**СОДЕРЖАНИЕ**

[Часть 1. ЯЗЫК SQL. СОЗДАНИЕ ТАБЛИЦ И ОГРАНИЧЕНИЙ НА SQL 3](#_Toc114857273)

[Цель работы 3](#_Toc114857274)

[Исходные данные 3](#_Toc114857275)

[Используемые программы 3](#_Toc114857276)

[Теоретические сведения 3](#_Toc114857277)

[Задание для лабораторной работы №3 (часть 1) 23](#_Toc114857278)

[Часть 2. СОЗДАНИЕ ЗАПРОСОВ НА ВЫБОРКУ. ОТБОР СТРОК ПО УСЛОВИЮ 28](#_Toc114857279)

[Цель работы 28](#_Toc114857280)

[Исходные данные 28](#_Toc114857281)

[Используемые программы 28](#_Toc114857282)

[Теоретические сведения 28](#_Toc114857283)

[Задание 31](#_Toc114857284)

[Ход работы 31](#_Toc114857285)

[Задание для лабораторной работы №3 (часть 2) 41](#_Toc114857287)

[Часть 3. СОЗДАНИЕ МНОГОТАБЛИЧНЫХ ЗАПРОСОВ. ЗАПРОСЫ НА СОЕДИНЕНИЕ 42](#_Toc114857288)

[Цель практической работы 42](#_Toc114857289)

[Исходные данные 42](#_Toc114857290)

[Используемые программы 42](#_Toc114857291)

[Теоретические сведения 42](#_Toc114857292)

[Задание для лабораторной работы №3 (часть 3) 53](#_Toc114857295)

[Часть 4. СОЗДАНИЕ ЗАПРОСОВ НА ГРУППИРОВКУ И СОРТИРОВКУ ДАННЫХ. ЗАПРОСЫ НА ИЗМЕНЕНИЕ. ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ ФУНКЦИЙ. 55](#_Toc114857296)

[Цель работы 55](#_Toc114857297)

[Исходные данные 55](#_Toc114857298)

[Используемые программы 55](#_Toc114857299)

[Задание 55](#_Toc114857300)

[Задание для лабораторной работы №3 (часть 4) 75](#_Toc114857301)

## ЛАБОРАТОРНАЯ РАБОТА №3

## Часть 1. ЯЗЫК SQL. СОЗДАНИЕ ТАБЛИЦ И ОГРАНИЧЕНИЙ НА SQL

### Цель работы

Изучение структурированного языка запросов Transact - SQL, являющегося основой системы программирования SQL Server, и приобретение навыков применение инструментальных средств разработки и программирования объектов создаваемых баз данных. Изучить SQL-операторы для работы с таблицами и индексами. Изучить sql- команды для создания, изменения и удаления таблиц. Изучить используемые в SQL Server типы ограничений. Изучить SQL-операторы для работы с ограничениями.

### Исходные данные

Исходными данными является индивидуальное задание и результат предыдущих лабораторных работ.

### Используемые программы

Программа "SQL Server Management Studio" и установленный сервер Microsoft SQL Server.

### Теоретические сведения

**Основы программирования на Transact - SQL**

**Выполнение операций над таблицами, индексами и ограничениями с помощью языка SQL**

Система программирования **SQL Server** относится к классу командно- интерпретирующих систем сверхвысокого уровня. Единицами действий системы являются команды, исполняемые в режиме интерпретации сразу же по мере их поступления в сервер. Основой этой системы программирования является проблемно- ориентированный структурированный язык запросов **(Structured Query Language) Transact - SQL**, который расширяется и развивает возможности стандарта **ANSI SQL - 92**.

**Transact - SQL включает следующие средства:**

* + - 1. данные различного типа баз данных и переменных;
      2. константы, стандартные и ограниченные идентификаторы;
      3. арифметические и логические выражения, включающие следующие операнды: константы, переменные, имена столбцов таблиц, функции, подзапросы и условные выражения, а также выражения, взятые в круглые скобки;
      4. SQL - команды для создания, изменения и удаления баз данных и их объектов, а также для определения запросов на ввод, обработку и извлечение данных;
      5. управляющие программные структуры, определяющие условия и порядок выполнения команд в заданной последовательности или пакете команд;
      6. встроенные (системные) и определяемые пользователем функции;
      7. встроенные (системные) и определяемые пользователем хранимые процедуры.

В системе могут храниться, помимо функций и процедур, последовательности (пакеты) команд, которые называются **скриптами**. Если скрипт описывает процесс создания базы данных, или каких-либо ее объектов, то такой скрипт называется **сценарием**. Сценарии позволяют переносить структуру базы данных от одного сервера к другому, а также структуру таблиц и других объектов в различные базы данных. Скрипты хранятся в текстовых файлах.

Функции и хранимые процедуры баз данных позволяют уменьшить объем запросов, передаваемых от клиента к серверу, что повышает общую производительность системы. Наличие исходного кода для этих объектов позволяет упростить сопровождение программных комплексов и внесение изменений в них.

Обычно все бизнес-правила и алгоритмы обработки данных реализуются на сервере баз данных и доступны конечному пользователю в виде набора функций и хранимых процедур, которые и представляют интерфейс обработки данных. Для обеспечения целостности данных, а также в целях безопасности, приложению обычно не предоставляется прямой доступ к данным. Вся работа ведется с помощью указанного интерфейса.

Подобный подход делает весьма простым изменение алгоритмов обработки данных и обеспечивает возможность расширения системы без внесения изменений в само приложение. Достаточно изменить хранимую процедуру на сервере баз данных, и сделанные изменения тотчас станут доступными всем пользователям сети.

**В языке Transact - SQL имеются следующие виды констант:**

1. **битовые**: 0 и 1;
2. **логические**: FALSE и TRUE;
3. **бинарные в шестнадцатеричном представлении**: 0\*9E70DA;
4. символьные: 'ABC'; "ABC" (если QUOTEDIDENTIFIER = OFF); N 'ABC' (Unicode); N "ABC" (Unicode);

5. **целые**: 1; 2; 175;

1. **с фиксированной точкой**: 12.35; - 16.753;
2. **с плавающей точкой**: 1.75Е5; 3.84Е - 3;

8. **для даты**: " April 15.2003"; "4/15/2003"; "20031207";

9. **для времени**: 14:30; 14:30:20:999; 4am; 4pm;

10. **денежные**: $100;?200; 2.15.

Комментарии в языке бывают двух типов: сточные, начинающиеся с двух символов минуса - и блочные, заключаемые символами /\* и \*/.

Все объекты базы данных должны иметь имена, которые используются в командах для ссылки на эти объекты. Любой объект базы данных должен быть уникально идентифицирован. Помимо программных имен сервер автоматически генерирует внутренние уникальные имена для идентификации объектов баз данных, например, PK Table X 014543FA.

**Программные имена** задаются идентификаторами двух типов:

1. стандартными идентификаторами: Table X; Key Col;
2. ограниченными идентификаторами: [My Table]; [Order]; "My Table"; "Order" (если QUOTEDIDENTIFIER = ON).

Стандартный идентификатор в качестве первого символа может иметь любую латинскую или русскую букву, знаки #, ##, @, @@ и знак подчеркивания \_. Последующими знаками, помимо указанных, могут быть еще и десятичные цифры.

Ограниченные идентификаторы могут включать и другие символы, в том числе зарезервированные слова. В этом случае они должны заключаться в квадратные скобки или двойные кавычки.

В соответствии с идеологией SQL Server каждый объект создается определенным пользователем и принадлежит той или иной базе данных. В свою очередь база данных

расположена на конкретном сервере. Из имен объекта, пользователя, базы данных и сервера создается **полное имя (complete name)** или **полностью определенное имя (full qualified name)**, записываемое в следующем виде:

**[[[server.].[database].[owner\_name].] objectname**

Варианты обращения к объектам базы данных:

**A.B.C.D; A.B..D; A..C.D; A..D; B.C.D; B..D; C.D; D**

Чтобы сослаться на конкретный столбец таблицы или представления, необходимо в полном имени указать пятый элемент: **А.В.C.E.**

**Операторами выражения** могут быть унарные (**+** и **-** ), бинарные арифметические операторы **(+, -, \*, %** ), оператор присваивания **(=**), строковая операция конкатенации (**+**), операторы сравнения **(=, >, <, <=, >=, =, !=** или **<>, !<, !>**), логические операторы (**NOT, AND, OR, ALL, ANY, BETWEEN, EXIST, IN, LIKE, SOME** ) и битовые операторы **(&,**

**|, ^**).

Константы, переменные, операнды и выражения используются при записи команд и программирования функций и хранимых процедур, которые, все вместе взятые, составляют основную часть системы программирования SQL Server и определяют ее выразительную мощь. Команды позволяют создавать, модифицировать и удалять базы данных и их объекты, формировать сложные запросы на ввод, обработку и извлечение данных из баз знаний, выполнять функции администрирования и обслуживания баз данных. Функции и хранимые процедуры реализуют разнообразные алгоритмы обработки данных или выполнение служебных функций сервера.

При формировании запросов очень часто используются специальные **логические операторы**, синтаксис которых записывают следующим образом:

**1.** Выражение **{= | < > | ! = | > | >= | ! >|, = | !<} ALL** подзапрос.

Здесь скалярное выражение вычисляется и сравнивается с каждым значением, возвращаемым подзапросом. Если сравнение дает истину для всех возвращаемых подзапросом значений, то этот оператор возвращает истину.

1. Если вместо **ALL** записать **SOME** или **ANY**, то результатом будет истина, если хотя в одной строке будет выполняться заданное сравнение.
2. Выражение **[NOT] BETWEEN Н Выражение AND В Выражение** возвращает истину, когда значение выражения лежит в диапазоне значений **Н выражения** и **В Выражения** (или не лежит).
3. Оператор **EXISTS** (подзапрос) возвращает значение истина, если подзапрос возвращает хотя бы одну строку.
4. Выражение **[NOT] IN (подзапрос \ выражение [,.. n])** возвращает значение истина, если значение левого выражения совпадает с одним из значений подзапроса или списка значений правых выражений (или не совпадает).
5. Выражение **[NOT] LIKE шаблон [ESCAPE знак]** дает истину, если значение выражения соответствует или не соответствует шаблону, в котором **"%"** означает любое количество произвольных символов, **"\_"** - один произвольный символ, **"[символы]"** - один из указанных в скобках, **"[символы]"** - все символы, кроме указанных. Знак после слова **ESCAPE** позволяет указать, что следующий за ним знак шаблона не является управляющим знаком шаблона, т.е. знаком **"%", "\_"** и т.д., а представляет обычный знак строки.

Раздел документации сервера T - SQL Help содержит описание каждой команды языка Transact - SQL B и набор примеров их использования. Синтаксис команды

определяется с помощью специального метаязыка, основанного на **нормальных формах Бекуса Наура (БНФ)**.

Ключевые слова определения без всякого изменения переходят в саму команду; метапеременные, написанные курсивом, должны быть заменены программными именами; разделители (запятая, равно, апостроф и т.д.) также переходят в качестве разделителей в команду.

**Метасинтаксические знаки имеют следующий смысл:**

*::=* — есть по определению;

*|* — выбор альтернативы;

*[ ]* — возможное отсутствие части определения;

*{ }* — объединение частей определения для выюора или повторения;

*[,...n]* — повторение предшествующей части 1, 2, ..., n раз с разделителем запятая для этой части (разделитель может быть любой;)

*<.*..*>* — метапеременная, которая имеет свое определение.

**Команды SQL для создания, удаления и изменения таблицами**

Часть языка SQL, которая управляет метаданными, называется **Data Definition Language (DDL)**.

**К DDL относятся операторы для определения любых содержащихся в базе данных объектов, в том числе и таблиц.**

Операторы, определяющие структуру таблиц в MS SQL Server, соответствуют стандарту SQL, и поэтому без изменений будут работать и во многих других СУБД.

Таблицы создаются, изменяются и удаляются соответственно командами Transact-SQL:

* **CREATE TABLE,**
* **ALTER TABLE**
* **DROP TABLE.**

При создании новой базы данных сервер автоматически создает **18 системных таблиц** для хранения информации о ее структуре и организации, доступ к которым со стороны пользователя запрещен.

Помимо основных и системных таблиц, которые, как правило, постоянно хранятся в базе данных, можно использовать **временные таблицы** для временного хранения информации, которые автоматически уничтожаются при закрытии соединения с базой данных.

В языке Transact-SQL используются следующие типы данных:

|  |  |
| --- | --- |
| **binary(n)** | -двоичные данные фиксированной длины до 8000 байт; для n байтов выделяется n+4 байта памяти; значения задаются с помощью 16- ичных чисел 0 х<шестнадцатеричные цифры>; функция DATALENGTH позволяет определить длину поля в байтах;  дополнение нулевыми байтами производится справа. |
| **image** | **-** двоичные данные длиной до 2 - 1; место выделяется в виде цепочки  страниц. |
| **char(n)** | -строковый тип данных фиксированной длины без поддержки Unicode длиной до 8000 байтов; данные зависят от установленной кодовой страницы; если для столбца не задана опция NULL, то строка при необходимости будет дополняться справа пробелами;  если эта опция задана, то дополнение пробелами будет иметь место |
|  | при условии ANSI\_PADDING=ON, в противном случае пробелы  добавляться не будут. |
| **var char(n)** | -строковый тип, как и char^^ro не с фиксированной длиной; если ANSI\_PADDING=OFF, то будет выполняться удаление конечных  пробелов, если ANSI\_PADDING=ON, то удаление пробелов производиться не будет. |
| **Nchar(n)** | -строковый типкак и char(n), но с поддержкой Unicode, поэтому максимальное количество символов составляет 4000; в этом случае  для строковых констант надо задавать впереди букву N: N'ABC'. |
| **Nvarchar(n)** | -строковый тип, как varchar(n), но с поддержкой Unicode. |
| **Text** | -строковый тип без поддержки Unicode длиной до 2 Гбайт; память выделяется страницами по 8 Кбайт, связываемыми в цепочку; можно использовать встроенные функции: DATALENGTH, PATINDEX,  SUBSTRING, TEXTPTR, TEXTVALID, READTEXT, SET TEXTSIZE, UPDATETEXT, WRITETEXT. |
| **Ntext** | -строковый тип как и text, но с поддержкой Unicode, поэтому длина  строки не более 1 Гбайта. |
| **Int** | -целый тип длиной в 4 байта и с диапазоном от -231 до 231-1. **Smalling**  -целый тип длиной в 2 байта с диапазоном от -215 до 215-1. **Tinyint** - целый тип длиной в 1 байт и диапазоном от 0 до 255. |
| **Bigint** | -целый тип длиной в 8 байт и с диапазоном от-263 до 263-1. **Decimal[(p[,s])]** -десятичный двоично-кодированный тип с p десятичными разрядами, из которых s - дробных; максимальное значение p достигает 38, поэтому диапазон значений составляет от -  (1038-1) до 1038-1. |
| **Numeric[(p[,s])]** | -тип, аналогичный типу decimal[(p[,s])]. |
| **Float[(n)]** | -плавающий (приблизительный) тип длиной в 4 байта и с диапазоном  от -1.79x10308 до 1.79x10308; значение n определяет количество бит для хранения мантисы и может принимать значения от 1 до 53. |
| **Real** | -плавающий тип, являющийся аналогом float(240). |
| **Datetime** | -тип данных для хранения даты (4 первых байта) и времени (4 последних байта) в диапазоне от 1.1.1753 и до 31.12.9999 года; дата хранится в виде смещения относительно базовой даты 1.1.1753, а время является количеством миллисекунд после полуночи; формат  для пользователя: MMM DD YYYY hh: |
| **Smalldatetime** | -тип данных для хранения даты (первых 2 байта) и времени (последние 2 байта) в диапазоне от 1.1.1900г. до 6.6.2079г., время задается с точностью до минуты. **Money** -тип данных для хранения больших денежных величин с точностью до 4 знаков после запятой в диапазоне от -922 337 203 685 477.5808 до +922 337 203 685 477.5807;  для хранения данных отводится 8 байт. **Smallmoney** -тип данных для хранения нормальных денежных величин с точностью до 4 знаков после запятой в диапазоне от -214 748.3648 до 214 748.3647; для  хранения данных отводится 4 байта. |
| **Bit** | -битовый (логический) тип со значениями 0 и 1; для хранения  выделяется 1 разряд байта памяти. |
| **Timestamp** | -тип данных временный штамп для учета числа изменений данных в записи (версий строки row version); значение timestamp уникально в  пределах базы данных и позволяет идентифицировать конкретное значение записи; тип аналогичен binary(8), если хранение NULL не |

|  |  |
| --- | --- |
|  | разрешено и varbinary(8), если разрешено. **Uniqueidentifier** -тип данных для хранения глобальных уникальных идентификаторов длиной в 16 байт, генерируемых функций NEWID и используемых для идентификации строк (записей); при генерации используется  номер сетевой карты компьютера и текущее время. |
| **Sysname** | -тип данных для хранения имен объектов базы данных; аналог nvarchar (128). **Sql\_variant** -вариантный тип данных для хранения данных любого типа, кроме text, ntext,image,timestamp; для получения информации о природе хранимых данных  используется функция SQL\_VARIANT\_PROPERTY(). |
| **Table** | -тип таблицы для временного хранения наборов данных с  использованием переменных. |

**Инструкция CREATE TABLE (Transact-SQL)**

**Для создания таблиц** используется оператор "**CREATE TABLE**", который приводит к созданию **пустой** таблицы без строк. При создании таблиц задается имя таблицы, описание набора столбцов с их именами, типами и размерами, а также ограничения на хранящуюся в таблице информацию.

Имена таблиц в пределах базы данных должны быть уникальны.

Каждый столбец в таблице должен иметь имя, уникальное в пределах таблицы, а также либо тип данных, ограничения целостности, либо выражение для вычисления значения столбца.

**Общий синтаксис**

**CREATE TABLE**

[ database\_name . [ schema\_name ] . | schema\_name . ] table\_name [ AS FileTable ]

( { <column\_definition> | <computed\_column\_definition>

**| <column\_set\_definition> | [ <table\_constraint> ] [ ,...n ] } )**

**[ ON { partition\_scheme\_name ( partition\_column\_name ) | filegroup**

**| "default" } ]**

**[ { TEXTIMAGE\_ON { filegroup | "default" } ]**

**[ FILESTREAM\_ON { partition\_scheme\_name | filegroup| "default" } ]**

**[ WITH ( <table\_option> [ ,...n ] ) ]**

[ ; ]

<column\_definition> ::=

column\_name <data\_type>

[ FILESTREAM ]

[ COLLATE collation\_name ] [ NULL | NOT NULL ]

[

[ CONSTRAINT constraint\_name ] DEFAULT constant\_expression ]

| [ IDENTITY [ ( seed ,increment ) ] [ NOT FOR REPLICATION ]

]

[ ROWGUIDCOL ] [ <column\_constraint> [ ...n ] ] [ SPARSE ]

<data type> ::=

[ type\_schema\_name . ] type\_name [ ( precision [ , scale ] | max |

[ { CONTENT | DOCUMENT } ] xml\_schema\_collection ) ]

<column\_constraint> ::=

[ CONSTRAINT constraint\_name ]

{ { PRIMARY KEY | UNIQUE }

[ CLUSTERED | NONCLUSTERED ] [

WITH FILLFACTOR = fillfactor

| WITH ( < index\_option > [ , ...n ] )

]

[ ON { partition\_scheme\_name ( partition\_column\_name )

| filegroup | "default" } ]

| [ FOREIGN KEY ]

REFERENCES [ schema\_name . ] referenced\_table\_name [ ( ref\_column ) ]

[ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ] [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ] [ NOT FOR REPLICATION ]

| CHECK [ NOT FOR REPLICATION ] ( logical\_expression )

}

<computed\_column\_definition> ::= column\_name AS computed\_column\_expression [ PERSISTED [ NOT NULL ] ]

[

[ CONSTRAINT constraint\_name ]

{ PRIMARY KEY | UNIQUE }

[ CLUSTERED | NONCLUSTERED ] [

WITH FILLFACTOR = fillfactor

| WITH ( <index\_option> [ , ...n ] )

]

| [ FOREIGN KEY ]

REFERENCES referenced\_table\_name [ ( ref\_column ) ] [ ON DELETE { NO ACTION | CASCADE } ]

[ ON UPDATE { NO ACTION } ] [ NOT FOR REPLICATION ]

| CHECK [ NOT FOR REPLICATION ] ( logical\_expression ) [ ON { partition\_scheme\_name ( partition\_column\_name )

| filegroup | "default" } ]

]

<column\_set\_definition> ::=

column\_set\_name XML COLUMN\_SET FOR ALL\_SPARSE\_COLUMNS

< table\_constraint > ::=

[ CONSTRAINT constraint\_name ]

{

{ PRIMARY KEY | UNIQUE }

[ CLUSTERED | NONCLUSTERED ]

(column [ ASC | DESC ] [ ,...n ] ) [

WITH FILLFACTOR = fillfactor

|WITH ( <index\_option> [ , ...n ] )

]

[ ON { partition\_scheme\_name (partition\_column\_name)

| filegroup | "default" } ]

| FOREIGN KEY

( column [ ,...n ] )

REFERENCES referenced\_table\_name [ ( ref\_column [ ,...n ] ) ]

[ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ] [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ] [ NOT FOR REPLICATION ]

| CHECK [ NOT FOR REPLICATION ] ( logical\_expression )

}

<table\_option> ::=

{

[DATA\_COMPRESSION = { NONE | ROW | PAGE }

[ ON PARTITIONS ( { <partition\_number\_expression> | <range> } [ , ...n ] ) ]]

[ FILETABLE\_DIRECTORY = <directory\_name> ]

[ FILETABLE\_COLLATE\_FILENAME = { <collation\_name> | database\_default

} ]

[ FILETABLE\_PRIMARY\_KEY\_CONSTRAINT\_NAME = <constraint\_name> ] [ FILETABLE\_STREAMID\_UNIQUE\_CONSTRAINT\_NAME =

<constraint\_name> ]

[ FILETABLE\_FULLPATH\_UNIQUE\_CONSTRAINT\_NAME =

<constraint\_name> ]

}

<index\_option> ::=

{

PAD\_INDEX = { ON | OFF }

| FILLFACTOR = fillfactor

| IGNORE\_DUP\_KEY = { ON | OFF }

| STATISTICS\_NORECOMPUTE = { ON | OFF }

| ALLOW\_ROW\_LOCKS = { ON | OFF}

| ALLOW\_PAGE\_LOCKS ={ ON | OFF}

| DATA\_COMPRESSION = { NONE | ROW | PAGE }

[ ON PARTITIONS ( { <partition\_number\_expression> | <range> } [ , ...n ] ) ]

}

<range> ::=

<partition\_number\_expression> TO <partition\_number\_expression>

Рассмотрим и расшируем назначение ключевых слов и аргументов команды

**CREATE TABLE**.

**Аргументы:**

***database\_name***

Имя базы данных, в которой создается таблица. Параметр **database\_name** должен указывать имя существующей базы данных. Если аргумент **database\_name** не указан, по умолчанию список стоп-слов создается в текущей базе данных. Имя входа для текущего соединения должно быть связано с идентификатором пользователя, существующего в базе данных, указанной аргументом **database\_name**, а этот пользователь должен обладать разрешениями **CREATE TABLE**.

