**Лабораторная работа №1: Создание баз данных**

**Утилита SQL Server Management Studio**

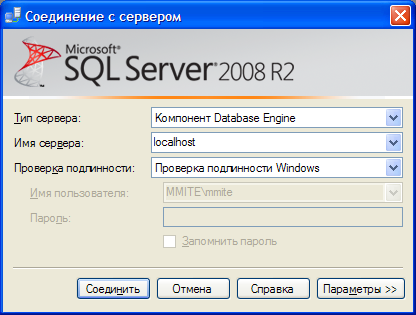
Подавляющую массу задач администрирования SQL Server можно выполнить в графической утилите SQL Server Management Studio. В ней можно создавать базы данных и все ассоциированные с ними объекты (таблицы, представления, хранимые процедуры и др.). Здесь вы можете выполнить последовательности инструкций Transact-SQL (запросы). В этой утилите можно выполнить 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» операционной системы выберите пункт «Microsoft SQL Server 2008 R2\Среда SQL Server Management Studio». Когда откроется окно программы, вас попросят подключиться к какому либо серверу баз данных SQL Server.

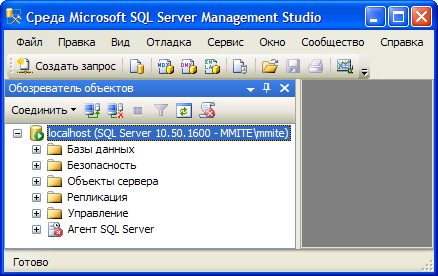
**Подключение к серверу**

В окне «Соединение с сервером» необходимо указать следующую информацию:

* *Тип сервера*. Здесь следует выбрать, к какой именно службе необходимо подключится. 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.



После подключения экземпляр сервера будет отображаться на панели «Обозреватель объектов».



Окно Management Studio имеет следующую структуру:

* Оконное 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.

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:

* *master*. Главная служебная база данных всего сервера. В ней 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.
* *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).
* *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.
* *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.

**Создание пользовательских баз данных**

База данных представляет собой группу файлов, хранящихся на жестком диске. Эти файлы могут относиться к трем типам: файлы с первичными данными, файлы с вторичными данными и файлы журнала транзакций. Любая база данных SQL Server содержит, по крайней мере, 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

**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.

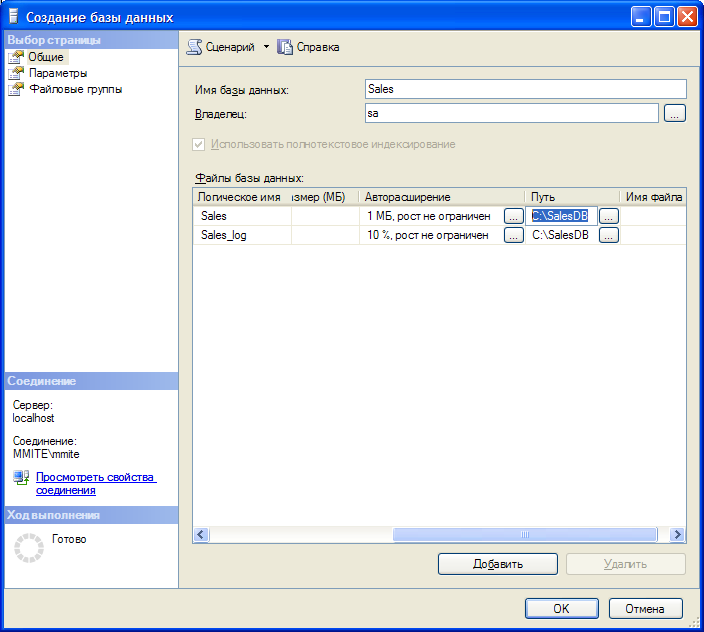
1. В окне «Обозреватель объектов» найдите и раскройте папку «Базы данных». Щелкните на ней правой кнопкой мыши и выберите команду «Создать базу данных…».
2. В открывшемся диалоговом окне «Создание базы данных» на странице «Общие» введите следующую информацию:

Имя базы данных: Sales

Владелец: sa

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

Для всех остальных параметров оставьте значения по умолчанию.



