Министерство образования Республики Беларусь
УО «Полесский государственный университет»

Т. В. КИСЕЛЬ

БАЗЫ ДАННЫХ

Методические указания по выполнению лабораторных работ
по дисциплине «Базы данных»
для студентов I ступени получения высшего образования специальности 1-40 05 01
«Информационные системы и технологии»

Пинск
ПолесГУ
2021

УДК 004.65(076.5)
ББК 32.972.1
К44

Р е ц е н з е н т ы:
директор ООО «Технопарк ”Полесье“»
А. В. Козырь;
кандидат физико-математических наук, доцент,
зав. кафедрой информационных технологий и интеллектуальных систем ПолесГУ
М. А. Романова

У т в е р ж д е н о
научно-методическим советом ПолесГУ

Кисель, Т. В.

К44 Базы данных : методические указания по выполнению лабораторных работ / Т. В. Кисель. – Пинск : ПолесГУ, 2021. – 76 с.

ISBN 978-985-516-710-6

Цель методических указаний – получение практических навыков создания баз данных в СУБД MS SQL Server, взаимодействия с базой данных с использованием средства администрирования SQL Server Management Studio и языка Transact-SQL.

Издание предназначено для студентов IT-специальностей и может быть использовано преподавателями при проведении практических и лабораторных занятий.

УДК 004.65(076.5)
ББК 32.972.1

ISBN 978-985-516-710-6

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

SQL Server является одной из наиболее популярных систем управления базами данных (СУБД) в мире. SQL Server был создан компанией Microsoft. Данная СУБД подходит для самых различных проектов: от небольших приложений до больших высоконагруженных проектов.

Центральным аспектом в MS SQL Server, как и в любой СУБД, является *база данных*. База данных представляет хранилище данных, организованных определенным способом.

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

- *таблицы* (хранят собственно данные);
- *представления* (views) – выражения языка SQL, которые возвращают набор данных в виде таблицы;
- *хранимые процедуры* (выполняют код на языке SQL по отношению к данным БД, получает данные или изменяет их);
- *функции* (также код SQL, который выполняет определенную задачу).

В SQL Server используется два типа баз данных: *системные* и *пользовательские*.

Системные базы данных необходимы серверу SQL для корректной работы. *Пользовательские базы данных* создаются пользователями сервера и могут хранить любую произвольную информацию. Их можно изменять и удалять, создавать заново. Это те базы данных, которые мы будем создавать и с которыми мы будем работать.

Для организации баз данных MS SQL Server использует *реляционную модель*. Реляционная модель предполагает хранение данных в виде таблиц, каждая из которых состоит из строк и столбцов. Каждая строка хранит отдельный объект, а в столбцах размещаются атрибуты этого объекта.

Для взаимодействия с базой данных применяется *язык SQL* (*Structured Query Language*). Выделяют две разновидности языка SQL: *PL-SQL* и *T-SQL*.

PL-SQL используется в таких СУБД, как Oracle и MySQL.

T-SQL (сокращенная форма от Transact-SQL) применяется в SQL Server. Поэтому в рамках текущего практикума будет рассматриваться именно T-SQL.

Для удобного управления базами данных и различными опциями и настройками в MS SQL Server предназначено специальное средство администрирования, которое называется *SQL Server Management Studio* (*SSMS*). Данную программу можно использовать для создания баз данных и их таблиц, написания и выполнения запросов к базе данных, а также для многого другого.

ЛАБОРАТОРНАЯ РАБОТА 1
СОЗДАНИЕ БАЗЫ ДАННЫХ

1. Средство администрирования SQL Server Management Studio (SSMS)

Подавляющую массу задач администрирования SQL Server можно выполнять в графической утилите SQL Server Management Studio. В ней можно создавать базы данных и все ассоциированные с ними объекты (таблицы, представления, хранимые процедуры и др.), выполнять последовательности инструкций Transact-SQL (запросы). В этой утилите можно выполнять типовые задачи обслуживания баз данных, такие как резервирование и восстановление. Здесь можно настраивать систему безопасности базы данных и сервера, просматривать журнал ошибок и многое другое.

Для запуска Management Studio в меню *«Пуск»* выберите пункт *«Microsoft SQL Server / Среда SQL Server Management Studio»*. Когда откроется окно программы, вас попросят подключиться к какому-либо серверу баз данных SQL Server.

Подключение к серверу

В окне *«Соединение с сервером»* необходимо указать следующую информацию (рисунок 1.1):

– *тип сервера* (здесь следует выбрать, к какой именно службе необходимо подключиться. Оставьте вариант *«Компонент Database Engine»*);

– *имя сервера* (позволяет указать, к какому серверу будет осуществляться подключение. По умолчанию имя SQL Server совпадает с именем компьютера. Выберите ваш локальный компьютер);

– *проверка подлинности* – способ аутентификации (можно выбрать *«Проверка подлинности Windows»* или *«Проверка подлинности SQL Server»)*. Первый способ использует учетную запись, под которой текущий пользователь осуществил вход в Windows. Вариант SQL Server использует свою собственную систему безопасности. Оставьте вариант проверки подлинности Windows.

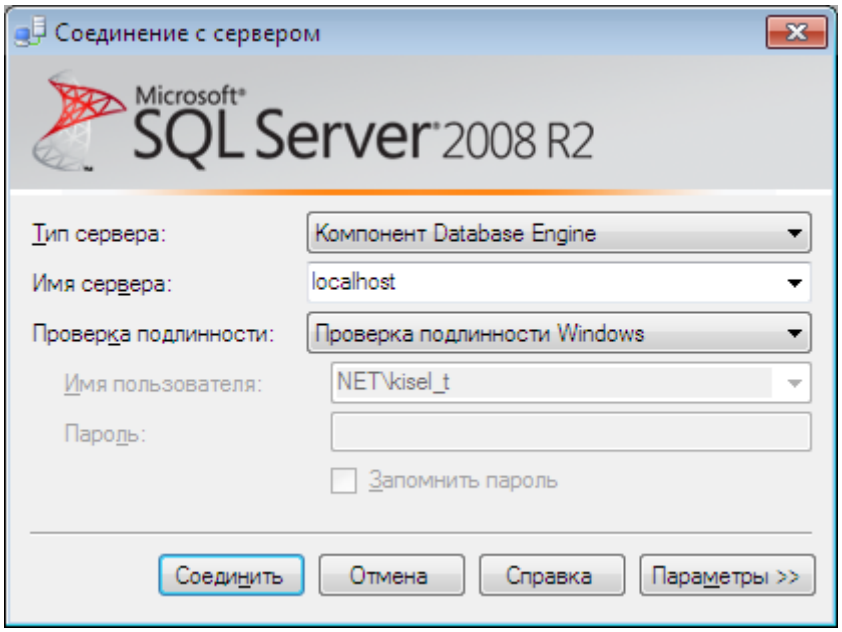


Рисунок 1.1 – Подключение к серверу

После подключения экземпляр сервера будет отображаться на панели *«Обозреватель объектов»* (рисунок 1.2).

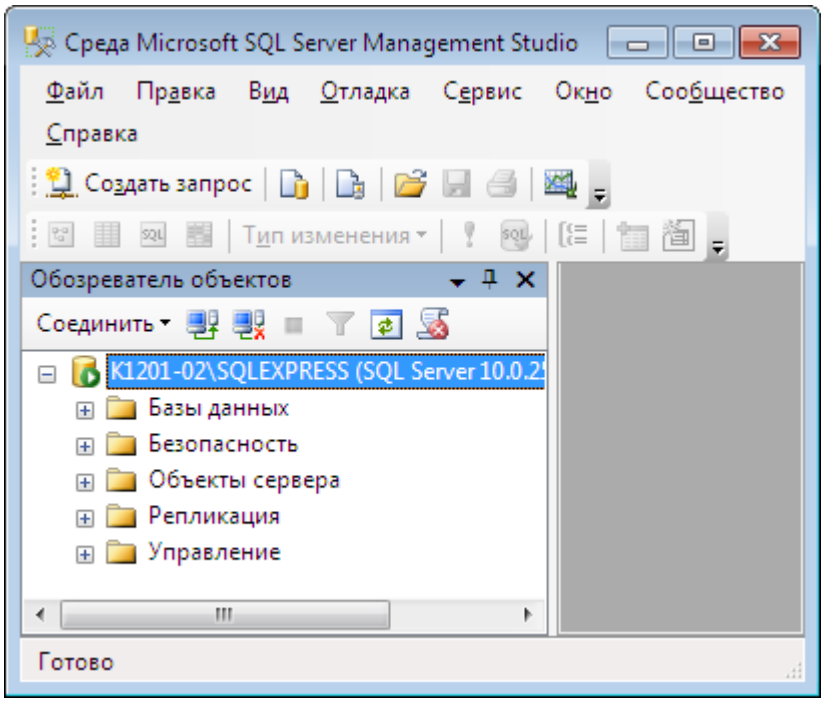


Рисунок 1.2 – Текущий экземпляр сервера

Окно графической утилиты SQL Server Management Studio

Окно «*Management Studio»* имеет следующую структуру:

– *меню* (содержит полный набор команд для управления сервером и выполнения различных операций);

– *панель инструментов* (содержит кнопки для выполнения наиболее часто производимых операций. Внешний вид данной панели зависит от выполняемой операции);

– панель *«Обозреватель объектов»* – это панель с древовидной структурой, отображающая все объекты сервера, а также позволяющая производить различные операции как с самим сервером, так и с его базами данных и их объектами. *Обозреватель объектов* является основным инструментом для разработки;

– *рабочая область* (в рабочей области производятся все действия с базой данных, а также отображается её содержимое).

Обзор служебных баз данных SQL Server

Прежде чем перейти к созданию своих собственных рабочих баз данных рассмотрим служебные базы данных SQL Server, которые создаются автоматически в процессе его установки. Если мы раскроем узел *«Базы данных – Системные базы данных»* в обозревателе объектов, то увидим следующий набор служебных баз данных (рисунок 1.3):

– **master** – главная служебная база данных всего сервера, в которой хранится общая служебная информация сервера: настройки его работы, список баз данных на сервере с информацией о настройках каждой базы данных и ее файлах, информация об учетных записях пользователей, серверных ролях и т. п.;

– **model** – эта база данных является шаблоном для создания новых баз данных в SQL Server. Если внести в нее изменения (например, создать набор таблиц), то эти таблицы будут присутствовать во всех создаваемых базах данных;

– **msdb** – эта база данных в основном используется для хранения информации службы SQL Server Agent (пакетных заданий, предупреждений и т. п.), но в нее записывается и другая служебная информация (например, история резервного копирования);

– **tempdb** – эта база данных предназначена для временных таблиц и хранимых процедур, создаваемых пользователями и самим SQL Server. Эта база данных создается заново при каждом запуске SQL Server.

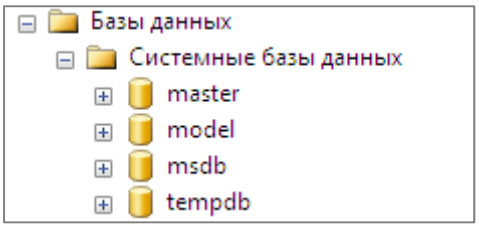


Рисунок 1.3 – Системные базы данных

2. Создание пользовательских баз данных

База данных представляет собой группу файлов, хранящихся на жестком диске. Эти файлы могут относиться к трем типам: первичные файлы данных, вторичные файлы данных и файлы журнала транзакций. Любая база данных SQL Server содержит два файла: первичный файл данных (с расширением **.mdf**) и файл журнала транзакций (с расширением **.ldf**). Существует два способа их создания:

– графически (с помощью SQL Server Management Studio);

– посредством кода Transact-SQL.

2.1. Создание баз данных в SQL Server Management Studio

Использование данной утилиты является самым простым способом создания базы данных. Создадим базу данных «*Sales*» (Продажи), которую позже заполним таблицами, представлениями и другими объектами, предназначенными для отдела продаж (рисунок 1.4).

1. В окне *«Обозреватель объектов»* найдите и раскройте папку *«Базы данных»*. Щелкните на ней правой кнопкой мыши и выберите команду *«Создать базу данных»*.

2. В открывшемся диалоговом окне *«Создание базы данных»* на странице *«Общие»* введите следующую информацию: имя базы данных – «*Sales*»; владелец – «*sa*».

3. В таблице *«Файлы базы данных»* измените путь к файлам данных и журнала на ваш каталог.

4. Для всех остальных параметров оставьте значения по умолчанию.

5. Для создания базы данных щелкните кнопку *«OK»*. Вы должны увидеть свою новую базу данных в окне *«Обозреватель объектов»*.

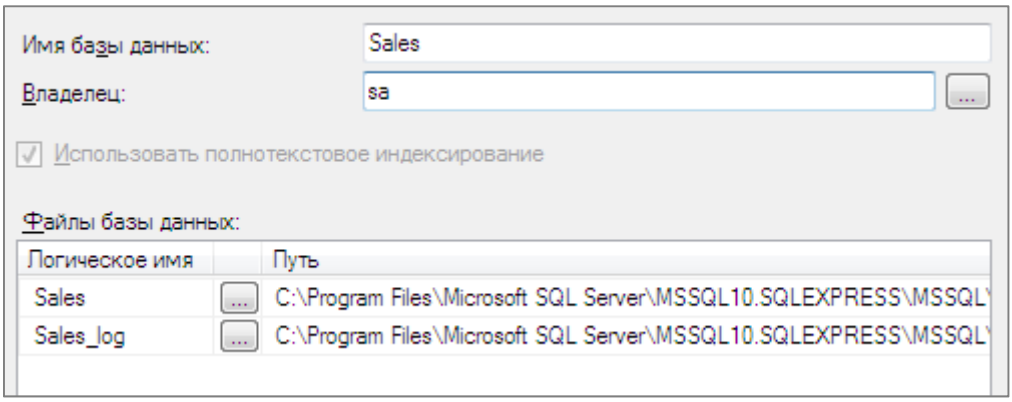


Рисунок 1.4 – Создание базы данных «Sales»

2.2. Создание баз данных с помощью Transact-SQL

Для программного создания базы данных используется инструкция CREATE DATABASE языка T-SQL. У инструкции CREATE DATABASE имеется обязательный параметр – имя базы данных. Кроме этого, у инструкции имеется ряд необязательных параметров, например расположение на диске (данный параметр указывает, где требуется хранить файлы базы данных). При выполнении инструкции CREATE DATABASE без дополнительных параметров для многих из них SQL Server использует значения по умолчанию.

Для создания базы данных в окне редактора запросов введите следующий код:

```
CREATE DATABASE TestDB
GO,
```

где CREATE – это команда языка T-SQL для создания объектов на SQL сервере;
командой DATABASE мы указываем, что хотим создать базу данных;
TestDB – это имя новой базы данных.

Чтобы выполнить инструкцию и создать базу данных с именем TestDB, нажмите клавишу *«F5»*. При создании базы данных SQL Server создает копию базы данных «*Model»* и присваивает ей указанное имя базы данных.

С помощью инструкции CREATE DATABASE можно задать абсолютно все параметры. Например, если заменить вышеуказанную инструкцию следующей, то база данных будет создана в каталоге DataBases на диске D.

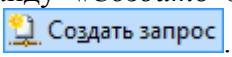
```
--Создание БД TestDB
  CREATE DATABASE TestDB
  ON  PRIMARY --Первичный файл
  (
      NAME = N'TestDB', --Логическое имя файла БД
      FILENAME = N'D:\DataBases\TestDB.mdf' --Имя и местоположение файла БД
  )
  LOG ON --Явно указываем файлы журналов
  (
     NAME = N'TestDB_log', --Логическое имя файла журнала
     FILENAME = N'D:\DataBases\TestDB_log.ldf' --Имя и местоположение файла журнала
  ) GO
```

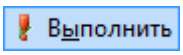
2.3. Создание базы данных на основе сгенерированного сценария

Сценарий создания новой базы данных может быть сгенерирован на основе уже существующей базы данных. Для этого в SQL Server Management Studio в контекстном меню узла *«Sales»* выберите команду *«Создать сценарий для базы данных – Используя CREATE – Буфер обмена»*. В результате в буфер обмена будет сохранен текст запроса на создание новой базы данных с параметрами, указанными при создании базы данных *«Sales»* в Management Studio.

Для проверки работоспособности сгенерированного запроса на создание базы данных удалим базу данных *«Sales»*. В контекстном меню базы данных выберите команду *«Удалить»* и в появившемся диалоговом окне нажмите кнопку *«OK»*. База данных со всеми файлами должна исчезнуть.

Чтобы воспользоваться сгенерированным заранее запросом на создание базы данных выполните следующие шаги:

1. В контекстном меню базы *«Sales»* выберите команду *«Создать запрос»* или щелкните соответствующую кнопку на панели инструментов 

.
2. В открывшемся окне редактора SQL вставьте из буфера обмена сгенерированный запрос.
3. Для запуска запроса на выполнение щелкните кнопку 

на панели инструментов или нажмите клавишу *«F5»*.
4. Обновите содержимое дерева обозревателя объектов командой *«Обновить»* из контекстного меню узла *«Базы данных»*. База данных *«Sales»* должна вновь появиться в списке доступных.

При необходимости перед выполнением текст запроса может быть изменен в этом же окне.

Рассмотрим основные опции сгенерированной инструкции CREATE DATABASE:

– имя новой базы данных (указывается непосредственно после ключевого слова CREATE DATABASE. В данном случае это *Sales*);

– ON – опция, которая указывает на файловую группу, представляет собой логическую группу вторичных файлов данных и используется для управления размещением пользовательских объектов, таких как таблицы и индексы;

– опция PRIMARY после аргумента ON используется для указания группы файлов PRIMARY, в которую по умолчанию входят все созданные файлы и которая является единственной группой файлов, содержащей первичный файл данных;

– NAME – логическое имя базы данных, которое будет применяться для ссылки на нее из кода T-SQL;

– FILENAME – это имя и путь файла базы данных, хранящегося на жестком диске.

– SIZE – исходный размер файлов данных;

– MAXSIZE – максимальный размер, до которого может расти база данных;

– FILEGROWTH – это приращение расширения файла;

– параметры в разделе LOG ON аналогичны параметрам в разделе CREATE DATABASE. Однако они определяют параметры файла журнала транзакций.

Общий синтаксис инструкции CREATE DATABASE со всеми возможными опциями можно посмотреть в справочной системе. Для этого в редакторе запросов выделите слова CREATE DATABASE и нажмите клавишу *«F1»*.

3. Отсоединение и присоединение базы данных

Для переноса базы данных на другой сервер необходимо отсоединить ее от текущего сервера. Для этого в контекстном меню базы данных *«Sales»* выберите команду *«Задачи –*

Отсоединить…». В диалоговом окне *«Отсоединение базы данных»* нажмите кнопку *«OK»* и убедитесь, что *Sales* исчезла из списка баз данных в дереве обозревателя объектов. Теперь файлы базы данных могут быть перенесены на другой сервер.

Для присоединения базы данных к серверу выберите в контекстном меню узла *«Базы данных»* команду *«Присоединить…»*. В диалоговом окне *«Присоединение базы данных»* с помощью кнопки *«Добавить…»* выберите созданный на предыдущих этапах файл Sales.mdf (ldf файл будет определен системой автоматически), измените владельца на *sa* и нажмите кнопку *«OK».* База данных «*Sales»* должна появиться в списке дерева обозревателя объектов.

Задание

Создайте базу данных *«Sales»* любым из вышеперечисленных способов.

ЛАБОРАТОРНАЯ РАБОТА 2
СОЗДАНИЕ ТАБЛИЦ И ОГРАНИЧЕНИЙ

Таблица – основной объект базы данных, предназначенный для хранения данных в структурированном виде.

Одним из основных правил организации баз данных является то, что в одной таблице должны храниться данные лишь об одном конкретном типе сущности (например, клиенты, товары, заказы и т. п.).

Данные в таблицах организованы по *полям* и *записям*. *Поля* содержат однородные значения (например, *фамилия, адрес, телефонный номер*). *Запись* – группа связанных полей, содержащих информацию об отдельном экземпляре сущности.

Любое поле таблицы характеризуется как минимум тремя обязательными свойствами:

- *имя столбца* (реализует способ обращения к конкретному полю в таблице. Рекомендуется всегда присваивать полям *смысловые имена*);
- *тип данных* (определяет, информация какого типа может храниться в данном поле);
- *разрешить значения null* (определяет, допустимо ли для данного поля отсутствие фактических данных, для обозначения которого используется так называемый *маркер пустого значения null)*.

2.1. Типы данных

При выборе типа данных для столбца следует отдавать предпочтение типу, который позволит хранить любые возможные для этого столбца значения и занимать при этом минимальное место на диске. Типы данных в MS SQL Server можно разделить на восемь категорий: *целочисленные данные, текстовые данные, десятичные данные, денежные типы данных, данные с плавающей точкой, типы данных даты и времени, двоичные типы данных, специализированные типы данных.*

Создание пользовательских типов данных

SQL Server позволяет на основе системных типов данных создавать пользовательские типы со всеми предварительно заданными параметрами, включая все ограничения и умолчания. В качестве примера создадим тип данных *«phone»*, который будет использоваться в таблице *«Customer»* для хранения телефонного номера клиента. Для его создания воспользуемся графическим интерфейсом утилиты *Management Studio*.

В дереве обозревателя объектов раскройте папки *«Базы данных – Sales – Программирование – Типы*». В контекстном меню узла «*Определяемые пользователем типы данных*» выберите команду *«Создать определяемый пользователем тип данных*» (рисунок 2.1).