***schema\_name***

Имя схемы, которой принадлежит новая таблица.

***table\_name***

Имя новой таблицы. Имена таблиц должны соответствовать правилам для идентификаторов. Аргумент **table\_name** может состоять не более чем из 128 символов, за исключением имен локальных временных таблиц (имена с одним префиксом номера #), длина которых не должна превышать 116 символов.

***AS FileTable***

Создает новую таблицу FileTable. Нет необходимости указывать столбцы, так как таблица FileTable имеет фиксированное схему. Дополнительные сведения о таблицах FileTable см. в разделе Таблицы FileTable (SQL Server).

***column\_name***

Имя столбца в таблице. Имена столбцов должны соответствовать правилам именования идентификаторов и быть уникальными в рамках таблицы. Аргумент **column\_name** может иметь длину не более 128 символов. Аргумент **column\_name** может быть опущен для столбцов, создаваемых с типом данных timestamp. Если аргумент **column\_name** не указан, столбцу типа timestamp по умолчанию присваивается имя timestamp.

**computed\_column\_expression**

Выражение, определяющее значение вычисляемого столбца. Вычисляемый столбец представляет собой виртуальный столбец, физически не хранящийся в таблице, если для него не установлен признак **PERSISTED**. Значение столбца вычисляется на основе выражения, использующего другие столбцы той же таблицы. Например, определение вычисляемого столбца может быть следующим:

**cost AS price \* qty**

Выражение может быть именем невычисляемого столбца, константой, функцией, переменной или любой их комбинацией, соединенной одним или несколькими операторами. Выражение не может быть вложенным запросом или содержать псевдонимы типов данных.

Вычисляемые столбцы могут использоваться в списках выбора, предложениях **WHERE, ORDER BY** и в любых других местах, в которых могут использоваться обычные выражения, за исключением следующих случаев.

* Вычисляемый столбец нельзя использовать ни в качестве определения ограничения **DEFAULT** или **FOREIGN KEY**, ни вместе с определением ограничения **NOT NULL**. Однако вычисляемый столбец может использоваться в качестве ключевого столбца индекса или части какого-либо ограничения **PRIMARY KEY** или **UNIQUE,** если значение этого вычисляемого столбца определяется детерминистическим выражением и тип данных результата разрешен в столбцах индекса.
* Вычисляемый столбец не может быть целевым столбцом инструкций

**INSERT** или **UPDATE**. Примечание

Каждая строка таблицы может содержать различные значения столбцов, задействованных в вычисляемом столбце; таким образом, значение вычисляемого столбца не будет одним и тем же в каждой строке.

**PERSISTED -** указывает, что компонент Компонент SQL Server Database Engine будет физически хранить вычисляемые значения в таблице и обновлять их при изменении любого столбца, от которого зависит вычисляемый столбец. Указание **PERSISTED** для вычисляемого столбца позволяет создать индекс по вычисляемому столбцу, который будет детерминистическим, но неточным.

**ON { <partition\_scheme> | filegroup | "default" }**

Указывает схему секционирования или файловую группу, в которой хранится таблица. Если аргумент **<partition\_scheme>** указан, таблица будет разбита на секции, хранимые в одной или нескольких файловых группах, указанных аргументом

<**partition\_scheme>**. Если указан аргумент **filegroup**, таблица сохраняется в файловой группе с таким именем. Это должна быть существующая файловая группа в базе данных. Если указано значение "**default**" или параметр **ON** не определен вообще, таблица сохраняется в файловой группе по умолчанию. Механизм хранения таблицы, указанный в инструкции **CREATE TABLE**, изменить в дальнейшем невозможно.

**TEXTIMAGE\_ON { filegroup| "default" }**

Указывают, что столбцы типов **text, ntext, image, xml, varchar(max), nvarchar(max), varbinary(max),** а также определяемых пользователем типов данных **CLR** (включая **geometry** и **geography**) хранятся в указанной файловой группе.

**FILESTREAM\_ON { partition\_scheme\_name | filegroup | "default" }**

Задает файловую группу для данных **FILESTREAM**.

Если таблица содержит данные **FILESTREAM** и является секционированной, необходимо включить предложение **FILESTREAM\_ON** и указать схему секционирования файловых групп файлового потока. В этой схеме секционирования должны использоваться те же функции и столбцы секционирования, что и в схеме секционирования для таблицы; в противном случае возникает ошибка.

***[ type\_schema\_name. ] type\_name***

Указывает тип данных столбца и схему, к которой он принадлежит. Тип данных может быть одним из следующих.

* Системный тип данных.
* Псевдонимы типа на основе системного типа данных SQL Server.
* Определяемый пользователем тип данных **CLR**. Прежде чем определяемые пользователем типы данных **CLR** можно будет использовать в определении таблицы, их нужно создать с помощью инструкции **CREATE TYPE**. Для создания столбца с определяемым пользователем типом данных **CLR** требуется разрешение **REFERENCES** на этот тип.

***precision***

Точность указанного типа данных.

***scale***

Масштаб указанного типа данных.

**CONTENT**

Указывает, что каждый экземпляр типа данных xml в столбце **column\_name** может содержать несколько элементов верхнего уровня. Аргумент **CONTENT** применим только к данным типа xml.

**DOCUMENT**

Указывает, что каждый экземпляр типа данных xml в столбце **column\_name** может содержать только один элемент верхнего уровня. Аргумент DOCUMENT применим только к данным типа xml.

**DEFAULT**

Указывает значение, присваиваемое столбцу в случае отсутствия явно заданного значения при вставке. Определения **DEFAULT** могут применяться к любым столбцам, кроме имеющих тип timestamp или обладающих свойством IDENTITY. Определения DEFAULT удаляются, когда таблица удаляется из памяти. В качестве значения по умолчанию могут использоваться только константы (например, символьные строки), скалярные функции (системные, определяемые пользователем или функции CLR) или значение NULL.

***constant\_expression***

Константа, значение NULL или системная функция, используемая в качестве значения столбца по умолчанию.

**IDENTITY**

Указывает, что новый столбец является столбцом идентификаторов. При добавлении в таблицу новой строки компонент Компонент Database Engine формирует для этого столбца уникальное последовательное значение. Столбцы идентификаторов обычно используются с ограничением **PRIMARY KEY** для поддержания уникальности идентификаторов строк в таблице.

Свойство **IDENTITY** присвоено столбцам типа **tinyint, smallint, int, bigint, decimal(p,0)** или **numeric(p,0).** Для каждой таблицы можно создать только один столбец идентификаторов. Ограниченные значения по умолчанию и ограничения DEFAULT не могут использоваться в столбце идентификаторов.

Необходимо указать как начальное значение, так и приращение, или же не указывать ничего. Если ничего не указано, применяется значение по умолчанию (**1,1**).

***seed***

Значение, используемое для самой первой строки, загружаемой в таблицу.

***increment***

Значение приращения, добавляемое к значению идентификатора предыдущей загруженной строки.

**NOT FOR REPLICATION**

В инструкции **CREATE TABLE** предложение **NOT FOR REPLICATION** может указываться для свойства **IDENTITY**, а также ограничений **FOREIGN KEY** и **CHECK**. Если это предложение указано для свойства **IDENTITY**, значения в столбцах идентификаторов не приращиваются, если вставку выполняют агенты репликации. Если ограничение сопровождается этим предложением, оно не выполняется, когда агенты репликации выполняют операции вставки, обновления или удаления.

**ROWGUIDCOL**

Указывает, что новый столбец является столбцом идентификаторов GUID строки. Только один столбец типа uniqueidentifier в таблице может быть назначен в качестве столбца ROWGUIDCOL. Применение свойства ROWGUIDCOL позволяет ссылаться на столбец с помощью ключевого слова $ROWGUID. Свойство ROWGUIDCOL может быть присвоено только столбцу типа uniqueidentifier. Ключевым словом ROWGUIDCOL нельзя обозначать столбцы определяемых пользователем типов данных.

**SPARSE**

Указывает, что столбец является разреженным столбцом. Хранилище разреженных столбцов оптимизируется для значений NULL. Для разреженных столбцов нельзя указать параметр NOT NULL. Дополнительные ограничения и сведения о разреженных столбцах см. в разделе Использование разреженных столбцов.

**FILESTREAM**

Допустимо только для столбцов типа varbinary(max). Указывает хранилище FILESTREAM для данных BLOB типа varbinary(max).

**COLLATE collation\_name**

Задает параметры сортировки для столбца. Могут использоваться параметры сортировки Windows или параметры сортировки SQL. Параметр **collation\_name** применим только к столбцам типов данных **char, varchar, text, nchar, nvarchar** и **ntext**. Если этот аргумент не указан, столбцу назначаются либо параметры сортировки определяемого пользователем типа, если столбец принадлежит к определяемому пользователем типу данных, либо установленные по умолчанию параметры сортировки для базы данных.

**CONSTRAINT**

Необязательное ключевое слово, указывающее на начало определения ограничения

**PRIMARY KEY, NOT NULL, UNIQUE, FOREIGN KEY или CHECK.**

***constraint\_name***

Имя ограничения. Имена ограничений должны быть уникальными в пределах схемы, к которой принадлежит таблица.

**NULL | NOT NULL**

Определяет, допустимы ли для столбца значения NULL.

**PRIMARY KEY**

Ограничение, которое обеспечивает целостность сущностей для указанного столбца или столбцов с помощью уникального индекса. Можно создать только одно ограничение **PRIMARY KEY** для таблицы.

**UNIQUE**

Ограничение, которое обеспечивает целостность сущностей для указанного столбца или столбцов с помощью уникального индекса. В таблице может быть несколько ограничений **UNIQUE**.

**FOREIGN KEY REFERENCES**

Ограничение, которое обеспечивает ссылочную целостность данных в этом столбце или столбцах. Ограничения **FOREIGN KEY** требуют, чтобы каждое значение в столбце существовало в соответствующем связанном столбце или столбцах в связанной таблице.

Ограничения **FOREIGN KEY** могут ссылаться только на столбцы, являющиеся ограничениями **PRIMARY KEY** или **UNIQUE** в связанной таблице или на столбцы, на которые имеются ссылки в индексе UNIQUE INDEX связанной таблицы. Внешние ключи в вычисляемых столбцах должны быть также помечены как PERSISTED.

***[ schema\_name.] referenced\_table\_name]***

Имя таблицы, на которую ссылается ограничение FOREIGN KEY, и схема, к которой она принадлежит.

**( ref\_column [ ,... n ] )**

Столбец или список столбцов из таблицы, на которую ссылается ограничение

**FOREIGN KEY**.

**ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }**

Определяет операцию, которая производится над строками создаваемой таблицы, если эти строки имеют ссылочную связь, а строка, на которую имеются ссылки, удаляется из родительской таблицы. Параметр по умолчанию — NO ACTION.

**NO ACTION**

Компонент Database Engine формирует ошибку, и выполняется откат операции удаления строки из родительской таблицы.

**CASCADE**

Если из родительской таблицы удаляется строка, соответствующие ей строки удаляются и из ссылающейся таблицы.

**SET NULL**

Все значения, составляющие внешний ключ, при удалении соответствующей строки родительской таблицы устанавливаются в NULL. Для выполнения этого ограничения внешние ключевые столбцы должны допускать значения NULL.

**SET DEFAULT**

Все значения, составляющие внешний ключ, при удалении соответствующей строки родительской таблицы устанавливаются в значение по умолчанию. Для выполнения этого ограничения все внешние ключевые столбцы должны иметь определения по умолчанию. Если столбец допускает значения NULL и значение по умолчанию явно не определено, значением столбца по умолчанию становится NULL.

**ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }**

Указывает, какое действие совершается над строками в изменяемой таблице, когда эти строки имеют ссылочную связь и строка родительской таблицы, на которую указывает ссылка, обновляется. Параметр по умолчанию — NO ACTION.

**NO ACTION**

Компонент Database Engine возвращает ошибку, а обновление строки родительской таблицы откатывается.

**CASCADE**

Соответствующие строки обновляются в ссылающейся таблице, если эта строка обновляется в родительской таблице.

**SET NULL**

Всем значениям, составляющим внешний ключ, присваивается значение NULL, когда обновляется соответствующая строка в родительской таблице. Для выполнения этого ограничения внешние ключевые столбцы должны допускать значения NULL.

**SET DEFAULT**

Всем значениям, составляющим внешний ключ, присваивается их значение по умолчанию, когда обновляется соответствующая строка в родительской таблице.

**CHECK**

Ограничение, обеспечивающее целостность домена путем ограничения возможных значений, которые могут быть введены в столбец или столбцы. Ограничения **CHECK** в вычисляемых столбцах должны быть также помечены как **PERSISTED**.

***logical\_expression***

Логическое выражение, возвращающее значения TRUE или FALSE.

***column***

Столбец или список столбцов (в скобках), используемый в ограничениях таблицы для указания столбцов, используемых в определении ограничения.

**[ ASC | DESC ]**

Указывает порядок сортировки столбца или столбцов, участвующих в ограничениях таблицы. Значение по умолчанию — ASC.

***partition\_scheme\_name***

Имя схемы секционирования, определяющей файловые группы, которым сопоставляются секции секционированной таблицы. Эта схема секционирования должна существовать в базе данных.

***[ partition\_column\_name. ]***

Указывает столбец, по которому будет секционирована таблица. Столбец должен соответствовать по типу данных, длине и точности столбцу, указанному в функции секционирования, используемой аргументом partition\_scheme\_name. Вычисляемый столбец, участвующий в функции секционирования, должен быть явно обозначен ключевым словом PERSISTED. Важно!

**WITH FILLFACTOR =fillfactor**

Указывает, насколько плотно компонент Компонент Database Engine должен заполнять каждую страницу индекса, используемую для хранения данных индекса.

**column\_set\_name XML COLUMN\_SET FOR ALL\_SPARSE\_COLUMNS**

Имя набора столбцов. Набор столбцов представляет собой нетипизированное XML- представление, в котором все разреженные столбцы таблицы объединены в структурированные выходные данные.

**< table\_option> ::=**

Указывает один или более параметров таблицы.

**DATA\_COMPRESSION**

Задает режим сжатия данных для указанной таблицы, номера секции или диапазона секций. Ниже приведены доступные параметры.

**ON PARTITIONS ( { <partition\_number\_expression> | <range> } [ ,...n ] )**

Указывает секции, к которым применяется параметр DATA\_COMPRESSION. Если таблица не секционирована, аргумент ON PARTITIONS приведет к формированию ошибки.

**Упрощенный синтаксис оператора создания таблицы:**

<определение\_таблицы> ::=

CREATE TABLE [ имя\_базы\_данных.[владелец].| владелец. ] имя\_таблицы

(<элемент\_таблицы>[,...n])

где

<элемент\_таблицы> ::=

{<определение\_столбца>}

| <имя\_столбца> AS <выражение>

|>ограничение\_таблицы<

Обычно владельцем таблицы (**dbo**) является тот, кто ее создал.

**<Выражение>** задает значение для вычисляемого столбца.

***<определение\_столбца> ::=***

{ имя\_столбца <тип\_данных>} [ [ DEFAULT <выражение> ]

| [ IDENTITY (начало, шаг) [NOT FOR REPLICATION]]]]

[ROWGUIDCOL][<ограничение\_столбца>][...n]]

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. Различные типы ограничений рассмотрим позже.

**Пример создания таблицы без ограничений.**

Для выполнения данных запросов предварительно откройте **среду Micrpsoft SQL Server Managmant Studio**, выполните соединение с сервером и откройте базу данных **Educator**. Нажмите на панели инструментов команду **Создать запрос**.

**Пример 1. Создание родительской таблицы Товар без ограничений**

CREATE TABLE Товар (КодТовара INT IDENTITY(1,1),

Название VARCHAR(50), Цена MONEY,

Тип VARCHAR(50), Сорт VARCHAR(50), Город VARCHAR(50),

Остаток INT);

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример1.sql** в папке **ФИО\_студента/Лаб3**.

Автоматическая генерация значения столбца **КодТовара** достигается за счет использования свойства **IDENTITY**, по умолчанию начальное значение, генерируемое с помощью **IDENTITY** равно 1, так же как и его приращение. Таким образом, следующее значение будет равно 2. Значения в **IDENTITY**-столбцах обязательно последовательные, то есть если приращение положительное, то следующее значение всегда больше предыдущего, если приращение отрицательное, то – всегда меньше. Приращение и начальное значение могут быть заданы, однако этот механизм чрезвычайно редко используется в реальных проектах.

**Создание ограничений**

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.

**Пример 2. Создание таблицы Товар с ограничением первичного ключа.**

CREATE TABLE Товар

(КодТовара INT IDENTITY(1,1) PRIMARY KEY,

Название VARCHAR(50),

Цена MONEY,

Тип VARCHAR(50),

Сорт VARCHAR(50),

Город VARCHAR(50),

Остаток INT);

***Примечание.*** Прежде чем выполнять sql-код удалите ранее созданную таблицу

**Товар** из базы данных и обновите ее.

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример2.sql** в папке **ФИО\_студента/Лаб3**.

**Первичные ключи более чем одного поля**

Ограничение **PRIMARY KEY** может также быть применено для многочисленных полей, составляющих уникальную комбинацию значений. Предположим, что ваш первичный ключ - это имя, и вы имеете первое имя и последнее имя сохраненными в двух различных полях (так что вы можете организовывать данные с помощью любого из них). Очевидно, что ни первое ни последнее имя нельзя заставить быть уникальным самостоятельно, но мы можем каждую из этих двух комбинаций сделать уникальной.

Мы можем применить ограничение таблицы **PRIMARY KEY** для пар:

**Пример 3. Создание таблицы Сотрудники с ограничением первичного ключа.**

CREATE TABLE Сотрудники

(

Фамилия char(10),

Имя char(10),

Город char(10),

PRIMARY KEY (Фамилия, Имя));

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример3.sql** в папке **ФИО\_студента/Лаб3**.

Одна проблема в этом подходе та, что мы можем вынудить появление уникальности - например, введя **Иванов Андрей** и **Иванов А.** Это может ввести в заблуждение, потому что ваши служащие могут не знать кто из них кто.

Обычно более надежный способ чтобы определять числовое поле, которое могло бы отличать одну строку от другой, это иметь первичный ключ, и применять ограничение **UNIQUE** для двух имен полей.

**Использование ограничений для исключения пустых (NULL) указателей**

Вы можете использовать команду **CREATE TABLE** чтобы предохранить поле от разрешения в нем пустых (**NULL**) указателей с помощью ограничения **NOT NULL**. Это ограничение накладывается только для разнообразных столбцов.

**NULL** - это специальное обозначение, которое отмечает поле как пустое. **NULL** может быть полезен, когда имеются случаи, когда вы хотите быть от них гарантированы. Очевидно, что первичные ключи никогда не должны быть пустыми, поскольку это будет подрывать их функциональные возможности. Кроме того, такие поля как имена, требуют в большинстве случаев, определенных значений. Например, вы вероятно захотите иметь информацию о **должности** занимаемым каждым сотрудником в таблице **Сотрудники**.

Если вы поместите ключевые слова **NOT NULL** сразу после типа данных (включая размер) столбца, любая попытка поместить значение **NULL** в это поле будет отклонена. В противном случае, SQL понимает, что **NULL** раз решен.

Например, давайте улучшим наше определение таблицы Сотрудники, не позволяя помещать NULL значения в столбец **Должность** :

**Пример 4. Создание таблицы Сотрудники с ограничением пустых значений.**

CREATE TABLE Сотрудники

(

Фамилия char(10),

Имя char(10) ,

Город char(10),

Должность сhar(10) NOT NULL, PRIMARY KEY (Фамилия, Имя));

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример4.sql** в папке **ФИО\_студента/Лаб3.**

Важно помнить, что любому столбцу с ограничением **NOT NULL** должно быть установлено значение в каждом предложении **INSERT** воздействующем на таблицу. При отсутствии **NULL**, SQL может не иметь значений для установки в эти столбцы, если конечно значение по умолчанию, описанное ранее в этой главе, уже не было назначено.

Если ваша система поддерживает использование **ALTER TABLE** чтобы добавлять новые столбцы к уже существующей таблице, вы можете вероятно помещать ограничение столбцов, типа **NOT NULL**, для этих новых столбцов. Однако, если вы предписываете новому столбцу значение **NOT NULL**, текущая таблица должна быть **пустой**!!!!

**Использование ограничений для уникальности значений**

В предыдущей практике мы обсудили использование уникальных индексов, чтобы заставить поля иметь различные значения для каждой строки. Эта практика - осталась с прежних времен, когда SQL поддерживал ограничение **UNIQUE**.

**Уникальность** - это свойство данных в таблице, и поэтому его более логично назвать как ограничение этих данных, а не просто как свойство логического отличия, связывающее объект данных (индекс).

Несомненно, уникальные индексы - один из самых простых и наиболее эффективных методов предписания уникальности. По этой причине, некоторые реализации ограничения **UNIQUE** используют уникальные индексы; то есть они создают индекс, не сообщая вам об этом. Остается фактом, что вероятность беспорядка в базе данных достаточно мала, если вы предписываете уникальность вместе с ограничением.

