**,Birthday

**FROM** Employees

**WHERE** Birthday **BETWEEN** '19800101' **AND** '19891231' -- сотрудники 80-х годов

**ORDER** **BY** Birthday

В некоторых случаях, дату удобнее задавать при помощи функции DATEFROMPARTS:  
**SELECT** **ID**,**Name**,Birthday

**FROM** Employees

**WHERE** Birthday **BETWEEN** **DATEFROMPARTS**(1980,1,1) **AND** **DATEFROMPARTS**(1989,12,31)

**ORDER** **BY** Birthday

## Немного о числах и их преобразованиях

В отличие от функции преобразования CAST, в функции CONVERT можно задать третий параметр, который отвечает за стиль преобразования (формат). Для разных типов данных может использоваться свой набор стилей, которые могут повлиять на возвращаемый результат.

**DECLARE** @min\_int int **SET** @min\_int=-2147483648

**DECLARE** @max\_int int **SET** @max\_int=2147483647

**SELECT**

-- (-2147483648)

@min\_int,**CAST**(@min\_int **AS** float),**CONVERT**(float,@min\_int),

-- 2147483647

@max\_int,**CAST**(@max\_int **AS** float),**CONVERT**(float,@max\_int),

-- numeric(16,6)

@min\_int/1., -- (-2147483648.000000)

@max\_int/1. -- 2147483647.000000

, в случае, если мы хотим получить результат типа numeric, с указанным количеством цифр после запятой, то мы можем в MS SQL применить трюк с умножением целого значения на (1., 1.0, 1.00 и т.д)

## Выполнение подзапросов

T-SQL поддерживает функциональность подзапросов (subquery), то есть таких запросов, которые могут встроены в другие запросы. Подзапрос выполняет команду и заключается в скобки.

### Коррелирующие подзапросы

Подзапросы бывают коррелирующими и некоррелирующими.

Но также существуют **коррелирующие подзапросы** (correlated subquery), результаты которых зависят от строк, которые выбираются в основном запросе.

Например, выберем все заказы из таблицы Orders, добавив к ним информацию о товаре:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT  CreatedAt,          Price,          (SELECT ProductName FROM Products          WHERE Products.Id = Orders.ProductId) AS Product  FROM Orders |

### Подзапросы в SELECT

В выражении SELECT мы можем вводить подзапросы четырьмя способами:

1. Использовать в условии в выражении WHERE
2. Использовать в условии в выражении HAVING
3. Использовать в качестве таблицы для выборки в выражении FROM
4. Использовать в качестве спецификации столбца в выражении SELECT

#### Получение набора значений

При использовании в операторах сравнения подзапросы должны возвращать одно скалярное значение. Но иногда возникает необходимость получить набор значений. Чтобы при использовании в операторах сравнения подзапрос мог возвращать набор значений, перед ним необходимо использовать один из операторов: **ALL**, **SOME** или **ANY**.

При использовании ключевого слова **ALL** условие в операции сравнения должно быть верно для всех значений, которые возвращаются подзапросом. Например, найдем все товары, цена которых меньше чем у любого товара фирмы Apple:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Products  WHERE Price < ALL(SELECT Price FROM Products WHERE Manufacturer='Apple') |

#### Подзапрос как спецификация столбца

Результат подзапроса может представлять отдельный столбец в выборке. Например, выберем все заказы и добавим к ним информацию о названии товара:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \*,  (SELECT ProductName FROM Products WHERE Id=Orders.ProductId) AS Product  FROM Orders |

### Подзапросы в команде UPDATE

В команде UPDATE подзапросы могут применяться:

1. В качестве устанавливаемого значения после оператора SET
2. Как часть условия в выражении WHERE

Так, увеличим количество купленных товаров на 2 в тех заказах, где покупатель Тоm:

|  |  |
| --- | --- |
| 1  2  3 | UPDATE Orders  SET ProductCount = ProductCount + 2  WHERE CustomerId=(SELECT Id FROM Customers WHERE FirstName='Tom') |

Оператор **EXISTS** позволяет проверить, возвращает ли подзапрос какое-либо значение. Как правило, этот оператор используется для индикации того, что какая-либо строка удовлетворяет условию. То есть фактически оператор EXISTS не возвращает строки, а лишь указывает, что в базе данных есть как минимум одна строка, которые соответствует данному запросу. Поскольку возвращения набора строк не происходит, то подзапросы с подобным оператором выполняются довольно быстро.

Применение оператора имеет следующий формальный синтаксис:

|  |  |
| --- | --- |
| 1 | WHERE [NOT] EXISTS (подзапрос) |

Например, найдем всех покупателей из таблицы Customer, которые делали заказы:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \*  FROM Customers  WHERE EXISTS (SELECT \* FROM Orders                    WHERE Orders.CustomerId = Customers.Id) |

Стоит отметить, что для получения подобного результата ы могли бы использовать и опеатор **IN**:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \*  FROM Products  WHERE Id NOT IN (SELECT ProductId FROM Orders) |

Но поскольку при применении EXISTS не происходит выборка строк, то его использование более оптимально и эффективно, чем использование оператора IN.

## Неявное соединение таблиц