В появившемся окне в текстовом поле *«Имя*» введите *phone*. В раскрывающемся списке «*Тип данных*» выберите *nchar*. В качестве длины введите *10*. Отметьте параметр «*Разрешить значения null*», чтобы иметь возможность не указывать телефонный номер при добавлении нового клиента. В секции «*Привязки*» оставьте пустые значения и нажмите кнопку *«ОК».* Созданный пользовательский тип данных должен появиться в дереве обозревателя объектов (рисунок 2.2).

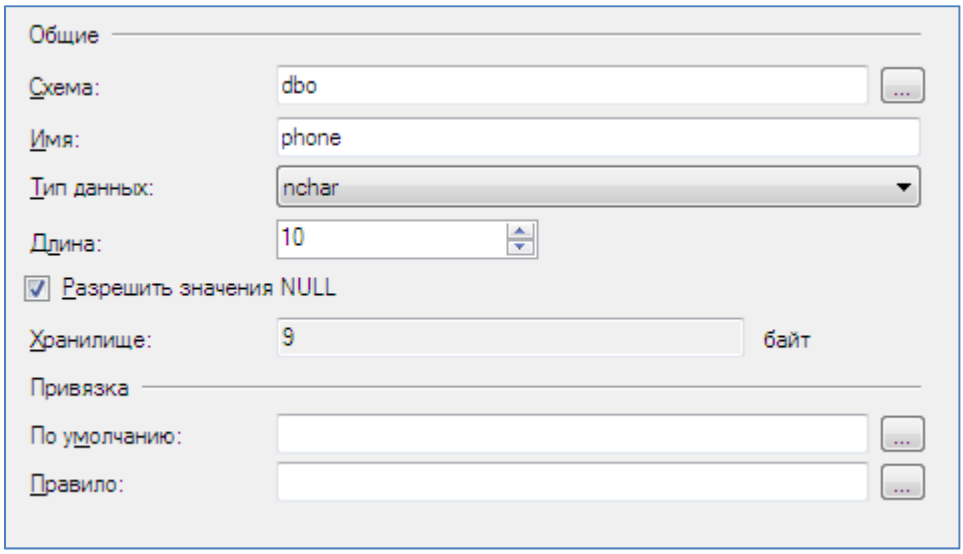


Рисунок 2.1. – Создание определяемого пользователем типа данных

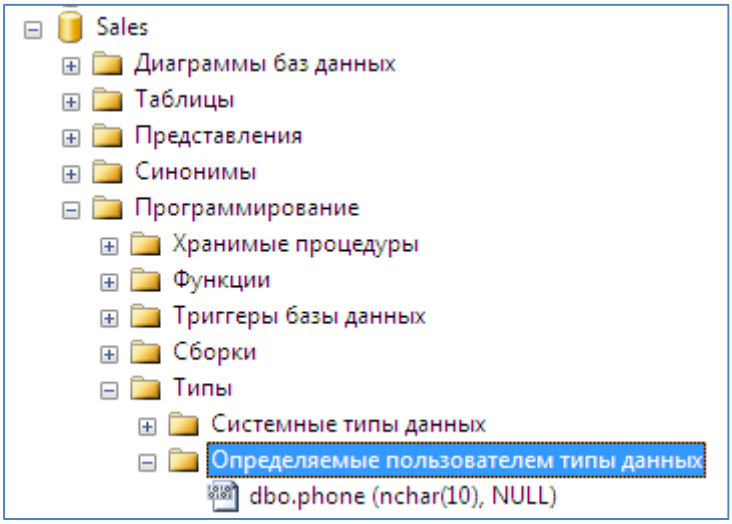


Рисунок 2.2 – Пользовательский тип данных phone

2.2. Создание таблиц

Создадим в базе данных *«Sales»* следующие таблицы (таблица 2.1).

Таблица 2.1 – Список таблиц базы данных «*Sales*» (Продажи)

№ п/п	Название таблицы	Краткое описание
1	Customer	Таблица предназначена для хранения информации о клиентах
2	City	Справочник городов
3	Category	Перечень категорий товаров
4	Product	Информация о товарах
5	Order	Данная таблица будет содержать подробную информацию о заказах
6	Ordered	Данная таблица будет содержать информацию о составе заказа (т. е. перечень товаров, входящих в заказ)

Таблица 2.2 – Описание таблицы *«Customer»* (Покупатель)

Имя столбца	Тип данных	Разрешить null	Описание
IdCust	int, identity	нет	Номер клиента
LName	nvarchar(20)	нет	Фамилия клиента
FName	nvarchar(20)	нет	Имя клиента
Gender	nchar(1)	нет	Пол
Birthday	date	да	Дата рождения
Phone	phone	нет	Телефонный номер клиента
Email	nvarchar(20)	да	Электронный адрес клиента
Address	nvarchar(70)	да	Адрес клиента
IdCity	int	нет	Ссылка на город
PostCode	nchar(5)	да	Почтовый индекс клиента

Таблица 2.3 – Описание таблицы *«City»* (Город)

Имя столбца	Тип данных	Разрешить null	Описание
IdCity	int, identity	нет	Код города
CityName	nvarchar(20)	нет	Название города

Таблица 2.4 – Описание таблицы *«Category»* (Категория)

Имя столбца	Тип данных	Разрешить null	Описание
IdCategory	int	нет	Номер категории
Name	nvarchar(50)	нет	Название категории
Description	nvarchar(100)	да	Краткое текстовое описание категории

Таблица 2.5 – Описание таблицы *«Product»* (Товар)

Имя столбца	Тип данных	Разрешить null	Описание
IdProd	int, identity	нет	Код товара
IdCategory	int	нет	Ссылка на категорию товара
Name	nvarchar(40)	нет	Наименование товара
Unit	nvarchar(10)	нет	Ед. измерения
Price	money	нет	Цена за единицу
Description	nvarchar(100)	да	Краткое текстовое описание товара
InStock	int	нет	Количество единиц на складе

Таблица 2.6 – Описание таблицы «*Order»* (Заказ)

Имя столбца	Тип данных	Разрешить null	Описание
IdOrd	int, identity	нет	Уникальный идентификационный номер заказа
IdCust	int	нет	Ссылка на номер клиента
OrdDate	smalldatetime	нет	Дата и время размещения заказа
Discount	float	да	Размер скидки
Paid	bit	да	Отметка об оплате (т. е. оплачено / не оплачено)

Таблица 2.7 – Описание таблицы «*Ordered*» (Заказано)

Имя столбца	Тип данных	Разрешить null	Описание
IdOrd	int	нет	Ссылка на номер заказа
IdProd	int	нет	Ссылка на номер товара
Amount	int	нет	Количество единиц товара в заказе

Таблицы можно создавать как в графическом интерфейсе (используя утилиту *Management Studio*), так и с помощью кода T-SQL. На данном этапе воспользуемся графическим способом. Сначала создадим таблицу *«Customer»*.

Для этого в дереве обозревателя объектов в базе данных *«Sales»* в контекстном меню узла *«Таблицы»* выберите команду *«Создать таблицу…»*. В рабочей области должна появиться вкладка с конструктором таблиц.

В первой строке в столбце *«Имя столбца»* введите *IdCust*, в столбце «*Тип данных*» выберите *int*. Убедитесь, что параметр «*Разрешить значения null*» отключен.

В нижней половине экрана в разделе «*Свойства столбцов*» введите описание поля и измените значение параметра «*Спецификация идентификатора / (Идентификатор)*» на «*Да*» для того, чтобы значения номера клиента формировались автоматически. Свойство «*Идентифицирующий столбец*» (*Identity*), обычно используемое совместно с типом данных *int*, предназначено для автоматического приращения значения на единицу при добавлении каждой новой записи. К примеру, клиент, добавленный в таблицу первым, будет иметь значение идентификатора *1*, вторым – *2*, третьим – *3* и т. д.

Аналогичным образом введите описания всех остальных полей и закройте окно конструктора таблиц (рисунок 2.3). Введите в качестве имени таблицы «*Customer*». Вновь созданная таблица должна появиться в дереве обозревателя объектов в папке «*Таблицы*».

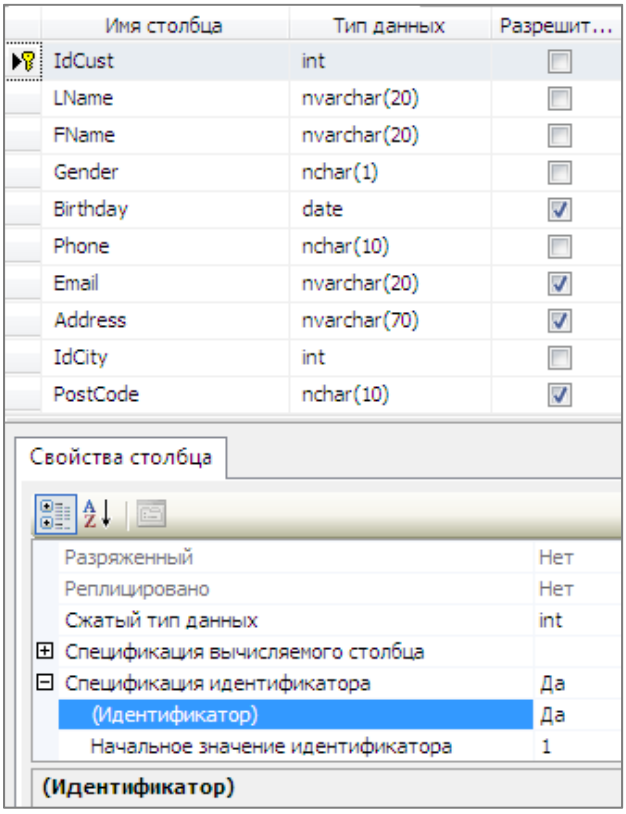


Рисунок 2.3 – Проект таблицы «*Customer*»

Задание. В соответствии с вышеприведенным описанием создайте оставшиеся пять таблиц: *City*, *Category*, *Product*, *Order* и *Ordered*.

2.3. Создание ограничений

Перед тем, как начать работать с таблицами, следует ограничить вводимые в них данные в целях обеспечения так называемой целостности данных, т. е. ограничить возникновение в базе данных некорректных или противоречивых данных вследствие добавления, изменения или удаления какой-либо записи (например, ввод отрицательной цены или количества товара).

Существует четыре типа целостности данных: *доменная*, *сущностная*, *ссылочная* и *пользовательская (или бизнес-правила)*.

Рассмотрим основные инструменты, предоставляемые в SQL Server для их реализации.

2.3.1. Обеспечение доменной целостности

Обеспечение *доменной целостности*, т. е. ограничение диапазона данных, вводимых пользователем в поле. Основными инструментами обеспечения доменной целостности являются *ограничения проверки* и *значения по умолчанию*.

Использование проверочных ограничений

Ограничения на проверку используются для ограничения данных, принимаемых полем, даже если они имеют корректный тип. Например, поле *«PostCode»* (*почтовый индекс*) имеет тип *nchar(10)*, т. е. теоретически оно может принимать буквы. Это может стать проблемой, поскольку не существует почтовых индексов с буквами. Рассмотрим, как создать ограничение на проверку, запрещающее вводить в это поле буквы.

1. В контекстном меню папки «*Ограничения*» таблицы *«Customer»* выберите команду *«Создать ограничение*».
2. В открывшемся окне «*Проверочные ограничения*» заполните следующие поля:

– *имя (CK_PostCode);*

– *выражение: ([PostCode] like '[0-9][0-9][0-9][0-9][0-9]'*). Данное выражение описывает ограничение, принимающее пять символов, которыми могут быть только цифры от 0 до 9;

– *описание (*ограничение на значения почтового индекса*)*.

3. Нажмите кнопку *«Закрыть*» и закройте конструктор таблиц (он был открыт, когда вы начали создавать ограничение) с сохранением изменений (рисунок 2.4).

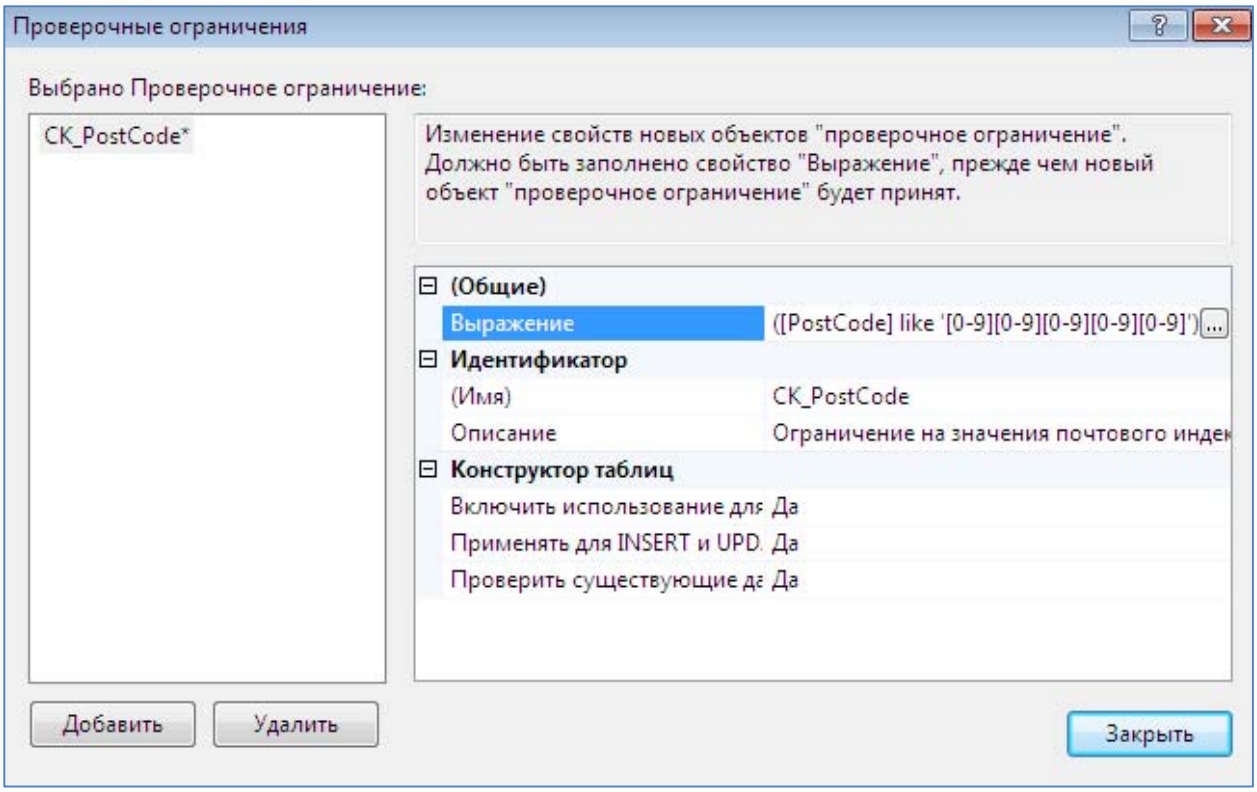


Рисунок 2.4 – Создание проверочных ограничений

Задание. Создайте ограничения в следующих таблицах.
Product **–** для полей *«InStock»* (*количество единиц товара на складе*) и *«Price»* (*цена за единицу*);
Order – для поля *«Discount»* (*размер скидки*);
Ordered – для поля *«Amount»* (*количество единиц товара в заказе*). Данные ограничения должны запрещать ввод отрицательных значений в указанные поля. Выражение проверки будет иметь вид (*Имя поля > 0*) для полей *«Price»* и *«Amount»*. Для полей *«InStock»* и *«Discount»* – (*Имя поля >= 0*).

Использование значений по умолчанию

Установка для полей значений по умолчанию это отличный способ избавить пользователя от излишней работы, если значения этих полей во всех записях, как правило, принимают одни и те же значения. Так в таблице заказов «*Order»* вполне логично определить по умолчанию значение поля «*OrdDate»* (*дата заказа*) в виде текущей даты. В этом случае при добавлении записи о новом заказе в случае пропуска этого поля оно будет автоматически заполняться значением системной даты. Для создания такого свойства выполните следующие шаги:

1. Раскройте папку *«Столбцы»* таблицы «*Order»* и в контекстном меню поля «*OrdDate*» выберите команду *«Изменить»*.
2. В свойстве столбца *«Значение или привязка по умолчанию»* введите *getdate()* (*функция T-SQL, возвращающая текущую системную дату*).
3. Щелкните кнопку *«Сохранить»* и выйдите из конструктора таблиц (рисунок 2.5).

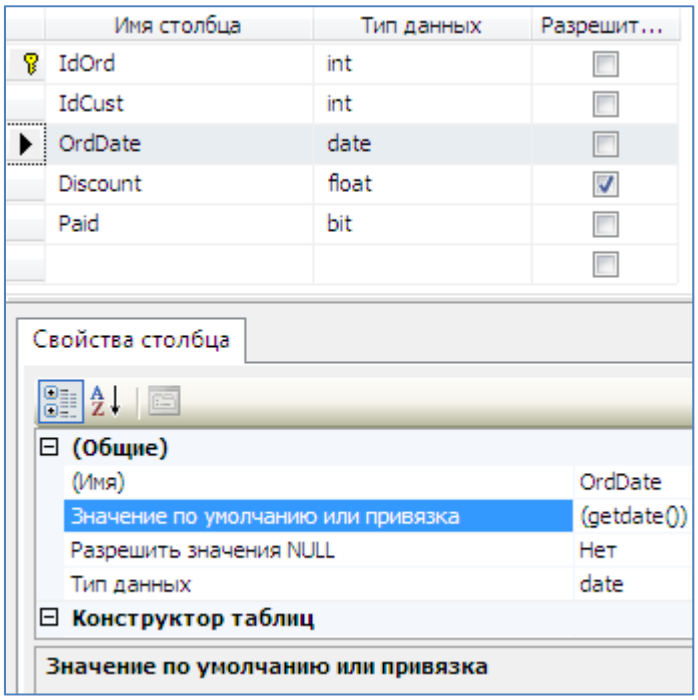


Рисунок 2.5 – Установка текущей даты по умолчанию

Задание. В таблице *«Product»* установите для поля *«InStock»* (*количество единиц товара на складе*) в качестве значения по умолчанию – 0 (нуль). В таблице *«Order»* установите для поля *«Paid»* (*отметка об оплате заказа*) в качестве значения по умолчанию – *False* (*не оплачено*).

2.3.2. Обеспечение сущностной целостности

Обеспечение гарантии уникальности записей в таблицах и предотвращение их дублирования. Основными инструментами обеспечения целостности сущностей являются *первичные ключи* и *ограничения уникальности*.

Создание первичных ключей

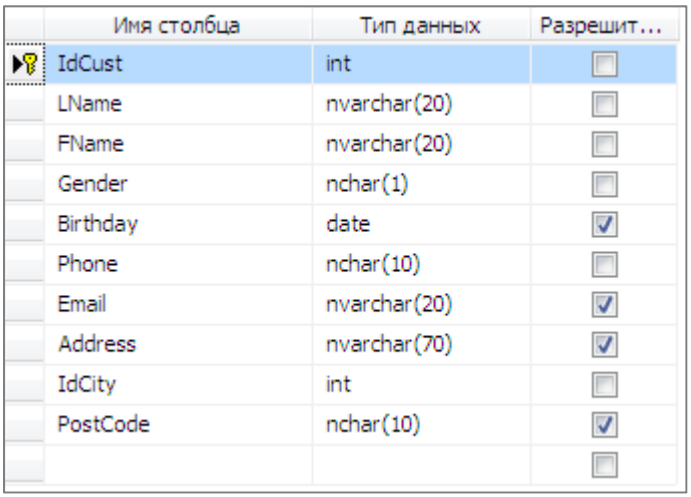
Первичный ключ используется для обеспечения гарантии уникальности каждой записи в таблице. Он состоит из одного (*простой ключ*) или нескольких (*составной ключ*) столбцов с гарантированно уникальными значениями для каждой записи таблицы. Если пользователь попытается ввести в поле первичного ключа дублирующее значение, будет сгенерирована ошибка и модификация данных будет отменена.

В качестве примера создадим первичный ключ для таблицы «*Customer*». В данном случае идеальным кандидатом на роль первичного ключа выступает столбец «*IdCust*», поскольку значения, содержащиеся в нем, являются уникальными по определению (*для него установлено свойство identity*). Следует отметить, что в качестве первичного ключа могут быть взяты и реальные атрибуты клиента, например ИНН, номер страхового свидетельства, серия и номер паспорта (*пример составного ключа*), но использование различных разновидностей, так называемых суррогатных ключей (*identity, uniqueidentifier*), обеспечивает большую степень сущностной целостности (поскольку реальные атрибуты могут все же со временем измениться) и является распространенной практикой. Для создания первичного ключа в таблице *«Customer»* выполните следующие шаги:

1. В контекстном меню таблицы *«Customer»* выберите команду *«Проект»*.

2. В окне конструктора таблиц щелкните правой кнопкой мыши на поле «*IdCust*», выберите команду «*Задать первичный ключ*» или нажмите кнопку  на панели инструментов. Обратите внимание на то, что слева от поля «*IdCust*» теперь отображается значок ключа, указывающий, что поле является первичным ключом.

3. Закройте конструктор таблиц с сохранением изменений (рисунок 2.6).



	Имя столбца	Тип данных	Разрешит...
▶	IdCust	int	☐
	LName	nvarchar(20)	☐
	FName	nvarchar(20)	☐
	Gender	nchar(1)	☐
	Birthday	date	☑
	Phone	nchar(10)	☐
	Email	nvarchar(20)	☑
	Address	nvarchar(70)	☑
	IdCity	int	☐
	PostCode	nchar(10)	☑
			☐

Рисунок 2.6 – Создание первичного ключа

Задание. Аналогичным образом создайте первичные ключи для остальных таблиц.