**Уникальность как ограничение столбца**

Время от времени, вы хотите убедиться, что все значения введеные в столбец отличаются друг от друга. Например, первичные ключи достаточно ясно это показывают. Если вы помещаете ограничение столбца **UNIQUE** в поле при создании таблицы, база данных отклонит любую попытку ввода в это поле для одной из строк, значения, которое уже представлено в другой строке. Это ограничение может применяться только к полям, которые были объявлены как непустые (**NOT NULL**), так как не имеет смысла позволить одной строке таблицы иметь значение **NULL**, а затем исключать другие строки с **NULL** значениями как дубликаты. Имеется дальнейшее усовершенствование нашей команды создания таблицы Сотрудники:

**Пример 5. Создание таблицы Сотрудники с ограничением уникальности.**

CREATE TABLE Сотрудники

(

Фамилия char(10) NOT NULL UNIQUE,

Имя char(10) NOT NULL UNIQUE,

Город char(10),

Должность сhar(10) NOT NULL, PRIMARY KEY ( Фамилия, Имя ));

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример5.sql** в папке **ФИО\_студента/Лаб3**.

Когда вы объявляете поле Фамилия уникальным, убедитесь, что в вашей базе данных не будет двух Ивановых или Петровых. В то же время — это не так уж необходимо с функциональной точки зрения - потому что поле Имя в качестве первичного ключа, все равно обеспечит отличие этих двух строк - что проще для людей, использующих данные в таблицах, чем помнить, что эти Ивановы не идентичны.

**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.

Когда вы вставляете строку в таблицу без указания значений в ней для каждого поля, SQL должен иметь значение по умолчанию для включения его в определенное поле, или же команда будет отклонена. Наиболее общим значением по умолчанию является - **NULL**. Это - значение по умолчанию для любого столбца, которому не было дано ограничение **NOT NULL** или который имел другое назначение по умолчанию.

Значение **DEFAULT (ПО УМОЛЧАНИЮ)** указывается в команде **CREATE TABLE** тем же способом что и ограничение столбца, хотя, с технической точки зрения, значение **DEFAULT** не ограничительного свойства - оно не ограничивает значения, которые вы можете вводить, а просто определяет, что может случиться если вы не введете любое из них.

Предположим, что вы работаете в г. Москва и подавляющее большинство ваших сотрудников живут в этом городе. Вы можете указать г. Москва в качестве значения поля **Город**, по умолчанию, для вашей таблицы **Сотрудников**:

**Пример 7. Создание таблицы Сотрудники с значением по умолчанию.**

CREATE TABLE Сотрудники

(

Фамилия char(10) NOT NULL UNIQUE,

Имя char(10) NOT NULL UNIQUE,

Город char(10) DEFAULT 'Москва',

Должность сhar(10) NOT NULL, PRIMARY KEY ( Фамилия, Имя ));

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример7.sql** в папке **ФИО\_студента/Лаб3**.

Конечно, вводить значение Москва в таблицу каждый раз, когда назначается новый сотрудник, не такая уж необходимость, и можно просто пре небречь им (не вводя его) даже если оно должно иметь некоторое значение.

Другой способ использовать значение по умолчанию - это использовать его как альтернативу для **NULL**. Так как **NULL** (фактически) неверен при любом сравнении, ином чем **IS NULL**, он может быть исключен с помощью большинства предикатов. Иногда, вам нужно видеть пустые значения ваших полей, не обрабатывая их каким-то определенным образом. Вы можете установить значение по умолчанию, типа нуль или пробел, которые функционально меньше по значению чем просто не установленное значение - пустое значение(**NULL**). Различие между ними и обычным **NULL** в том, что SQL будет обрабатывать их также, как и любое другое значение.

**Ограничение проверочное (CHECK)**

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.

**Проверка значений полей**

Конечно, имеется любое число ограничений, которые можно устанавливать для данных вводимых в ваши таблицы, чтобы видеть, например, находятся ли данные в соответствующем диапазоне или правильном формате, о чем SQL естественно не может знать заранее. По этой причине, SQL обеспечивает вас ограничением **CHECK**, которое позволяет вам установить условие, которому должно удовлетворять значение, вводимое в таблицу, прежде чем оно будет принято.

Ограничение **CHECK** состоит из ключевого слова **CHECK** сопровождаемого предложением предиката, который использует указанное поле. Любая попытка модифицировать или вставить значение поля, которое могло бы сделать этот предикат неверным - будет отклонена.

Давайте рассмотрим таблицу **Продавцы**. Столбец комиссионных выражается десятичным числом и поэтому может быть умножен непосредственно на сумму приобретений в результате чего будет получена сумма комиссионных (в долларах) продавца с установленным справа значком доллара ($). Кто-то может использовать понятие процента, однако ведь, можно об этом и не знать. Если человек введет по ошибке 14 вместо .14 чтобы указать в процентах свои комиссионные, это будет расценено как 14.0, что является законным десятичным значением, и будет нормально воспринято системой. Чтобы предотвратить эту ошибку, мы можем наложить ограничение столбца - **CHECK** чтобы убедиться, что вводимое значение меньше чем 1.

**Пример 8. Создание таблицы Продавцы и с проверкой значений полей**

CREATE TABLE Продавцы

( КодПродавца integer NOT NULL PRIMARY KEY,

Фамилия char(10) NOT NULL UNIQUE,

Город char(10),

Комиссионные decimal CHECK (Комиссионные < 1 ));

**Использование - CHECK, чтобы предопределять допустимое вводимое значение**

Мы можем также использовать ограничение CHECK чтобы защитить от ввода в поле определенных значений, и таким образом предотвратить ошибку.

Например, предположим, что единствеными городами в которых мы имели ведомства сбыта являются Лондон, Барселона, Сан Хосе, и Нью Йорк. Если вам известны все продавцы работающие в каждом из этих ведомств, нет необходимости позволять ввод других значений. Если же нет, использова ние ограничения может предотвратить опечатки и другие ошибки.

**Пример 8. Создание таблицы Продавцы и с проверкой значений полей, чтобы предопределять допустимое вводимое значение**

CREATE TABLE Продавцы

( КодПродавца integer NOT NULL PRIMARY KEY,

Фамилия char(10) NOT NULL UNIQUE,

Город char(10)

CHECK (Город IN (' Лондон ', 'Барселона', ' Сан Хосе ', ' Нью Йорк ')),

Комиссионные decimal CHECK (Комиссионные < 1 ));

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример8.sql** в папке **ФИО\_студента/Лаб3**.

Конечно, если вы собираетесь сделать это, вы должны быть уверены, что ваша компания не открыла уже новых других ведомств сбыта. Большинство программ баз данных поддерживают команду **ALTER TABLE**, которая позволяет вам изменять определение таблицы, даже когда она находится в использовании. Однако, изменение или удаление ограничений не всегда возможно для этих команд, даже там, где это вроде бы поддерживается.

Если вы использовали систему, которая не может удалять ограничения, вы будете должны создавать (**CREATE**) новую таблицу и передавать информацию из старой таблицы в нее всякий раз, когда вы хотите изменить ограничение. Конечно же Вы не захотите делать это часто, и со временем вообще перестанете это делать.

**Проверка условий, базирующийся на многочисленных полях**

Вы можете также использовать **CHECK** в качестве табличного ограничения. Это полезно в тех случаях, когда вы хотите включить более одного поля строки в условие. Предположим, что комиссионные 0.15 и выше, будут разрешены только для продавца из Барселоны. Вы можете указать это со следующим табличным ограничением **CHECK:**

**Пример 9. Создание таблицы Продавцы и с проверкой значений полей, базирующийся на многочисленных полях**

CREATE TABLE Продавцы2

( КодПродавца integer NOT NULL PRIMARY KEY,

Фамилия char(10) NOT NULL UNIQUE,

Город char(10) ,

Комиссионные decimal,

CHECK (Комиссионные < 0.15 OR Город='Барселона'));

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример9.sql** в папке **ФИО\_студента/Лаб3**.

Как вы можете видеть, два различных поля должны быть проверены чтобы определить, верен предикат или нет. Имейте в виду, что это - два разных поля одной и той же строки. Хотя вы можете использовать многочисленные поля, SQL не может проверить более одной строки одновременно. Вы не можете, например, использовать ограничение **CHECK,** чтобы удостовериться что все комиссионные в данном городе одинаковы. Чтобы сделать это, SQL должен всякий раз просматривая другие строки таблицы, когда вы модифицируете или вставляете строку, видеть, что значение комиссионных указано для текущего города. SQL этого делать не умеет.

Фактически, вы могли бы использовать сложное ограничение **CHECK** для вышеупомянутого, если бы знали заранее, каковы должны быть комиссионные в разных городах.

**Самостоятельно** измените ограничение в примере 9 на следующее:

* Если комиссионные равны 0.15 , то будут разрешены только для продавца из Лондона
* Если комиссионные равны 0.14 , то будут разрешены только для продавца из Барселоны
* Если комиссионные равны 0.13 , то будут разрешены только для продавца из Сан-Хосе
* Если комиссионные равны 0.12 , то будут разрешены только для продавца из Нью-Йорка

Вы получили идею. Чем налагать такой комплекс ограничений, вы могли бы просто использовать представление с предложением **WITH CHECK OPTION,** которое имеет все эти условия в своем предикате. Пользователи могут обращаться к представлению таблицы вместо самой таблицы. Одним из преимуществ этого будет то, что процедура изменения в ограничении не будет такой болезненной или трудоемкой. Представление с **WITH CHECK OPTION** - хороший заменитель ограничению **CHECK**.

**Пример 10. Создание таблицы Клиент с ограничениями.**

CREATE TABLE Клиент

(КодКлиента INT IDENTITY(1,1) PRIMARY KEY,

Фирма VARCHAR(50) NOT NULL,

Фамилия VARCHAR(50)NOT NULL,

Город VARCHAR(50) NOT NULL,

Телефон CHAR(10) NOT NULL

CHECK (Телефон LIKE '[1-9][0-9]-[0-9][0-9]-[0-9][0-9]'));

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример10.sql** в папке **ФИО\_студента/Лаб3**.

**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 столбцы 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**.

Если пользователь предпринимает попытку удалить из родительской таблицы строку, на которую ссылается одна или более строк дочерней таблицы, язык SQL предоставляет следующие возможности:

**CASCADE** - выполняется удаление строки из родительской таблицы, сопровождающееся автоматическим удалением всех ссылающихся на нее строк дочерней таблицы;

**SET NULL** - выполняется удаление строки из родительской таблицы, а во внешние ключи всех ссылающихся на нее строк дочерней таблицы записывается значение NULL;

**SET DEFAULT** - выполняется удаление строки из родительской таблицы, а во внешние ключи всех ссылающихся на нее строк дочерней таблицы заносится значение, принимаемое по умолчанию;

**NO ACTION** - операция удаления строки из родительской таблицы отменяется. Именно это значение используется по умолчанию в тех случаях, когда в описании внешнего ключа фраза ON DELETE опущена.

Те же самые правила применяются в языке SQL и тогда, когда значение потенциального ключа родительской таблицы обновляется.

Определитель **MATCH** позволяет уточнить способ обработки значения **NULL** во внешнем ключе.

При определении таблицы предложение **FOREIGN KEY** может указываться произвольное количество раз.

В операторе **CREATE TABLE** используется необязательная фраза **DEFAULT**, которая предназначена для задания принимаемого по умолчанию значения, когда в операторе **INSERT** значение в данном столбце будет отсутствовать.

Фраза **CONSTRAINT** позволяет задать имя ограничению, что позволит впоследствии отменить то или иное ограничение с помощью оператора **ALTER TABLE**.

**Пример 11. Создание таблицы Склад с ограничениями первичного ключа и внешнего ключа.**

CREATE TABLE Сделка

(КодСделки INT IDENTITY(1,1) PRIMARY KEY,

КодТовара INT NOT NULL,

КодКлиента INT NOT NULL,

Количество INT NOT NULL DEFAULT 0,

Дата DATETIME NOT NULL DEFAULT GETDATE(),

Остаток INT,

CONSTRAINT fk\_Товар FOREIGN KEY(КодТовара) REFERENCES Товар, CONSTRAINT fk\_Клиент FOREIGN KEY(КодКлиента) REFERENCES

Клиент);

**Выполните sql-код**. Обновите базу данных и просмотрите созданную таблицу.

Сохраните sql-запрос под именем **Пример11.sql** в папке **ФИО\_студента/Лаб3**.

**Изменение таблиц**

Для внесения изменений в уже созданные таблицы стандартом SQL предусмотрен оператор **ALTER TABLE**, предназначенный для выполнения следующих действий:

* добавление в таблицу нового столбца;
* удаление столбца из таблицы;
* добавление в определение таблицы нового ограничения;
* удаление из определения таблицы существующего ограничения;
* задание для столбца значения по умолчанию;
* отмена для столбца значения по умолчанию.

**Оператор изменения таблицы имеет следующий обобщенный формат:**

<изменение\_таблицы> ::=

ALTER TABLE имя\_таблицы

[ADD [COLUMN]имя\_столбца тип\_данных [ NOT NULL ][UNIQUE]

[DEFAULT <значение>][ CHECK (<условие\_выбора>)]] [DROP [COLUMN] имя\_столбца [RESTRICT | CASCADE ]]

[ADD [CONSTRAINT [имя\_ограничения]] [{PRIMARY KEY (имя\_столбца [,...n])

|[UNIQUE (имя\_столбца [,...n])}

|[FOREIGN KEY (имя\_столбца\_внешнего\_ключа [,...n])

REFERENCES имя\_род\_таблицы [(имя\_столбца\_род\_таблицы [,...n])],

[ MATCH {PARTIAL | FULL}

[ON UPDATE {CASCADE| SET NULL | SET DEFAULT | NO ACTION}]

[ON DELETE {CASCADE| SET NULL | SET DEFAULT | NO ACTION}]

|[CHECK(<условие\_выбора>)][,...n]}] [DROP CONSTRAINT имя\_ограничения

[RESTRICT | CASCADE]]

**[ALTER [COLUMN] SET DEFAULT <значение>] [ALTER [COLUMN] DROP DEFAULT]**

Здесь параметры имеют то же самое назначение, что и в определении оператора

**CREATE TABLE**.

Оператор **ALTER TABLE** реализован не во всех диалектах языка SQL. В некоторых диалектах он поддерживается, однако не позволяет удалять из таблицы уже существующие столбцы.

В дополнение к уже названным параметрам определим параметр **{ENABLE | DISABLE} TRIGGER ALL\_,** предписывающий задействовать или отключить конкретный триггер или все триггера, связанные с таблицей.

**Пример 12. Удаление ограничения внешнего ключа.**

ALTER TABLE Сделка DROP CONSTRAINT fk\_Товар

**Пример 13. Добавления ограничения внешнего ключа, реализующего каскадные обновления и изменения.**

ALTER TABLE Сделка ADD CONSTRAINT fk\_Товар FOREIGN KEY (КодТовара) REFERENCES Товар ON UPDATE CASCADE ON DELETE CASCADE

**Пример 14. Пример создания вычисляемого поля.**

ALTER TABLE Товар ADD Налог AS Цена\*0.05

**Пример 15. Пример удаления поля**

ALTER TABLE Товар DROP COLUMN Остаток

**Удаление таблиц**

Удаление таблицы выполняется командой:

DROP TABLE имя\_таблицы

Удалить можно любую таблицу, даже системную. К этому вопросу нужно подходить очень осторожно. Однако удалению не подлежат таблицы, если существуют объекты, ссылающиеся на них. К таким объектам относятся таблицы, связанные с удаляемой таблицей посредством внешнего ключа. Поэтому, прежде чем удалять родительскую таблицу, необходимо удалить либо ограничение внешнего ключа, либо дочерние таблицы. Если с таблицей связано хотя бы одно представление, то таблицу также удалить не удастся. Кроме того, связь с таблицей может быть установлена со стороны функций и процедур. Следовательно, перед удалением таблицы необходимо удалить все объекты базы данных, которые на нее ссылаются, либо изменить их таким образом, чтобы ссылок на удаляемую таблицу не было.

**Самостоятельно** удалите таблицу **Продавцы**.

### Задание для лабораторной работы №3 (часть 1)

**Самостоятельно,** используя **команды языка SQL,** в базе данных **Университет создать**:

* + 1. Новую таблицу под именем **STUDENT** (**Студент)** с помощью sql-операторов с полями:

**STUDENT\_ID** – целого типа для уникальной идентификации записей в таблице первичный ключ тип счетчик,

**SUTNAME** – текстового типа для обозначения имени студента,

**SUTFNAME** - текстового типа для обозначения фамилии,

**STIPEND** – действительного типа для обозначения стипендии. При этом на это поле наложено ограничение числом – величина размера стипендии должна быть меньше 500 руб.

**KURS** - целого типа для обозначения курса. При этом на это поле наложено ограничение – курс на котором может учиться студент может принимать значение от 1 до 4,

**CITY** - текстового типа для обозначения города,

**BIRTDAY** –типа даты/времени для обозначения день рождения,

**GROUP** - текстового типа для обозначения студенческой группы,

**KOD\_KAFEDRU** – целого типа для обозначения названия кафедры, на которой учится студент. Поле **KOD\_KAFEDRU** из таблицы **STUDENT** и поле **KOD\_KAFEDRU** из таблицы **KAFEDRA** связаны тем, что описывают одни и те же данные, т.е. содержат идентификаторы кафедр, информация о которых содержит база данных. Более того, значение идентификаторов кафедр, которые допустимы в таблице **STUDENT**, должны выбираться только из списка значений поля **KOD\_KAFEDRU**, т.е. принадлежащих реально описанных в базе данных кафедрам. Т.е. между этими полями имеется прямая связь. Т.о. поле **KOD\_KAFEDRU** из таблицы **STUDENT** будет являться внешним ключом.

Кроме того, при определении таблицы **STUDENT** запрещено использовать значение **NULL** – значений для столбцов **STUDENT\_ID, SUTNAME, SUTFNAME**.

В качестве первичного ключа принято значение столбца **STUDENT\_ID**.

***Выполните sql-код****. Обновите базу данных и просмотрите созданную таблицу.*

*Сохраните sql-запрос под именем* ***Студент.sql*** *в папке* ***ФИО\_студента/Лаб3.***

* + 1. Новую таблицу под именем **TEACHER** (**Преподаватели)** с помощью sql- операторов. Эта таблица содержит информацию о преподавателях вуза. Каждый преподаватель может работать на одной кафедре, иметь множество лекционных занятий и быть куратором более чем одной группы.

Описание столбцов таблицы TEACHER

**KOD\_TEACHER** Уникальный идентификатор преподавателя. Является первичным ключом

**KOD\_KAFEDRU** Уникальный идентификатор кафедры, на которой работает преподаватель. Является внешним ключом

**NAME\_TEACHER** Фамилия преподавателя

**INDEF\_KOD** Идентификационный код. Является уникальным для преподавателя

**DOLGNOST** Должность, может принимать только определенные значения из списка, такие как 'профессор', 'доцент', 'старший преподаватель', 'ассистент'. Значение по умолчанию 'ассистент'.

**ZVANIE** Научное звание, может принимать только определенные

значения из списка, такие как 'к.т.н', 'к.г.у', 'к.с.н', 'к.ф.м.н.', 'д.т.н', 'д.г.у', 'д.с.н', 'д.ф.м.н', 'нет'. Значение по умолчанию 'нет'.

**SALARY** ставка зарплаты. Значение по умолчанию 1000 руб. Зарплата должна быть больше нуля.

**RISE** надбавка к зарплате. Ее значение по умолчанию =0 и не может быть отрицательным числом.

**DATA\_HIRE** дата приема на работу. По умолчанию текущая дата.

**BIRTHDAY** день рождения

**POL** пол, может принимать только определенные значения из списка, 'ж', 'Ж', 'м', 'М'

**TEL\_TEACHER** Телефон. Может принимать значения только в виде '[1-9][0-9]- [0-9][0-9]-[0-9][0-9]'.

В качестве первичного ключа принято значение столбца **KOD\_TEACHER**.

Поле **KOD\_KAFEDRU** из таблицы **TEACHER** и поле **KOD\_KAFEDRU** из таблицы **KAFEDRA** связаны тем, что описывают одни и те же данные, т.е. содержат идентификаторы кафедр, информация о которых содержит база данных. Более того, значение идентификаторов кафедр, которые допустимы в таблице **TEACHER**, должны выбираться только из списка значений поля **KOD\_KAFEDRU**, т.е. принадлежащих реально описанных в базе данных кафедрам. Т.е. между этими полями имеется прямая связь. Т.о. поле **KOD\_KAFEDRU** из таблицы **TEACHER** будет являться внешним ключом.

***Выполните sql-код****. Обновите базу данных и просмотрите созданную таблицу.*

*Сохраните sql-запрос под именем* ***Преподаватель.sql*** *в папке* ***ФИО\_студента/Лаб3.***

**САМОСТОЯТЕЛЬНО** используя **команды языка SQL!!!**:

Создать **на языке Transact-SQL** файл базы данных согласно номеру варианта (присвоить ей новое имя, несовпадающее с именем базы данных, созданной в лаб.№2). База данных разрабатывается на основе спроектированной концептуальной модели данных в лаб.№1.

**Создать программно на языке SQL** все таблицы, с указанием первичных и внешних ключей и ограничения целостности.

Все программные инструкции команд SQL сохранять в файлах с расширением

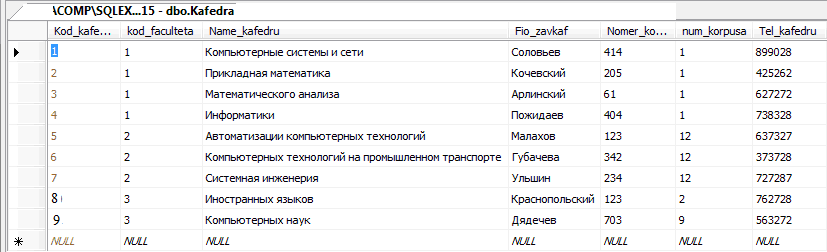
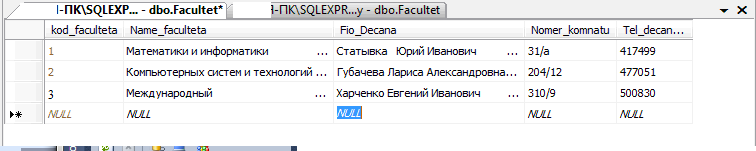
**\*.sql** в папке ***ФИО\_студента/Лаб3****.*

Заполнить таблицы данными по 5 записей в каждой.