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

**Создание баз данных с помощью Transact-SQL**

Для программного создания базы данных (например, в программе установки приложения) используется инструкция CREATE DATABASE языка T-SQL (сокращенная форма от Transact-SQL). Данная инструкция может включать в себя множество опций, определяющих различные параметры новой базы данных.

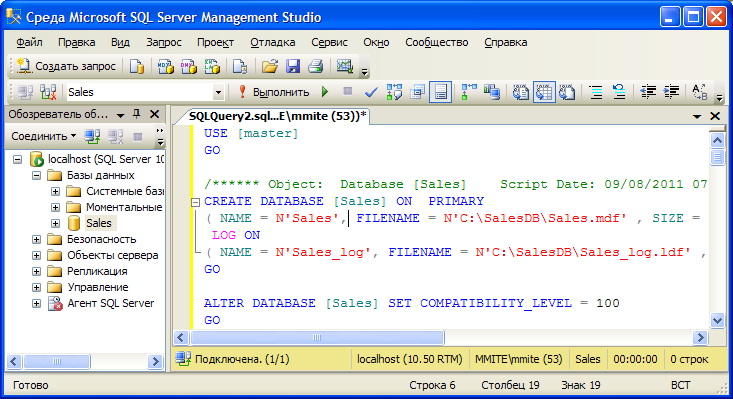
Сценарий создания новой базы данных может быть сгенерирован на основе уже существующей базы данных. Для этого в SQL Server Management Studio в контекстном меню узла «Sales» выберите команду «Создать сценарий для базы данных – Используя CREATE – Буфер обмена». В результате в буфер обмена будет сохранен 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. Это опция указывает на файловую группу, которая 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

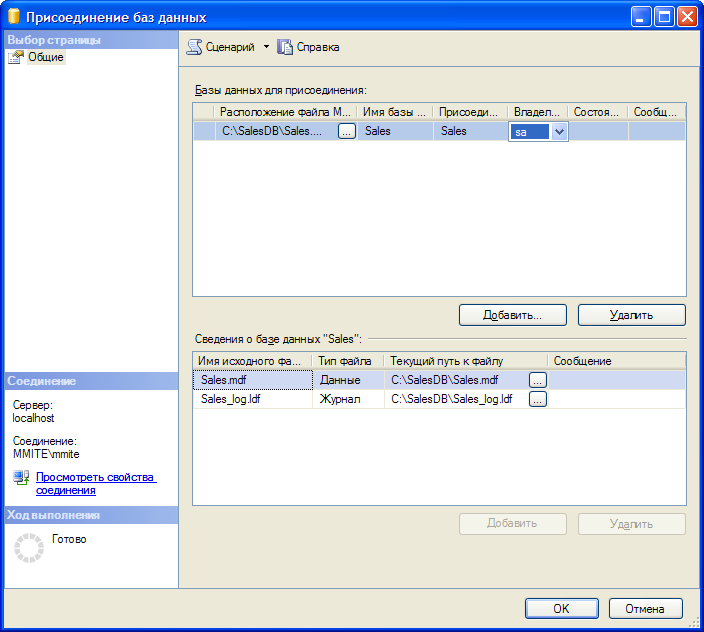
Параметры в разделе LOG ON аналогичны параметрам в разделе CREATE DATABASE. Однако они определяют параметры файла журнала транзакций.

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.

**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.



**Лабораторная работа №2: Создание таблиц и ограничений**

*Таблицы* представляют собой объекты базы данных, используемые непосредственно для хранения всех данных. Одним из самых главных 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 (или столбцы таблицы) содержат определенный тип информации, например, фамилию, адрес, телефонный номер. Запись (или строка таблицы) - 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.