Таблица	*Первичный ключ* (PK)
City	IdCity
Category	Id Category
Product	IdProd
Order	IdOrd
Ordered	IdOrd, IdProd (*составной первичный ключ*)

Использование ограничений на уникальность

Между ограничениями первичного ключа и ограничениями на уникальность существует *два отличия*. *Первое* состоит в том, что первичные ключи используются вместе с внешними ключами для обеспечения целостности ссылок. *Второе* отличие заключается в том, что ограничения на уникальность позволяют вставлять в его поля пустые значения (*null*), чего нельзя делать с первичными ключами. Во всем остальном они служат одной цели – обеспечить уникальность данных, вставляемых в поле. Ограничение на уникальность следует использовать в тех случаях, когда нужно гарантировать, что дублирующие значения не будут добавляться в поле, не являющееся частью первичного ключа, в частности, все потенциальные ключи должны быть организованы в виде ограничений уникальности. Хорошим примером такого поля, требующего ограничение на уникальность, является поле ИНН или серия и номер паспорта, поскольку эти поля должны быть уникальными у каждого человека. Такого идеального кандидата на роль уникального ограничения в нашей таблице «*Customer*» нет. Поэтому создадим его по полю «*Phone*», которое также повторяться у разных клиентов не должно.

1. Для открытия конструктора таблиц в контекстном меню таблицы «*Customer*» выберите команду «*Проект*». На панели инструментов нажмите на кнопку «*Управление индексами и ключами*»  .

2. В открывшемся окне «*Индексы и ключи*» щелкните кнопку «*Добавить*» и введите следующие параметры для нового уникального ключа (рисунок 2.7):

- столбцы – *Phone;*
- тип – *Уникальный ключ;*
- (имя) – *CK_Phone.*

3. Закройте конструктор таблиц с сохранением изменений.

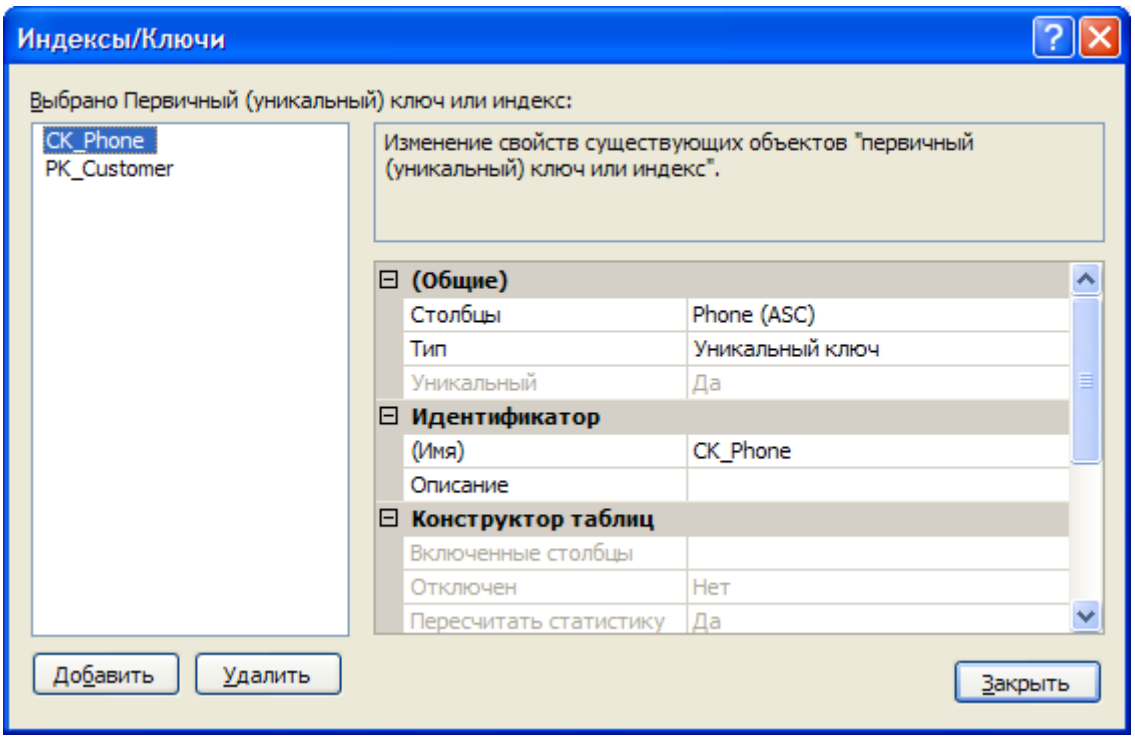


Рисунок 2.7 – Создание уникального ключа

Задание. Аналогичным образом создайте ограничение уникальности по полю *«CityName»* таблицы *«City»*, чтобы обеспечить отсутствие в справочнике городов с одинаковыми названиями, а также по полю *«Description»* таблицы *«Product»*, чтобы иметь возможность отличить один товар от другого.

2.3.3. Обеспечение целостности ссылок

Сейчас в базе данных *«Sales»* имеется шесть таблиц, которые тесно взаимосвязаны между собой, и данные, содержащиеся в них, должны быть согласованы и непротиворечивы. Например, в таблице *«Order»* не должно быть записей о заказах для клиента, данные о котором отсутствуют в таблице *«Customer»*. Чтобы гарантировать отсутствие в базе данных таких записей, необходимо обеспечить целостность ссылок.

Суть обеспечения целостности ссылок очевидна из названия: данные в одной таблице, ссылающиеся на данные из другой таблицы, защищены от некорректного обновления. В терминологии SQL Server это называется *декларативной ссылочной целостностью*

и достигается путем связывания первичного ключа одной из таблиц с внешним ключом другой таблицы (*создается так называемое ограничение внешнего ключа*).

Внешний ключ используется в комбинации с первичным для связывания двух таблиц по общему столбцу (столбцам). К примеру, можно связать таблицы *«Customer»* и *«Order»* по столбцу *«IdCust»*, который присутствует в обеих таблицах. Поскольку поле *«IdCust»* таблицы *«Customer»* является его первичным ключом, можно использовать поле *«IdCust»* таблицы *«Order»* в качестве внешнего ключа, который свяжет эти две таблицы. После организации такого ограничения будет невозможно добавить запись в таблицу *«Order»*, если в таблице *«Customer»* нет записи с соответствующим значением *«IdCust»*. Кроме того, при отсутствии каскадирования невозможно удалить запись из таблицы *«Customer»* при наличии связанных с ней записей в таблице *«Order»*, поскольку нельзя оставлять заказ без информации о клиенте. Для создания описанного ограничения внешнего ключа в *Management Studio* выполните следующие шаги:

1. В контекстном меню папки *«Ключи»* таблицы *«Order»* выберите команду *«Создать внешний ключ…»* (рисунок 2.8).

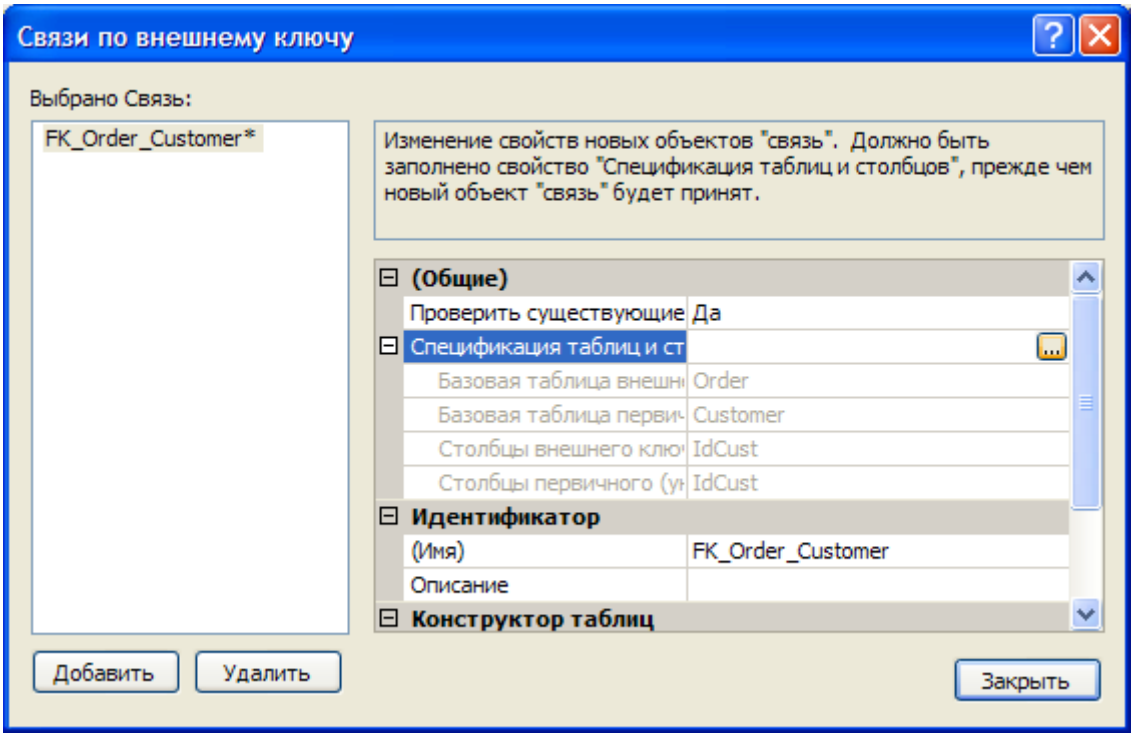


Рисунок 2.8 – Создание внешнего ключа

2. В открывшемся окне *«Отношения внешнего ключа»* заполните следующие поля:

– *(имя) – FK_Order_Customer;*

– *спецификация таблиц и столбцов*. Для заполнения данного блока щелкните на кнопке с многоточием и в появившемся окне *«Таблицы и столбцы»* в качестве таблицы первичного ключа выберите пункт *«Customer»*, а полей связи – пункт *«IdCust»* (рисунок 2.9).

3. Закройте все открывшиеся окна с сохранением изменений.

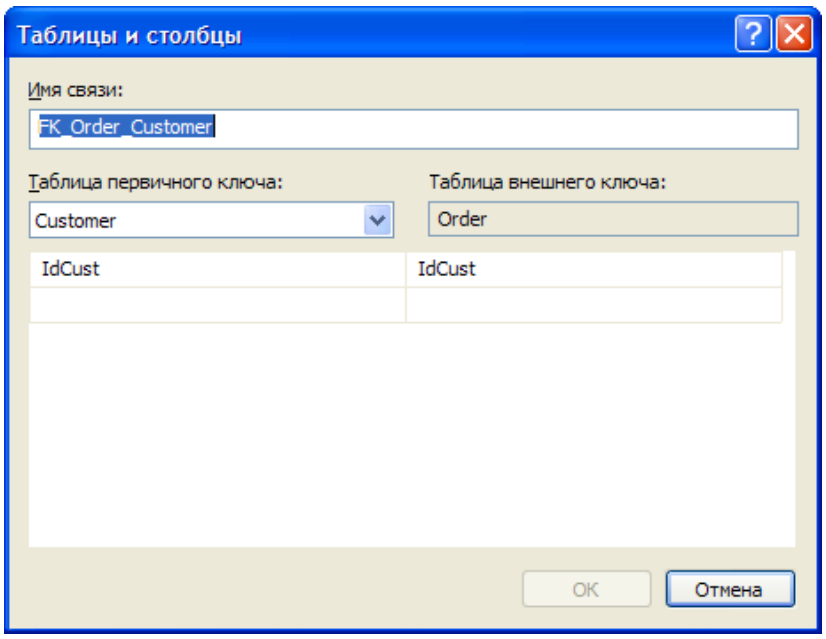


Рисунок 2.9 – Связь двух таблиц по общему полю

Использование каскадной ссылочной целостности

При наличии ограничения внешнего ключа с параметрами по умолчанию вы не можете удалить запись или изменить значение первичного ключа главной таблицы в случае наличия связанных записей в подчиненной таблице (*в которой организовано ограничение внешнего ключа*). Однако это поведение можно изменить, используя *каскадную ссылочную целостность*.

Настроить правила каскадирования можно при создании ограничения внешнего ключа в окне *«Связи по внешнему ключу»*, изменяя значения параметров *«Правило обновления»* и *«Правило удаления»* блока *«Спецификация INSERT* и *UPDATE»*. Оба этих параметра могут содержать четыре значения, описанные в таблице 2.8.

Таблица 2.8 – Правила каскадного удаления (обновления)

Настройка	Правило удаления	Правило обновления
Нет действия	Невозможно удалить в главной таблице строку, на которую есть ссылки в подчиненной	Невозможно обновить значения полей первичного ключа главной таблицы при наличии связанных записей в подчиненной
Каскадно	При удалении строки в главной таблице все связанные строки в подчиненной также будут удалены	При обновлении значений полей первичного ключа главной таблицы соответствующим образом будут изменены и их значения во всех связанных строках подчиненной таблицы
Присвоить Null	При удалении строки в главной таблице во всех связанных строках подчиненной таблицы полям вторичного ключа будет присвоено значение Null	При обновлении значений полей первичного ключа главной таблицы во всех связанных строках подчиненной таблицы полям вторичного ключа будет присвоено значение Null
Присвоить значение по умолчанию	При удалении строки в главной таблице во всех связанных строках подчиненной таблицы полям вторичного ключа будут присвоены значения по умолчанию	При обновлении значений полей первичного ключа главной таблицы во всех связанных строках подчиненной таблицы полям вторичного ключа будут присвоены значения по умолчанию

Задание. Создайте ограничение внешнего ключа *FK_Ordered_Order* в таблице «*Ordered*» для связи таблиц «*Order*» и «*Ordered*» по полю «*IdOrd*». При этом настройте правило каскадного удаления, установив в качестве параметра «*Спецификация INSERT*» и «*UPDATE\Правило удаления*» значение «*Каскадно*», что приведет к автоматическому удалению всех товаров из заказа при удалении самого заказа.

2.4. Использование диаграмм баз данных

Диаграммы базы данных представляют собой графическое отображение схемы (*целиком или частично*) базы данных с таблицами и столбцами, а также связей между ними. Создадим диаграмму базы данных:

1. В контекстном меню папки «*Диаграммы базы данных*» выберите команду «*Создать диаграмму базы данных*».
2. В диалоговом окне «*Добавление таблиц*» выберите все таблицы и нажмите на кнопку «*Добавить*».
3. Добавив таблицы, нажмите кнопку «*Закрыть*» и увидите созданную диаграмму базы данных (рисунок 2.10).

Используя диаграмму базы данных, ограничения внешнего ключа можно создавать значительно быстрее, лишь перетаскивая поля из одной таблицы в другую. В качестве примера создадим внешний ключ в таблице *«Customer»* по полю *«IdCity»* для связи с таблицей *«City»*.

1. Выделите в таблице *«City»* поле *«IdCity»* и, не отпуская кнопку мыши, перетащите его на поле *«IdCity»* таблицы *«Customer»*.
2. В диалоговых окнах *«Таблицы и столбцы»* и *«Связь по внешнему ключу»* примите настройки по умолчанию.
3. Сохраните диаграмму базы данных под именем *«SchemeDB»*.
4. Расположите таблицы в канонической форме (главные таблицы выше подчиненных) в соответствии с вышеприведенным рисунком.

Задание. Аналогичным образом создайте связь между таблицами *«Product»* и *«Ordered»* по полю *«IdProduct»*. Окончательный список связей между таблицами со всеми их характеристиками представлен в таблице 2.9.

Таблица 2.9 – Список связей между таблицами

Главная таблица	Подчиненная таблица	Поле связи (внешний ключ)	Правила каскадирования
City	Customer	IdCity	Без действия
Customer	Order	IdCust	Без действия
Order	Ordered	IdOrd	Каскадное удаление
Product	Ordered	IdProd	Без действия
Category	Product	IdCategory	Без действия

Задание. После настройки всех ограничений заполните таблицы данными для последующей работы. Для этого в контекстном меню таблицы выберите команду *«Изменить первые 200 строк»* и в появившейся в рабочей области вкладке введите новые записи, заполняя все необходимые столбцы. В процессе внесения данных проверьте работоспособность всех созданных ранее ограничений.

Ограничения проверки: попробуйте ввести в поле *«PostCode»* (*почтовый индекс*) таблицы *«Customer»* нечисловые значения, а в поля *«InStock»* и *«Price»* таблицы *«Product»*, а также в поле *«Amount»* таблицы *«Ordered»* – отрицательные значения.

Значения по умолчанию: убедитесь, что при пропуске полей *«OrdDate»* и *«InStock»* таблиц *«Order»* и *«Product»* для них устанавливаются значения по умолчанию в виде текущей системной даты и нуля соответственно.

Ограничения первичного и уникального ключа: попробуйте ввести в таблицы записи с дублирующими значениями первичного или уникального ключа.

Ограничения внешнего ключа: попробуйте ввести несогласованные данные в связанные таблицы (например, заказ для несуществующего клиента) или удалить запись из любой главной таблицы при наличии связанных записей в подчиненной при отсутствии правил каскадирования.

Правила каскадирования: убедитесь, что при удалении записи из таблицы *«Order»* все связанные записи из таблицы *«Ordered»* удаляются автоматически.

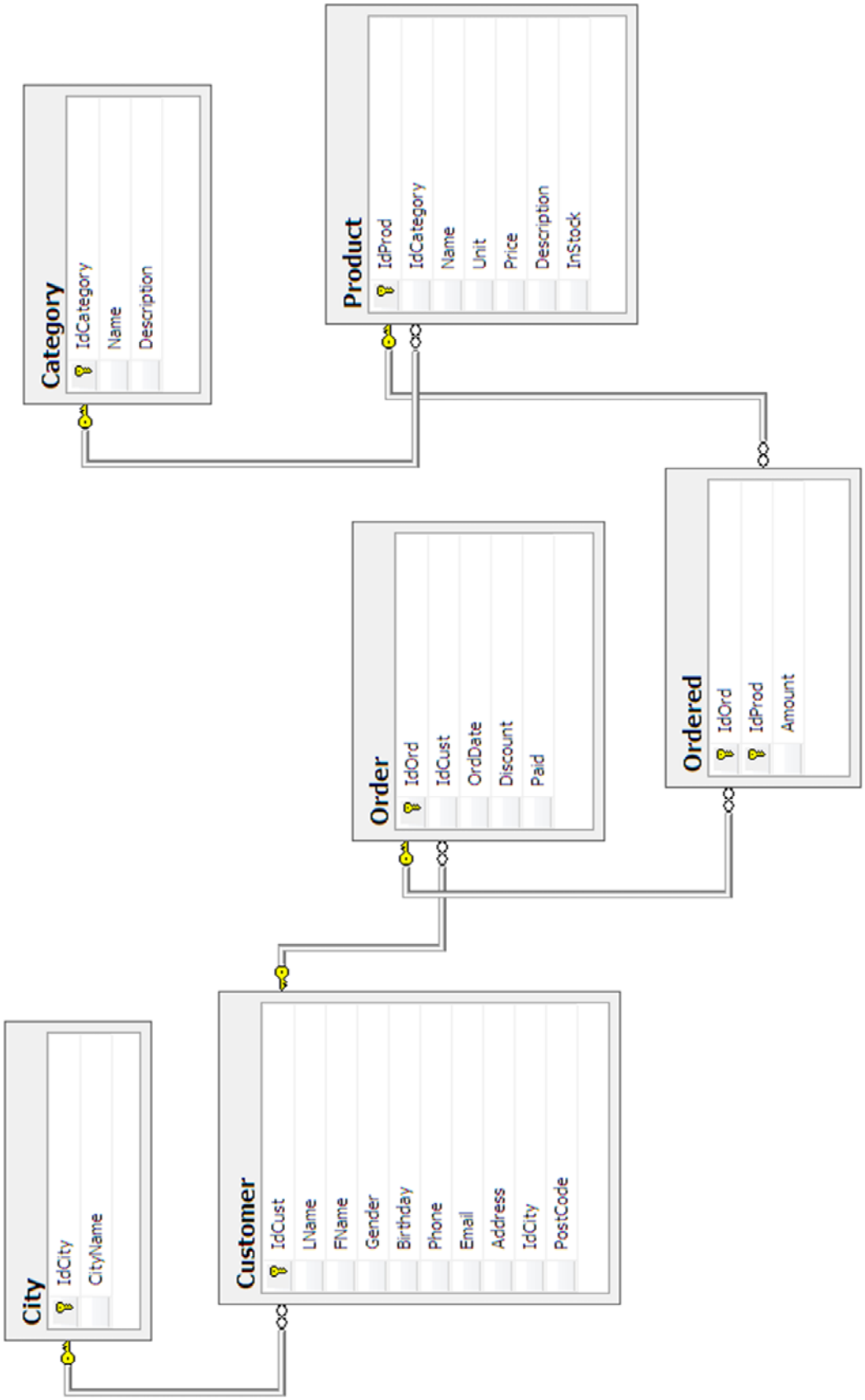


Рисунок 2.10 – Схема базы данных

ЛАБОРАТОРНАЯ РАБОТА 3
ОСНОВЫ TRANSACT-SQL

SQL – это аббревиатура выражения Structured Query Language (язык структурированных запросов). SQL основывается на реляционной алгебре и специально разработан для взаимодействия с реляционными базами данных.

SQL является, прежде всего, информационно-логическим языком, предназначенным для описания хранимых данных, их извлечения и модификации. SQL не является языком программирования. Вместе с тем конкретные реализации языка, как правило, включают различные процедурные расширения.

Язык SQL представляет собой совокупность операторов, которые можно разделить на четыре группы:

- DDL (Data Definition Language) – операторы определения (описания) данных;
- DML (Data Manipulation Language) – операторы манипуляции данными;
- DCL (Data Control Language) – операторы определения доступа к данным;
- TCL (Transaction Control Language) – операторы управления транзакциями.