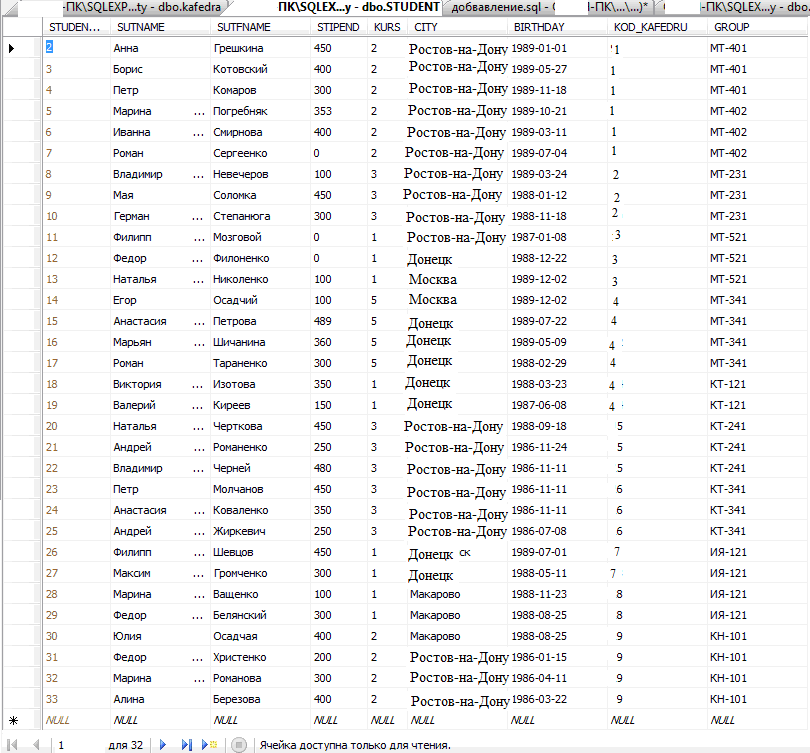
Создать текстовый отчет, в котором отобразить sql-команды разработанных запросов и скриншоты результатов работы из СУБД **SQL Server Management Studio**.

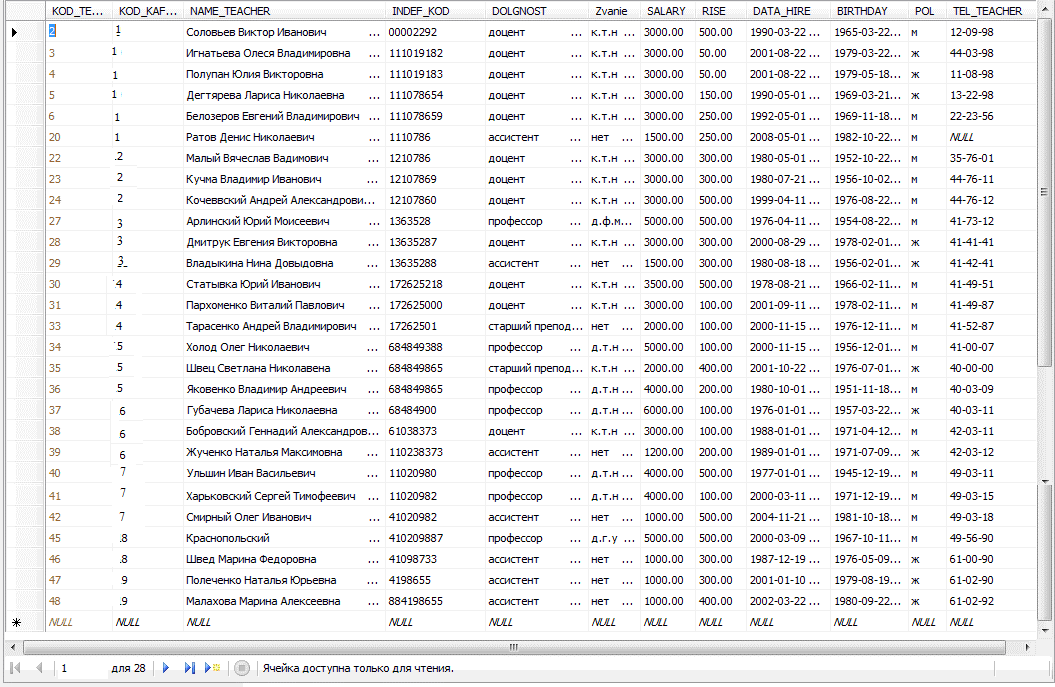
Самостоятельно заполните вручную данными таблицы Студент и Преподаватель согласно рис. 1-2, приведенным ниже.

Также ранее должны были введены следующие данные:



**Новые данные ввести вручную.**





## Часть 2. СОЗДАНИЕ ЗАПРОСОВ НА ВЫБОРКУ. ОТБОР СТРОК ПО УСЛОВИЮ

### Цель работы

Изучить используемый в реляционных СУБД оператор извлечения данных из таблиц. Получить навыки работы с оператором SELECT в программе ‘SQL Server Management Studio’.

### Исходные данные

Исходными данными является индивидуальное задание и результат предыдущих практических работ.

### Используемые программы

Программы ‘ SQL Server Management Studio ‘.

### Теоретические сведения

В SQL имеется единственный оператор, который предназначен для выборки данных из базы данных. Оператор относится к подмножеству **DML**.

Ниже приведен почти полный синтаксис оператора **SELECT**.

**SELECT** [**DISTINCT | ALL**]

**{\* |** <величина> [, <величина> ...]**}** [**INTO** :Переменная [, :Переменная ...]] **FROM** <tableref> [, <tableref> ...] [**WHERE** <условие поиска>]

[**GROUP BY** Колонка [, Колонка ...]] [**HAVING** <условие поиска>] [**UNION** [**ALL**] <select\_expr>] [**ORDER BY** <список сортировки>];

**<величина>** = **{**Колонка **|** :Переменная **|** <константа>

**|** <выражение> **|** <функция>

**|** udf **(**[<величина> [, <величина> ...]]**)**

**| NULL | USER}** [**AS** Псевдоним]

**<константа> =** Число **|** 'Строка'

**<выражение>** = SQL выражение, возвращающее единичное значение

<функция> =

**COUNT (\* |** [**ALL**] <величина> | **DISTINCT** <величина>**)**

**| SUM (**[**ALL**] <величина> **| DISTINCT** <величина>**)**

**| AVG** ([**ALL**] <величина> **| DISTINCT** <величина>**)**

**| MAX (**[**ALL**] <величина> **| DISTINCT** <величина>**)**

**| MIN (**[**ALL**] <величина> **| DISTINCT** <величина>**)**

**| CAST(**<величина> **AS** <тип данных>**)**

**| UPPER (**<величина>**)**

**| GEN\_ID (**Имя\_Генератора, <величина>**)**

**<tableref> = {**<joined\_table> **|** table **|** view

**|** procedure[**(**<величина> [, <величина> ...]**)**]**}**

[Псевдоним]

**<joined\_table> =** <tableref> <join\_type> **JOIN** <tableref>

**ON** <условие поиска> **| (**<joined\_table>**)**

<join\_type> = [**INNER**] **| {LEFT | RIGHT | FULL }** [**OUTER**]

**<условие поиска> =**

<величина> <оператор сравнения>

**{**<величина> **| (**<select\_one>**)}**

**|** <величина> [**NOT**] **BETWEEN** <величина> **AND** <величина>

**|** <величина> [**NOT**] **LIKE** <величина>

**|** <величина> [**NOT**] **IN**

**(**<величина> [, <величина> ...] **|** <select\_list>**)**

**|** <величина> **IS** [**NOT**] **NULL**

**|** <величина> **{**>= **|** <=**}** <величина>

**|** <величина> [**NOT**] **{**= **|** < **|** >**}** <величина>

**| {ALL | SOME | ANY} (**<select\_list>**)**

**| EXISTS (**<select\_expr>**)**

**| SINGULAR (**<select\_expr>**)**

**|** <величина> [**NOT**] **CONTAINING** <величина>

**|** <величина> [**NOT**] **STARTING** [**WITH**] <величина>

**| (**<условие поиска>**)**

**| NOT** <условие поиска>

**|** <условие поиска> **OR** <условие поиска>

**|** <условие поиска>**AND** <условие поиска>

**<оператор сравнения> =**

**{**= **|** < **|** > **|** <= **|** >= **|** !< **|** !> **|** <> **|** !=**}**

**<select\_one>** = оператор SELECT, выбирающий одну колонку и возвращающий ровно одно значение

**<select\_list>** = оператор SELECT, выбирающий одну колонку, возвращающий ноль или много значений

**<select\_expr>** = оператор SELECT, выбирающий несколько величин и возвращающий ноль или много значений

**<список сортировки> =**

**{**Колонка **|** Номер**}** [**ASC | DESC**]

[, <список сортировки> ...]

Некоторые параметры, входящие в этот оператор, описаны в табл.

**Описание параметров оператора SELECT**

|  |  |
| --- | --- |
| **Параметр** | **Описание** |
| **DISTINCT | ALL** | **DISTINCT** – предотвращает дублирование данных, которые будут извлечены.  **ALL** (по умолчанию) – приведет к извлечению всех |
| **{\* |** <величина> [,  <величина> ...]**}** | Звездочка (**\***) означает, что надо извлекать все колонки из указанных таблиц.  <величина> [, <величина> ...] – извлекает список указанных |
| **INTO** : Переменная ,  :Переменная ...] | Используется только в триггерах и хранимых процедурах для операторов SELECT, возвращающих не более одной строки.  Указывается список переменных, в которые извлекаются величины |
| **FROM <tableref> [,**  **<tableref> ...]** | Указывает список таблиц, просмотров и хранимых  процедур, из которых извлекаются данные. Список может включать соединения и соединения могут быть |
| **table** | Имя существующей в базе данных таблицы |
| **view** | Имя существующего базе данных просмотра |
| **procedure** | Имя существующей хранимой процедуры, предназначенной для использования в операторе SELECT |
| **Псевдоним** | Короткое альтернативное имя для таблицы, просмотра или  колонки. После описания в <tableref>, псевдоним может использоваться для ссылок на таблицу или просмотр |
| **join\_type** | Задает тип соединения, которое может быть внутренним или  внешним |
| **WHERE** <условие поиска> | Указывает условие, которое ограничивает количество извлекаемых строк |
| **GROUP BY** Колонка [, Колонка ...] | Разбивает результат запроса на группы, содержащие все строки с идентичными значениями указанными в списке |
| **HAVING**  <условие поиска> | Использует совместно с GROUP BY. Задает условие, которое ограничивает количество возвращаемых групп |
| **UNION** [**ALL**] | Объединяет результаты нескольких запросов. Все запросы должны извлекать одинаковое количество столбцов, тип данных каждого столба первого запроса должен совпадать с типом данных других запросов, имена столбцов в разных запросах могут отличаться. Необязательный параметр ALL |
| **ORDER BY**  <список сортировки> | Указывает колонки, по которым будет производиться сортировка извлекаемых строк. Можно указывать либо имена колонок, либо их порядковые номера в списке извлекаемых колонок. Если указать ASC, то строки будут выдаваться в  порядке возрастания значений сортируемых полей, если DESC |

Как видно из синтаксиса оператора **SELECT,** обязательными являются только предложение **SELECT** с перечнем выдаваемых колонок и предложение **FROM**.

**Пример простейшего оператора SELECT:**

*-- Выдать перечень всех служащих:*

**SELECT \* FROM** Employee;

Ниже приведено несколько упрощенных вариантов синтаксиса оператора

**SELECT**, помогающих научиться составлять простые запросы.

**Упрощенный синтаксис внутреннего соединения** (стандарт SQL-92):

**SELECT** Колонка [, Колонка ...] **| \***

**FROM** <tableref\_left> [**INNER**] **JOIN** <tableref\_right> [**ON** <условие поиска>]

[**WHERE** <условие поиска>];

**Упрощенный синтаксис внешнего соединения:**

**SELECT** Колонка [, Колонка ...] **| \* FROM** <tableref\_left>

**{LEFT | RIGHT | FULL}** [**OUTER**] **JOIN**

<tableref\_right>

[**ON** <условие поиска>] [**WHERE** <условие поиска>];

**Упрощенный синтаксис использования подзапроса:**

**SELECT** [**DISTINCT**] Колонка [, Колонка ...]

**FROM** <tableref> [, <tableref> ...]

**WHERE**

**{**expression **{**[**NOT**] **IN |** <оператор сравнения>**}**

**|** [**NOT**] **EXISTS**

**}**

**(SELECT** [**DISTINCT**] Колонка [, Колонка ...]

**FROM** <tableref> [, <tableref> ...]

**WHERE** <условие поиска>

**)**;

### Задание

Практическую работу следует выполнять в следующем порядке:

1. Изучить синтаксис оператора **SELECT** и примеры запросов к учебной базе данных ‘**University.mdf**’.
2. Выполнить в окне SQL Editor запросы к базе данных, согласно приведенным в практической работе образцам выполнения запросов и сохранять каждый под именами ‘**Lab3-k.sql’**, где k – номер запроса по порядку, в своей рабочей папке. Каждый запрос должен иметь комментарии с описанием, а файл в целом должен иметь комментарии со сведениями об авторе и дате создания.

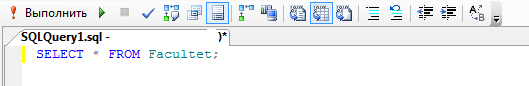
### Ход работы

**Примечание.** У вас должны быть перед выполнением этой практической работы созданы все таблицы базы данных университета, созданы ключи, а также заполнены данными.

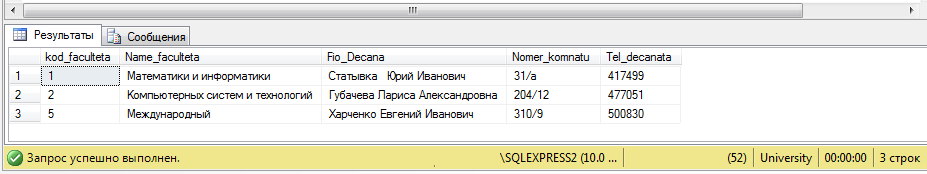
**Выполнение sql-запросов**

Для выполнения запросов **SELECT** в программе ‘SQL Server Management Studio’ необходимо выполнить следующие действия:

1. Подключиться к базе данных и выполнить команду ‘**Создать запрос**’. В результате откроется окно ‘**Конструктора запросов**’.



1. Ввести текст запроса согласно рис.1.
2. Нажать на панели инструментов кнопку  [**Выполнить**] .
3. Если запрос правильный, то в результате произойдет его выполнение и результат будет отображен на вкладке ‘**Результаты**’.



1. Количество извлеченных в результате выполнения запроса строк отображается над сеткой с данными справа. На рис.2 там содержится строка ‘3 строк’. В данном примере извлечено столько строк, сколько требуется, чтобы заполнить сетку (в ней помещается только 3 строки) \* .
2. Чтобы узнать, сколько всего строк соответствуют выполненному оператору, надо перейти в конец отображаемого набора данных.

Чтобы выполнить другой запрос, надо вернуться на вкладку ‘**Редактора**’, создать новый запрос и повторить те же действия.

К тексту ранее выполнявшихся правильных запросов можно вернуться, если перейти на вкладку ‘**История**’.

*Примеры создания запросов с отбором строк по условию****.***

SQL дает возможность определить критерии отбора необходимых строк во фразе WHERE предложения SELECT. В этом случае строки исходных таблиц будут включены в результирующую только если строка соответствует указанным критериям. *Условие -* это выражение, которое может быть истинным или ложным (логическое выражение или предикат), то есть принимать логические значения TRUE или FALSE соответственно. В результирующую таблицу включаются только те строки, для которых указанное во фразе WHERE условие равно TRUE (иными словами, которые удовлетворяют заданному условию).

В случае одной таблицы механизм работы предложения SELECT с фразой WHERE следующий.

1. Из таблицы, указанной во фразе **FROM**, выбирается очередная строка.
2. Она проверяется на соответствие условию во фразе **WHERE**.
3. Если результат равен **TRUE**, строка включается в результирующую таблицу и форматируется в соответствии с фразой **SELECT**, а если он равен **FALSE**, строка пропускается.

Далее будут рассмотрены основные выражения, допустимые для условия во фразе WHERE.

**Использование простейших условий**

Простейшими считаются условия, в которых используются операторы сравнения и логические операторы.

Хотя такие условия являются простейшими в смысле семантики конструкций, они могут иметь довольно сложную структуру из многих операторов сравнений и вложенных друг в друга логических связок.

**Операторы сравнения**

Особенностью операторов сравнения является то, что независимо от типов операндов их результатом являются логические значения. Предположим, вы хотите получить список всех профессоров.

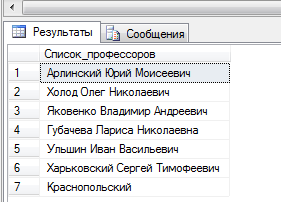
Создайте новый запрос, введите sql-запрос, выполните его, сохраните его в рабочую папку ЛАБ3\_SQL под именем 1.sql.

**Запрос 1. Вывести фамилии профессоров.**

SELECT NAME\_TEACHER AS ‘Cписок профессоров’ FROM TEACHER

WHERE DOLGNOST = 'профессор';

Чтобы выполнить sql-команду нажмите на панели редактора кнопку . В результате выполнения данного кода будут выданы все профессора. Например,



Слово 'пpофeccop' в запросе является строковой константой, поэтому ее следует заключить в кавычки. Обратите внимание, что мы указали фразу SELECT без ключевого слова DISTINCT, так как тогда от нас была бы скрыта информация о существовании среди профессоров однофамильцев. Чтобы при выводе результирующий столбец имел содержательный заголовок, мы поименовали его как Список профессоров.

Это первый пример использования предиката над строковым типом данных. Здесь столбец строкового типа сравнивается со строковой константой. Запрос выполнен правильно, однако нужно всегда помнить о том, что предикаты над строками являются чувствительными к регистру букв. Например, предикат 'ИВАНОВ' = 'Иванов' будет ложным. Поэтому, если для некоторого профессора его должность была введена в таблицу TEACHER как 'Профессор', он не будет найден по условию WHERE DOLGNOST = 'профессор'.

Чтобы на предикаты над строками не влиял регистр букв, нужно использовать обычно имеющиеся в СУБД функции преобразования букв в прописные и строчные. В стандарте SQL, например, указаны функции UPPER и LOWER, выполняющие такие преобразования. Следовательно, для предыдущего запроса правильней будет записать условие фразы WHERE одним из следующих способов:

**WHERE LOWER(DOLGNOST) = 'профессор' WHERE UPPER(DOLGNOST) = 'ПРОФЕССОР'**

**Самостоятельно** измените в исходном запросе строку условия с использованием функции изменения регистра.

Чтобы сохранить запрос нажмите правую кнопу и из контекстного меню выберите Сохранить в файл. Присвойте имя 1.sql и сохраните в папку Лаб3\_SQL.

**Запрос 2.** Найти всех студентов с стипендией, превышающей 300 руб.

В sql-редакторе создайте новый запрос и введите следующий код: **SELECT SUTNAME, SUTFNAME**

**FROM STUDENT WHERE STIPEND > 300;**

**Выполните его.**

Приведем несколько примеров использования операторов сравнения для столбцов строкового и временного типа. Обратите внимание, что сравнивать можно не только значение столбца с константой, но и значения столбцов между собой.

**Запрос 3. Вывести фамилии и должности преподавателей, принятых на работу после 01.01.2002.**

SELECT NAME\_TEACHER AS 'Фамилия', DOLGNOST AS 'Должность' FROM TEACHER

WHERE DATA\_HIRE > '1/01/2002';

**Запрос 4. Вывести фамилии и должности преподавателей, фамилии которых в алфавитном порядке располагаются после фамилии Сычева.**

**SELECT NAME\_TEACHER, DOLGNOST FROM TEACHER**

WHERE UPPER(NAME\_TEACHER) > 'Сычева';

**Самостоятельно создайте запрос 5**. Вывести фамилии преподавателей, у которых надбавка меньше ставки в 2,5 и более раз.

Логические операторы

**Операндами и результатом логических операторов являются логические значения.**

Стандартными логическими операторами являются **AND, OR** и **NOT**. Их действие показано в так называемых истинностных таблицах. Использование операторов сравнения вместе с логическими операторами предоставляет возможность формулировать составные условия для отбора строк таблиц.

**Использование логического оператора AND**

Логический оператор **AND** во многих случаях действует как связка ‘и’ в русском языке. Рассмотрим несколько примеров с использованием этого оператора.

**Запрос 6. Вывести фамилии студентов, проживающих в городе Макарово и имеющих стипендию больше 100 руб.**

SELECT SUTFNAME FROM STUDENT

WHERE CITY = 'Макарово' AND STIPEND >100;

Самостоятельно создайте запрос 7. **Вывести фамилии преподавателей, которые являются профессорами и ставка которых превышает 4500.**

**Самостоятельно создайте запрос 8.** Вывести фамилии студентов учащихся на кафедре под порядковым номером 2 (Прикладная математика) со стипендией в диапазоне 100-500 руб.

**Использование логического оператора OR**

Логический оператор OR во многих случаях действует как связка ‘или’ в русском языке. Рассмотрим несколько примеров.

**Запрос 9**. Вывести названия кафедр, расположенных либо в 1 либо в 8 корпусе.

SELECT NAME\_KAFEDRU, NUM\_KORPUSA FROM KAFEDRA

WHERE NUM\_KORPUSA = 1 OR NUM\_KORPUSA = 8;

**Использование логического оператора NOT**

Логический оператор NOT в русском языке передается словами ‘не’ и ‘кроме’.