**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:

1. Целочисленные данные

* bit (1 байт). Может хранить только значения 0, 1 или null (пустое значение, сообщающее об отсутствии данных). Его удобно использовать в качестве индикатора состояния – включено/выключено, да/нет, истина/ложь.
* tinyint (1 байт).Целые значения от 0 до 255.
* smallint (2 байта). Диапазон значений от -215 (-32768) до 215 (3767).
* int (4 байта). Может содержать целочисленные данные от -231 (-2147483648) до 231 (21474833647).
* bigint (8 байт). Включает в себя данные от -263 (9223372036854775808) до 263 (9223372036854775807). Удобен для хранения очень больших чисел, не помещающихся в типе данных int.

1. Текстовые данные

* char. Содержит символьные не Unicode-данные фиксированной длины до 8000 знаков.
* varchar. Содержит символьные не Unicode-данные переменной длины до 8000 знаков.
* nchar. Содержит данные Unicode фиксированной длины до 4000 символов. Подобно всем типам данных Unicode его удобно использовать для хранения небольших фрагментов текста, которые будут считываться разноязычными клиентами.
* nvarchar. Содержит данные Unicode переменной длины до 4000 символов.

1. Десятичные данные

* decimal. Содержит числа с фиксированной точностью от -1038-1 до 1038-1. Он использует два параметра: точность и степень. Точностью называется общее количество знаков, хранящееся в поле, а степень – это количество знаков справа от десятичной запятой.
* numeric. Это синоним типа данных decimal – они идентичны.

1. Денежные типы данных

* money (8 байт). Содержит денежные значения от -263 до 263 с десятичной точностью от денежной единицы. Удобен для хранения денежных сумм, превышающих 214768,3647.
* smallmoney (4 байта). Содержит значения от -214748,3648 до 214748,3647 с десятичной точностью.

1. Данные с плавающей точкой

* float. Содержит числа с плавающей запятой от -1,79Е+38 до 1,79Е+38.
* real. Содержит числа с плавающей запятой от -3,40Е+38 до 3,40Е+38.

1. Типы данных даты и времени

* datetime (8 байт). Содержит дату и время в диапазоне от 1 января 1753 года до 31 декабря 9999 года с точностью 3,33 мс.
* smalldatetime (4 байта). Содержит дату и время, начиная от 1 января 1900 года и заканчивая 6 июнем 2079, с точностью до 1 минуты.

1. Двоичные типы данных

* binary. Содержит двоичные данные фиксированной длины до 8000 байт.
* varbinary. Содержит двоичные данные переменной длины до 8000 байт.

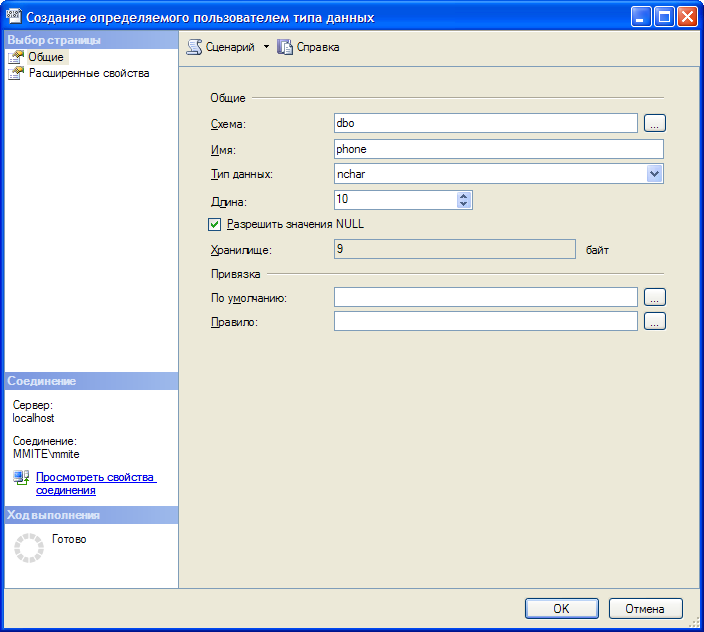
1. Специализированные типы данных

* sql\_variant. Используется для хранения значения с различными типами данных.
* timestamp. Используется для установки временных меток записей при вставке, которые соответствующим образом обновляются. Удобен для отслеживания изменений в данных.
* uniqueidentifier. Глобальный уникальный идентификатор.
* xml. Используется для хранения целых документов или фрагментов XML.