3.1. Оператор SELECT: базовые возможности

Рассмотрим более подробно группу операторов DML, которая содержит 4 составных оператора, наименования которых определяют основное их предназначение:

- оператор SELECT обеспечивает выборку множества кортежей отношений (строк таблиц), удовлетворяющих заданным ограничениям, и (возможно) обработку выбранных данных;
- оператор INSERT обеспечивает вставку (добавление) в таблицы новых строк с заданными значениями их атрибутов;
- оператор UPDATE обновляет значения атрибутов в строках таблиц, удовлетворяющих заданным ограничениям;
- оператор DELETE удаляет из таблиц строки, удовлетворяющие заданным ограничениям.

Среди перечисленных DML-операторов оператор SELECT, обеспечивающий выборку строк таблиц, является основным, т. к. выборка строк производится перед их удалением (DELETE), обновлением значений (UPDATE) или генерацией перед их вставкой (INSERT) в таблицу.

Оператор SELECT позволяет извлекать информацию из одной или нескольких таблиц базы данных и даже из нескольких баз данных. Результаты выполнения инструкции SELECT помещаются в еще одну таблицу, называемую *результирующим набором (result set)*.

Составной оператор выборки SELECT включает 6 именованных разделов (SELECT, FROM, WHERE, ORDER BY, GROUP BY и HAVING), из которых первые два являются обязательными.

Результатом выполнения оператора SELECT является виртуальное отношение, схема которого определяется списком атрибутов, заданным в разделе SELECT этого оператора, а состав кортежей – параметрами остальных его разделов.

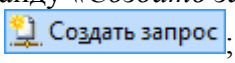
Рассмотрим простейшие SQL-запросы на выборку данных (оператор SELECT). SELECT – наиболее часто используемый SQL-оператор. Он предназначен для выборки информации из таблиц. Чтобы при помощи оператора SELECT извлечь данные из таблицы, нужно указать как минимум две вещи – что вы хотите выбрать и откуда.

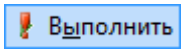
Выборка отдельных столбцов

Пример 1: SELECT Name FROM Product

В приведенном выше операторе используется оператор SELECT для выборки одного столбца под названием «Name» из таблицы «Product». Искомое имя столбца указывается сразу после ключевого слова «SELECT», а ключевое слово «FROM» указывает на имя таблицы, из которой выбираются данные.

Для создания и тестирования данного запроса в Management Studio выполните следующие шаги:

- в контекстном меню базы *«Sales»* выберите команду *«Создать запрос»* или щелкните соответствующую кнопку на панели инструментов 

;
- в открывшемся окне создания нового запроса введите описанную выше инструкцию SQL;
- для запуска запроса на выполнение щелкните кнопку 

на панели инструментов или нажмите клавишу *«F5»*. В нижней части экрана должен появиться результат (рисунок 3.1).

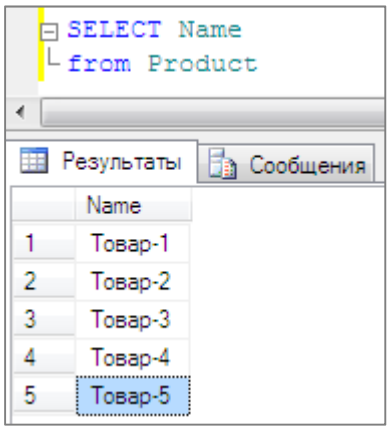


Рисунок 3.1 – Результат выполнения инструкции SELECT

Management Studio позволяет сохранять пакеты SQL. Это полезно для сохранения сложных запросов, которые будут повторно запускаться в будущем. Для этого щелкните кнопку  на панели инструментов. По умолчанию файлы запросов сохраняются с расширением .sql. В дальнейшем сохраненный запрос может быть открыт командой *«Открыть файл»*.

Выборка нескольких столбцов

Для выборки из таблицы нескольких столбцов используется тот же оператор SELECT. Отличие состоит в том, что после ключевого слова SELECT необходимо через запятую указать несколько имен столбцов.

Пример 2: SELECT Name, InStock FROM Product

Выборка всех столбцов

Помимо возможности осуществлять выборку определенных столбцов (одного или нескольких), при помощи оператора SELECT можно запросить все столбцы, не перечисляя каждый из них. Для этого вместо имен столбцов вставляется групповой символ «звездочка» (*). Это делается следующим образом.

Пример 3: SELECT * FROM Product

Создание вычисляемых полей

Инструкция SELECT кроме имен столбцов таблиц может также включать так называемые вычисляемые поля. Вычисляемых полей на самом деле в таблицах базы данных нет. Они создаются «на лету» SQL-оператором SELECT. Рассмотрим следующий пример:

Пример 4: SELECT LName +' '+ FName FROM Customer

Здесь создается вычисляемое поле. Оно позволяет объединить (произвести конкатенацию) с помощью оператора + фамилия, пробел и имя клиента в одно поле (столбец) (рисунок 3.2).

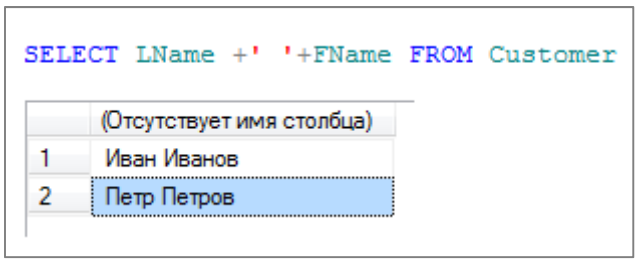


Рисунок 3.2 – Результат объединения двух полей в одно

Еще одним способом использования вычисляемых полей является выполнение математических операций над выбранными данными.

Пример 5: SELECT datediff (year, Birthday, GETDATE()) FROM Customer

Здесь с помощью функций datediff определяется количество полных лет каждого клиента.

Оператор AS. Именование результирующих столбцов

Так как вычисляемых полей нет в таблицах базы данных, выходной столбец не имеет названия – *«Отстутствует имя столбца»* (рисунок 3.2).

С помощью оператора AS можно изменить название выходного столбца или определить его псевдоним. Рассмотрим следующий пример.

Пример 6: SELECT LName +' '+ FName AS 'Фамилия и имя клиента' FROM Customer

В этом примере вычисляемому полю с помощью оператора AS дан псевдоним *«Фамилия и имя клиента»* (рисунок 3.3).

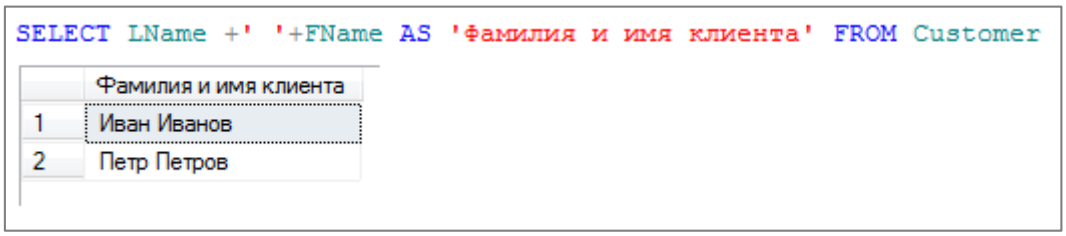


Рисунок 3.3 – Определение псевдонима выходного столбца

Псевдоним может быть задан и для обычного столбца таблицы.

Пример 7: SELECT IdCust AS 'Номер клиента', LName +' '+ FName AS 'Фамилия и имя клиента' FROM Custome

В этом примере столбцу *«IdCust»* задан псевдоним *«Номер клиента»* (рисунок 3.4).

```
SELECT  IdCust AS 'Номер клиента',
        LName +' '+FName AS 'Фамилия и имя клиента'

FROM Customer
```

	Номер клиента	Фамилия и имя клиента
1	1	Иван Иванов
2	5	Петр Петров

Рисунок 3.4 – Результат изменения названия выходного столбца

3.2. Сортировка данных

В результате выполнения запроса на выборку данные выводятся в том порядке, в котором они находятся в таблице. Для точной сортировки выбранных при помощи оператора SELECT данных используется предложение ORDER BY. В этом предложении указывается имя одного или нескольких столбцов, по которым необходимо отсортировать результаты.

Пример 8: SELECT IdProd, Name, InStock FROM Product ORDER BY InStock

Это выражение идентично предыдущему, за исключением предложения ORDER BY, которое указывает СУБД отсортировать данные по возрастанию значений столбца *«InStock»*.

Пример 9: SELECT LName, FName, Phone FROM Customer ORDER BY LName

В данном случае получим отсортированный в алфавитном порядке список клиентов по фамилии.

Сортировка по нескольким столбцам

Чтобы осуществить сортировку по нескольким столбцам, просто укажите их имена через запятую. В следующем коде выбираются три столбца, а результат сортируется по двум из них: сначала по категории товара, а затем по названию товара.

Пример 10
```
SELECT IdCategory, Name, InStock
FROM Product
ORDER BY IdCategory, Name
```

Важно понимать, что при сортировке по нескольким столбцам порядок сортировки будет таким, который указан в запросе. Другими словами, в примере, приведенном выше, продукция сортируется по столбцу *«Name»*, только если существует несколько строк с одинаковыми значениями *«IdCategory»*. Если никакие значения столбца *«IdCategory»* не совпадают, данные по столбцу *«Name»* сортироваться не будут.

Сортировка с использованием сложного выражения на основе столбцов

В качестве критерия сортировки также можно использовать сложное выражение на основе столбцов.

Пример 11
SELECT Name, InStock*Price
FROM Product
ORDER BY InStock*Price

Сортировка по псевдониму столбца

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

Пример 12
SELECT Name, InStock * Price AS TotalSum
FROM Product
ORDER BY TotalSum

Указание направления сортировки

В предложении ORDER BY можно также использовать порядок сортировки по убыванию. Для этого необходимо указать ключевое слово DESC. В следующем примере продукция сортируется по количеству в убывающем порядке плюс по названию продукта.

Пример 13
SELECT IdProd, Description, InStock
FROM Product
ORDER BY InStock DESC, Description

Ключевое слово DESC применяется только к тому столбцу, после которого оно указано. В предыдущем примере ключевое слово DESC было указано для столбца *«InStock»*, но не для *«Description».* Таким образом, столбец *«InStock»* отсортирован в порядке убывания, а столбец *«Description»* – в возрастающем порядке (принятым по умолчанию).

3.3. Фильтрация данных

В таблицах баз данных обычно содержится много информации и довольно редко возникает необходимость выбирать все строки таблицы. Гораздо чаще бывает, что нужно извлечь какую-то часть данных таблицы для каких-либо действий или отчетов.

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

Пример 14: SELECT * FROM Product WHERE InStock = 0

Этот оператор извлекает значения всех столбцов из таблицы товаров, но показывает не все строки, а только те, значение в столбце *«InStock»* (количество товаров на складе) которых равно 0, т. е. только список отсутствующих на складе товаров.

При совместном использовании предложений ORDER BY и WHERE предложение ORDER BY должно следовать после WHERE.

В предыдущем примере проводилась проверка на равенство, т. е. определялось, содержится ли в столбце указанное значение. В T-SQL можно применять следующие операции сравнения (таблица 3.1).

Таблица 3.1 – Операции сравнения, применяемые в T-SQL

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

В качестве условия могут использоваться и более сложные выражения.

Пример 15: SELECT * FROM Product WHERE Price * InStock < 1000

Логические операторы

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

AND: операция логического И. Она объединяет два выражения (*выражение 1 AND выражение 2*). Только если оба этих выражения одновременно истинны, то и общее условие оператора AND также будет истинно (то есть если и первое условие истинно, и второе).

OR: операция логического ИЛИ. Она также объединяет два выражения (*выражение 1 OR выражение 2*). Если хотя бы одно из этих выражений истинно, то общее условие оператора OR также будет истинно (то есть если или первое условие истинно, или второе).

NOT: операция логического отрицания. Если выражение в этой операции ложно, то общее условие истинно (*NOT-выражение*).

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

Например, выберем все товары некоторой категории, количество которых на складе превышает заданное значение.

Пример 16: SELECT * FROM Product WHERE IdCategory = 1 AND InStock < 100

Теперь выберем товары первой либо второй категории, запасы которых ниже некоторого значения. Так как оператор AND имеет более высокий приоритет, необходимо использовать скобки, чтобы переопределить порядок операций.

Пример 17: SELECT * FROM Product WHERE (IdCategory = 1 OR IdCategory = 2) AND InStock < 100

Применение оператора NOT

Выберем все товары, кроме товаров заданной категории.

Пример 18: SELECT * FROM Product WHERE NOT IdCategory = 1

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

Пример 19: SELECT * FROM Product WHERE IdCategory < > 1

Применение оператора IS NULL

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

Например, выберем всех клиентов, для которых не указан контактный телефон.

Пример 20: SELECT * FROM Customer WHERE Phone IS NULL

Если необходимо получить строки, у которых поле *«Phone»* не равно NULL, то можно использовать оператор NOT.

Пример 21: SELECT * FROM Customer WHERE Phone IS NOT NULL

Применение оператора IN

Оператор IN позволяет определить набор значений, которые должны иметь столбцы: *WHERE-выражение [NOT] IN (выражение)*.

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

Например, выберем товары первой, второй либо третьей категорий.

Пример 22: SELECT * FROM Product WHERE IdCategory IN (1, 2, 3)

Можно было бы все эти значения проверить и через оператор OR.

Пример 23: SELECT * FROM Product WHERE IdCategory = 1 or IdCategory = 2 or IdCategory = 3

Но использование оператора IN гораздо удобнее, особенно если подобных значений очень много.

С помощью оператора NOT можно найти все строки, которые, наоборот, не соответствуют набору значений.

Пример 24: SELECT * FROM Product WHERE IdCategory NOT IN (1, 2, 3)

Применение оператора BETWEEN

Для поиска диапазона значений можно использовать оператор BETWEEN. Его синтаксис немного отличается от других операторов предложения WHERE, так как для него требуются два значения (начальное и конечное): *WHERE-выражение [NOT] BETWEEN начальное_значение AND конечное_значение*.

Например, получим все товары, цена которых от 50 до 100 (начальное и конечное значения также включаются в диапазон).

Пример 25: SELECT * FROM Product WHERE Price BETWEEN 50 AND 100

Предыдущий запрос можно также записать, используя операции сравнения.

Пример 26: SELECT * FROM Product WHERE Price >= 50 AND Price <= 100

Если требуется, наоборот, выбрать те строки, которые не попадают в заданный диапазон, то применяется оператор NOT.

Пример 27: SELECT * FROM Product WHERE Price NOT BETWEEN 50 AND 100

Также можно использовать более сложные выражения. Например, получим товары, запасы которых на складе на определенную сумму.

Пример 28: SELECT * FROM Product WHERE Price * InStock BETWEEN 500 AND 1000

Задание. Используя операцию BETWEEN, составьте на языке SQL запрос на выборку следующих данных: список всех заказов за определенный период времени (например, сентябрь 2021 года), отсортированный по дате заказа. Проделайте то же самое, только вместо оператора BETWEEN используйте операции сравнения.

Применение оператора LIKE

Оператор LIKE используется для сопоставления с образцом, т. е. он сравнивает значения столбца с указанным шаблоном. Столбец может быть любого символьного типа данных или типа дата. Общая форма оператора LIKE выглядит таким образом:

WHERE column [NOT] LIKE выражение шаблона

Для определения шаблона может применяться ряд специальных символов подстановки (таблица 3.2).

Таблица 3.2 – Специальные символы подстановки

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

Некоторые примеры использования подстановок:

1. Инструкция Select * From Customer WHERE FName LIKE 'A%' выполняет поиск и выдает всех клиентов, имена которых начинаются на букву ‘А’.

2. Инструкция Select * From Customer WHERE LName LIKE '_етров' выполняет поиск и выдает всех клиентов, фамилии которых состоят из шести букв и заканчиваются сочетанием ‘етров’ (Петров, Ветров и т. п.).

3. Инструкция Select * From Customer WHERE LName LIKE '[Л-С]омов' выполняет поиск и выдает всех клиентов, фамилии которых заканчиваются на ‘омов’ и начинаются на любую букву в промежутке от ‘Л’ до ‘С’, например Ломов, Ромов, Сомов и т. п.

4. Инструкция Select * From Customer WHERE LName LIKE 'ив[^а]%' выполняет поиск и выдает всех клиентов с фамилиями, начинающимися на ‘ив’, в которых третья буква отличается от ‘а’.

Задание. Составьте на языке SQL запросы на выборку следующих данных:
1) список всех товаров, в описании которых встречается словосочетание «пальмовое масло» (либо другое), с указанием их остатка на складе;
2) сведения обо всех клиентах с фамилией, оканчивающейся буквой «в»;
3) список товаров, в названии которых есть буква «а» или «в»;
4) сведения обо всех клиентах, в электронном адресе которых есть символ «_» (нижнее подчеркивание) или «.» (точка);
5) сведения обо всех клиентах, номер телефона которых содержит комбинацию «29».

3.4. Группировка

Агрегатные функции

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

- AVG (находит среднее значение);
- SUM (находит сумму значений);
- MIN (находит наименьшее значение);
- MAX (находит наибольшее значение);
- COUNT (находит количество строк в запросе).

В качестве аргумента все агрегатные функции принимают выражение, которое представляет критерий для определения значений. Зачастую в качестве выражения выступает название столбца, над значениями которого надо проводить вычисления.

Выражение в функциях AVG и SUM должно представлять числовое значение. Выражение в функциях MIN, MAX и COUNT может представлять числовое или строковое значение или дату.

Все агрегатные функции за исключением COUNT(*) игнорируют значения NULL.

Функция Avg

Функция Avg возвращает среднее значение на диапазоне значений столбца таблицы.

Найдем среднюю цену товаров из таблицы *«Product»*.

Пример 29: SELECT AVG(Price) AS AveragePrice FROM Product

Для поиска среднего значения в качестве выражения в функцию передается столбец *«Price».* Для получаемого значения устанавливается псевдоним AveragePrice, хотя можно его и не устанавливать.

Также мы можем применить фильтрацию. Например, найти среднюю цену для товаров определенной категории.

Пример 30: SELECT AVG(Price) FROM Product WHERE IdCategory = 1

Кроме того, мы можем находить среднее значение для более сложных выражений. Например, найдем среднюю сумму всех товаров, учитывая их количество на складе.

Пример 31: SELECT AVG(Price * InStock) FROM Product

Функция Count

Функция Count вычисляет количество строк в выборке. Есть две формы этой функции. Первая форма COUNT(*) подсчитывает число строк в выборке.

Пример 32: SELECT COUNT(*) FROM Product

Вторая форма функции вычисляет количество строк по определенному столбцу, при этом строки со значениями NULL игнорируются. К примеру, определим количество оплаченных заказов.

Пример 33: SELECT Count(Paid) FROM Order Where Paid = 'True'

Функции Min и Max

Функции Min и Max возвращают соответственно минимальное и максимальное значение по столбцу. Например, найдем минимальную цену среди товаров.

Пример 34: SELECT MIN(Price) FROM Product

Поиск максимальной цены.

Пример 35: SELECT MAX(Price) FROM Product

Функция Sum

Функция Sum вычисляет сумму значений столбца. Например, подсчитаем общее количество товаров на складе.

Пример 36: SELECT SUM(InStock) FROM Product

Также вместо имени столбца может передаваться вычисляемое выражение. Например, найдем общую стоимость всех имеющихся товаров.

Пример 37: SELECT SUM(InStock * Price) FROM Product.

Операторы ALL и DISTINCT

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

По умолчанию вместо DISTINCT применяется оператор ALL, который выбирает все строки. Так как этот оператор неявно подразумевается при отсутствии DISTINCT, то его можно не указывать.

Задание. Составьте на языке SQL запрос на выборку следующих данных:
1) суммарная стоимость всех заказов;
2) заказ с максимальной стоимостью;
3) количество заказов за определенный период;
4) размер скидки по каждому заказу.

Запросы с группировкой строк. Операторы GROUP BY и HAVING

Оператор GROUP BY

Описанные выше агрегатные функции применялись ко всей таблице. Однако часто при создании отчетов появляется необходимость в формировании промежуточных итоговых значений, то есть относящихся к данным не всей таблицы, а ее частей. 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).

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 выражение 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.

Самым простым вариантом использования фразы GROUP BY является группировка по значениям одного столбца. Например, определим количество товара каждой категории.

Пример 38
```
SELECT IdCategory, COUNT(*) AS 'Кол-во'
FROM Product
GROUP BY IdCategory
```

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. Например, определим количество товаров в каждой категории, названия которых начинаются с буквы «А».

Пример 39
```
SELECT IdCategory, COUNT(*) AS 'Кол-во'
FROM Product
WHERE Name LIKE 'A%'
GROUP BY IdCategory
```

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

Пример 40
```
SELECT OrdDate AS 'Дата заказа', count(Paid) AS 'Кол. оплаченных заказов'
FROM [Order]
WHERE Paid = 1
GROUP BY OrdDate, Paid
```

Задание. Определите с помощью SQL-запроса:
1) количество оплаченных заказов на каждую дату;
2) количество оформленных заказов на каждую дату, но еще не оплаченных.

Оператор HAVING

Для отбора строк среди полученных групп применяется фраза 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:

1. Фразой WHERE отбираются строки, удовлетворяющие указанному в ней условию.
2. Фраза GROUP BY группирует отобранные строки.
3. Фразой HAVING отбираются группы, удовлетворяющие указанному в ней условию.

Значение условия, указываемого во фразе HAVING, должно быть уникальным для всех строк каждой группы. Поэтому правила использования имен столбцов и агрегатных функций во фразе HAVING такие же, как и для фразы SELECT при наличии фразы GROUP BY. Это значит, что во фразе HAVING в качестве операндов сравнения можно использовать только группируемые столбцы или агрегатные функции.

С помощью следующего запроса определим города, количество клиентов из которых больше 10.