**Запрос 10**. Вывести названия всех факультетов, кроме факультета математики и информатики.

SELECT NAME\_FACULTETA FROM FACULTET

WHERE NOT LOWER(NAME\_FACULTETA) = 'математики и информатики';

Обратите внимание, что оператор NOT должен предшествовать выражению сравнения, а не ставиться перед оператором сравнения. То есть запись LOWER(NAME\_FACULTETA) NOT =''математики и информатики'' будет неверной. Учитывая, что отрицанием оператора = является оператор <>, вместо указанного условия можно было бы записать LOWER(NAME\_FACULTETA) <> ''математики и информатики''. Это относится ко всем операторам сравнения, так как каждый из них имеет оператор, являющийся его отрицанием.

**Комбинирование логических операторов**

Логические операторы можно объединять, формируя составные условия. Возможность комбинирования обеспечивается тем, что любой логический оператор возвращает истинностное значение, а значит, его результат может использоваться в другом логическом операторе. Рассмотрим несколько примеров.

**Запрос 11**. Вывести фамилии, должность, ставку и надбавку ассистентов, у которых либо ставка меньше 550, либо надбавка больше 60.

SELECT NAME\_TEACHER, DOLGNOST, Salary, Rise FROM TEACHER

WHERE LOWER(DOLGNOST) ='ассистент' AND

(Salary < 550 OR Rise > 60);

**Использование выражений над столбцами**

До сих пор в выражениях фразы WHERE мы использовали в качестве значения одного или обоих аргументов имена столбцов. Однако аргументами могут быть и выражения над столбцами.

**Запрос 12.** Показать фамилии преподавателей, чья зарплата (ставка плюс надбавка) превышает 3500.

SELECT NAME\_TEACHER AS ‘Фамилия преподавателя’, Salary + Rise AS ‘Его зарплата’

FROM TEACHER

WHERE Salary + Rise > 3500;

**Запрос 13.** Показать фамилии преподавателей, половина зарплаты которых превышает пятикратную надбавку.

SELECT NAME\_TEACHER FROM TEACHER

WHERE (Salary + Rise) / 2 > 5 \* Rise;

**Использование специальных операторов**

В SQL имеются операторы сравнения, позволяющие проверять значения столбцов и выражений над ними на соответствие некоторым специальным условиям:

Принадлежность множеству;

принадлежность диапазону;

соответствие шаблону;

соответствие регулярному выражению;

неопределенное значение.

В этом разделе вы узнаете, как их использовать и как с их помощью создавать составные условия.

**Проверка на принадлежность множеству**

Оператор IN позволяет проверить, входит ли значение в указанное множество значений. В простейшем случае этот оператор имеет следующий синтаксис:

имя\_столбца [NOT] IN (список\_эначений)

Здесь список значений представляет собой перечень разделенных запятыми констант, тип которых должен соответствовать типу столбца, чье имя приведено слева. Семантика этого предиката такова: он принимает значение TRUE, если значение столбца соответствует одной из констант списка. Приведем пример.

**Запрос 14**. Вывести названия и номер корпуса кафедр, расположенных в корпусах 1, 3, 12.

SELECT Name\_Kafedru, NUM\_KORPUSA FROM KAFEDRA

WHERE NUM\_KORPUSA IN ('1', '3', '12');

**Использование отрицания**

Так как предикат IN возвращает истинностное значение, к нему можно применить логическое отрицание. Для этого следует воспользоваться нотацией NOT IN. В этом случае предикат будет истинным, если значение столбца не входит в указанный список.

**Запрос 15**. Вывести названия и номер корпуса кафедр, расположенных в любых корпусах, кроме 1, 3, или 12.

SELECT Name\_Kafedru AS ‘Название кафедры’, NUM\_KORPUSA AS “Корпус’

FROM KAFEDRA

WHERE NUM\_KORPUSA NOT IN (‘1', 'З', '12');

**Использование выражений над столбцами**

В левой части оператора IN вместо имени столбца можно использовать любое допустимое над столбцами таблицы выражение языка.

**Запрос 16**. Вывести фамилии преподавателей, зарплата которых (ставка + надбавка) равна 800, 900, 1000 или 1200.

SELECT NAME\_TEACHER AS ‘Фамилия преподавателя’, Salary + Rise AS ‘Зарплата преподавателя’

FROM TEACHER

WHERE Salary + Rise IN (800, 900, 1000, 1200);

Более того, элементами списка в правой части оператора IN тоже могут быть выражения над столбцами, как это показано в следующем примере:

**Запрос 17.**

SELECT NAME\_TEACHER, Salary, Salary + Rise FROM TEACHER

WHERE Salary + Rise IN (Salary + 100, Salary + 200, Salary + 300, Salary + 400, Salary + 500);

**Проверка на принадлежность диапазону значений**

Еще одной формой проверки вхождения элемента во множество является проверка на его принадлежность диапазону значений. Для этого применяется предикат BETWEEN, который определяет нахождение значения столбца между указанными минимальным и максимальным значениями. Синтаксис предиката следующий:

имя\_столбца [NOT] BETWEEN минимум AND максимум

Проверять можно значения числовых, строковых и временных типов (для строк символов предполагается алфавитное упорядочение). Оператор BETWEEN является включающим - это означает, что крайние значения диапазона включаются в допустимые.

**Запрос 18**. Вывести фамилии преподавателей со ставкой в диапазоне 1000-2000.

SELECT NAME\_TEACHER FROM TEACHER

WHERE Salary BETWEEN 1000 AND 2000;

**Использование строковых значений**

Использование в операторе BETWEEN в качестве границ диапазона строковых значений имеет особенности, связанные с упорядочением (это же относится и к другим операторам сравнения).

**Запрос 19. Вывести фамилии преподавателей, начинающиеся на буквы от ‘3’ до ‘Л’.**

**SELECT NAME\_TEACHER FROM TEACHER**

**WHERE UPPER(NAME\_TEACHER) BETWEEN 'З' AND 'Л';**

Среди строк результата нет фамилий, начинающихся на букву ‘Л’. Дело в том, что при сравнении строк символов разной длины SQL предварительно дополняет более короткую строку символами пробела, а он в упорядочениях символов предшествует всем остальным. Поэтому строка, состоящая из буквы ‘Л’ (дополненная пробелами), всегда будет меньше любой другой строки, в которой за начальной буквой 'Л’ следуют отличающиеся от пробела символы.

Чтобы это учесть, в качестве верхнего значения диапазона лучше всего указывать следующую по алфавиту букву (в данном случае — ‘М’).

**Использование отрицания**

Так как предикат BETWEEN возвращает истинностное значение, к нему можно применить логическое отрицание. Для этого следует воспользоваться нотацией NOT BETWEEN, в которой предикат будет истинным, только если значение столбца не входит в указанный диапазон. Представление отрицания нотацией NOT BETWEEN введено в язык для большей наглядности, так как с предикатом BETWEEN можно стандартным образом использовать логический оператор NOT (то есть ставить отрицание ко всему выражению, а не к предикату):

NOT (имя\_столбца BETWEEN минимум AND максимум)

Круглые скобки в данном случае можно и опустить, так как они не меняют порядка исполнения операторов. В нотации NOT BETWEEN крайние значения в диапазон не включаются.

**Запрос 20. Вывести названия и номер корпуса кафедр, которые не расположены в корпусах 1 и 3.**

SELECT Name\_Kafedru, NUM\_KORPUSA FROM KAFEDRA

WHERE NUM\_KORPUSA NOT BETWEEN ‘1’ AND '3';

SELECT Name\_Kafedru, NUM\_KORPUSA FROM KAFEDRA

WHERE NOT (NUM\_KORPUSA BETWEEN ‘1’ AND '3');

**Использование выражений над столбцами**

Как и в предикате IN, вместо имени столбца и границ диапазона можно использовать любое допустимое в языке выражение над столбцами таблицы, включая и функции.

**Запрос 21. Показать фамилии преподавателей, принятых на работу между 01.01.2000 и 12.12.2001.**

SELECT NAME\_TEACHER, DATA\_HIRE FROM TEACHER

WHERE DATA\_HIRE BETWEEN ‘01/01/2000’ AND '12/12/2001';

**Запрос 22.** Вывести данные преподавателей, зарплата которых (ставка + надбавка) находится в диапазоне от удвоенной величины надбавки до утроенной надбавки плюс 50.

SELECT NAME\_TEACHER, Salary + Rise, 2 \* Rise, 3 \* Rise + 50

FROM TEACHER

WHERE Salary + Rise BETWEEN 2 \* Rise AND 3 \* Rise + 50;

**Проверка на соответствие шаблону**

Когда необходимо отобрать строки таблицы, в которых значение некоторого столбца совпадает с заданной строкой символов, следует использовать обычное сравнение, как это показано выше. Однако во многих случаях можно не знать точное представление в базе данных интересующего значения. Название одной и той же кафедры, например, может храниться в одном из следующих вариантов: ‘базы данных’, ‘организация баз данных’, ‘информационные системы и базы данных’, ‘базы данных и знаний’.

Такая же ситуация возникает, когда не известно точное написание фамилии преподавателя, название дисциплины, факультета и т. п. Специально для таких случаев предназначен оператор сравнения LIKE, позволяющий отобрать из таблицы строки на основе частичного соответствия. Упрощенный синтаксис оператора следующий:

имя\_столбца [NOT] LIKE шаблон [ESCAPE символ\_пропуска]

Его можно использовать только с символьными значениями.

**Использование шаблона**

Оператор LIKE сравнивает значение столбца с множеством значений, определяемых шаблоном. Он представляет собой строку, в которой помимо обычных символов, составляющих основу поискового выражения, можно использовать так называемые подстановочные символы (иногда они называются групповыми символами). Имеется всего два подстановочных символа, различающихся тем, что именно на их месте может стоять:

% — любая последовательность символов, включая их отсутствие;

\_ — один любой символ.

Подстановочные символы могут находиться в любом месте шаблона в любом наборе.

Например, шаблону '%Иван%' соответствуют строки 'Иван', 'Иванов', 'Иванченко', 'Петр Иванович', а шаблону 'л\_с\_' - 'лист', 'леса', 'лоск' (ноне 'лес', 'листок', 'плес').

Оператор LIKE, как и все другие, работающие с символьными строками, чувствителен к регистру букв, поэтому при его использовании мы рекомендуем использовать уже известные вам функции UPPER() и L0WER().

**Запрос 23.** Найти фамилии преподавателей на букву ‘М’.

SELECT NAME\_TEACHER FROM TEACHER

WHERE UPPER(NAME\_TEACHER) LIKE 'М%';

Имейте в виду, что если вы запишете условие фразы WHERE как UPPER(NAME\_TEACHER) = 'М%' или даже как 'М%' LIKE UPPER(NAME\_TEACHER),

фамилии преподавателей будут сравниваться со строкой ' М%'. Во втором случае выражение является синтаксически правильным оператором LIKE, однако в нем строка 'М%' не выступает в качестве шаблона, так как расположена перед ключевым словом LIKE.

**Запрос 24.** Указать преподавателей, в фамилиях которых первой буквой является ‘М’, а четвертой – ‘ы’.

**SELECT NAME\_TEACHER FROM TEACHER**

WHERE NAME\_TEACHER LIKE 'Мы%’;

**Запрос 25.** Вывести названия кафедр, в которых присутствует словосочетание ‘анализ’ (в различных грамматических формах).

SELECT Name\_Kafedru FROM KAFEDRA

WHERE LOWER(Name\_Kafedru) LIKE '%анализ%';

В левой части оператора LIKE может находиться не только имя столбца, но и любое допустимое над столбцами выражение, как это показано в следующем примере.

**Запрос 26.** Указать преподавателей, в фамилию и название должности которых входит в сумме не меньше пяти букв ‘о’.

SELECT NAME\_TEACHER, DOLGNOST FROM TEACHER

WHERE LOWER(NAME\_TEACHER + DOLGNOST) LIKE

'%o%o%o%o%o%';

**Проверка на неопределенное значение**

Как мы уже отмечали, наличие значения NULL во фразе WHERE приводит к тому, что условие принимает истинностное значение UNKNOWN и соответствующая строка не включается в результат. Детальное описание работы с неопределенным значением вы можете найти в уроке 10, а здесь мы покажем, как обрабатывать значение NULL во фразе WHERE.

Чтобы проверить столбец на неопределенное значение, следует применить унарный оператор IS NULL, имеющий такой синтаксис:

имя\_столбца IS [NOT] NULL

Этот оператор принимает истинностное значение TRUE, если столбец имеет неопределенное значение, и FALSE — в противном случае. В нотации IS NOT NULL его действие обратное.

**Запрос 27**. Вывести фамилии преподавателей, у которых не задан номер телефона или идентификационный код.

**SELECT NAME\_TEACHER, INDEF\_KOD, TEL\_TEACHER FROM TEACHER**

**WHERE INDEF\_KOD IS NULL OR TEL\_TEACHER IS NULL;**

### Задание для лабораторной работы №3 (часть 2)

Для созданной базы данных, согласно номеру варианта, самостоятельно создать **на языке Transact-SQL 15 запросов с отбором строк по условию:**

* 3 простейших запроса с использованием операторов сравнения;
* 3 запроса с использованием логических операторов AND, OR и NOT;
* 2 запроса на использование комбинации логических операторов;
* 2 запроса на использование выражений над столбцами;
* 2 запроса с проверкой на принадлежность множеству;
* 2 запроса с проверкой на принадлежность диапазону значений;
* 2 запроса с проверкой на соответствие шаблону;
* 2 запроса с проверкой на неопределенное значение.

Все программные инструкции команд SQL сохранять в файлах с расширением

**\*.sql** в папке ***ФИО\_студента/Лаб3***

*.*

Для каждого запроса сформулировать текстовое задание, которое должно быть выполнено к базе данных.

Создать текстовый отчет, в котором отобразить sql-команды разработанных запросов и скриншоты результатов работы из СУБД **SQL Server Management Studio**.

## Часть 3. СОЗДАНИЕ МНОГОТАБЛИЧНЫХ ЗАПРОСОВ. ЗАПРОСЫ НА СОЕДИНЕНИЕ

### Цель практической работы

Изучить используемый в реляционных СУБД оператор извлечения данных из таблиц. Получить навыки работы с оператором SELECT в программе ‘SQL Server Management Studio’.

### Исходные данные

Исходными данными является индивидуальное задание и результат предыдущих практических работ.

### Используемые программы

Программы ‘ SQL Server Management Studio ‘.

### Теоретические сведения

При проектировании стремятся создавать таблицы, в каждой из которых содержалась бы информация об одном и только одном типе сущности. Это облегчает модификацию базы данных и поддержку ее целостности. Именно так мы поступили, создавая учебную базу данных. Однако сущности могут быть взаимосвязанными. Кафедры связаны с факультетами по признаку вхождения в их состав, преподаватели работают на кафедрах, студенты учатся на кафедрах и т. д.

Связь между таблицами устанавливается за счет размещения специального столбца первичного ключа одной таблицы, которая называется родительской, в другой таблице, которая называется дочерней. Столбец (или совокупность столбцов) дочерней таблицы, определенный для связи с родительской таблицей, называется внешним ключом.

Наличие внешних ключей является основной для инициирования поиска по многим таблицам.

Одна из наиболее важных особенностей предложения SELECT — это способность использования связей между различными таблицами, а также вывода содержащейся в них информации. Операция, которая приводит к соединению из двух таблиц всех пар строк, для которых выполняется заданное условие, называется соединением таблиц. Для того чтобы указать соединяемые таблицы, их следует перечислить через запятую во фразе FROM.

**Декартово произведение таблиц**

Соединение таблиц - это частный случай операции декартового произведения (или просто произведения). Декартово произведение двух таблиц — это таблица, состоящая из всех возможных пар строк обеих таблиц. Это определение можно естественным образом расширить на любое количество таблиц. В SQL декартово произведение выражается указанием имен перемножаемых таблиц во фразе FROM и указанием всех их столбцов во фразе SELECT.

Так, произведение таблиц FACULTET и KAFEDRA выражается следующим образом:

SELECT \*

FROM FACULTET, KAFEDRA

Так как результирующая таблица содержит много столбцов, и они не помещаются по ширине страницы, мы приведем только интересующие нас столбцы произведения этих таблиц.

**Запрос 28.** Декартово произведение таблиц.

SELECT FACULTET.Name\_faculteta, FACULTET. Kod\_faculteta, KAFEDRA. Kod\_faculteta, KAFEDRA.Name\_Kafedru

FROM FACULTET, KAFEDRA;

Каждая строка таблицы факультетов оказалась соединенной с каждой строкой таблицы кафедр, в результате получилось 27 строк (3 факультета х 9 кафедр = 27 комбинаций).

В произведении может участвовать много таблиц. Например, произведение таблиц факультетов, кафедр и преподавателей записывается следующим образом:

SELECT \*

FROM FACULTET, KAFEDRA, TEACHER

**Условие соединения**

Соединение таблиц может быть указано во фразе WHERE или во фразе FROM. Сначала рассмотрим первый вариант. Большинство запросов, имеющих несколько таблиц во фразе FROM, содержат фразу WHERE, в которой указаны условия, попарно сравнивающие столбцы из различных таблиц. Такое условие называется условием соединения. В этом случае SQL предполагает сцепление только тех пар строк из разных таблиц, для которых условие соединения принимает истинное значение. Теоретически при соединении сначала выполняется декартово произведение указанных таблиц в одну, а затем из нее отбираются строки согласно условию соединения. Естественно, ни одна СУБД не работает таким образом.

Фраза WHERE помимо условия соединения может также содержать другие условия, каждое из которых ссылается на столбцы соединенной таблицы. Эти условия производят отбор строк соединенной таблицы.

Соединения можно разделить на следующие категории.

Внутренние соединения (типичные операции соединения, использующие такие операторы сравнения, как = или <>). Они включают эквивалентные соединения и естественные соединения.

Внутренние соединения используют оператор сравнения для установки соответствия строк из двух таблиц на основе значений общих столбцов в каждой таблице. Примером может быть получение всех строк, в которых идентификационный номер студента одинаковый как в таблице students, так и в таблице courses.

Внешние соединения. Внешние соединения бывают левыми, правыми и полными.

Если внешние соединения задаются в предложении FROM, они указываются с одним из следующих наборов ключевых слов.

LEFT JOIN или LEFT OUTER JOIN

Результирующий набор левого внешнего соединения включает все строки из левой таблицы, заданной в предложении LEFT OUTER, а не только те, в которых соединяемые столбцы соответствуют друг другу. Если строка в левой таблице не имеет совпадающей строки в правой таблице, результирующий набор строк содержит значения NULL для всех столбцов списка выбора из правой таблицы.

RIGHT JOIN или RIGHT OUTER JOIN

Правое внешнее соединение является обратным для левого внешнего соединения. Возвращаются все строки правой таблицы. Для левой таблицы возвращаются значения NULL каждый раз, когда строка правой таблицы не имеет совпадающей строки в левой таблице.

FULL JOIN или FULL OUTER JOIN

Полное внешнее соединение возвращает все строки из правой и левой таблицы. Каждый раз, когда строка не имеет соответствия в другой таблице, столбцы списка выбора другой таблицы содержат значения NULL. Если между таблицами имеется соответствие, вся строка результирующего набора содержит значения данных из базовых таблиц.

**Перекрестные с соединения**

Перекрестное соединение возвращает все строки из левой таблицы. Каждая строка из левой таблицы соединяется со всеми строками из правой таблицы. Перекрестные соединения называются также декартовым произведением.

Таблицы или представления в предложении FROM могут указываться в любом порядке с внутренним соединением или полным внешним соединением. Однако важен порядок таблиц или представлений, заданных при использовании левого или правого внешнего соединения.

**Соединение таблиц по равенству**

Если таблицы соединяются по равенству значений пары столбцов (группы столбцов) из различных таблиц, такая операция называется соединением таблиц по равенству. Соединение по равенству, в отличие от декартового произведения, позволяет соединить только те пары строк, которые действительно взаимосвязаны друг с другом. Так, например, мы можем соединить таблицы факультетов и кафедр по условию FACULTET.Kod\_faculteta = KAFEDRA.Kod\_faculteta. В таком варианте мы соединяем таблицы осмысленно, так как каждая строка таблицы FACULTET соединяется только со строками соответствующих кафедр. На базе таблиц FACULTET и KAFEDRA мы получаем таблицу со столбцами из обеих таблиц, имеющую строки с понятным смыслом. Можно также сказать, что в таблицу KAFEDRA вместо столбца Kod\_faculteta мы вставляем все характеристики (столбцы) соответствующего факультета из таблицы FACULTET.

Соединение таблиц используется, когда необходимо вывести значения столбцов:

* разных таблиц;
* одной таблицы, но отвечающих условию, заданному на другой таблице.

Эти два варианта, а также их комбинация, характерны для любого вида соединения, а не только по равенству. Перейдем к рассмотрению примеров.

**Вывод столбцов разных таблиц**

Этот вид запросов характерен тем, что фраза WHERE содержит только условие соединения, а список фразы SELECT содержит имена столбцов из различных таблиц.

**Запрос 29**. Вывести названия кафедр и номера их групп.

SELECT Name\_Kafedru, [Group] FROM KAFEDRA, STUDENT

WHERE KAFEDRA.kod\_kafedru = STUDENT.kod\_kafedru;

или

SELECT Name\_Kafedru, student.[GROUP] FROM KAFEDRA, STUDENT

WHERE KAFEDRA.kod\_kafedru = STUDENT.kod\_kafedru;

Мы привели два варианта запроса. В первом имена столбцов не уточняются именами таблиц, а во втором — уточняются. В данном случае это не имеет значения, оба запроса корректны.

**Уточнение имен столбцов**

До тех пор, пока запрос относится к одной таблице, обращение к столбцам по их именам не вызывает проблем — в таблице все имена столбцов должны быть неповторяющимися. Однако как только запрос соединяет несколько таблиц, может возникнуть неоднозначность при ссылках на столбцы с одинаковыми именами из разных таблиц. Для разрешения этой неоднозначности во фразах SELECT и WHERE (как и в некоторых других фразах) имена столбцов необходимо уточнять именами таблиц.