**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.

1. 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».
2. 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.
3. В секции «Привязки» оставьте пустые значения и щелкните на кнопке Ok. Созданный пользовательский тип данных должен появиться в дереве обозревателя объектов.



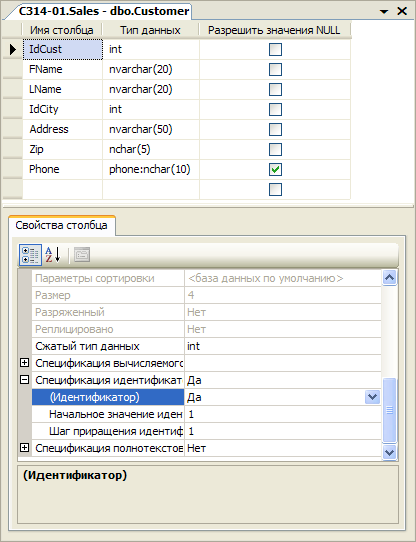
**Создание таблиц**

Создадим в базе данных Sales пять таблицы. Первая таблица, Customer, будет хранить информацию о клиентах, вторая таблица City – справочник городов, третья, Product, - информацию о товарах, четвертая, Order, будет содержать подробную информацию о заказах и пятая, OrdItem, - о составе заказа (перечне товаров входящих в заказ). Ниже представлены все поля этих таблиц и их основные свойства.

|  |  |  |  |
| --- | --- | --- | --- |
| Имя столбца | Тип данных | Разрешить null | Описание |
| Customer | | | |
| IdCust | int, identity | нет | Уникальный идентификационный номер клиента, на который можно ссылаться в других таблицах |
| FName | nvarchar(20) | нет | Имя клиента |
| LName | nvarchar(20) | нет | Фамилия клиента |
| IdCity | int | нет | Ссылка на номер города |
| Address | nvarchar(50) | нет | Адрес клиента |
| Zip | nchar(5) | нет | Почтовый индекс клиента |
| Phone | phone | да | Телефонный номер клиента |
| City | | | |
| IdCity | int, identity | нет | Уникальный идентификационный номер города |
| CityName | nvarchar(20) | нет | Название города |
| Product | | | |
| IdProd | int, identity | нет | Уникальный идентификационный номер для каждого товара |
| Description | nvarchar(100) | нет | Короткое текстовое описание товара |
| InStock | int | нет | Количество единиц продукта на складе |
| Order | | | |
| IdOrd | int, identity | нет | Уникальный идентификационный номер заказа |
| IdCust | int | нет | Ссылка на номер клиента |
| OrdDate | smalldatetime | нет | Дата и время размещения заказа |
| OrdItem | | | |
| IdOrd | int | нет | Ссылка на номер заказа |
| IdProd | int | нет | Ссылка на номер товара |
| Qty | int | нет | Количество единиц товара в заказе |
| Price | money | нет | Цена товара |

Таблицы можно создавать как в графическом интерфейсе (в утилите Management Studio), так и с помощью кода T-SQL. Воспользуемся самым простым, графическим способом. Сначала создадим таблицу Customer:

1. 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.
2. В первую строку в столбце «Имя столбца» введите IdCust, в столбце «Тип данных» выберите int. Убедитесь что параметр «Разрешить значения null» отключен.
3. 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.
4. Аналогичным образом введите описания всех остальных полей и закройте окно конструктора таблиц. Введите в качестве имени таблицы Customer. Вновь созданная таблица должна появиться в дереве обозревателя объектов в папке «Таблицы».