Пример 41

```
SELECT IdCity
FROM Customer
GROUP BY IdCity
HAVING COUNT(*)>10
```

Задание. Сформулируйте на языке SQL запросы на выборку следующих данных:
1) список всех заказов с указанием их суммарной стоимости;
2) список клиентов, которые за указанный период совершили более 3 заказов.

ЛАБОРАТОРНАЯ РАБОТА 4
ВЛОЖЕННЫЕ ЗАПРОСЫ (ПОДЗАПРОСЫ)

Во всех рассмотренных ранее примерах значения столбцов сравниваются с выражением, константой или набором констант. Кроме таких возможностей сравнения язык Transact-SQL позволяет сравнивать значения столбца с результатом другой инструкции SELECT. Такая конструкция, где предложение WHERE инструкции SELECT содержит одну или больше вложенных инструкций SELECT, называется **подзапросом** (subquery).

Первая инструкция SELECT подзапроса называется ***внешним запросом*** *(outer query)*, а внутренняя инструкция (или инструкции) SELECT, используемая в сравнении, называется ***вложенным запросом*** *(inner query).* Первым выполняется вложенный запрос, а его результат передается внешнему запросу.

Подзапросы можно использовать внутри инструкций SELECT, INSERT, UPDATE, DELETE или внутри другого вложенного запроса.

В инструкции SELECT подзапросы можно использовать четырьмя способами:

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

Существует два типа подзапросов: **независимые** (простые) и **связанные** (коррелированные). В независимых подзапросах вложенный запрос логически выполняется ровно один раз. Связанный запрос отличается от независимого тем, что его значение зависит от переменной, получаемой от внешнего запроса. Таким образом, вложенный запрос связанного подзапроса выполняется каждый раз, когда система получает новую строку от внешнего запроса.

4.1. Независимые (простые) подзапросы

Рассматривая независимые подзапросы, следует выделить три частных случая:

1. Подзапросы, возвращающие единственное значение.
2. Подзапросы, возвращающие список (набор) значений из одного столбца таблицы.
3. Подзапросы, возвращающие набор записей (таблицу).

Тип возвращаемого подзапросом значения определяет, как можно его использовать и какие операторы можно применять в содержащем выражении для взаимодействия с результатом вложенного запроса.

Независимый подзапрос может применяться со следующими операторами: операторами сравнения; оператором IN; операторами SOME (ANY) и ALL.

4.1.1. Подзапросы, возвращающие единственное значение

Пример 1. Допустим, из таблицы Goods требуется выбрать все товары, у которых цена выше средней.

```
Select *
From Product
Where Price > (SELECT AVG (Price) FROM Product)
```

Чтобы получить нужные товары, вначале надо выполнить подзапрос на получение средней цены товара: SELECT AVG (Price) FROM Product.

Пример 2. С помощью следующего запроса выберем из таблицы Product данные обо всех товарах определенной категории:

```
SELECT      *
FROM        Product
WHERE       IdCategory = (SELECT      idCategory
                          FROM        Category
                          WHERE       Name = 'Планшеты')
```

В данном запросе сначала выполняется подзапрос (SELECT idCategory FROM Category WHERE Name = 'Планшеты'). Этот подзапрос возвращает единственную запись, содержащую единственное поле – идентификатор категории *«Планшеты»*. Далее выполняется внешний запрос, который выводит все столбцы таблицы *«Goods»* и записи, в которых значение столбца *«idCategory»* равно значению, полученному с помощью подзапроса. Таким образом, сначала выполняется подзапрос, а затем внешний запрос, использующий результат подзапроса. Хотя в данном случае подзапрос прекрасно справится с поставленной задачей, стоит отметить, что это не самый эффективный способ для извлечения данных из других таблиц, так как в рамках T-SQL для извлечения данных из разных таблиц можно использовать оператор JOIN, который будет рассмотрен далее.

Задание. Составьте на языке SQL запросы на выборку следующих данных:
1) список заказов, в которых содержится определенный товар;
2) список покупателей из таблицы *«Customer»*, у которых есть оформленные, но не оплаченные заказы в таблице *«Order»*.

4.1.2. Подзапросы, возвращающие набор значений (из одного столбца таблицы). Операции группового сравнения ALL, SOME (ANY)

Подзапрос, вообще говоря, может возвращать несколько значений. Чтобы в этом случае в условии внешнего оператора WHERE можно было использовать операторы сравнения, требующие единственного значения, используются кванторы, такие как ALL (все) и SOME (ANY) (некоторый). Операторы ALL, SOME (ANY) используют в тех случаях, когда необходимо проверить условие на соответствие с каждым значением, которое вернул подзапрос. Они, как и оператор EXISTS, работают только с подзапросами.

При использовании ключевого слова ALL условие в операции сравнения должно быть верно для всех значений, которые возвращаются подзапросом.

Для примера, в каждой категории определим самый дорогой товар.

Пример 3

```
SELECT idProd, Name, IdCategory, Price
FROM Product p1
WHERE p1.Price > ALL (    SELECT p2.Price
                          FROM Product p2
                          -- учесть только товары данной категории
                          WHERE p2.IdCategory = p1.IdCategory
                          -- исключить сравнение со своей же ценой
                          AND p2.IdProd <> p1.IdProd          )
```

Если в данном случае опустить ключевое слово ALL, то мы получим ошибку. Допустим, если подзапрос возвращает значения val1, val2 и val3, то условие фильтрации фактически было бы аналогично объединению этих значений через оператор AND: WHERE Price > val1 AND Price > val2 AND Price > val3.

В тоже время подобный запрос гораздо проще переписать другим образом:

Пример 4

```
SELECT idProd, Name, IdCategory, Price
FROM Product p1
WHERE p1.Price > (   SELECT MAX (p2.Price)
                     FROM Product p2
                     WHERE p2.IdCategory = p1.IdCategory
                     AND p2.IdProd <> p1.IdProd          )
```

При применении ключевых слов SOME или ANY условие в операции сравнения должно быть истинным хотя бы для одного из значений, возвращаемых подзапросом. По действию оба этих оператора аналогичны, поэтому можно применять любое из них.

Задания:
1. Составьте запрос с использованием операторов SOME или ANY.
2. Попробуйте выразить составленный вами запрос (с использованием операторов SOME или ANY) при помощи других операторов.

Когда вложенный запрос возвращает набор значений, также можно использовать оператор IN. Выберем, например, всех покупателей из таблицы *«Customer»*, у которых нет заказов в таблице *«Order».*

Пример 5

```
SELECT * FROM Customer WHERE IdCust NOT IN (SELECT IdCust FROM [Order])
```

4.1.3. Подзапросы, возвращающие набор записей

Подзапрос можно вставлять не только в операторы WHERE и HAVING, но и в оператор FROM.

```
SELECT    t.столбец1, t.столбец2, ... , t.столбецn
FROM      (SELECT ... ) t
WHERE     ...
```

Здесь таблице, возвращаемой подзапросом в операторе FROM, присваивается псевдоним t, а внешний запрос выделяет столбцы этой таблицы и, возможно, записи в соответствии с некоторым условием, которое указано в операторе WHERE.

4.2. Связанные (коррелированные) запросы

Все приведенные до сих пор запросы не зависели от своих содержащих выражений, т. е. могли выполняться самостоятельно и представлять свои результаты для проверки. Связанный подзапрос (коррелированный), напротив, зависит от содержащего выражения, из которого он ссылается на один или более столбцов. В отличие от несвязанного подзапроса, который выполняется непосредственно перед выполнением содержащего выражения, связанный подзапрос выполняется по разу для каждой строки-кандидата (это строки, которые предположительно могут быть включены в окончательные результаты). Например, следующий запрос использует связанный подзапрос для подсчета количества заказов у каждого клиента. Затем основной запрос выбирает тех клиентов, у которых больше одного заказа.

Пример 6

```
SELECT *
FROM Customer c
WHERE 1 < (SELECT COUNT(*) FROM [Order] o WHERE o.IdCust = c.IdCust)
```

Ссылка на c.IdCust в самом конце подзапроса – это то, что делает этот подзапрос связанным. Чтобы подзапрос мог выполняться, основной запрос должен поставлять значения для c.IdCust. В данном случае основной запрос извлекает из таблицы *«Customer»* все строки и выполняет по одному подзапросу для всех клиентов, передавая в него соответствующий Id клиента при каждом выполнении. Если подзапрос возвращает значение больше одного, условие фильтрации выполняется и строка добавляется в результирующий набор.

Связанные подзапросы часто используются с условиями сравнения (в предыдущем примере <) и вхождения в диапазон, но самый распространенный оператор, применяемый в условиях со связанными подзапросами, – это оператор EXISTS (существует). Оператор EXISTS применяется, если требуется показать, что связь есть, а количество связей при этом не имеет значения. Например, следующий запрос возвращает список всех товаров, которые когда-либо заказывали.

Пример 7
```
SELECT IdProd, Description
FROM Product p
WHERE EXISTS (SELECT * FROM Ordered o WHERE o.IdProd = p.IdProd)
```

При использовании оператора EXISTS подзапрос может возвращать ни одной, одну или много строк, а условие просто проверяет, возвращены ли в результате выполнения подзапроса строки (все равно сколько). Если взглянуть на блок SELECT подзапроса, можно увидеть, что он состоит из единственного литерала *. Для условия основного запроса имеет значение только факт наличия возвращенных строк, а что именно было возвращено подзапросом – не важно. Поэтому подзапрос может возвращать все, что вам вздумается, но все же при использовании EXISTS принято задавать SELECT *.

Для поиска подзапросов, не возвращающих строки, можно использовать оператор EXISTS совместно с оператором отрицания NOT. В частности, чтобы предыдущий запрос возвращал все товары, которые ни разу не заказывались, его можно модифицировать следующим образом.

Пример 8
```
SELECT IdProd, Description
FROM Product p
WHERE NOT EXISTS (SELECT * FROM Ordered o WHERE o.IdProd = p.IdProd)
```

Подзапрос как спецификация столбца

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

Пример 9
```
SELECT *, (    SELECT       Category.Name
FROM           Category
WHERE          IdCategory=Goods.IdCategory)      AS Category
FROM Product
```

В данном случае для каждой строки из таблицы *«Orders»* будет выполняться подзапрос, результат которого зависит от столбца *«ProductId»*. И каждый подзапрос может возвращать различные данные.

Задание. Составьте запрос, используя подзапрос в качестве одного из столбцов выборки.

ЛАБОРАТОРНАЯ РАБОТА 5
МНОГОТАБЛИЧНЫЕ ЗАПРОСЫ
(ГОРИЗОНТАЛЬНОЕ СОЕДИНЕНИЕ ТАБЛИЦ)

Операции соединения

При проектировании базы данных 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. Однако сущности могут быть взаимосвязанными. Клиенты связаны с городами по признаку проживания, заказы осуществляют клиенты, товары входят в состав заказов и т. д. Связь между таблицами устанавливается за счет размещения столбца первичного ключа одной таблицы, которая называется родительской, в другой взаимосвязанной таблице, которая называется дочерней. Столбец (или совокупность столбцов) дочерней таблицы, определенный для связи с родительской таблицей, называется внешним ключом. Так, таблица *«Customer»* содержит столбец *«IdCity»*, который для каждой строки клиента содержит значение первичного ключа того города, в котором проживает данный клиент.

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**.

5.1. Связь таблиц при помощи WHERE-условия

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.

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.

Если таблицы соединяются по равенству значений пары столбцов (группы столбцов) из различных таблиц, такая операция называется соединением таблиц по равенству. Соединение по равенству позволяет соединить только те пары строк, которые действительно взаимосвязаны друг с другом. Например, мы можем соединить таблицы городов и клиентов по условию City.IdCity = Customer.IdCity. В таком варианте мы соединяем таблицы осмысленно, т. к. каждая строка таблицы *«Customer»* соединяется только с одной строкой соответствующего города. На базе таблиц *«City»* и *«Customer»* мы получаем таблицу со столбцами из обеих таблиц, имеющую строки с понятным смыслом. Можно также сказать, что в таблицу *«Customer»* вместо столбца *«IdCity»* мы вставляем все характеристики (столбцы) соответствующего города из таблицы *«City»*.

Соединение таблиц используется, когда необходимо вывести значения столбцов:

- из разных таблиц;
- из 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.

Рассмотрим примеры.

Пример 1

```
SELECT FName, LName, CityName
FROM Customer, City
WHERE Customer.IdCity = City.IdCity
```

Этот запрос возвращает список всех клиентов с указанием названий городов, в которых они проживают. Этот вид запросов характерен тем, что фраза WHERE содержит только условие соединения, а список фразы SELECT содержит имена столбцов из различных таблиц.

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. Запишем предыдущий запрос с полным уточнением имен.

Пример 2

```
SELECT Customer.FName, Customer.LName, City.CityName
FROM Customer, City
WHERE Customer.IdCity = City.IdCity
```

В этом запросе мы уточнили имена столбцов во фразах SELECT и WHERE, хотя в предложении SELECT это было не обязательно, т. к. используются неповторяющиеся имена. Тем не менее рекомендуется при соединении таблиц для наглядности уточнять имена столбцов. Однако на практике для задания более лаконичных имен часто используют короткие синонимы таблиц, по которым можно сослаться на них в любых других местах запроса. Синоним указывается сразу после имени таблицы в предложении FROM. Предыдущий запрос с использование синонимов для таблиц можно записать более компактным образом.

Пример 3

```
SELECT k.FName, k.LName, c.CityName
FROM Customer k, City c
WHERE k.IdCity = c.IdCity
```

Рассмотрим пример, когда фраза WHERE содержит не только условие соединения. Следующий запрос отбирает всех клиентов из Пинска с фамилией Иванов.

Пример 4

```
SELECT K.IdCust, k.FName
FROM Customer k, City c
WHERE k.IdCity = c.IdCity AND k.LName = 'Иванов' AND c.CityName = 'Пинск'
```

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

Соединение более двух таблиц

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.

Пример 5

Список всех клиентов, которые когда-либо заказывали товар с кодом 1.

```
SELECT DISTINCT c.IdCust, c.FName, c.LName
FROM Customer c, [Order] o, Ordered ord
WHERE c.IdCust = o.IdCust AND o.IdOrd = ord.IdOrd AND ord.IdProd = 1
```

Сформулируем общую процедуру составления многотабличного запроса:

– 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;

– 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;

– во фразе FROM перечислить необходимые таблицы;

– во фразе WHERE соединить таблицы запроса и при необходимости задать условия отбора строк в базовых таблицах запроса;

– во фразе SELECT перечислить выводимые столбцы.

Задания:

1. Составьте запрос, возвращающий список товаров в заказе. Результирующая таблица должна включать следующие поля: номер заказа, название товара, цена за ед., количество ед. в заказе.
2. Составьте запрос, возвращающий список заказов и их стоимость (поля: номер заказа, дата заказа, стоимость заказа).
3. Составьте запрос, определяющий заказ с максимальной стоимостью.

5.2. Соединения с использованием фразы FROM

Все рассмотренные выше типы и способы соединения таблиц можно (и рекомендуется, поскольку соединения во фразе WHERE считаются устаревшими) осуществлять и с помощью фразы FROM. В ней, в соответствии со стандартом SQL, можно не только перечислить имена таблиц, участвующих в запросе, но и указать их соединение, используя следующий синтаксис: таблица [INNER | {FULL | LEFT | RIGHT} [OUTER]] JOIN таблица {ON условие} (таблица 5.1).

Таблица 5.1 – Операции горизонтального соединения таблиц

Краткий синтаксис	Полный синтаксис	Описание
1	2	3
JOIN	INNER JOIN	Из строк левой и правой таблиц объединяются и возвращаются только те строки, по которым выполняются условия соединения
LEFT JOIN	LEFT OUTER JOIN	Возвращаются все строки левой таблицы (ключевое слово LEFT). Данными правой таблицы дополняются только те строки левой таблицы, для которых выполняются условия соединения. Для недостающих данных вместо строк правой таблицы вставляются NULL-значения
RIGHT JOIN	RIGHT OUTER JOIN	Возвращаются все строки правой таблицы (ключевое слово RIGHT). Данными левой таблицы дополняются только те строки правой таблицы, для которых выполняются условия соединения. Для недостающих данных вместо строк левой таблицы вставляются NULL-значения

Продолжение таблицы 5.1

1	2	3
FULL JOIN	FULL OUTER JOIN	Возвращаются все строки левой и правой таблиц. Если для строк левой и правой таблиц выполняются условия соединения, то они объединяются в одну строку. Для строк, для которых не выполняются условия соединения, NULL-значения вставляются на место левой таблицы либо на место правой таблицы в зависимости от того, данных какой таблицы в строке не имеется
CROSS JOIN	–	Объединение каждой строки левой таблицы со всеми строками правой таблицы. Этот вид соединения иногда называют декартовым произведением

Внутреннее соединение

В операторе JOIN внутреннее соединение указывается ключевым словом INNER (впрочем, его можно опустить, т. к. соединение двух таблиц является внутренним по умолчанию). Условие соединения указывается после ключевого слова ON. В этом случае внутреннее соединение с помощью фразы FROM JOIN очень похоже на соединение с использованием фразы WHERE. Запишем первый пример с предыдущего раздела с использование оператора JOIN.

Пример 6. Список всех клиентов с указанием названий городов, в которых они проживают.

```
SELECT FName, LName, CityName
FROM Customer k JOIN City c ON k.IdCity = c.IdCity
```

При соединении с использованием фразы FROM дополнительное условие можно для увеличения наглядности запроса помещать во фразу WHERE. В этом случае второй пример с предыдущего раздела примет такой вид.

Пример 7. Список всех клиентов из Минска с фамилией Иванов.

```
SELECT k.FName, k.LName, c.CityName
FROM Customer k INNER JOIN City c ON k.IdCity = c.IdCity
WHERE k.LName = 'Иванов' AND c.CityName = 'Минск'
```

Внешнее соединение

Все соединения таблиц, рассмотренные до сих пор, являются внутренними. Во всех примерах вместо ключевого слова JOIN можно писать INNER JOIN (внутреннее соединение). Из таблицы, получаемой при внутреннем соединении, отбраковываются все записи, для которых нет соответствующих записей одновременно в обеих соединяемых таблицах. При внешнем соединении такие несоответствующие записи сохраняются. В этом и заключается отличие внешнего соединения от внутреннего.

С помощью специальных ключевых слов LEFT OUTER, RIGHT OUTER и FULL OUTER, написанных перед JOIN, можно выполнить соответственно левое, правое и полное соединение. В SQL-выражении запроса таблица, указанная слева от оператора JOIN, называется левой, а указанная справа от него – правой.

При левом внешнем соединении несоответствующие записи, имеющиеся в левой таблице, сохраняются в результатной таблице, а имеющиеся в правой – удаляются. Значения столбцов из правой таблицы во всех строках, не имеющих соответствия с левой таблицей, принимают значение NULL.

При правом внешнем соединении несоответствующие записи, имеющиеся в правой таблице, сохраняются в результатной таблице, а имеющиеся в левой – удаляются. Значения столбцов из левой таблицы во всех строках, не имеющих соответствия с правой таб-

лицей, принимают значение NULL. Соответственно, левое и правое внешние соединения различаются только порядком следования таблиц.

При полном внешнем соединении двух таблиц результирующая таблица содержит все строки внутреннего соединения этих таблиц, а также не включенные им строки и первой, и второй таблиц, дополненные значениями NULL для отсутствующих столбцов.

В следующем примере возвращается полный список городов с указанием количества клиентов из каждого из них.

Пример 8

```
SELECT c.CityName, a.CountCity
FROM City c LEFT OUTER JOIN (SELECT IdCity, COUNT(*) AS CountCity FROM Customer GROUP BY IdCity) a ON c.IdCity = a.IdCity
ORDER BY c.CityName
```

Обратите внимание, что таблица *«City»* соединяется не с таблицей, а с подзапросом, которому задан псевдоним a.

Задания:

1. Если в предыдущем запросе заменить левое внешнее соединение на внутреннее, то из результата будут потеряны города, из которых нет ни одного клиента. Проверьте это, заменив LEFT OUTER JOIN на INNER JOIN, и объясните причину разницы.
2. Составьте на языке SQL запросы на выборку следующих данных (с использование оператора JOIN для соединения таблиц):
   1) список всех товаров, которые когда-либо заказывал заданный клиент;
   2) список всех клиентов, не имеющих ни одного заказа.

ЛАБОРАТОРНАЯ РАБОТА 6
ЗАПРОСЫ НА МОДИФИКАЦИЮ ДАННЫХ

Запросы, рассмотренные ранее, были направлены на то, чтобы получить данные, содержащиеся в существующих таблицах базы данных. Главным ключевым словом таких запросов на выборку данных является SELECT. Запросы на выборку данных всегда возвращают виртуальную таблицу, которая отсутствует в базе данных и создается временно лишь для того, чтобы представить выбранные данные пользователю. При создании и дальнейшем сопровождении базы данных обычно возникает задача добавления новых и удаления ненужных записей, а также изменения содержимого ячеек таблицы. В SQL для этого предусмотрены операторы INSERT (вставить), DELETE (удалить) и UPDATE (изменить). Запросы, начинающиеся с этих ключевых слов, не возвращают данные в виде виртуальной таблицы, а изменяют содержимое уже существующих таблиц базы данных. Запросы на модификацию (добавление, удаление и изменение) данных могут содержать вложенные запросы на выборку данных из той же самой таблицы или из других таблиц, однако сами не могут быть вложены в другие запросы. Таким образом, операторы INSERT, DELETE и UPDATE в SQL-выражении могут находиться только в самом начале.