**Запрос 30. Вывести названия факультетов и их кафедр.**

SELECT FACULTET.NAME\_FACULTETA, KAFEDRA.Name\_Kafedru FROM FACULTET, KAFEDRA

WHERE FACULTET.Kod\_faculteta = KAFEDRA.Kod\_faculteta;

В этом запросе мы уточнили имена столбцов во фразах SELECT и WHERE, хотя во втором случае это не обязательно, так как используются неповторяющиеся имена. Тем не менее, рекомендуем при соединении таблиц для наглядности уточнять имена столбцов. Обратите внимание на то, что в предыдущем примере отсутствует факультет математики

— на нем нет кафедр.

**Вывод столбцов с условием отбора**

Вариант, когда отбираются строки одной таблицы, а условие задается с участием другой, используется довольно часто. Приведем примеры.

**Запрос 31. Вывести названия кафедр факультета Математики и информатики.**

SELECT KAFEDRA.Name\_Kafedru AS ‘Кафедры факультета математики и информатики’

FROM FACULTET, KAFEDRA

WHERE FACULTET. Kod\_faculteta = KAFEDRA. Kod\_faculteta AND LOWER(FACULTET.NAME\_FACULTETA) = 'математики и информатики';

**Запрос 32. Вывести фамилии доцентов кафедры информатики.**

SELECT TEACHER.NAME\_TEACHER AS ‘Доценты кафедры информатики’ FROM KAFEDRA, TEACHER

WHERE KAFEDRA.kod\_kafedru = TEACHER. kod\_kafedru AND LOWER(KAFEDRA.Name\_Kafedru) = ‘информатики' AND LOWER(TEACHER.DOLGNOST) = 'доцент';

В последнем запросе помимо условия соединения используется также отбор строк по условиям, заданным для разных таблиц.

**Синонимы таблиц**

Синонимы таблиц часто используются для задания более лаконичного имени таблицы, по которому можно сослаться на нее в любых других местах запроса. Приведем пример.

**Запрос 33. Вывести названия кафедр, на которых имеются студенты со стипендией >200 руб.**

SELECT DISTINCT k.Name\_Kafedru FROM KAFEDRA k, STUDENT s

WHERE k.Kod\_kafedru = s. Kod\_kafedru AND s.Stipend > 400;

**Запросы по трем и более таблицам**

SQL позволяет формулировать запросы, которые предполагают использование трех и более таблиц. При этом следует применять ту же методику соединения, что и для двух таблиц. Рассмотрим простой пример соединения трех таблиц.

**Запрос 34. Вывести названия тех кафедр факультета математики и информатики, на которых работают профессора.**

SELECT DISTINCT KAFEDRA.Name\_Kafedru FROM FACULTET, KAFEDRA, TEACHER

WHERE FACULTET.Kod\_faculteta = KAFEDRA.Kod\_faculteta AND KAFEDRA.Kod\_kafedru = TEACHER.Kod\_kafedru AND FACULTET.Name\_faculteta ='Математики и информатики' AND TEACHER.DOLGNOST = 'профессор';

Для ответа на запрос необходимы три таблицы: на таблицах факультетов и преподавателей заданы условия отбора, а из таблицы кафедр следует вывести столбец названий. Поэтому три необходимые таблицы указываются во фразе FROM, а во фразе WHERE производится их соединение по условию равенства первичного и внешнего ключей:

FACULTET. Kod\_faculteta = KAFEDRA. Kod\_faculteta -- соединение таблиц факультетов и кафедр

KAFEDRA. Kod\_kafedru = TEACHER. Kod\_kafedru -- соединение таблиц кафедр и преподавателей

Таблица, образующаяся в результате соединений, будет иметь столько же строк, сколько имеется в таблице преподавателей (если все преподаватели работают на кафедрах). Выясним, почему это так, но сначала заметим, что результат соединения таблиц не зависит от порядка соединения. Поэтому рассмотрим случай, когда сначала мы соединяем таблицы кафедр и преподавателей, а затем результат соединяем с таблицей факультетов.

Так как между таблицами кафедр и преподавателей существует связь типа один- ко-многим, их соединение фактически означает приписывание к строке каждого преподавателя данных о его кафедрах. Количество строк этого соединения будет равным количеству преподавателей. Связь между таблицами факультетов и кафедр также имеет тип один-ко-многим, поэтому второе соединение означает, что к каждой строке таблицы, полученной после первого соединения, приписываются данные о факультете кафедры. Таким образом, количество строк останется равным числу преподавателей.

Вернемся к запросу. Последние два условия фразы WHERE отбирают строки из соединенной таблицы, а во фразе SELECT указан выводимый столбец. Ключевое слово DISTINCT указано в нем потому, что названия кафедр в соединенной таблице могут повторяться.

**Запрос 35. Вывести фамилии ассистентов факультета математики и информатики.**

SELECT TEACHER.NAME\_TEACHER AS ‘Ассистенты ф-та математики и информатики’

FROM FACULTET, KAFEDRA, TEACHER

WHERE FACULTET.Kod\_faculteta = KAFEDRA.Kod\_faculteta AND KAFEDRA.Kod\_kafedru = TEACHER.Kod\_kafedru AND FACULTET.Name\_faculteta ='Математики и информатики' AND TEACHER.DOLGNOST = 'ассистент';

В этом случае для ответа нужны две таблицы — факультетов и преподавателей. Однако они связаны между собой опосредованно, через таблицу кафедр. Поэтому для соединения таблиц факультетов и преподавателей следует использовать таблицу кафедр.

Сформулируем общую процедуру составления многотабличного запроса и приведем пример ее использования.

1. Определить множество таблиц, необходимых для ответа на запрос. В это множество должны входить таблицы, на столбцах которых сформулированы условия, а также те, столбцы которых необходимо вывести. Это так называемые базовые таблицы запроса.
2. В структуре взаимосвязанных таблиц найти путь, соединяющий базовые таблицы. Это так называемый путь вычисления запроса. В результате вы получите перечень таблиц, необходимых для формулировки запроса. Это так называемые таблицы запроса.
3. Во фразе FROM перечислить необходимые таблицы.
4. Во фразе WHERE соединить таблицы запроса и при необходимости задать условия отбора строк в базовых таблицах запроса.
5. Во фразе SELECT перечислить выводимые столбцы.

**Вывод всех столбцов соединяемой таблицы**

В многотабличном запросе конструкция SELECT \* означает выбор всех столбцов соединенной таблицы. Например, результирующая таблица следующего запроса состоит из 21 столбца: 5 столбцов таблицы факультетов, 6 столбцов таблицы кафедр и 10 столбцов таблицы преподавателей.

**Запрос 36. SELECT \***

FROM FACULTET f, KAFEDRA k, TEACHER t

WHERE f.Kod\_faculteta = k.Kod\_faculteta AND k.Kod\_kafedru = t.Kod\_kafedru;

При наличии в запросе многих таблиц конструкция SELECT \* становится не очень практичной. В связи с этим в различных СУБД предоставляется возможность использовать во фразе SELECT многотабличных запросов выражение имя\_таблицы.\* для указания вывода всех столбцов конкретной таблицы. Например:

SELECT f.\*, k.FIO\_ZAVKAF, t.\*

FROM FACULTET f, KAFEDRA k, TEACHER t

WHERE f.Kod\_faculteta = k.Kod\_faculteta AND k.Kod\_kafedru = t.Kod\_kafedru;

**Другие виды соединений no равенству**

Логическая связь между таблицами поддерживается взаимосоответствием столбцов первичного и внешнего ключей. Все рассмотренные до сих под запросы для соединения таблиц использовали именно эту связь. Однако SQL позволяет связывать таблицы по любой паре столбцов, которые имеют сравнимые типы данных, независимо от того, имеет ли эта связь какой-либо смысл. Рассмотрим ряд примеров.

**Запрос 37. Если фамилия заведующего кафедры совпадает с фамилией декана какого-нибудь из факультетов, вывести название этой кафедры вместе с названием соответствующего факультета.**

SELECT k.Name\_Kafedru AS ‘Название кафедры’, f.NAME\_FACULTETA AS ‘Название факультета’

FROM FACULTET f, KAFEDRA k WHERE f.FIO\_DECANA = k.FIO\_ZAVKAF;

**Запрос 38. Вывести пары названий кафедр и фамилий преподавателей, у которых совпадают первичные ключи.**

SELECT k.Name\_Kafedru AS ‘Название кафедры’, t.Name\_Teacher AS ‘Фамилия преподавателя’

FROM KAFEDRA k, TEACHER t

WHERE k.Kod\_kafedru = t.Kod\_kafedru;

Если первый запрос не лишен смысла, то последний абсолютно бессмысленный, так как в учебной базе данных первичные ключи лишены какого-либо содержания и используются только для идентификации строк своих таблиц.

**Самосоединение таблицы**

Как правило, взаимосвязи существуют и в пределах одной таблицы. В одних случаях эти связи являются явными, например, когда внешний ключ ссылается на первичный ключ той же самой таблицы. В других случаях эта связь присутствует неявно, например, кафедры могут быть связаны между собой на основании того свойства, что располагаются в одном корпусе.

Для ответа на такие запросы следует осуществлять соединение таблицы со своей копией. Такое соединение иногда называют самосоединением таблицы. Несмотря на кажущуюся искусственность идеи самосоединения таблиц, существует множество запросов, которые требуют именно такого соединения. На приводимых далее примерах вы убедитесь в этом.

Чтобы произвести соединение таблицы со своей копией, необходимо указать во фразе FROM имя одной и той же таблицы два или большее количество раз, а во фразе WHERE — условие их самосоединения.

Однако в этом случае возникает следующая проблема — как ссылаться на столбцы различных копий таблицы. До сих пор проблема ссылки на столбцы с одинаковыми именами из разных таблиц разрешалась уточнением имени столбца именем таблицы. В нашем же случае соединяемые таблицы имеют одинаковые имена. Для разрешения этой проблемы без синонимов таблиц уже не обойтись. В нашем случае различным вхождениям одной и той же таблицы приписываются различные синонимы и именно по этим синонимам производится обращение к столбцам. Приведем примеры использования самосоединения.

**Запрос 39. Вывести фамилии преподавателей, зарплата которых больше, чем у преподавателя Сидорова.**

SELECT needed.NAME\_TEACHER

FROM TEACHER needed, TEACHER given

WHERE needed.Salary + needed.Rise > given.Salary + given.Rise AND given.NAME\_TEACHER = 'Игнатьева Олеся Владимировна';

**Симметричное соединение и удаление избыточности**

При самосоединении по равенству обычно возникают избыточные строки.

Рассмотрим следующий запрос.

**Запрос 40. Вывести названия кафедр, располагающихся в том же корпусе, что и кафедра информатики.**

SELECT needed.Name\_Kafedru

FROM KAFEDRA needed, KAFEDRA given

WHERE needed.NUM\_KORPUSA = given.NUM\_KORPUSA AND

given.Name\_Kafedru = 'Информатики';

Обратите внимание, что в результат включена и сама кафедра информатики. Для того чтобы избавиться от ненужной результирующей строки, следует добавить условие отбора:

needed.Name\_Kafedru <> ' Информатика’

При самосоединении по равенству можно получить симметричную результирующую таблицу. Суть симметричности заключается в том, что в таблице содержатся строки:

с одинаковыми значениями всех столбцов;

со всеми возможными перестановками значений столбцов.

**Запрос 41. Вывести пары номеров групп, которые принадлежат одной кафедре.**

SELECT g1.[Group], g2.[Group], g1.kod\_kafedru FROM STUDENT g1, STUDENT g2

WHERE g1.kod\_kafedru = g2.kod\_kafedru;

ПРИМЕЧАНИЕ В этом запросе мы дополнительно вывели столбец с номером (первичным ключом) кафедры для большей наглядности.

Результирующая таблица оказалась симметричной, и в связи с этим содержит избыточные строки.

Простой способ избежать этого состоит в том, чтобы наложить ограничение на выбираемые пары значений таким образом, чтобы первое выдаваемое значение было меньше другого (или предшествовало ему в алфавитном порядке). Это делает результат асимметричным, поэтому пары с одинаковыми значениями, а также пары, заданные в обратном порядке, не будут появляться. Покажем это на примере варианта предыдущего запроса.

SELECT g1.[Group], g2.[Group], g1.kod\_kafedru FROM STUDENT g1, STUDENT g2

WHERE g1.kod\_kafedru = g2.kod\_kafedru AND g1.[Group] <g2.[Group];

**Проверка правильности данных**

амосоединение можно использовать для проверки корректности данных. Например, мы точно знаем, что в нашем вузе нет однофамильцев, занимающих разные должности. С помощью самосоединения таблицы преподавателей мы можем убедиться, что их нет и в базе данных.

**Запрос 42. Указать преподавателей-однофамильцев, которые занимают различные должности.**

SELECT tch1.NAME\_TEACHER AS ‘Препод. с различ. должностями’ FROM TEACHER tch1, TEACHER tch2

WHERE tch1.NAME\_TEACHER = tch2.NAME\_TEACHER AND tch1.DOLGNOST <> tch2.DOLGNOST;

**Внешнее соединение таблиц**

Предположим, необходимо вывести список факультетов и их кафедр. Это достигается соединением таблиц FACULTET и KAFEDRA по равенству значений первичного и внешнего ключей и выбором столбцов с названиями факультетов и кафедр. Но в таком случае, если на факультете кафедр нет, он не будет включен в результат.

Для того чтобы в списке присутствовали все факультеты, даже без кафедр, необходимо использовать внешнее соединение, которое расширяет возможности обычного соединения. Внешнее соединение возвращает строки, которые удовлетворяют условию соединения, а также те строки одной из таблиц, для которых в другой не нашлось удовлетворяющих условию соединения строк.

Внутренние соединения возвращают результат, когда в обеих таблицах есть хотя бы одна строка, соответствующая условиям соединения. Внутренние соединения исключают строки, не соответствующие ни одной строке в другой таблице. Однако внешние соединения возвращают все строки хотя бы из одной таблицы или представления, упомянутых в предложении FROM, если они удовлетворяют условиям поиска WHERE или HAVING.

Все строки, получаемые из левой таблицы, образуют левое внешнее соединение, а строки, получаемые из правой таблицы, — правое внешнее соединение. Все строки их обеих таблиц возвращаются в полном внешнем соединении.

Для внешних соединений в предложении FROM SQL Server использует ключевые слова ISO:

LEFT OUTER JOIN или LEFT JOIN;

RIGHT OUTER JOIN или RIGHT JOIN;

FULL OUTER JOIN или FULL JOIN.

**Работа с левыми внешними соединениями**

Рассмотрим примеры.

Рассмотрим соединение таблиц KAFEDRA и TEACHER по столбцам kod\_kafedru. В результате будут выведены только те кафедры, для которых были написаны преподаватели.

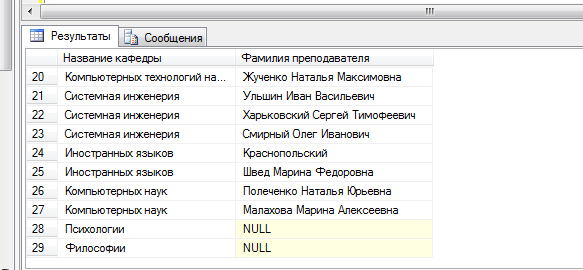
Чтобы включить в результаты все кафедры, независимо от того, были ли написаны их преподаватели, используйте левое внешнее соединение ISO. Пример запроса:

**Запрос 43**. Вывести фамилии всех преподавателей с указанием их кафедры, если она есть.

SELECT KAFEDRA.Name\_Kafedru AS 'название кафедры', TEACHER.NAME\_TEACHER AS 'фамилия преподавателя'

FROM KAFEDRA LEFT OUTER JOIN TEACHER

ON KAFEDRA.kod\_kafedru = TEACHER.kod\_kafedru;

Ключевые слова **LEFT OUTER JOIN** включают в вывод все строки таблицы KAFEDRA независимо от того, есть ли для них соответствующие значения в столбце kod\_kafedru таблицы TEACHER. Обратите внимание на то, что в результатах, где для кафедры нет соответствующего преподавателя, строки содержат значение NULL в столбце Фамилия преподавателя.

**Работа с правыми внешними соединениями**

Рассмотрим соединение таблиц **KAFEDRA** и **TEACHER** по столбцам **kod\_kafedru**. Оператор правого внешнего соединения ISO, **RIGHT OUTER JOIN**, включает в результаты все строки второй таблицы независимо от того, есть ли для них совпадающие данные в первой таблице.

Чтобы включить в результаты всех преподавателей независимо от того, есть ли связанные с ними кафедры, используйте правое внешнее соединение ISO. Пример запроса Transact-SQL и результаты правого внешнего соединения:

**Запрос 44. Вывести названия всех кафедр с указанием фамилий преподавателей, если они есть.**

SELECT KAFEDRA.Name\_Kafedru AS 'название кафедры', TEACHER.NAME\_TEACHER AS 'фамилия преподавателя'

FROM KAFEDRA RIGHT OUTER JOIN TEACHER

ON KAFEDRA.kod\_kafedru = TEACHER.kod\_kafedru;

**Внешнее соединение и условие отбора**

При внешнем соединении можно применять и дополнительные условия отбора строк. Как видно из следующих двух примеров, если условие относится к столбцам таблицы, к которой не применяется оператор внешнего соединения, то внешнее соединение происходит.

**Запрос 45. Вывести названия всех кафедр корпуса 1 с указанием их преподавателей, если они есть.**

SELECT KAFEDRA.Name\_Kafedru AS 'название кафедры', TEACHER.NAME\_TEACHER AS 'фамилия преподавателя'

FROM KAFEDRA LEFT OUTER JOIN TEACHER

ON KAFEDRA.kod\_kafedru = TEACHER.kod\_kafedru

WHERE KAFEDRA.NUM\_KORPUSA = '1';

**Запрос 46. Вывести названия всех кафедр с указанием их преподавателей, если они есть, ставка которых больше 3000.**

SELECT KAFEDRA.Name\_Kafedru AS 'Название кафедры', TEACHER.NAME\_TEACHER AS 'Фамилия преподавателя'

FROM KAFEDRA RIGHT OUTER JOIN TEACHER

ON KAFEDRA.kod\_kafedru = TEACHER.kod\_kafedru

WHERE TEACHER.salary > 3000;

**Работа с полными внешними соединениями**

Чтобы сохранить в выводе не соответствующие друг другу строки из обеих таблиц, включив их в результаты соединения, используйте полное внешнее соединение. SQL Server предоставляет оператор полного внешнего соединения, **FULL OUTER JOIN**, включающий все строки из обеих таблиц вне зависимости от того, есть ли в них совпадающие значения.

**Использование перекрестных соединений**

Перекрестное соединение, не имеющее предложения WHERE, выполняет декартово произведение таблиц, вовлеченных в объединение. Размер результирующего набора декартова произведения вычисляется, как произведение количества строк в первой таблице на количество строк во второй таблице. Следующий пример показывает перекрестное соединение Transact-SQL.

SELECT KAFEDRA.Name\_Kafedru AS 'название кафедры', TEACHER.NAME\_TEACHER AS 'фамилия преподавателя'

FROM KAFEDRA CROSS JOIN TEACHER

ORDER BY KAFEDRA.kod\_kafedru;

Результирующий набор содержит 297 строк (в **KAFEDRA** имеется 11 строк, а в таблице

**TEACHER** существует 27 строк; 11, умноженное на 27, равно 297).

**Внешнее соединение трех и более таблиц**

В запросе может быть использовано внешнее соединение более чем двух таблиц.

Хотя в операции соединения указываются всего две таблицы, предложение FROM может содержать несколько операций объединения. Это позволяет соединять в одном запросе несколько таблиц.

При этом следует помнить, что если к столбцу таблицы А применен оператор внешнего соединения с таблицей В, то никакой другой столбец таблицы А не может содержать оператор внешнего соединения с таблицей, отличающейся от В.

В следующем примере внешнее соединение применяется для трех таблиц — факультетов, кафедр и преподавателей.

**Запрос 47. Вывести список всех факультетов с указанием их кафедр и преподавателей.**

SELECT f.NAME\_FACULTETA AS 'Факультет',

k.Name\_Kafedru AS 'Кафедра', t.NAME\_TEACHER AS 'Преподаватель' FROM FACULTET f JOIN KAFEDRA k

ON f.kod\_faculteta =k.kod\_faculteta JOIN TEACHER t

ON k.kod\_kafedru = t.kod\_kafedru;

Следующий запрос Transact-SQL выполняет поиск наименований всех факультетов определенной кафедры и имена преподавателей этих кафедр.

Обратите внимание, что ни один из соединяемых столбцов — ни kod\_faculteta, ни kod\_kafedru, не включается в результаты. Тем не менее, соединение возможно только при использовании Kafedrа в качестве промежуточной таблицы.

Среднюю таблицу соединения, Kafedrа, можно назвать таблицей преобразования, или промежуточной таблицей, так как Kafedrа является промежуточной точкой объединения, которая находится между двумя другими участвующими в объединении таблицами.

При наличии в инструкции нескольких операторов соединения, применяющихся либо при соединении более двух таблиц, либо при соединении более двух пар столбцов, выражения соединения могут быть связаны операторами AND или OR.

### Задание для лабораторной работы №3 (часть 3)

Для созданной базы данных, согласно номеру варианта, самостоятельно создать **на языке Transact-SQL многотабличные запросы:**

* 2 запроса с использованием декартового произведения двух таблиц;
* 3 запроса с использованием соединения двух таблиц по равенству;
* 2 запроса с использованием соединения двух таблиц по равенству и условием отбора;
* 2 запроса с использованием соединения по трем таблицам;
* создать копии ранее созданных запросов на соединение по равенству на запросы с использованием внешнего полного соединения таблиц (**JOIN**).
* 2 запроса с использованием левого внешнего соединения;
* 2 запроса на использование правого внешнего соединения;
* 2 запроса с использованием симметричного соединения и удаление избыточности.

Все программные инструкции команд SQL сохранять в файлах с расширением

**\*.sql** в папке ***ФИО\_студента/Лаб3****.*

Для каждого запроса сформулировать текстовое задание, которое должно быть выполнено к базе данных.

Создать текстовый отчет, в котором отобразить sql-команды разработанных запросов и скриншоты результатов работы из СУБД **SQL Server Management Studio**.

## Часть 4. СОЗДАНИЕ ЗАПРОСОВ НА ГРУППИРОВКУ И СОРТИРОВКУ ДАННЫХ. ЗАПРОСЫ НА ИЗМЕНЕНИЕ. ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ ФУНКЦИЙ.

### Цель работы

Изучить используемый в реляционных СУБД оператор извлечения данных из таблиц SELECT и выполнение группировки и сортировки данных. Изучить синтаксис языка модификации данных. Научится использовать встроенные функции в запросах.

### Исходные данные

Исходными данными является индивидуальное задание и результат предыдущих практических работ.

### Используемые программы

Программа SQL Server Management Studio.

### Задание

Практическую работу следует выполнять в следующем порядке:

1. Изучить синтаксис создания запросов с использованием функций, группировки и сортировки данных, язык манипулирования данных на примерах запросов, использовать встроенные функции к учебной базе данных "**University.mdf**".
2. Выполнить в окне SQL Editor **запросы** к базе данных "**University.mdf**", согласно приведенным в практической работе образцам выполнения запросов и сохранить их в файле "**Lab3.sql**" в своей рабочей папке.

**Тема 1. Создание запросов с использованием функций**

Функции SQL подобны любым другим операторам языка в том смысле, что они производят действия с данными и возвращают результат в качестве своего значения. Функции имеют тип, который определяется типом возвращаемого значения, поэтому можно говорить о числовых, строковых, временных функциях и т. д. От обычных операторов функции отличаются форматом представления:

**имя\_функции[(аргумент[, аргумент]...)]**

Этот формат допускает, что функции могут иметь ноль, один или более аргументов, причем при отсутствии аргументов круглые скобки не используются.

Имеется два основных класса функций SQL: встроенные и определяемые пользователем.

Встроенными являются функции, предопределенные в SQL. Ко второму классу относятся функции, которые пишутся пользователями на специальном языке, обеспечивающем использование всех возможностей SQL. Каждая СУБД использует для этого свой собственный язык.

SQL Server содержит множество встроенных функций, а также поддерживает создание определяемых пользователем функций.

В SQL определено множество встроенных функций различных категорий. На этом уроке мы рассмотрим:

–агрегатные (или групповые) функции, оперирующие значениями столбцов множества строк и возвращающие одно значение;

–функции одной строки, использующие в качестве аргументов значения столбцов одной строки и возвращающие одно значение.

**Встроенные функции (Transact-SQL)**

SQL Server содержит множество встроенных функций, а также поддерживает создание определяемых пользователем функций. Категории встроенных функций перечислены на этой странице.

Типы функций

|  |  |
| --- | --- |
| Функция | Описание |
| Функции, возвращающие наборы  строк. | Возвращают объект, который можно использовать  так же, как табличные ссылки в SQL-инструкции. |
| Агрегатные функции | Обрабатывают коллекцию значений и возвращают  одно результирующее значение. |
| Ранжирующие функции | Возвращают ранжирующее значение для каждой  строки в секции. |
| Скалярная функция (описывается далее) | Обрабатывают и возвращают одиночное  значение. Скалярные функции можно применять везде, где выражение допустимо. |

Скалярные функции

|  |  |
| --- | --- |
| Категория функции | Описание |
| Функции конфигурации | Возвращают сведения о текущей конфигурации. |
| Функции преобразования | Поддержка приведения и преобразования типов  данных. |
| Функции работы с курсорами | Возвращают сведения о курсорах. |
| Функции и типы данных даты и времени | Выполняют операции над исходными значениями даты  и времени, возвращают строковые и числовые значения, а также значения даты и времени. |
| Логические функции | Выполнение логических операций. |
| Математические функции | Выполняют вычисления, основанные на числовых значениях, переданных функции в виде аргументов, и  возвращают числовые значения. |
| Функции метаданных | Возвращают сведения о базах данных и объектах баз  данных. |
| Функции безопасности | Возвращают данные о пользователях и ролях. |
| Строковые функции | Выполняют операции со строковым (char или varchar) исходным значением и возвращают строковое или  числовое значение. |
| Системные функции | Выполняют операции над значениями, объектами и параметрами экземпляра SQL Server и возвращают  сведения о них. |
| Системные статистические  функции | Возвращают статистические сведения о системе. |
| Функции обработки текста и изображений | Выполняют операции над текстовыми или графическими исходными значениями или столбцами и  возвращают сведения о |

**Агрегатные функции**

Аргументами агрегатных функций могут быть как столбцы таблиц, так и результаты выражений над ними. Агрегатные функции и сами могут включаться в другие арифметические выражения. В стандарте SQL определены следующие виды агрегатных функций: унарные, бинарные, инверсного распределения, гипотетические функции множеств.

Мы будем рассматривать только определенные в стандарте SQL унарные агрегатные функции. Их перечень представлен в табл. 1.1. Конкретные СУБД расширяют этот список.

AVG – среднее MIN - минимум

CHECKSUM\_AGG – Возвращает контрольную сумму значений в группе.

Значения NULL не учитываются.

SUM – сумма

COUNT – количество

STDEV – среднее квадратическое отклонение

COUNT\_BIG – Возвращает количество элементов в группе.

STDEVP – Возвращает статистическое стандартное отклонение всех значений в указанном выражении.

GROUPING – Указывает, является ли указанное выражение столбца в списке GROUP BY статистическим или нет. В результирующем наборе функция GROUPING возвращает 1 (статистическое выражение) или ноль (нестатистическое выражение).

VAR – дисперсия

GROUPING\_ID – Представляет собой функцию, которая вычисляет уровень группирования.

VARP – Возвращает статистическую дисперсию для заполнения всех значений в указанном выражении.

MAX – максимум

Общий формат унарной агрегатной функции следующий:

**имя\_функции([ALL | DISTINCT] выражение) [FILTER (WHERE условие)]**

где **DISTINCT** указывает, что функция должна рассматривать только различные значения аргумента, a ALL — все значения, включая повторяющиеся (этот вариант используется по умолчанию). Фраза FILTER позволяет дополнительно отобрать строки таблицы, столбец которой используется в качестве аргумента функции.

Агрегатные функции применяются во фразах SELECT и HAVING. Здесь мы рассмотрим их использование во фразе SELECT. В этом случае выражение в аргументе функции применяется ко всем строкам входной таблицы фразы SELECT. Кроме того, во фразе SELECT нельзя использовать и агрегатные функции, и столбцы таблицы (или выражения с ними) при отсутствии фразы GROUP BY, которую мы рассмотрим в теме 2.

**Функция COUNT**

Функция COUNT имеет два формата. В первом случае возвращается количество строк входной таблицы, во втором случае — количество значений аргумента во входной таблице:

**C0UNT(\*)**

**C0UNT([DISTINCT | ALL] выражение)**

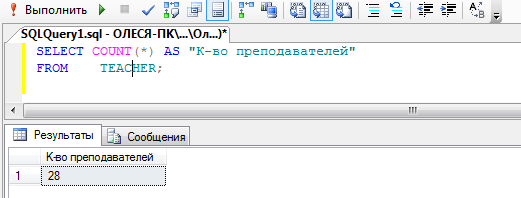
Простейший способ использования этой функции - подсчет количества строк в таблице (всех или удовлетворяющих указанному условию). Для этого используется первый вариант синтаксиса.

Создайте новый запрос, введите sql-запрос, выполните его, сохраните его в рабочую папку ЛАБ3\_SQL под именем 1.sql.

**Запрос 1. Информация о скольких преподавателях имеется в базе данных?** SELECT COUNT(\*) AS "К-во преподавателей"

FROM TEACHER;

Чтобы выполнить sql-команду нажмите на панели редактора кнопку Выполнить. В результате выполнения данного кода будет подсчитано кол-во всех преподавателей. Например,



**Самостоятельно** создать запрос 2. Сколько ассистентов не имеют телефонов?

Запросы с агрегатными функциями можно строить и с использованием нескольких таблиц, так как входная таблица и в этом случае будет только одна.

**Самостоятельно** создать запрос 3. Сколько кафедр на факультете математики и информатики?

Во втором варианте синтаксиса функции COUNT в качестве аргумента может быть использовано имя отдельного столбца. В этом случае подсчитывается количество либо всех значений в этом столбце входной таблицы, либо только неповторяющихся (при использовании ключевого слова DISTINCT).

**Запрос 4. На скольких различных должностях работают преподаватели кафедры «Компьютерные системы и сети»?**

SELECT C0UNT(DISTINCT DOLGNOST) FROM KAFEDRA d, TEACHER t

WHERE d.KOD\_KAFEDRU = t.KOD\_KAFEDRU AND

LOWER(d.NAME\_KAFEDRU) = 'Компьютерные системы и сети';

**Функция SUM**

Эта агрегатная функция подсчитывает сумму значений аргумента для всех строк входной таблицы. Аргумент должен иметь числовой тип или быть временным промежутком. В качестве аргумента может выступать имя столбца или выражение над столбцами входной таблицы. В этой функции также допускается использовать ключевые слова DISTINCT и ALL. Приведем примеры.

**Запрос 5. Какая суммарная ставка всех ассистентов?**

SELECT SUM(Salary)

FROM TEACHER

WHERE LOWER(DOLGNOST) = 'ассистент';

**Функция AVG**

Агрегатная функция AVG подсчитывает среднее значение аргумента для всех строк входной таблицы. Аргумент должен иметь числовой тип или быть временным промежутком. В качестве аргумента может выступать имя столбца или выражение над столбцами входной таблицы. Допускается использовать ключевые слова DISTINCT и ALL. Приведем ряд примеров.

**Самостоятельно** создать запрос 6. Какая средняя ставка среди всех преподавателей?

**Самостоятельно** создать запрос 7. Какое среднее значение ставки в вузе?

В данном запросе используйте ключевое слово DISTINCT, чтобы применить AVG не ко всем имеющимся в таблице TEACHER ставкам, а только к различным значениям ставки.

**Функции MIN и MAX**

Эти функции позволяют находить максимальное (МАХ) и минимальное (MIN) значения аргумента для всех строк входной таблицы. Хотя и в этом допускается использование ключевых слов DISTINCT и ALL, они не оказывают влияния на результат. Аргумент этих функций может быть любого типа, для которого определено упорядочение, то есть числовой, строковый и временной.

**Запрос 8. Какова максимальная зарплата преподавателя?**

SELECT MAX(Salary + Rise)

FROM TEACHER;

**Самостоятельно** создать запрос 9. Когда в последний раз (максимальная дата приема на работу) принимали на работу преподавателя на кафедру информатики?

**Выражения с использованием агрегатных функций**

Агрегатные функции не только могут иметь выражение в своем аргументе, но и сами могут использоваться в выражениях.

**Запрос 10. Вывести процентное соотношение суммарной ставки к суммарной зарплате и наоборот.**

SELECT SUM(Salary)\*100/SUM(Rise) AS

"Процент зарплаты к зарплате", SUM(Rise)\*100/SUM(Salary) AS "Процент зарплаты к ставке"

FROM TEACHER;

**Однострочные функции**

Напомним, что эти функции используют в качестве аргумента одно значение (одного столбца одной строки таблицы) и возвращают в качестве своего результата также единственное значение. Мы рассмотрим эти функции по типам их аргументов.

**Строковые функции**

Эти функции используют в качестве аргумента строку символов и в качестве результата возвращают также символьную строку. Стандарт SQL предлагает варианты таких функций и для двоичных строк.

**Функции UPPER, LOWER**

Эти функции мы уже рассматривали и многократно использовали. Они имеют следующий формат:

**UPPER(cтрокa)**

**LOWER(cтрокa)**

Приведем для них примеры .

**Запрос 11**. Вывести фамилии всех преподавателей прописными буквами.

SELECT UPPER(NAME\_TEACHER) AS "Все прописные”

FROM TEACHER;

Аналогично можно вывести все фамилии преподавателей строчными буквами.

**Числовые функции над числами**

Эти функции возвращают числовые значения на основании заданных в аргументе значений того же типа. Числовые функции используются для обработки данных, а также в условиях их поиска. Стандарт SQL предлагает ряд числовых функций с очевидной семантикой. Часть функций перечислены ниже:

ABS абсолютное значение

DEGREES Возвращает для значения угла в радианах соответствующее значение в градусах.

RAND – Возвращает псевдослучайное значение типа float от 0 до 1. EXP экспонента

ROUND - Возвращает числовое значение, округленное до указанной длины или точности.

FLOOR Возвращает наибольшее целое число, меньшее или равное указанному числовому выражению.

LOG логорифм SIN - синус

LOG10 десятичный логорифм SQRT – корень квадратный

PI число 3.14 SQUARE – квадрат числа

POWER Возвращает значение указанного выражения, возведенное в заданную степень.

TAN - тангенс

Особой функцией является WIDTH\_BUCKET, с помощью которой можно легко строить гистограммы:

WIDTH\_BUCKET(число, миминум, максимум, количество)

Некоторые СУБД расширяют приведенный выше набор функций, включая другие числовые функции, например вычисления обычных и гиперболических тригонометрических функций.

**Временные функции**

Эти функции используют в качестве аргумента типы даты, времени, временной отметки или временного промежутка. Тип возвращаемого значения не всегда соответствует типу аргумента.

Функции даты и времени в Transact-SQL деляться на Функции, получающие значения системной даты и времени Функции, получающие компоненты даты и времени

Функции, получающие значения даты и времени из их компонентов Функции, получающие разность даты и времени

Функции, изменяющие значения даты и времени

Функции, устанавливающие или получающие формат сеанса Функции, проверяющие значения даты и времени.

**Функции, получающие компоненты даты и времени.**

Функция извлекает из операнда указанный компонент и возвращает его в виде

DATENAME ( datepart , date )

Здесь date - это выражение временного типа, а datepart - временная единица, которая может иметь одно из следующих значений: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND и т.д.

DATEPART ( datepart,date ) - Возвращает целое число, представляющее указанный компонентdatepart указанной даты date.

DAY (date) - Возвращает целое число, представляющее день указанной даты date.

MONTH ( date ) - Возвращает целое число, представляющее месяц указанной даты date.

YEAR (date) - Возвращает целое число, представляющее год указанной даты date.

Рассмотрим пример.

**Запрос 12. Вывести фамилии всех преподавателей, родившихся в 1979 году**.

SELECT Name\_teacher, BIRTHDAY

FROM TEACHER

WHERE DATENAME(YEAR, BIRTHDAY)=1979;

**Функции, получающие значения системной даты и времени**

Функция CURRENT\_TIMESTAMP - Возвращает значение типа datetime2(7), которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Смещение часового пояса не включается.

Эта функция возвращает текущую дату. Аргументов она не имеет.

Функция GETDATE ( ) Возвращает значение типа datetime2(7), которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Смещение часового пояса не включается.

Функция GETUTCDATE ( ) Возвращает значение типа datetime2(7), которое содержит дату и время компьютера, на котором запущен экземпляр SQL Server. Возвращаемые дата и время отображаются в формате UTC.

Многие СУБД существенно расширяют список функций, оперирующих датой и временем. Далее мы приведем некоторые из важных функций этого типа, которые используются в Oracle. Напоминаем, что тип DATE в Oracle содержит в себе как дату, так и время.

**Функции, получающие значения даты и времени из их компонентов**

Функция DATEADD (datepart, number , date )

Возвращает новое значение datetime, добавляя интервал к указанной части datepart заданной даты date.

Добавляет к дате, указанной в первом аргументе, количество месяцев второго аргумента.

**Dateadd (компонент, кол-во , дата)**

Здесь кол-во - это количество прибавляемых лет, месяцев, дней и т.д., а компонент

* временная единица, которая может иметь одно из следующих значений: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND.

Например, DATEADD(month, 1, '2006-08-30')

**Запрос 13. Осуществить пересчет даты приема на работу преподавателя на фамилию, начинающуюся на букву C в сторону увеличения на 3 месяца.**

SELECT NAME\_TEACHER, DATA\_HIRE AS ' Дата приема ',

DATEADD(month, 3, DATA\_HIRE) AS ' Плюс 3 месяца '

FROM TEACHER

WHERE (NAME\_TEACHER) LIKE 'С%';

**Функция EOMONTH**

**EOMONTH (start\_date [, month\_to\_add ])**

Возвращает дату последнего дня того месяца, который указан в аргументе. Обычно используется для определения, сколько дней осталось до конца месяца.

**LAST\_DAY(дата)**

**Функция DATEDIFF**

DATEDIFF (datepart, startdate, enddate)

Возвращает количество пересеченных границ (целое число со знаком), указанных аргументом datepart, за период времени, указанный аргументами startdate и enddate.

**Запрос 14. Например, если вы хотите узнать, сколько месяцев уже проработал Статывка, можно выполнить такой запрос:**

SELECT 'Статывка проработал ' || ROUND(DATEDIFF(month,GETDATE(), DATA\_HIRE),1) ||'месяцев' AS "Стаж Статывки" FROM TEACHER

WHERE NAME\_TEACHER LIKE 'Статыв%';

**Функция NEXT\_DAY**

Возвращает ближайшую к первому параметру дату, в которой название дня недели совпадает с указанным во втором параметре.

**NEХТ\_DАУ(дата, день\_недели)**

**Функции преобразования**

Стандарт SQL предлагает единственную функцию преобразования данных из одного типа в другой — это функция CAST.

**Функция CAST и CONVERT**

Производит преобразование выражения, заданного первым аргументом, в тип, заданный вторым аргументом. Преобразование допускается только для определенных пар типов данных.

CAST(expression AS data\_type [ ( length ) ])

CONVERT(data\_type [ ( length ) ] , expression [ , style ])

Например

CAST(10.3496847 AS money)

CAST(10.6496 AS int)

**Тема 2. Группировка и сортировка**

В этом уроке мы рассмотрим еще три фразы предложения SELECT, а именно:

**HAVING, GROUP BY и ORDER BY.**

Первая из них позволяет группировать строки таблицы и применять к созданным группам агрегатные функции.

Рассмотрим простейшие варианты группировки. Фраза HAVING используется вместе с фразой GROUP BY и позволяет формулировать условия на группах строк для дополнительного отбора.

Наконец, фраза ORDER BY позволяет сортировать строки результирующей таблицы.

**Запросы с группировкой строк**

Часто при создании отчетов появляется необходимость в формировании промежуточных итоговых значений, то есть относящихся к данным не всей таблицы, а ее частей.

Именно для этого предназначена фраза GROUP BY. Она позволяет все множество строк таблицы разделить на группы по признаку равенства значений одного или нескольких столбцов (и выражений над ними).

Фраза GROUP BY должна располагаться вслед за фразой WHERE (если она отсутствует, то за фразой FROM).

Общий синтаксис фразы GROUP BY следующий:

**GROUP BY выражение[, выражение]...**

При наличии фразы GROUP BY фраза SELECT применяется к каждой группе, сформированной фразой группировки. В этом случае и действие агрегатных функций, указанных во фразе SELECT, будет распространяться не на всю результирующую таблицу, а только на строки в пределах каждой группы. Каждое выражение в списке фразы SELECT должно принимать единственное значение для группы, то есть оно может быть:

* константой;
* агрегатной функцией, которая оперирует всеми значениями аргумента в пределах группы и агрегирует их в одно значение (например, в сумму);
* выражением, идентичным стоящему во фразе GROUP BY;
* выражением, объединяющим приведенные выше варианты.

Рассмотрим возможности фразы GROUP BY, переходя от простых вариантов ее использования к более сложным.

**Группировка по одному столбцу**

Группировка по значениям одного столбца является самым простым вариантом использования фразы GROUP BY. Приведем примеры.

**Запрос 15. Для каждого корпуса подсчитать количество находящихся в нем кафедр.**

SELECT NUM\_KORPUSA AS “Корпус”,

СOUNT(\*) AS "K-вo кафедр" FROM KAFEDRA

GROUP BY NUM\_KORPUSA ;

**Самостоятельно** создать запрос 16. Для каждой из должностей указать суммарный фонд заработной платы.

Если в запросе используются фразы и WHERE, и GROUP BY, строки, не удовлетворяющие условию фразы WHERE, исключаются до выполнения группировки. Вследствие этого группировка производится только по тем строкам, которые удовлетворяют условию.

В случае многотабличных запросов сначала производится соединение таблиц, а затем их группировка. Приведем примеры.

**Самостоятельно** создать запрос 17. Для каждого факультета, расположенного в корпусе 1, вывести количество групп и общее количество студентов по каждой кафедре.

**Группировка по нескольким столбцам**

SQL позволяет группировать строки таблицы и по нескольким столбцам. В этом случае имена столбцов перечисляются во фразе GROUP BY через запятую.

**Запрос 18. Для каждого факультета, расположенного в корпусе 1, вывести сколько учится студентов по каждой группе.**

SELECT f.Name\_faculteta,

s."GROUP", count(s."GROUP") AS "Кол-во студентов в группе" FROM FACULTET f, KAFEDRA d, STUDENT s

WHERE f.KOD\_FACULTETA = d.KOD\_FACULTETA AND

d.KOD\_kafedru = s.KOD\_kafedru AND d.NUM\_KORPUSA = '1'

GROUP BY f.Name\_faculteta,s."GROUP";

**Самостоятельно** создать запрос 19. Для каждой кафедры и должности вывести суммарную и среднюю зарплату преподавателей.

Даже при группировке по двум и более столбцам этот вариант фразы GROUP BY обеспечивает только один уровень группировки. Так, приведенный выше запрос обеспечивает только одну итоговую строку для пары значений кафедра-должность.

**Использование выражений**

Хотя стандарт SQL не допускает группировку по выражениям над столбцами, некоторые СУБД такую возможность предоставляют. В этом случае во фразе SELECT также можно использовать выражение группировки, однако нельзя выводить по отдельности столбцы, участвующие в этом выражении.

**Запрос 20**. Для каждого значения зарплаты, не превышающего 1500, вывести это значение и количество преподавателей, такую зарплату получающих.

SELECT Salary + Rise, C0UNT(\*) FROM TEACHER

WHERE Salary + Rise <= 1500 GROUP BY Salary + Rise;

**Вложение агрегатных функций**

Если фраза GROUP BY в запросе отсутствует, то во фразе SELECT нельзя вкладывать агрегатные функции друг в друга. Например, следующий запрос приведет к ошибке:

SELECT AVG(MIN(Salary)) FROM TEACHER;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - -

ORA-00978: вложенная групповая функция без GROUP BY

Однако при наличии фразы GROUP BY такое вложение допускается. Оно интерпретируется следующим образом: сначала для каждой группы выполняется вложенная агрегатная функция, затем к полученной таким образом промежуточной таблице применяется внешняя агрегатная функция. Двойное вложение, например MAX(AVG(MIN(Salary))), недопустимо. Приведем пример.