6.1. Добавление новых записей

Для вставки записей в таблицу используется оператор INSERT, который имеет несколько форм: *INSERT INTO название таблицы VALUES (список значений)*. Вставляет запись в указанную таблицу и заполняет эту запись значениями из списка, указанного за ключевым словом VALUES. При этом первое в списке значение вводится в первый столбец таблицы, второе значение – во второй столбец и т. д. Порядок столбцов задается при создании таблицы.

Данная форма оператора INSERT не очень надежна, поскольку нетрудно ошибиться в порядке вводимых значений. Более надежной и гибкой является следующая форма: *INSERT INTO название таблицы (список столбцов) VALUES (список значений)*. Вставляет запись в указанную таблицу и вводит в заданные столбцы значения из указанного списка. При этом в первый столбец (из списка столбцов) вводится первое значение (из списка значений), во второй столбец – второе значение и т. д. Порядок имен столбцов в списке может отличаться от их порядка, заданного при создании таблицы. Столбцы, которые не указаны в списке, заполняются значением NULL. Рекомендуется использовать именно данную форму оператора INSERT.

Следующий запрос добавляет новую запись в справочник городов. Обратите внимание, что столбец *«IdCity»* не задается, поскольку он является счетчиком и заполняется СУБД автоматически.

Пример 1

```
INSERT INTO City(CityName)
VALUES('Минск')
```

Следующая форма оператора INSERT (*INSERT INTO название таблицы (список столбцов) SELECT ...*) вставляет в указанную таблицу записи, возвращаемые запросом на выборку. На практике нередко требуется загрузить в одну таблицу данные из другой таблицы. Например, следующий запрос вставляет в таблицу *«City»* сразу два города, возвращаемых запросом с объединением.

Пример 2
INSERT INTO City(CityName)
SELECT 'Минск' UNION SELECT 'Брест'

Задание. Используя разные формы оператора INSERT, добавьте информацию в следующие таблицы:
1) в таблицу *«Customer»* добавьте 20 клиентов;
2) в таблицу *«Category»* добавьте 5 товарных категорий, а в таблицу *«Product»* – по 5 товаров каждой категории.

6.2. Удаление записей

Для удаления записей из таблицы применяется оператор DELETE: *DELETE FROM название таблицы WHERE-условие.*

Данный оператор удаляет из указанной таблицы записи (а не отдельные значения столбцов), которые удовлетворяют указанному условию. Условие – это логическое выражение, различные конструкции которого были рассмотрены в предыдущих лабораторных занятиях.

Следующий запрос удаляет записи из таблицы *«Customer»*, в которой значение столбца *«LName»* равно 'Иванов'. Если в таблице содержатся сведения о нескольких клиентах с фамилией Иванов, то все они будут удалены.

Пример 3
DELETE FROM Customer
WHERE LName = 'Иванов'

В операторе WHERE может находиться подзапрос на выборку данных (оператор SELECT). Подзапросы в операторе DELETE работают точно так же, как и в операторе SELECT. Следующий запрос удаляет всех клиентов из города Пинск, при этом уникальный идентификатор города возвращается с помощью подзапроса.

Пример 4
DELETE FROM Customer
WHERE IdCity IN (SELECT IdCity FROM City WHERE CityName = 'Пинск’)

Transact-SQL расширяет стандартный SQL, позволяя использовать в инструкции DELETE еще одно предложение FROM. Это расширение, в котором задается соединение, может быть использовано вместо вложенного запроса в предложении WHERE для указания удаляемых строк. Оно позволяет задавать данные из второго FROM и удалять соответствующие строки из таблицы в первом предложении FROM. В частности, предыдущий запрос может быть переписан следующим образом:

Пример 5
DELETE FROM Customer
FROM Customer k INNER JOIN City c ON k.IdCity = c.IdCity AND c.CityName = 'Пинск'

Операция удаления записей из таблицы является опасной в том смысле, что связана с риском необратимых потерь данных в случае семантических (но не синтаксических) ошибок при формулировке SQL-выражения. Чтобы избежать неприятностей, перед удалением записей рекомендуется сначала выполнить соответствующий запрос на выборку, чтобы просмотреть, какие записи будут удалены. Так, например, перед выполнением рас-

смотренного ранее запроса на удаление не помешает выполнить соответствующий запрос на выборку.

Пример 6
```
SELECT *
FROM Customer k INNER JOIN City c ON k.IdCity = c.IdCity AND c.CityName = 'Минск'
```

Для удаления всех записей из таблицы достаточно использовать оператор DELETE без ключевого слова WHERE. При этом сама таблица со всеми определенными в ней столбцами сохраняется и готова для вставки новых записей. Например, следующий запрос удаляет записи обо всех товарах.

Пример 7
```
DELETE FROM Goods
```

Задание. Сформулируйте на языке SQL запрос на удаление всех заказов, не имеющих в составе ни одного товара (т. е. все пустые заказы).

6.3. Изменение данных

Для изменения значений столбцов таблицы применяется оператор UPDATE (изменить, обновить). Чтобы изменить значения в одном столбце таблицы в тех записях, которые удовлетворяют некоторому условию, следует выполнить такой запрос: *UPDATE название таблицы SET имя столбца = значение WHERE условие.*

За ключевым словом SET (установить) следует выражение равенства, в левой части которого указывается имя столбца, а в правой – выражение, значение которого следует сделать значением данного столбца. Эти установки будут выполнены в тех записях, которые удовлетворяют условию в операторе WHERE.

Чтобы одним оператором UPDATE установить новые значения сразу для нескольких столбцов, вслед за ключевым словом SET записываются соответствующие выражения равенства, разделенные запятыми: *UPDATE название таблицы SET имя столбца1 = значение1, имя столбца2 = значение2, ..., имя столбцаN = значениеN WHERE-условие.*

Например, следующий запрос изменяет фамилию клиента с кодом 5.

Пример 8
```
UPDATE Customer
SET LName='Петрова'
WHERE IdCust = 5
```

Использование оператора WHERE в инструкции UPDATE не обязательно. Если он отсутствует, то указанные в SET изменения будут произведены для всех записей таблицы.

Так же как и в инструкции DELETE, условие в операторе WHERE инструкции UPDATE может содержать подзапросы, в том числе и связанные.

Transact-SQL расширяет стандартный SQL, позволяя использовать в инструкции UPDATE предложение FROM (по аналогии с DELETE). Это расширение, в котором задается соединение, может быть использовано вместо вложенного запроса в предложении WHERE для указания обновляемых строк. Если обновляемый объект тот же самый, что и объект в предложении FROM, и в предложении FROM имеется только одна ссылка на этот объект, псевдоним объекта указывать не обязательно. Если обновляемый объект встречается в предложении FROM несколько раз, одна и только одна ссылка на этот объ-

ект не должна указывать псевдоним таблицы. Все остальные ссылки на объект в предложении FROM должны включать псевдоним объекта.

Предположим, что требуется сделать 5 % скидку по тем заказам клиентов, суммарная стоимость которых превышает 1000. Для этого следует изменить значения столбца *«Discount»* таблицы *«Order»*. Однако эти изменения должны быть выполнены, только если суммарная стоимость заказа превышает 1000. Таким образом, в качестве критерия обновления записей в таблице *«Order»* может быть задан запрос, возвращающий список всех заказов с суммарной стоимостью более 1000.

Операция изменения записей, как и их удаление, связана с риском необратимых потерь данных в случае семантических ошибок при формулировке SQL-выражения. Например, стоит только забыть написать оператор WHERE, и будут обновлены значения во всех записях таблицы. Чтобы избежать подобных неприятностей, перед обновлением записей рекомендуется выполнить соответствующий запрос на выборку, чтобы просмотреть, какие записи будут изменены.

Задания:

1. Установите с помощью запроса на обновление 5 % скидку по тем заказам клиентов, суммарная стоимость которых превышает 1000.
2. Составьте на языке SQL запрос, имитирующий поступление на склад новой партии определенного товара (обновите столбец *«InStock»* в таблице *«Goods»).*

ЛАБОРАТОРНАЯ РАБОТА 7
СОЗДАНИЕ ПРЕДСТАВЛЕНИЙ

Представления – это именованные запросы на выборку данных (инструкции SELECT на языке T-SQL), хранящиеся в базе данных. Подобно таблицам представления также состоят из полей и записей. Однако в отличие от таблиц они не содержат каких-либо данных. Представления всегда основываются на таблицах и используются для получения данных, хранящихся в этих таблицах, в определенных разрезах. Представления позволяют достичь более высокой защищенности данных, а также предоставляют проектировщику средства настройки пользовательской модели.

Механизм представлений может использоваться по нескольким причинам. Он предоставляет мощный и гибкий механизм защиты, позволяющий скрыть некоторые части базы данных от определенных пользователей. Пользователь не будет иметь сведений о существовании каких-либо атрибутов или кортежей, отсутствующих в доступных ему представлениях (горизонтальное и вертикальное разбиение таблиц). Также он позволяет организовать доступ пользователей к данным наиболее удобным для них образом, поэтому одни и те же данные в одно и то же время могут рассматриваться разными пользователями совершенно различными способами (в частности, переименование атрибутов). Он упрощает сложные операции с базовыми отношениями. Например, если представление будет определено на основе соединения двух отношений, то пользователь сможет выполнять над ним простые унарные операции выборки и проекции, которые будут автоматически преобразованы средствами СУБД в эквивалентные операции с выполнением соединения базовых отношений. Одной из важнейших причин использования представлений является стремление к упрощению многотабличных запросов. После определения представления с соединением нескольких таблиц можно будет использовать простейшие однотабличные запросы к этому представлению вместо сложных запросов с выполнением того же самого многотабличного соединения.

Все эти примеры демонстрируют определенную степень логической независимости от данных, достигаемую за счет использования представлений. Однако на самом деле представления позволяют добиться и более важного типа логической независимости от данных, связанной с защитой пользователей от реорганизаций концептуальной схемы. Например, если в отношение будет добавлен новый атрибут, то пользователи не будут даже подозревать о его существовании, пока определения их представлений не включают этот атрибут. Если существующее отношение реорганизовано или разбито на части, то использующее его представление может быть переопределено так, чтобы пользователи могли продолжать работать с данными в прежнем формате.

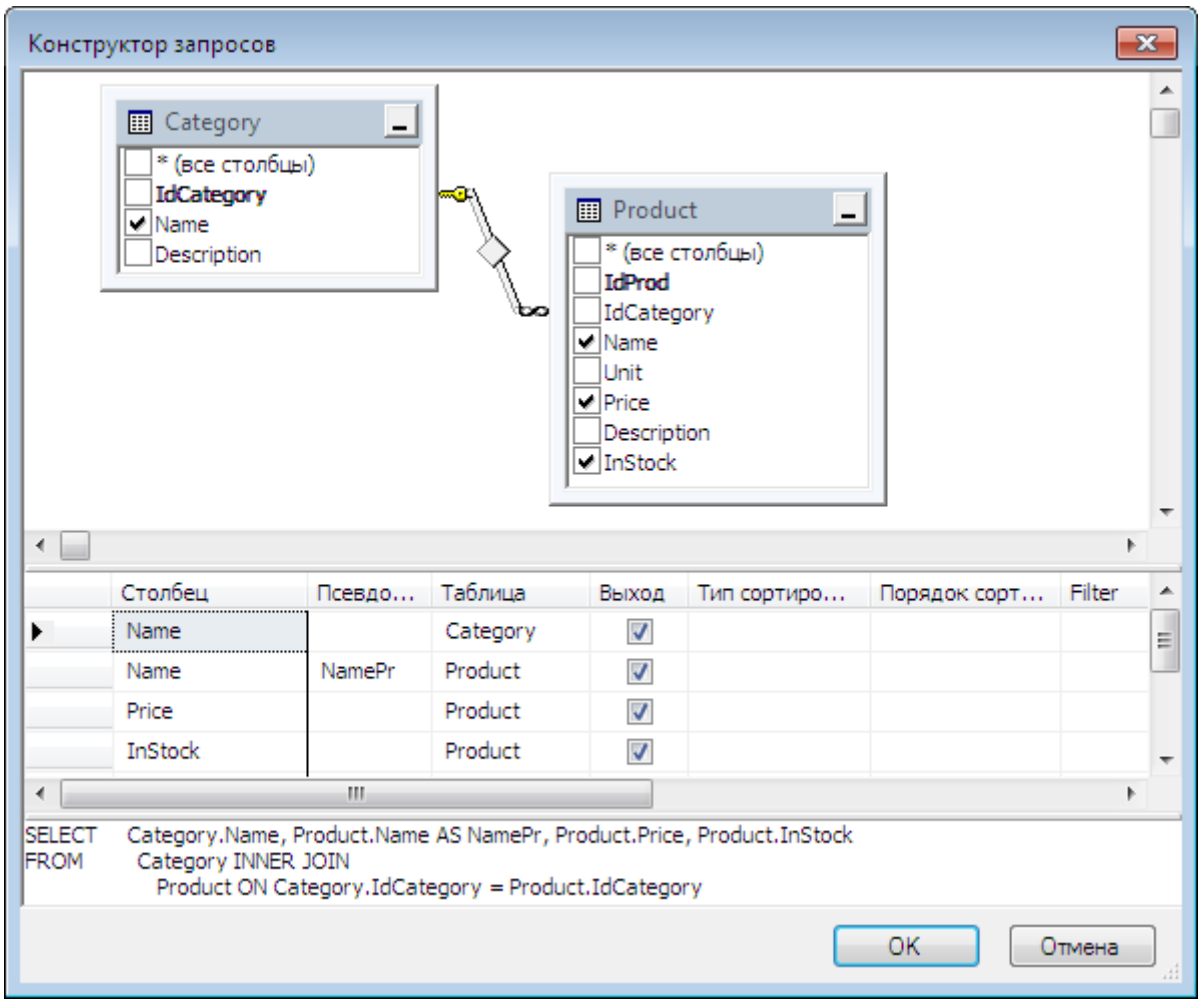
Представления можно создать в SQL Server с помощью SQL Server Management Studio или Transact-SQL.

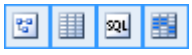
Создание представлений в Management Studio

В SQL Server Management Studio представления можно создавать, редактировать, выполнять и вставлять в другие запросы. Представление может быть создано только в текущей базе данных.

Поскольку представление является не чем иным, как сохраненной инструкцией SELECT, его создание начинается с проектирования этой инструкции.

Команда *«Создать представление»* в контекстном меню позволит запустить конструктор запросов в режиме создания представлений.



Окно конструктора состоит из четырех панелей:  панель диаграмм, область критериев, панель *«SQL»* и панель результатов.

Панель диаграммы (область схемы). В запросе может участвовать множество таблиц и представлений. Для связывания их отношениями с целью формирования предложения FROM инструкции SELECT можно использовать это графическое представление.

Панель сетки (область критериев). Панель *«Критерии»* содержит сетку, подобную электронной таблице, в которой указываются параметры (например, какие столбцы данных нужно отображать, какие строки выбирать, как группировать строки и т. д.).

Панель SQL. На этой панели можно в текстовом виде посмотреть и отредактировать формируемую инструкцию SELECT.

Панель результатов показывает сетку с данными, полученными запросом или представлением. В конструкторе запросов и представлений панель показывает результаты последнего выполненного запроса SELECT.

Кнопка *«Сохранить»*  панели инструментов запускает сценарий фактического создания представления в базе данных. Следует отметить, что для сохранения инструкция SELECT должна быть свободна от ошибок.

После создания представления его можно редактировать в Management Studio, выделив название и выбрав в контекстном меню команду *«Проект»*.

Для тестирования инструкции SELECT представления в конструкторе запросов нажмите кнопку *«Выполнить код SQL»*  или клавишу <*F5*>.

Контекстное меню представления также содержит команды управления его полнотекстовой индексацией и его переименования. Свойства приложения содержат расширенные параметры и разрешения системы безопасности. Для удаления представления из базы данных достаточно выделить его, выбрать в контекстном меню команду *«Удалить»* или нажать одноименную клавишу.

Создание представлений с помощью кода SQL

Представлениями можно управлять в редакторе запросов, выполняя сценарии SQL, которые используют команды языка DDL: CREATE, ALTER и DROP. Основной синтаксис создания представления следующий:

```
CREATE VIEW имя представления
AS
инструкция SELECT
```

Например, чтобы создать представление v_Product, возвращающее список товаров с указанием товарной категории, программным путем в окне запросов должна быть выполнена следующая команда:

Пример 1

```
CREATE VIEW v_Product
AS
SELECT   Category.Name, Product.Name AS NamePr, Product.Price, Product.InStock
FROM     Category INNER JOIN Product ON Category.IdCategory = Product.IdCategory
```

Попытка создать представление, которое уже существует, вызовет ошибку. Когда представление создано, инструкцию SELECT можно с легкостью отредактировать с помощью команды ALTER:

```
ALTER имя представления
AS
(новая) инструкция SELECT
```

Если изменение представления предполагает и изменение прав доступа на него, предпочтительнее удалить его и создать заново, поскольку удаление представления также приводит и к удалению разрешений доступа, установленных ранее.

Чтобы удалить представление из базы данных, используйте команду DROP:

```
DROP VIEW имя представления
```

Предложение ORDER BY и представления

Представления служат источником данных для других запросов и не поддерживают сортировку внутри себя. Например, следующий код извлекает данные из представления v_Product и упорядочивает их по полям Category.Name и Product.Name.

Предложение ORDER BY не является частью представления v_Product, а применяется к нему с помощью вызова инструкции SQL:

Пример 2

```
SELECT Name, NamePr, Price, InStock
FROM dbo.v_Product
ORDER BY Name, NamePr
```

Выполнение представлений

Представление не может быть выполнено само по себе. Инструкция SELECT, на основе которой создано представление, может быть выполнена, однако в этой форме, с технической стороны, инструкция SQL не является представлением. Представление может быть полезно только как источник данных в запросе.

Именно поэтому контекстное меню *«Открыть представление»* утилиты Management Studio автоматически генерирует простой запрос, извлекая из представления все столбцы. Представление отображает только результаты. Однако включение других панелей конструктора запросов позволяет увидеть и сам запрос, извлеченный из представления.

Панель SQL отобразит представление в предложении FROM инструкции SELECT. Именно в такой форме на представление ссылаются пользователи:

```
SELECT * FROM v_Product
```

Задание. Создайте представление, возвращающее список заказов, с указанием имени клиента, даты заказа и количества товаров в каждом заказе. Таким образом, результат должен включать следующие атрибуты: IdOrd, OrdDate, IdCust, FName, LName, количество товаров в заказе.

ЛАБОРАТОРНАЯ РАБОТА 8
ПРОГРАММИРОВАНИЕ НА T-SQL

Синтаксис и соглашения T-SQL

Правила формирования идентификаторов

Все объекты в SQL Server имеют имена (идентификаторы). Примерами объектов являются таблицы, представления, хранимые процедуры и т. д. Идентификаторы могут включать до 128 символов (в частности, буквы, символы _ @ $ # и цифры). Первый символ всегда должен быть буквенным. Для переменных и временных таблиц используются специальные схемы именования. Имя объекта не может содержать пробелов и совпадать с зарезервированным ключевым словом SQL Server, независимо от используемого регистра символов. Путем заключения идентификаторов в квадратные скобки в именах объектов можно использовать запрещенные символы.

Завершение инструкции

Стандарт ANSI SQL требует помещения в конце каждой инструкции точки с запятой. В то же время при программировании на языке T-SQL точка с запятой не обязательна.

Комментарии

Язык T-SQL допускает использование комментариев двух стилей: ANCI и языка C. Первый из них начинается с двух дефисов и заканчивается в конце строки:

```
-- это однострочный комментарий стиля ANSI.
```

Также комментарии стиля ANSI могут вставляться в конце строки инструкции:

```
1. SELECT CityName -- извлекаемые столбцы;
2. FROM City -- исходная таблица;
3. WHERE IdCity = 1 -- ограничение на строки.
```

Редактор SQL может применять и удалять комментарии во всех выделенных строках. Для этого нужно выбрать соответствующие команды в меню «*Правка*» или на панели инструментов  .

Комментарии стиля языка C начинаются с косой черты и звездочки (/*) и заканчиваются теми же символами в обратной последовательности. Этот тип комментариев лучше использовать для комментирования блоков строк, таких как заголовки или большие тестовые запросы.

```
/*
Пример
многострочного
комментария
*/
```

Одним из главных достоинств комментариев стиля C является то, что многострочные запросы в них можно выполнять, даже не раскомментируя.

Пакеты T-SQL

Пакет в T-SQL – это команды или инструкции SQL, которые объединены в одну группу, при этом SQL сервер будет компилировать и выполнять их как одно целое.

Для того чтобы дать понять SQL серверу, что вы передаете пакет команд, необходимо указывать ключевое слово *GO* после всех команд, которые вы хотите объединить в пакет.

Локальные переменные будут видны только в пределах того пакета, в котором они были созданы, т. е. обратиться к переменной после завершения пакета вы уже не сможете.

Переменные

Переменные T-SQL создаются с помощью команды DECLARE, имеющей следующий синтаксис:

```
DECLARE @имя_переменной тип_данных
```

Все имена локальных переменных должны начинаться символом @. Например, для объявления локальной переменной UStr, которая хранит до 16 символов Unicode, можно использовать следующую инструкцию:

```
DECLARE @UStr varchar(16)
```

Используемые для переменных типы данных в точности совпадают с существующими в таблицах. В одной команде DECLARE через запятую может быть перечислено несколько переменных. К примеру, в следующем примере создаются две целочисленные переменные a и b:

```
DECLARE @a int, @b int
```

Область определения переменных (т. е. срок их жизни) распространяется только на текущий пакет. По умолчанию только что созданные переменные содержат пустые значения NULL и до включения в выражения должны быть инициализированы.

Задание значений переменных

В настоящее время в языке SQL предусмотрены два способа задания значения переменной – для этой цели можно использовать оператор SELECT или SET. С точки зрения выполняемых функций эти операторы действуют почти одинаково, не считая того, что оператор SELECT позволяет получить исходное присваиваемое значение из таблицы, указанной в операторе SELECT.

Оператор SET обычно используется для задания значений переменных в такой форме, какая более часто встречается в процедурных языках. В качестве типичных примеров применения этого оператора можно указать следующие:

```
SET @a = 1;
SET @b = @a * 1.5
```

Обратите внимание на то, что во всех этих операторах непосредственно осуществляются операции присваивания, в которых используются либо явно заданные значения, либо другие переменные. С помощью оператора SET невозможно присвоить переменной значение, полученное с помощью запроса. Запрос должен быть выполнен отдельно, только после этого полученный результат может быть присвоен с помощью оператора SET. Например, попытка выполнения такого оператора вызывает ошибку:

```
DECLARE @c int
```

```
SET @c = COUNT(*) FROM City
SELECT @c
```

а следующий оператор выполняется вполне успешно:

```
DECLARE @c int
SET @c = (SELECT COUNT(*) FROM City)
SELECT @c
```

Оператор SELECT обычно используется для присваивания значений переменным, если источником информации, которая должна быть сохранена в переменной, является запрос. Например, действия, осуществляемые в приведенном выше коде, гораздо чаще реализуются с помощью оператора SELECT:

```
DECLARE @c int
SELECT @c = COUNT(*) FROM City
SELECT @c
```

Обратите внимание на то, что данный код немного понятнее (в частности, он более лаконичен, хотя и выполняет те же действия).

Таким образом, можно сформулировать следующее общепринятое соглашение по использованию того и другого оператора:

- оператор SET используется, если должна быть выполнена простая операция присваивания значения переменной, т. е. если присваиваемое значение уже задано явно в форме определенного значения или в виде какой-то другой переменной;
- оператор SELECT применяется, если присваивание значения переменной должно быть основано на запросе.

Использование переменных в запросах SQL

Одним из полезных свойств языка T-SQL является то, что переменные могут использоваться в запросах без необходимости создания сложных динамических строк, встраивающих переменные в программный код. Динамический SQL продолжает свое существование, но одиночное значение можно изменить проще – с помощью переменной.

Везде, где в запросе может использоваться выражение, может использоваться и переменная. В следующем примере продемонстрировано использование переменной в предложении WHERE:

```
DECLARE @IdProd int
SET @IdProd = 1
SELECT [Description]
FROM Goods
WHERE IdProd = @IdProd
```

Глобальные системные переменные

В SQL Server имеется более тридцати глобальных переменных, не имеющих параметров, которые определяются и поддерживаются системой. Все глобальные переменные имеют префикс в виде двух символов @. Вы можете извлечь значение любой из них с помощью простого запроса SELECT, как в следующем примере:

```
SELECT @@CONNECTIONS
```

Здесь используется глобальная переменная @@CONNECTIONS для извлечения количества подключений к SQL Server со времени запуска программы.

Среди наиболее часто применяемых системных переменных можно отметить следующие:

- @@ERROR – переменная, которая содержит номер ошибки, возникшей при выполнении последнего оператора T-SQL в текущем соединении. Если ошибка не обнаружена, содержит 0. Значение этой системной переменной переустанавливается после выполнения каждого очередного оператора. Если требуется сохранить содержащееся в ней значение, то это значение следует переносить в локальную переменную сразу же после выполнения оператора, для которого должен быть сохранен код ошибки.
- @@IDENTITY – переменная, которая содержит последнее идентификационное значение, вставленное в базу данных в результате выполнения последнего оператора INSERT. Если в последнем операторе INSERT не произошла выработка идентификационного значения, системная переменная @@IDENTITY содержит NULL. Это утверждение остается справедливым, даже если отсутствие идентификационного значения было вызвано аварийным завершением при выполнении оператора. А если с помощью одного оператора осуществляется несколько операций вставки, этой системной переменной присваивается только последнее идентификационное значение.
- @@ROWCOUNT – одна из наиболее широко используемых системных переменных. Возвращает информацию о количестве строк, затронутых последним оператором. Обычно применяется для контроля ошибок, отличных от тех, которые относятся к категории ошибок этапа прогона программы. Например, если в программе обнаруживается, что после вызова на выполнение оператора DELETE с конструкцией WHERE количество затронутых строк равно нулю, то можно сделать вывод, что произошло нечто непредвиденное. После этого сообщение об ошибке может быть активизировано вручную.

Следует отметить, что с версии SQL Server 2000 глобальные переменные принято называть функциями. Название глобальные сбивало пользователей с толку, позволяя думать, что область действия таких переменных шире, чем у локальных. Глобальным переменным часто ошибочно приписывалась возможность хранить информацию, независимо от того, включена она в пакет либо нет, что, естественно, не соответствовало действительности.

Средства управления потоком команд. Программные конструкции

В языке T-SQL предусмотрена большая часть классических процедурных средств управления ходом выполнения программы, в т. ч. условная конструкция и циклы.

Оператор IF. . .ELSE

Операторы IF. . .ELSE действуют в языке T-SQL в основном так же, как и в любых других языках программирования. Общий синтаксис этого оператора имеет следующий вид:

```
IF логическое выражение
SQL инструкция I BEGIN Блок SQL инструкций END
[ELSE
SQL инструкция | BEGIN Блок SQL инструкций END]
```

В качестве логического выражения может быть задано практически любое выражение, результат вычисления которого приводит к возврату булева значения.

Следует учитывать, что выполняемым по условию считается только тот оператор, который непосредственно следует за оператором IF (ближайшим к нему). Вместо одного

оператора можно предусмотреть выполнение по условию нескольких операторов, объединив их в блок кода с помощью конструкции BEGIN…END.

В приведенном ниже примере условие IF не выполняется, что предотвращает выполнение следующего за ним оператора.

```
IF 1 = 0
  PRINT 'Первая строка'
PRINT 'Вторая строка'
```

Необязательная команда ELSE позволяет задать инструкцию, которая будет выполнена в случае, если условие IF не будет выполнено. Подобно IF, оператор ELSE управляет только непосредственно следующей за ним командой или блоком кода, заключенным между BEGIN…END.

Несмотря на то, что оператор IF выглядит ограниченным, его предложение условия может включать в себя мощные функции, подобно предложению WHERE. В частности, это выражения IF EXISTS().

Выражение IF EXISTS() использует в качестве условия наличие какой-либо строки, возвращенной инструкцией SELECT. Так как ищутся любые строки, список столбцов в инструкции SELECT можно заменить звездочкой. Этот метод работает быстрее, чем проверка условия @@ROWCOUNT>0, потому что не требуется подсчет общего количества строк. Как только хотя бы одна строка удовлетворяет условию IF EXISTS(), запрос может продолжать выполнение.

В следующем примере выражение IF EXISTS используется для проверки наличия у клиента с кодом 1 каких-либо заказов перед удалением его из базы. Если по данному клиенту есть информация хотя бы по одному заказу, удаление не производится.

```
IF EXISTS(SELECT * FROM [Order] WHERE IdCust = 1)

PRINT 'Невозможно удалить клиента, поскольку в базе имеются связанные с ним записи'
ELSE
    BEGIN
        DELETE Customer
        WHERE IdCust = 1
        PRINT 'Удаление произведено успешно'
    END
```

Операторы WHILE, BREAK и CONTINUE

Оператор WHILE в языке SQL действует во многом так же, как и в других языках, с которыми обычно приходится работать программисту. По сути, в этом операторе до начала каждого прохода по циклу проверяется некоторое условие. Если перед очередным проходом по циклу проверка условия приводит к получению значения TRUE, осуществляется проход по циклу, в противном случае выполнение оператора завершается.

Оператор WHILE имеет следующий синтаксис:

```
WHILE Логическое выражение
SQL инструкцияI
[BEGIN
   [BREAK]
   Блок SQL инструкций
```

```
   [CONTINUE]
END]
```

Безусловно, с помощью оператора WHILE можно обеспечить выполнение в цикле только одного оператора (по аналогии с тем, как обычно используется оператор IF), но на практике конструкции WHILE, за которыми не следует блок BEGIN. . .END, соответствующий полному формату оператора, встречаются редко.

Оператор BREAK позволяет немедленно выйти из цикла, не ожидая того, как будет выполнен проход до конца цикла и произойдет повторная проверка условного выражения.

Оператор CONTINUE позволяет прервать отдельную итерацию цикла. Кратко можно описать действие оператора CONTINUE так, что он обеспечивает переход в начало цикла WHILE. Сразу после обнаружения оператора CONTINUE в цикле, независимо от того, где он находится, происходит переход в начало цикла и повторное вычисление условного выражения (а если значение этого выражения больше не равно TRUE, осуществляется выход из цикла).

Следующий короткий сценарий демонстрирует использование оператора WHILE для создания цикла:

```
DECLARE @Temp int;
SET @Temp = 0;
WHILE @Temp < 3
  BEGIN
        PRINT @Temp;
        SET @Temp = @Temp + 1;
  END
```

Здесь в цикле целочисленная переменная @Temp увеличивается с 0 до 3 и на каждой итерации ее значение выводится на экран.

Оператор RETURN

Оператор RETURN используется для остановки выполнения пакета, а следовательно, хранимой процедуры и триггера (рассматриваются в следующих лабораторных занятиях).

ЛАБОРАТОРНАЯ РАБОТА 9
СОЗДАНИЕ ХРАНИМЫХ ПРОЦЕДУР

Хранимая процедура – это наиболее часто используемая в базах данных программная структура, представляющая собой оформленный особым образом сценарий (вернее, пакет), который хранится в базе данных, а не в отдельном файле. Хранимые процедуры отличаются от сценариев тем, что в них допускается использование входных и выходных параметров, а также возвращаемых значений, которые фактически не могут использоваться в обычном сценарии.

Хранимая процедура представляет собой просто имя, связанное с программным кодом T-SQL, который хранится и исполняется на сервере. Она может содержать практически любые конструкции или команды, исполнение которых поддерживается в SQL Server. Процедуры можно использовать для изменения данных, возврата скалярных значений или целых результирующих наборов. Хранимые процедуры являются основным интерфейсом, который должен использоваться приложениями для обращения к любым данным в базах данных. Хранимые процедуры позволяют не только управлять доступом к базе данных, но также изолировать код базы данных для упрощения обслуживания.

Как серверные программы хранимые процедуры имеют ряд преимуществ:

- хранимые процедуры хранятся в компилированном виде, поэтому выполняются быстрее, чем пакеты или запросы;
- выполнение обработки данных на сервере, а не на рабочей станции, значительно снижает нагрузку на локальную сеть;
- хранимые процедуры имеют модульный вид, поэтому их легко внедрять и изменять. Если клиентское приложение вызывает хранимую процедуру для выполнения некоторой операции, то модификация процедуры в одном месте влияет на ее выполнение у всех пользователей;
- хранимые процедуры можно рассматривать как важный компонент системы безопасности базы данных. Если все клиенты осуществляют доступ к данным с помощью хранимых процедур, то прямой доступ к таблицам может быть запрещен, все действия пользователей будут находиться под контролем. Что еще важнее, хранимые процедуры скрывают от пользователя структуру базы данных и разрешают ему выполнение только тех операций, которые запрограммированы в хранимой процедуре.

Управление хранимыми процедурами

Хранимые процедуры управляются посредством инструкций языка определения данных (DDL) CREATE, ALTER и DROP.

Общий синтаксис T-SQL кода для создания хранимой процедуры имеет следующий вид:

```
CREATE PROC | PROCEDURE <procedure_name>
   [ <@parameter> <data_type> [ = <default> ] [ OUT | OUTPUT ] ] [ ,...n ]
AS
[ BEGIN ] <sql_statements> [ END ]

<procedure_option> ::=
   [ ENCRYPTION ]
   [ RECOMPILE ]
   [ EXECUTE_AS_Clause ]
```

Структура этого оператора соответствует основному синтаксису CREATE <Object Type> <Object Name>, лежащему в основе любого оператора CREATE. Единственная отличительная особенность состоит в том, что в нем допускается использовать ключевое слово PROCEDURE или PROC. Оба эти варианта являются допустимыми: PROC является лишь сокращением от PROCEDURE.

Каждая процедура должна иметь уникальное в рамках базы данных имя (procedure_name), соответствующее правилам для идентификаторов объектов.

Процедуры могут иметь любое число входных параметров (@parametr) заданного типа данных (data_type), которые используются внутри процедуры как локальные переменные. При выполнении процедуры для каждого из объявленных формальных параметров должны быть переданы фактические значения. Или же для входного параметра может быть определено значение по умолчанию (default), которое должно быть константой или равняться NULL. В этом случае процедуру можно выполнить без указания значения соответствующего аргумента. Применение входных параметров не обязательно.

Можно также указать выходные параметры (помеченные как OUTPUT), позволяющие хранимой процедуре вернуть одно или несколько скалярных значений в подпрограмму, из которой она была вызвана. При создании процедур можно задать три параметра. При создании процедуры с параметром ENCRYPTION SQL Server шифрует определение процедуры. При задании параметра RECOMPILE SQL Server перекомпилирует хранимую процедуру при каждом ее запуске. Параметр EXECUTE AS определяет контекст безопасности для процедуры.

В конце определения хранимой процедуры вслед за ключевым словом AS должно быть приведено непосредственно тело процедуры (sql_statements) в виде кода из одной или нескольких инструкций языка T-SQL.

Инструкция DROP удаляет хранимую процедуру из базы данных. Инструкция ALTER изменяет содержимое всей хранимой процедуры. Для внесения изменений предпочтительнее использовать инструкцию ALTER, а не комбинацию инструкций удаления и создания, т. к. последний метод удаляет все разрешения.

Пример хранимой процедуры без параметров

Самая простая хранимая процедура возвращает результаты, не требуя никаких параметров. В этом плане она похожа на обычный запрос. В следующем примере создается простая хранимая процедура, которая извлекает информацию обо всех заказах с 01.01.2020.

```
CREATE PROCEDURE spr_getOrders
AS
SELECT IdOrd, IdCust, OrdDate
FROM [Order]
WHERE (OrdDate >= '01.01.2020')
RETURN
```

Чтобы протестировать новую процедуру, откройте новый запрос SQL Server и выполните следующий код:

```
EXEC spr_getOrders
```

Команда EXECUTE или сокращенно EXEC выполняет указанную хранимую процедуру. В данном случае хранимая процедура вернет все строки из таблицы *«Order»*, в которых значение поля *«OrdDate»* больше 1 января 2020 года, в соответствии с содержащимся в нем запросом на выборку.

Применение входных параметров

Хранимая процедура предоставляет определенные процедурные возможности (а если она применяется в инфраструктуре .NET, такие возможности становятся весьма значительными), а также обеспечивает повышение производительности, но в большинстве обстоятельств хранимая процедура не позволяет добиться многого, если не предусмотрена возможность передать ей некоторые данные, указывающие на то, какие действия должны быть выполнены с ее помощью. В частности, основная проблема, связанная с предыдущей хранимой процедурой (spr_getOrders), состоит в ее статичности. Если пользователям потребуется информация о заказах за другой период времени, то эта процедура им не поможет. Поэтому необходимо предусмотреть возможность передачи в нее соответствующих входных параметров, которые позволили бы динамически изменять период выборки.

Параметры, передаваемые хранимой процедуре, перечисляются через запятую в инструкции CREATE (ALTER) PROCEDURE непосредственно после ее имени. При объявлении входного параметра необходимо указать имя параметра, тип данных и, возможно, значение по умолчанию. В общем случае объявление входного параметра имеет следующий вид:

@parameter_name [AS] datatype [= default|NULL]

Правила определения входных параметров во многом аналогичны объявлению локальных переменных. Каждый из параметров должен начинаться с символа @. Для хранимой процедуры он является локальной переменной. Как и все локальные переменные, параметры должны объявляться с допустимыми встроенными или определяемыми пользователями типами данных СУБД SQL Server.

Значительные различия между объявлениями параметров хранимых процедур и объявлениями переменных начинают впервые обнаруживаться, когда дело касается значений, заданных по умолчанию. Прежде всего, при инициализации переменным всегда присваиваются NULL-значения, а на параметры это правило не распространяется. В действительности, если в объявлении параметра не предусмотрено заданное по умолчанию значение, то подразумевается, что этот параметр должен быть обязательным и что при вызове хранимой процедуры должно быть указано его начальное значение. Чтобы задать предусмотренное по умолчанию значение, необходимо добавить знак равенства (=) после обозначения типа данных, а затем указать применяемое по умолчанию значение. Благодаря этому пользователи получают возможность при вызове хранимой процедуры принимать решение о том, следует ли задать другое значение параметра или воспользоваться значением, предусмотренным по умолчанию.

В следующем примере хранимая процедура spr_getOrders дополняется двумя входными параметрами, позволяющими явно указать период выборки.

```
ALTER PROCEDURE [dbo].[spr_getOrders]
  @dateBegin datetime,
  @dateEnd datetime
AS
SELECT IdOrd, IdCust, OrdDate
FROM [Order]
WHERE (OrdDate BETWEEN @dateBegin AND @dateEnd)
RETURN
```

При вызове хранимой процедуры фактические значения параметров могут быть заданы либо с учетом позиции, либо по имени, а в самой вызываемой хранимой процедуре

способ, применяемый для передачи параметров, не играет особой роли, поскольку для всех параметров, независимо от способа их передачи в процедуру, используется одинаковый формат объявления. Если хранимой процедуре передается множество параметров с учетом их позиции в объявлении, то они должны сохранять порядок, указанный в определении. Можно также передавать параметры в любом порядке, но при этом указывать их имена. Если эти два метода смешиваются, то после первого явного указания имени параметра все остальные должны использовать тот же метод.

В следующих трех примерах продемонстрированы вызовы хранимых процедур и передача им параметров с использованием исходного порядка и имен:

```
EXEC spr_getOrders '01.01.2020', '01.07.2020'
EXEC spr_getOrders
  @dateBegin = '01.01.2020',
  @dateEnd = '01.07.2020'
EXEC spr_getOrders '01.01.2020', @dateEnd = '01.07.2020'
```

В объявлении хранимой процедуры для двух указанных параметров не были предусмотрены значения, применяемые по умолчанию, поэтому оба параметра рассматриваются как обязательные. Это означает, что для успешного вызова хранимой процедуры необходимо предоставить оба параметра. В этом можно легко убедиться, осуществив попытку снова вызвать хранимую процедуру, указав только один параметр или вообще не указывая параметры.

Применение выходных параметров

Выходные параметры позволяют хранимой процедуре возвращать данные вызывающей программе. Для определения выходных параметров используется ключевое слово OUT[PUT], которое обязательно как при определении процедуры, так и при ее вызове. В самой хранимой процедуре выходные параметры являются локальными переменными. В вызывающей процедуре или пакете выходные переменные должны быть предварительно определены, чтобы получить результирующие значения. Когда выполнение хранимой процедуры завершается, текущее значение параметра передастся локальной переменной вызывающей программы.

В следующем примере выходной параметр используется для возвращения уникального идентификатора вновь добавленного товара.

```
CREATE PROCEDURE spr_addProduct
  @Description nvarchar(100),
  @InStock int = 0,
  @IdProd int OUT
AS

INSERT Product([Description], InStock)
VALUES (@Description, @InStock)
SET @IdProd = @@IDENTITY

RETURN
Пример вызова:
DECLARE       @IdProd int

EXEC spr_addProduct
  @Description = N'Новый товар',
```

```
  @IdProd = @IdProd OUTPUT

SELECT        @IdProd as N'@IdProd'
```

Обратите внимание на то, что при вызове процедуры переданы значения не для всех параметров. Параметр *«InStock»* является необязательным, поскольку для него указано значение по умолчанию в виде нуля, которое и будет использовано, если для него не будет явно предоставлено другое значение. При этом если бы вызов хранимой процедуры и передача значений происходили с использованием позиционных параметров, то пришлось бы заполнять каждую позицию в списке параметров по меньшей мере до того, как встретился бы последний параметр, для которого должно быть предусмотрено значение.

Подтверждение успешного или неудачного завершения работы с помощью возвращаемых значений. Использование команды RETURN

Любая вызываемая на выполнение хранимая процедура возвращает значение независимо от того, предусмотрен ли в ней возврат значения или нет. По умолчанию после успешного завершения процедуры СУБД SQL Server автоматически возвращает значение, равное нулю. Чтобы передать некоторое возвращаемое значение из хранимой процедуры обратно в вызывающий код, достаточно применить оператор RETURN:

```
RETURN [<Целое число>]
```

Обратите внимание на то, что возвращаемое значение должно быть обязательно целочисленным.

Возвращаемые значения предназначены исключительно для указания на успешное или неудачное завершение хранимой процедуры и позволяют даже обозначить степень или характер успеха или неудачи. Использование возвращаемого значения для возврата фактических данных, таких как идентификационное значение или данные о количестве строк, затронутых хранимой процедурой, рассматривается как недопустимая практика программирования. Возвращаемое значение 0 указывает на успешное выполнение процедуры и установлено по умолчанию. Компания Microsoft зарезервировала значения от –99 до –1 для служебного пользования. Разработчикам для возвращения состояния ошибки пользователю рекомендуется использовать значения –100 и меньше.

Одной из наиболее важных особенностей оператора RETURN является то, что его выполнение приводит к безусловному завершению работы и выходу из хранимой процедуры. Это означает, что, независимо от местонахождения оператора RETURN в коде хранимой процедуры, после его выполнения больше не будет выполнена ни одна строка кода. Под безусловным завершением работы подразумевается, что действие, предусмотренное оператором RETURN, осуществляется независимо от того, в каком месте кода он обнаруживается. С другой стороны, допускается наличие в коде хранимой процедуры нескольких операторов RETURN, а выполнение этих операторов происходит, только если к этому приводит обычная структура управления процессом выполнения кода.