**Запрос 21**. Вывести среднее значение среди минимальных и максимальных ставок для каждой группы преподавателей, занимающих одну должность, а также минимальное и максимальное значения среди средних ставок.

SELECT AVG(MIN(Salary)) AS AVG\_MIN, AVG(MAX(Salary)) AS AVG\_MAX, MIN(AVG(Salary)) AS MIN\_AVG, MAX(AVG(Salary)) AS MAX\_AVG

FROM TEACHER GROUP BY Dolgnost;

**Условие отбора групп**

Предположим, что нужно вывести номера кафедр, у которых суммарное количество работающих профессоров более 1. Приведенная ниже формулировка запроса является неверной:

SELECT KOD\_kafedru FROM TEACHER

WHERE count(dolgnost) > 1 and dolgnost=’профессор’ GROUP BY KOD\_kafedru;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - -

WHERE count(dolgnost) > 3 and dolgnost=’профессор’;

**\***

Ошибка в строке 3;

CRA-00934: групповая функция здесь не разрешена

Дело в том, что фраза WHERE проверяет на соответствие условию строки исходных таблиц, а мы указали в ней агрегатную функцию. Для отбора строк среди полученных групп следует применять фразу HAVING. Она играет такую же роль для групп, что и фраза WHERE для исходных таблиц, и может использоваться лишь при наличии фразы GROUP BY. В предложении SELECT фразы WHERE, GROUP BY и HAVING обрабатываются в следующем порядке.

Фразой WHERE отбираются строки, удовлетворяющие указанному в ней условию. Фраза GROUP BY группирует отобранные строки.

Фразой HAVING отбираются группы, удовлетворяющие указанному в ней условию. В связи с вышесказанным, предыдущий запрос необходимо записать так.

Перепишем тогда запрос так:

**Запрос 22**. Вывести номера кафедр, у которых суммарное количество работающих профессоров более 1.

SELECT KOD\_kafedru as "Номер кафедры", Count(\*) as "Кол-во профессоров на кафедре"

FROM TEACHER

WHERE dolgnost='профессор' GROUP BY KOD\_kafedru having count(dolgnost) > 1;

**Использование столбцов группировки во фразе HAVING**

Рассмотрим использование во фразе HAVING условий отбора, заданных для группируемых столбцов (или выражений над ними). Для этого усложним предыдущий запрос.

**Запрос 23**. Вывести названия кафедр факультета математики и информатики, на которых работают один и более профессоров. Указать также количество профессоров и их суммарную зарплату.

SELECT d.Name\_kafedru, Count(\*), SUM(t.salary + t.Rise) FROM FACULTET f, KAFEDRA d, TEACHER t

WHERE f.KOD\_FACULTETA = d.KOD\_FACULTETA AND

d.KOD\_kafedru = t.KOD\_kafedru AND LOWER(f.Name\_faculteta) = 'математики и информатики' AND LOWER(t.Dolgnost ) = 'профессор'

GROUP BY d.Name\_kafedru HAVING COUNT(\*) > 0;

**Фраза HAVING без фразы GROUP BY**

Выше мы указали, что фраза HAVING может использоваться лишь при наличии фразы GROUP BY. Из этого правила синтаксис SQL допускает только одно исключение: когда вся таблица интерпретируется как одна группа. В этом случае в списке фразы SELECT можно использовать только константы, агрегатные функции и выражения над ними. Приведем примеры.

**Запрос 24**. Если суммарная зарплата всех преподавателей превышает 15 000, вывести их минимальную ставку, максимальную надбавку и суммарную зарплату.

SELECT MIN(Salary), MAX(Rise), SUM(Salary + Rise)

FROM TEACHER

HAVING SUM(Salary + Rise) > 15000;

При наличии фразы WHERE сначала производится отбор строк согласно ее условию, и только после этого применяется условие фразы HAVING.

**Запрос 25.** Если суммарная зарплата всех ассистентов превышает 2500, вывести их среднюю ставку, среднюю надбавку и суммарную зарплату.

SELECT AVG(Salary), AVG(Rise), SUM(Salary + Rise)

FROM TEACHER

WHERE LOWER(Dolgnost ) = 'ассистент'

HAVING SUM(Salary + Rise) > 2500;

На практике фраза HAVING очень редко используется без фразы GROUP BY, из- за чего такая возможность предоставляется не во всех СУБД.

**Сортировка результирующих строк**

Как мы уже отмечали, строки в таблицах базы данных неупорядочены. Также неупорядочены и строки результирующей таблицы запроса, однако для их упорядочения в предложении SELECT можно воспользоваться фразой ORDER BY. Она сортирует по значению указанных в ней столбцов (и выражений над столбцами) строки результирующей таблицы запроса. Синтаксис этой фразы следующий:

ORDER BY спецификация\_сортировки[. спецификация\_сортировки]...

где спецификация\_сортировки имеет такой синтаксис:

**выражение\_сортировки [направление\_сортировки] [положение\_NULL]**

Сортировать можно по столбцам (выражениям) тех типов, для которых определены операции сравнения. Это относится, в частности, к символьным строкам, числам и временным значениям. Можно указывать направление сортировки и место расположения строк, имеющих значение NULL для выражений сортировки.

Далее рассмотрим общие способы упорядочения результирующих строк.

**Сортировка по столбцу или выражению**

Сортировать строки результирующей таблицы запроса можно по отдельным столбцам, совокупности столбцов, а также по одному или нескольким выражениям над столбцами. Ниже рассматриваются все эти варианты.

**Сортировка по столбцу**

Простейший вариант сортировки - это сортировка по одному из столбцов результирующей таблицы.

**Запрос 26.** Вывести алфавитный список фамилий профессоров и доцентов.

SELECT NAME\_TEACHER FROM TEACHER

wheRE LOwER(Dolgnost ) =’профессор’ OR

LOWER(Dolgnost ) = 'доцент'

ORDER BY NAME\_TEACHER;

**Сортировка по выражению над столбцами**

Упорядочивать строки можно не только по значению столбца, но и по значению выражения над столбцами.

**Запрос 27.** Вывести фамилии ассистентов и их зарплату по ее возрастанию.

SELECT Name\_teacher, Salary + Rise FROM TEACHER

WHERE LOWER(Dolgnost ) = 'асcистент’ ORDER BY Salary + Rise;

**Направление сортировки**

Во всех до сих пор приводимых примерах сортировка производилась в порядке возрастания значений. В SQL такой порядок определен по умолчанию. Однако есть возможность и явно указать направление сортировки с помощью ключевых слов ASC (по возрастанию) и DESC (по убыванию), которые следует располагать после имени сортируемого столбца (выражается).

**Запрос 28**. Вывести фамилии ассистентов и дату их приема на работу по возрастанию даты.

SELECT Name\_teacher, Data\_hire

FROM TEACHER

WHERE LOWER(Dolgnost) = 'ассистент' ORDER BY Data\_hire ASC;

Самостоятельно создать запрос 29. Вывести фамилии доцентов в обратном алфавитном порядке и их зарплату.

**Тема 3. Внесение изменений в базу данных**

**Добавление новых данных**

Новые данные добавляются оператором INSERT. Наименьшей единицей информации, которую можно добавить в реляционную базу данных, является одна строка таблицы.

Немного упрощенный синтаксис оператора INSERT имеет вид:

INSERT INTO Имя\_Таблицы [(Колонка [, Колонка ...])]

{VALUES(<величина> [, <величина> ...]) | <оператор SELECT>};

<величина> = {:Переменная | <константа> | <выражение>

| <функция> | udf([<величина> [, <величина> ...]])

| NULL | USER}

<константа> = Число | 'Строка'

<функция> = CAST(<величина> AS <тип данных>) UPPER(<величина>)

GEN\_ID(Имя\_Генератора, <величина>)

<выражение> = SQL выражение, возвращающее единичное значение

В этой практической работе рассматривается только первый вариант оператора INSERT.

Пример, когда в качестве вставляемых величин применены константы:

INSERT INTO Person(Pr\_ID, Pr\_LastName, Pr\_FirstName)

VALUES(150, 'Иванов', 'Петр');

**Удаление существующих данных**

Для удаления строк из таблицы используется оператор DELETE. Вот его упрощенный синтаксис:

DELETE FROM Имя\_Таблицы [WHERE <условие поиска>];

<условие поиска> = как в операторе SELECT

Если не использовать предложение WHERE, то будут удалены все строки в таблице.

-- Удаление всех служащих:

DELETE FROM Employee;

-- Удаление всех людей с номерами 150 и больше:

DELETE FROM Person WHERE Pr\_ID >= 150;

Отбирать строки для удаления не обязательно только на основании содержимого этих строк. Можно составить условие для удаляемых строк, опираясь на данные из других таблиц. Для составления таких условий необходимо сначала изучить оператор SELECT.

**Обновление существующих данных**

Оператор UPDATE обновляет значения одного или нескольких столбцов в выбранных строках одной таблицы. Строки для обновления указываются в предложении WHERE. Если пропустить предложение WHERE, то изменятся все строки таблицы.

UPDATE Имя\_Таблицы

SET Колонка = <величина>[, Колонка = <величина>...] [WHERE <условие поиска>]

<величина> = { Колонка | :Переменная | <константа>

| <выражение> | <функция>

| udf([<величина> [, <величина> ...]]) | NULL | USER}

<выражение> = SQL выражение, возвращающее единичное значение

<условие поиска> = как в операторе SELECT

**Примеры:**

-- Увеличить зарплату всем служащим на 10%:

UPDATE Employee

SET Salary = 1.1\*Salary;

/\* Увеличить зарплату всем служащим, которые имеют зарплату меньше 10000 на 15%: \*/

UPDATE Employee

SET Salary = 1.15\*Salary; WHERE Salary <= 10000;

Отбирать строки для изменения, как и для удаления, можно с использование подчиненного запроса SELECT, который позволит учитывать в условии поиска изменяемых строк данные из других таблиц.

Например, можно выполнить такой запрос: увеличить зарплату на 10% всем служащим, работающим в отделе продаж, которые обслужили за последний месяц клиентов больше чем в полтора раза, чем в среднем по их отделу.

**Задание**

Добавление новых строк

Предложение INSERT вставляет строки в таблицу базы данных. Есть три разновидности этой команды:

**INSERT VALUES INSERT SELECT**

**INSERT DESALT VALUES**

Первая из них производит вставку в таблицу явно заданной строки, вторая разновидность – вставку группы строк, выбранных в результате выполнения запроса, а третья — вставку значений по умолчанию.

**Вставка отдельных строк**

Предложение INSERT... VALUES выполняет вставку в таблицу одной строки. Его удобно использовать для небольших операций, когда в таблицу нужно вставить несколько строк. Синтаксис этого предложения следующий:

INSERT INTO имя\_таблицы [<имя\_стол6ца[, имя\_столбца]...)]

VALUES (значение[, значение]...);

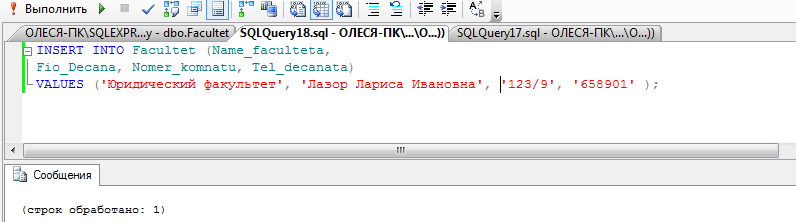
**Указание вставляемых столбцов**

Этот формат предполагает указания имени таблицы, в которую производится вставка, списка имен столбцов, в которые будут вставляться значения, и списка собственно вставляемых в строку значений. При этом следует придерживаться следующих правил:

–вставляемые данные должны согласовываться с типами данных указанных столбцов;

–размеры данных должны соответствовать размерам столбцов;

–порядок данных во фразе VALUES должен соответствовать порядку перечисления столбцов.

Запрос 30. Выполнить вставку одной строки в таблицу FACULTET

В таблицу вставляется строка со значениями, указанными в списке фразы VALUES, причем расположение значений в списке соответствует расположению соответствующих столбцов в списке столбцов таблицы.

Примечание. Чтобы данные были добавлены, не забудьте нажать на кнопки , а затем  для подтверждения внесенных изменений в таблицу. Просмотрите внесенные изменения в таблицу FACULTET.

В этом примере мы перечислили столбцы в том порядке, в каком они были определены при создании таблицы, однако это не обязательно. При желании порядок перечисления имен столбцов в команде INSERT можно изменить.

Список имен столбцов может быть не полным. Можно указывать только те из них, значения которых известны для вставляемой строки, Столбцы, отсутствующие в списке, будут принимать значения NULL для вводимой строки.

Самостоятельно создайте запрос 31. Выполнить вставку одной строки в таблицу KAFEDRA для столбцов name\_kafedru, fio\_zavkaf, kod\_faculteta с данными 'Психологии', 'Иванова', 5.

В этом примере в водимой строке отсутствуют значения столбцов Nomer\_komnatu, Num\_korpusa и Tel\_kafedru. В базе данных они примут значение NULL.**Поддержка ограничений целостности**

Помните, некоторые из столбцов или наборов столбцов могут иметь ограничения целостности PRIMARY KEY и NOT NULL. Такие столбцы не могут принимать значения NULL.

Приведенные выше рассуждения относятся ко всем ограничениям целостности, определенным для таблиц. При попытке ввода данных (как, впрочем, и при обновлении и удалении) СУБД проверяет возможное нарушение объявленных ограничений целостности. И если это так, команда будет отклонена с выдачей соответствующего уведомления.

**Использование выражений**

В качестве вставляемых значений могут использоваться выражения.

**Самостоятельно** создайте запрос 32.Ввести в таблицу TEACHER данные (50, 10, ' Капуста Леонид Владимирович',

1271, 1271/3, ‘доцент’, GETDATE()-1)

Здесь мы указали, что надбавка равна третьей части ставки (1271 / 3),а дата приема на работу на один день меньше текущей даты (CURRENT\_DATE -1).

**Результат запроса в качестве вставляемого значения**

Вместо вставляемого значения можно использовать запрос. Это оказывается очень удобным в том случае, когда вставляемое значение присутствует в базе данных.

**Запрос 33**.Например, в следующем предложении в качестве фамилии заведующего вновь вставляемой кафедры выбирается фамилия декана факультета «Компьютерных наук и технологий» .

INSERT INTO KAFEDRA (name\_kafedru, kod\_faculteta, fio\_zavkaf) VALUES ( 'Философии', 5, (SELECT fio\_decana

FROM FACULTET WHERE LOWER(Name\_faculteta) = 'международный'));

**Обновление существующих данных**

Целью предложения обновления является изменение значений отдельных столбцов всех или удовлетворяющих указанному условию строк таблицы. Упрощенный синтаксис предложения следующий:

UPDATE имя\_таблицы [[AS] синоним]

SET имя\_столбца = выражение[, имя\_столбца = выражение]...

[WHERE условие];

Его элементы означают следующее:

имя\_таблицы — имя обновляемой таблицы;

синоним — синоним обновляемой таблицы для ссылки на нее в подзапросе; имя\_столбца - имя обновляемого столбца;

выражение - допустимое в SQL выражение соответствующего типа, значение которого присваивается обновляемому столбцу;

условие - допустимое в SQL выражение условия, которое используется для отбора обновляемых строк.

По одному предложению UPDATE обновлению подвергаются строки только одной базовой таблицы.

**Обновление всех строк**

Как видно из определения синтаксиса команды UPDATE, фраза WHERE является факультативной. При ее отсутствии все строки таблицы подвергаются обновлению согласно фразе SET.

Во фразе SET можно одновременно изменять значения нескольких столбцов таблицы.

**Запрос 34**. Например, в следующем примере всем преподавателям увеличивается ставка на 12 % и надбавка на 7 %:

UPDATE TEACHER

SET Salary = Salary + Salary \* 0.12, Rise = Rise + Rise \* 0.08;

Во фразе SET в правой части оператора присваивания может использоваться любое допустимое в SQL выражение того же типа, что и столбец, имя которого приведено слева от оператора присваивания. Если в этом выражении используется имя столбца целевой таблицы, для вычисления выражения применяется значение этого столбца в текущей строке, которое было перед обновлением.

**Примечание**. Чтобы данные были добавлены, не забудьте нажать на кнопки , а затем  для подтверждения внесенных изменений в таблицу. Просмотрите внесенные изменения в таблицу TEACHER.

**Обновление по условию**

Данный вариант использует фразу WHERE. В этом случае обновляются столбцы только тех строк таблицы, на которых выполняется условие фразы WHERE. Рассмотрим несколько примеров.

**Запрос 35**. Увеличить всем ассистентам зарплату и надбавку на 10 %:

UPDATE TEACHER

SET Salary = Salary \* 1.1, Rise = Rise \* 1.1

WHERE LOWER(Dolgnost ) = 'ассистент';

**Самостоятельно** создать запрос 36. Установить, что деканат юридического факультета переместился в комнату 232 8 корпуса.

**Подзапросы во фразе WHERE**

Во фразе WHERE можно использовать подзапросы, как мы это делали в предложении SELECT. Это дает возможность отбирать строки для обновления на основе информации из других таблиц.

**Запрос 37**. Например, увеличить ставку всех преподавателей кафедры прикладной математики факультета математики и информатики в полтора раза:

UPDATE TEACHER

SET Salary = Salary \* 1.5

WHERE KOD\_kafedru = (SELECT KOD\_kafedru FROM KAFEDRA WHERE LOWER(Name\_kafedru) = 'прикладная математика');

**Подзапросы во фразе SET**

До сих пор новые значения представляли собой константы или выражения с использованием значений обновляемой строки. Однако если такие значения присутствуют в других строках обновляемой таблицы или вообще в других таблицах, можно воспользоваться подзапросом. В этом случае допускается использовать две формы фразы SET:

**SET {имя\_столбца | (список\_инен\_столбцов)} = (подзапрос)**

В обоих вариантах подзапрос должен возвращать одну строку. В первом случае он также должен возвращать значение одного столбца, а во втором возвращаемая строка должна содержать столько значений, сколько столбцов приведено в списке имен столбцов. При этом производится присвоение значений строки подзапроса соответствующим столбцам из списка слева.

**Запрос 38.** Установить всем ассистентам надбавку, равную 70 % текущей средней надбавки по вузу.

UPDATE TEACHER

SET Rise = (SELECT SUM(Rise) \* 0.7 / COUNT(\*) FROM TEACHER)

WHERE LOWER(Dolgnost ) = 'ассистент';

**Удаление существующих строк**

Удалять строки из таблицы можно с помощью предложения DELETE. Оно удаляет только строки целиком, а не индивидуальные значения столбцов. Синтаксис команды следующий:

DELETE FROM имя\_таблицы [[AS] синоним] [WHERE условие];

При использовании предложения DELETE вы прежде всего обнаружите, что предупреждающая подсказка, как правило, не выдается. Обычно, когда пользователь удаляет какой-либо объект операционной среды, он получает сообщение типа «Вы уверены (Д/Н)?». В системах, поддерживающих SQL, строки удаляются без такого сообщения. Поэтому будьте внимательны.

В зависимости от применения фразы WHERE предложение DELETE позволяет удалить отдельную строку, несколько или все строки таблицы. Строки могут быть и не удалены. При использовании предложения DELETE помните о следующем:

нельзя удалить значение отдельного столбца (используйте для этого предложение UPDATE);

как и предложения INSERT и UPDATE, удаление строк может нарушить ограничения целостности;

сама таблица не удаляется (используйте для этого предложение DROP TABLE).

**Удаление всех строк таблицы**

Чтобы удалить все содержимое таблицы, не нужно использовать фразу WHERE. Помните, что вы удаляете не саму таблицу, а только все ее строки.

**Запрос 39. Удалить содержимое таблицы Сотрудники базы данных Educator.**

use Educator

DELETE FROM Cотрудники;

**Удаление по условию**

Обычно нужно удалять только некоторые строки из таблицы. Чтобы определить, какие строки будут удалены, нужно использовать условие во фразе WHERE. Приведем несколько примеров.

Самостоятельно создать **запрос 40**. Удалить сведения об ассистентах, которые были приняты на работу до 01.01.1986.

**Удаление одной строки**

Чтобы удалить одну конкретную строку, нужно сформулировать условие таким образом, чтобы оно идентифицировало эту единственную строку. Обычно для этого в условии используются первичный ключ таблицы или уникальный набор столбцов.

Самостоятельно создать **запрос 41**. Удалить всех преподавателей под фамилией

Швец.

### Задание для лабораторной работы №3 (часть 4)

Для созданной базы данных, согласно номеру варианта, самостоятельно создать **на языке Transact-SQL многотабличные запросы:**

* 2 запроса с использованием функции **COUNT**;
* 2 запроса с использованием функции **SUM**;
* 2 запроса с использованием функций **UPPER, LOWER**;
* 2 запроса с использованием временных функций;
* 2 запроса с использованием группировки по одному столбцу;
* 2 запроса на использование группировки по нескольким столбцам;
* 2 запроса с использованием условия отбора групп **HAVING**;
* 2 запроса с использованием фразы **HAVING** без фразы **GROUP BY;**
* 2 запроса с использованием сортировки по столбцу;
* 2 запроса на добавление новых данных в таблицу;
* 2 запроса на добавление новых данных по результатам запроса в качестве вставляемого значения;
* 2 запроса на обновление существующих данных в таблице;
* 2 запроса на обновление существующих данных по результатам подзапроса во фразе **WHERE;**
* 2 запроса на удаление существующих данных.

Все программные инструкции команд SQL сохранять в файлах с расширением

**\*.sql** в папке ***ФИО\_студента/Лаб3****.*

Для каждого запроса сформулировать текстовое задание, которое должно быть выполнено к базе данных.

Создать текстовый отчет, в котором отобразить sql-команды разработанных запросов и скриншоты результатов работы из СУБД **SQL Server Management Studio**.