В предыдущем примере при попытке добавления в таблицу *«Product»* информации о новом товаре с дублирующим названием могла произойти ошибка, поскольку по наименованию товара организовано ограничение уникальности. Расширим хранимую процедуру spr_addProduct, предусмотрев предварительную проверку на наличие указанного товара в базе и индикацию об успешности операции через выходной параметр.

```
ALTER PROCEDURE [dbo].[spr_addProduct]
  @Description nvarchar(100),
```

```
  @InStock int = 0,
  @IdProd int OUT
AS
IF EXISTS(SELECT * FROM Product WHERE [Description] = @Description)
  RETURN -100
INSERT Product([Description], InStock)
VALUES (@Description, @InStock)
SET @IdProd = @@IDENTITY
RETURN 0
```

При вызове хранимой процедуры, если ожидается выходное значение, команда EXEC должна использовать целочисленную переменную:

```
EXEC @локальная_переменная = имя_хранимой_процедуры;
DECLARE       @return_value int,
              @IdProd int

EXEC    @return_value = spr_addProduct
              @Description = N'Новый товар',
              @IdProd = @IdProd OUTPUT

IF @return_value = 0
  BEGIN
    PRINT 'Товар успешно добавлен'
    SELECT @IdProd as N'@IdProd'
  END
ELSE
  BEGIN
    PRINT 'При добавлении товара произошла ошибка'
    SELECT 'Return Value' = @return_value
  END
```

Задание. Создайте хранимые процедуры, реализующие следующие действия:
1. Возврат списка всех заказов, содержащих заданный товар (по *IdProd*).
2. Определение количества клиентов, не имеющих ни одного заказа. Результат должен возвращаться через выходной параметр.
3. Удаление из базы данных информации об определенном клиенте (по *IdCust*). Если с данными клиентов имеются связанные записи (заказы), удаление должно быть отменено. Возвращаемое значение должно определять успешность выполнения операции.

ЛАБОРАТОРНАЯ РАБОТА 10
ФУНКЦИИ

Системные функции

SQL Server содержит богатый набор встроенных системных функций, которые формально подразделяются на следующие группы: статистические, функции настройки, функции работы с курсором, функции даты и времени, математические, функции работы с наборами строк, функции безопасности, строковые, системные статистические, функции обработки текста и изображений и прочие. Полный список системных функций, сгруппированных в отдельные папки по вышеуказанным категориям, можно увидеть в Management Studio в узле *«Программирование – Функции – Системные функции»* дерева обозревателя объектов. Рассмотрим некоторые из наиболее часто используемых скалярных (возвращающих одно значение) встроенных системных функций.

Информационные функции

В архитектуре «клиент/сервер» полезно знать, кем является конкретный клиент. В этом смысле очень полезными окажутся следующие четыре функции, особенно при сборе информации для аудита:

1. User_name(). Возвращает имя текущего клиента, каким он представился базе данных. Когда пользователю открыт доступ к базе данных, его имя может отличаться от регистрационного имени входа на сервер.
2. Suser_sname(). Возвращает регистрационное имя пользователя, под которым он вошел на SQL Server. Даже если тот был аутентифицирован как член одной из групп пользователей Windows, функция все равно возвращает имя его учетной записи Windows.
3. Host_name(). Возвращает имя рабочей станции пользователя.
4. App_name(). Возвращает имя приложения, подключенного к SQL Server.

Пример 1

```
SELECT
USER_NAME() AS 'Имя пользователя БД',
SUSER_SNAME() AS 'Имя входа',
HOST_NAME() AS 'Имя рабочей станции',
APP_NAME() AS 'Имя приложения'
```

Строковые функции

SQL Server поддерживает больше двух десятков функций для манипулирования строками. Рассмотрим несколько самых полезных из них.

1. Substring(строка, начальная_позиция, длина). Возвращает фрагмент строки. Первым параметром является сама строка, вторым – номер символа, с которого вырезается фрагмент, третьим – длина вырезаемого фрагмента. Например, результатом инструкции SELECT SUBSTRING('abcdefg', 3, 2) будет подстрока ‘cd’.
2. Stuff(строка, позиция_вставки, число_удаляемых_символов, вставляемая_строка). Противоположная по характеру функции substring() функция stuff() вставляет одну строку в другую, при этом в позиции вставки может быть удалено заданное количество символов исходной строки. Например, результатом инструкции SELECT STUFF('abcdefg', 3, 2, '123') будет строка 'ab123efg'.

3. Replace(строка, строка). Заменяет заданные фрагменты строки другой строкой. Например, функция REPLACE('abacad', 'a', 'e') возвращает строку ‘ebeced’.

4. Charindex(символ_поиска, строка, начальная_позиция). Возвращает позицию заданного символа в строке. Например, инструкция SELECT CHARINDEX('c', 'abcdefg', 1) вернет результат 3.

5. Patindex(% шаблон%, строка). Выполняет поиск по шаблону, который может содержать в строке символы макроподстановки. В следующем примере ищется первое вхождение в строку символа c или d: SELECT PATINDEX('%[cd]%', 'abdedefg'). Результатом данного запроса будет число 3.

6. Right(строка, число) и Left(строка, число). Возвращает крайнюю правую или левую часть строки. Например, результатом запроса SELECT LEFT('abcdefg',2) будет ‘ab’.

7. Len(строка). Возвращает длину строки.

8. Rtrim(строка) и Ltrim(строка). Эти функции удаляют соответственно пробелы в начале и в конце строки.

9. Upper(строка) и Lower(строка). Преобразует символы строки в верхний или нижний регистр.

Функции работы с датой и временем

Для манипулирования значениями даты и времени SQL Server предлагает девять функций. Некоторые из этих функций используют аргумент datepart, определяющий фрагмент, к которому применяется операция. В следующей таблице перечислены все возможные значения аргумента datepart.

Константа	**Значение**
yy или yyyy	Год
qq или q	Квартал
mm или m	Месяц
wk или ww	Неделя
dw или w	День недели
dy или y	День года
dd или d	День
hh	Час
mi или n	Минута
ss или s	Секунда
ms	Миллисекунда

Например, функция DATEADD принимает в качестве аргументов маркер datepart, величину приращения и исходную дату. Она возвращает результат добавления указанной величины в единицах, указанных в аргументе datepart, к текущей дате. Таким образом, чтобы добавить к текущей дате три дня, вы можете использовать следующий код:

Пример 2

```
PRINT DATEADD(d, 3, GETDATE())
```

Ниже приведен полный список доступных функций даты и времени.

1. Dateadd(datepart, величина, дата_начала). Добавляет к дате указанную величину.

2. Datediff(datepart, величина, дата_начала). Выводит количество единиц времени, заданных в аргументе datepart, между двумя датами.

3. Datename(datepart, дата). Возвращает текстовые имена (например, имя месяца или дня недели), соответствующие заданной дате.

4. Datepart(datepart, дата). Извлекает определенный фрагмент из заданной даты.

5. Day(дата). Извлекает день из даты.
6. Getdate. Возвращает текущее время и дату.
7. Getutcdate. Возвращает текущую дату и время, преобразованное в формат универсального синхронизированного времени (UTC).
8. Month(дата). Извлекает месяц из даты.
9. Year(дата). Извлекает год из даты.

В следующем примере запрос возвращает список всех заказов, сделанных в сентябре месяце, с указанием дня недели:

Пример 3
```
SELECT *, DATENAME(dw, OrdDate) AS 'День недели'
FROM [Order]
WHERE MONTH(OrdDate) = 9
```

Функции преобразования данных

Для явных преобразований одного типа данных в другой в SQL Server используют функции cast() и convert().

1. Cast(исходные_данные AS тип_данных). Стандарт ANSI SQL рекомендует явное преобразование одного типа данных в другой. Даже если такое преобразование может быть выполнено неявно сервером, использование функции cast() гарантирует получение нужного типа. Функция cast() программируется несколько отлично от других. Вместо разделения двух своих аргументов запятой используется ключевое слово AS, за которым следует требуемый тип данных.

Пример 4
```
SELECT [Description], 'Остаток на складе: ' + CAST(InStock AS varchar(10))
FROM Goods
```

2. Convert(тип_данных, выражение [стиль]). Эта функция возвращает значение, преобразованное в другой тип данных с произвольным форматированием. Эта функция не предусмотрена стандартом ANSI SQL. Первым ее аргументом является желаемый тип данных, применяемый к выражению. Аргумент стиль предполагает применение к результату некоторого стиля. Стиль обычно применяется при преобразовании из типа даты-времени в символьный и наоборот. Как правило, одно- или двухцифровой стиль предполагает двухцифровой год, а трехцифровой – четырехцифровой год. К примеру, стиль 1 подразумевает следующий формат данных: 01/01/03; в тоже время стиль 3 – 01/01/2003.

Пример 5
```
SELECT CONVERT(nvarchar(25), GETDATE(), 1)
SELECT CONVERT(nvarchar(25), GETDATE(), 100)
```

Функции для обработки пустых значений

Часто пустое значение нужно преобразовать в некоторое допустимое, чтобы данные можно было понять или чтобы выражение имело результат. Пустые значения требуют специальной обработки при использовании в выражениях, и язык SQL содержит ряд функций, специально предназначенных для работы с пустыми значениями. Функции isnull () и coalesce() преобразуют пустые значения в пригодные для использования, а функция nullif() создает пустое значение, если выполняется определенное условие.

Наиболее часто используемой функцией, предназначенной для работы с пустыми значениями, является isnull(). Эта функция в качестве аргумента принимает одно выражение или столбец, а также подстановочное значение. Если первый аргумент является допустимым значением (т. е. не пустым), эта функция возвращает его. Однако если первый аргумент представляет собой пустое значение, то возвращается значение второго аргумента. Общий синтаксис функции следующий:

Isnull(исходное_выражение, замещающее_значение).

Следующий пример подставляет строку 'не указан' вместо пустого значения там, где для клиента не определен телефон.

Пример 6
```
SELECT FName, LName, ISNULL(Phone, 'не указан') AS Phone
FROM Customer
```

Функция coalesce() принимает список выражений или столбцов и возвращает первое значение, которое окажется непустым. Ее общий синтаксис следующий:

Coalesce(выражение, выражение, ...)

В следующем примере продемонстрирована функция coalesce(), возвращающая первое непустое значение (в данном случае это 3): SELECT COALESCE(NULL, 1+NULL, 1+2, 'abc')).

Иногда пустое значение нужно создать на месте заменяющего его суррогатного. Если база данных заполнена значениями n/a или пустыми строками там, где должны находиться пустые значения, вы можете воспользоваться функцией nullif() и расчистить базу данных.

Функция nullif() принимает два аргумента. Если они равны, то возвращается пустое значение, в противном случае возвращается первый параметр.

Следующий фрагмент кода преобразует все пробелы в столбце «FName» в пустые значения.

```
SELECT NULLIF(LTRIM(RTRIM(FName)),'') AS FName
FROM Customer
```

Пользовательские функции

Кроме широкого выбора встроенных функций SQL Server предоставляет также возможность создавать свои собственные функции, содержащие часто используемый код.

Пользовательские функции обладают следующими преимуществами:

- с их помощью можно внедрить в запросы сложную логику;
- создавая новые функции, можно проектировать сложные выражения;
- эти функции обладают всеми достоинствами представлений, поскольку могут использоваться в предложении FROM инструкции SELECT и в выражениях, а также могут быть задействованы в схеме. К тому же пользовательские функции могут принимать параметры, в то время как представления – нет;
- они обладают достоинствами хранимых процедур, т. к. могут быть скомпилированы и оптимизированы таким же способом;
- главным аргументом против использования пользовательских функций является вопрос переносимости. Пользовательские функции привязаны к SQL Server, и любую базу данных, использующую множество таких функций, будет сложно или даже невозможно

перенести на другую платформу СУБД без существенной переработки. Эта задача усложняется тем, что также должны быть переписаны и все инструкции SELECT, в которые внедрены пользовательские функции. Если в будущем планируется развертывание базы данных на других платформах, то лучше заменить все пользовательские функции представлениями или хранимыми процедурами;

- пользовательские функции во многом аналогичны хранимым процедурам, но отличаются от них перечисленными ниже особенностями:
  - пользовательские функции могут возвращать значения, относящиеся к большинству типов данных SQL Server. Не допускается использовать в качестве типов возвращаемых значений лишь такие типы, как text, ntext, image, cursor и timestamp;
  - пользовательские функции не должны иметь побочных эффектов. По существу, в пользовательских функциях не допускается выполнение каких-либо действий, выходящих за пределы действия самой функции. Например, в них нельзя модифицировать таблицы, отправлять электронную почту и вносить изменения в значения параметров системы или базы данных;
  - пользовательские функции во многом аналогичны функциям, используемым в классических языках программирования. Эти функции принимают несколько параметров и возвращают одно значение. Различия между пользовательскими функциями SQL Server и функциями многих процедурных языков программирования состоят в том, что в них передача параметров осуществляется по значению, поэтому для них не предусмотрен способ передачи параметров, подобный применению ссылки или передачи указателей. Тем не менее пользовательские функции удобны тем, что позволяют возвращать данные в виде специальной таблицы.

Пользовательские функции подразделяются на три типа:

- скалярные, возвращающие одно значение;
- внедренные табличные, аналогичные представлениям;
- сложные табличные, создающие в программном коде результирующий набор данных.

Скалярные функции

Скалярными называют те функции, которые возвращают одно значение. Эти функции могут принимать множество параметров, выполнять вычисления, но в результате выдают одно значение. Эти функции могут использоваться в любых выражениях, даже участвующих в ограничениях проверки. Значение возвращается функцией с помощью оператора return – эта команда должна завершать скалярную функцию.

В скалярных пользовательских функциях не допускаются операции обновления базы данных, но в то же время они могут работать с локальными временными таблицами. Они не могут возвращать данные BLOB (двоичные большие объекты) таких типов, как text, image и ntext, равно как табличные переменные и курсоры.

Скалярные функции создаются, изменяются и удаляются с помощью тех же инструкций DDL, что и другие объекты, хотя синтаксис немного отличается, чтобы определить возвращаемое значение:

```
CREATE FUNCTION имя_функции (входные_параметры)
RETURNS тип_данных
AS
BEGIN
текст_ функции
```

```
RETURN выражение
END
```

В списке входных параметров должны быть указаны типы данных и значения по умолчанию (в случае необходимости) аналогично хранимым процедурам (параметр = умолчание). Параметры функции отличаются от параметров хранимых процедур тем, что даже если определены значения по умолчанию, параметры все равно должны присутствовать в вызове функции (т. е. параметры с определенными по умолчанию значениями все равно обязательны). Чтобы запросить значение по умолчанию при вызове функции, ей передается ключевое слово default.

Следующая скалярная функция выполняет простую арифметическую операцию; ее второй параметр имеет значение по умолчанию:

```
CREATE FUNCTION dbo.Multiply (@A int, @B int = 3)
RETURNS INT
BEGIN
  RETURN @A * @B
END
```

Скалярные функции могут использоваться в любом месте выражений, где допустимо одно значение. Пользовательские скалярные функции должны всегда вызываться с помощью двухкомпонентного имени (владелец.имя). В следующем примере продемонстрирован вызов ранее созданной функции Multiply:

```
SELECT dbo.Multiply(3,4)
SELECT dbo.Multiply(7, DEFAULT)
```

Следующий код создает функцию, возвращающую имя заданного клиента в формате Фамилия И.

```
CREATE FUNCTION getFICust (@IdCust int)
RETURNS varchar(25)
AS
BEGIN
  DECLARE @result varchar(25)
  SET @result = 'NULL'

  SELECT @result = LName + ' ' + SUBSTRING(FName, 1, 1) + '.'
  FROM Customer
  WHERE IdCust = @IdCust

  RETURN @result
END
```

Тестирование созданной функции:

```
SELECT dbo.getFICust(IdCust) AS CustName
FROM Customer
ORDER BY LName, FName
```

Задание. Создайте скалярную пользовательскую функцию, возвращающую общую стоимость заказа.

Внедренные табличные функции

Второй тип пользовательских функций очень похож на представления. Оба обрамляют сохраняемую инструкцию SELECT. Внедренные табличные функции имеют все достоинства представлений, добавляя к ним использование параметров и возможность компиляции. Как и в представлениях, если инструкция SELECT обновляема, то обновляемой является и функция.

Внедренная табличная функция не имеет в своем теле блока BEGIN ... END – вместо этого возвращается результирующий набор данных инструкции SELECT в виде таблицы с заданным именем.

```
CREATE FUNCTION имя_функции (параметры)
RETURNS Table AS
RETURN (инструкция_SELECT)
```

Следующая внедренная табличная функция является функциональным эквивалентом представления v_Customer, созданного в лаб. занятии № 6.

```
CREATE FUNCTION fCustomers ()
RETURNS TABLE
AS
RETURN
(
  SELECT Customer.IdCust, Customer.FName, Customer.LName, City.CityName
  FROM Customer INNER JOIN
       City ON Customer.IdCity = City.IdCity
)
```

Для извлечения данных с помощью функции fCustomers вызовите ее в предложении FROM инструкции SELECT:

```
SELECT *
FROM dbo.fCustomers()
ORDER BY LName, FName
```

Одним из преимуществ внедренных табличных функций по сравнению с представлениями является возможность первых включать параметры в предварительно скомпилированные инструкции SELECT. Представления не могут иметь параметров и обычно ограничивают результат с помощью добавления предложения WHERE в инструкцию SELECT, вызывающую представление. В качестве примера рассмотрим функцию, возвращающую список клиентов из заданного города.

```
CREATE FUNCTION [dbo].[fCustomersForCity] (@IdCity int = NULL)
RETURNS TABLE
AS
RETURN
(
  SELECT IdCust, FName, LName
  FROM Customer
  WHERE IdCity = @IdCity OR @IdCity IS NULL
)
```

Если функция вызывается с параметром по умолчанию, то возвращается список всех клиентов:

```
SELECT *
FROM dbo.fCustomersForCity(DEFAULT)
```

Если же в качестве параметра передается уникальный идентификатор города, то скомпилированная инструкция SELECT в функции вернет только клиентов из города с заданным кодом:

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

Табличные функции с множеством инструкций

Пользовательские табличные функции с множеством инструкций комбинируют способность скалярных функций содержать сложный программный код со способностью внедренных табличных функций возвращать результирующий набор данных. Этот тип функций создает табличную переменную, а затем заполняет ее в теле функции. Сформированная таблица впоследствии возвращается функцией и может использоваться в инструкциях SELECT.

Главным преимуществом таких функций является то, что результирующий набор данных может формироваться в пакете инструкций, а затем напрямую использоваться в инструкциях SELECT. Это позволяет использовать этот тип функций вместо хранимых процедур.

Синтаксис, используемый для создания табличных функций с множеством инструкций, практически такой же, как и для создания скалярных функций:

```
CREATE FUNCTION имя_функции (входные_параметры)
RETURNS @имя_таблицы TABLE (столбцы)
AS
BEGIN
  Программный код заполнения табличной переменной
  RETURN
END
```

Следующая процедура позволяет создать табличную пользовательскую функцию с множеством инструкций, которая возвращает основной результирующий набор данных:

- в начале инструкции CREATE FUNCTION создается табличная переменная;
- в теле функции с помощью инструкций INSERT заполняют переменную;
- после выполнения функции значение табличной переменной передается во внешнюю процедуру как результат функции.

Запишем предыдущую функцию в виде многооператорной функции:

```
CREATE FUNCTION [dbo].[fCustomersByCity2]
(
  @IdCity int = NULL
)
RETURNS
@Result TABLE
(
```

```
  IdCust int,
  FName nvarchar(20),
  LName nvarchar(20)
)
AS
BEGIN
  IF (@IdCity IS NULL)
    INSERT @Result
    SELECT IdCust, FName, LName
    FROM Customer
  ELSE
    INSERT @Result
    SELECT IdCust, FName, LName
    FROM Customer
    WHERE IdCity = @IdCity

  RETURN
END
```

Задание. Создайте пользовательские функции, возвращающие:
- список всех товаров, которые не были ни разу заказаны;
- список всех заказов за заданный период времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Куликов, С. С. Реляционные базы данных в примерах : практическое пособие для программистов и тестировщиков / С. С. Куликов. – Минск : Четыре четверти, 2020. – 424 с.

2. Игнатьева, О. В. Прикладное программирование и базы данных : учебно-методическое пособие для практических работ / О. В. Игнатьева ; ФГБОУ ВО РГУПС. – Ростов н/Д, 2017. – 206 с.

3. Волк, В. К. Базы данных. Проектирование и программирование : учебное пособие : в 2 ч. / В. К. Волк. – Курган : Изд-во Курганского гос. ун-та, 2018. – Ч. 1. – 178 с.

4. Тригуб, Н. А. Базы данных : лаб. практикум / Н. А. Тригуб. – М. : Изд. Дом НИТУ «МИСиС», 2019. – 68 с.

5. Куликов, С. С. Работа с MySQL, MS SQL Server и Oracle в примерах : учеб.-метод. пособие : в 2 ч. / С. С. Куликов, Е. Е. Фадеева. – Минск : БГУИР, 2019. – Ч. 1. – 287 с.

Учебное издание

Кисель Татьяна Васильевна

Базы данных

Методические указания по выполнению лабораторных работ

Ответственный за выпуск *Ю. В. Чечун*

Редактор *Т. И. Сакович*
Корректор *Ю. В. Цвикевич*

Подписано в печать 12.11.2021 г. Формат 60×84/8.
Бумага офсетная. Гарнитура «Таймс». Ризография.
Усл. печ. л. 8,83. Уч.-изд. л. 3,63.
Тираж 57 экз. Заказ № 276.

Отпечатано в редакционно-издательском отделе
Полесского государственного университета
225710, г. Пинск, ул. Днепровской флотилии, 23.