**Задание для самостоятельной работы:** В соответствие с вышеприведенным описанием создайте оставшиеся четыре таблицы: City, Product, Order и OrdItem.

**Создание ограничений**

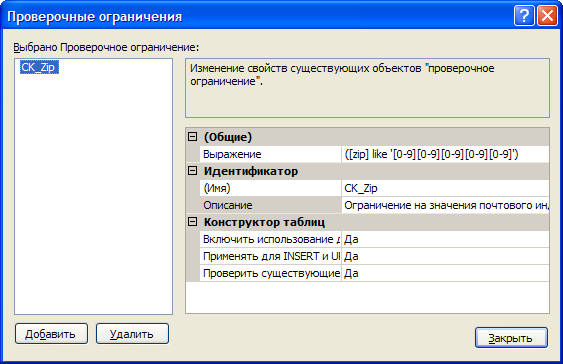
Перед тем как начать работать с таблицами следует ограничить вводимые в них данные в целях обеспечения так называемой целостности данных, т. е. ограничить возникновение в базе данных некорректных или противоречивых данных вследствие добавления, изменения или удаления какой-либо записи, например, ввод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

**Обеспечение доменной целостности.** 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 Zip (почтовый индекс) имеет тип nchar(5), т.е. чисто 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\_Zip
   * Выражение: ([zip] 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. Щелкните на кнопке «Закрыть» и закройте конструктор таблиц (он был открыт, когда вы начали создавать ограничение) с сохранением изменений.

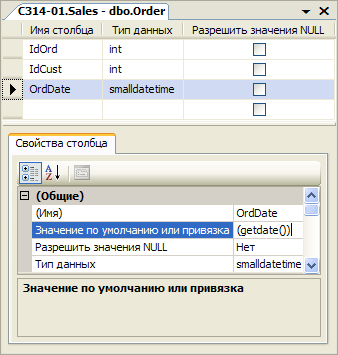


**Задание для самостоятельной работы:** Создайте ограничения для полей InStock таблицы Product и Qty, Price таблицы OrdItem, запрещающие ввод в них отрицательных значений. В данном случае выражение проверки будет иметь вид (Имя поля > 0) для полей Qty, Price и (Instock>=0) для столбца InStock.

**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(). Эта функция T-SQL возвращает текущую системную дату.
3. Щелкните на кнопке Сохранить и выйдите из конструктора таблиц.



**Задание для самостоятельной работы:** Установите для поля InStock (количество единиц продукта на складе) таблицы Product в качестве значения по умолчанию ноль.

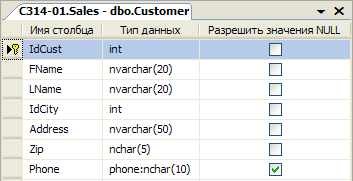
**Обеспечение сущностной целостности**. Обеспечение гарантии уникальности записей в таблицах и предотвращение их дублирования. Основными инструментами обеспечения целостности сущностей являются первичные ключи и ограничения уникальности.

**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 поля 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 вместе взятые (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 и 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



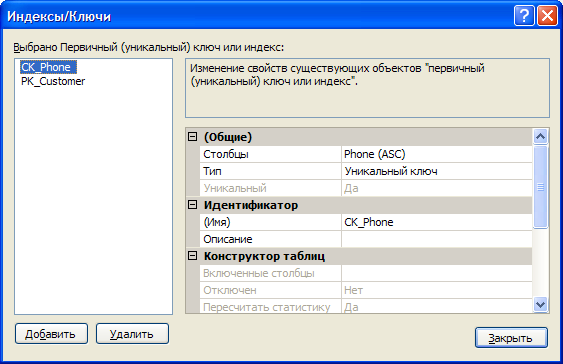
**Задание для самостоятельной работы:** Аналогичным образом создайте первичные ключи для остальных таблиц в соответствие с ниже приведенной таблицей.

|  |  |
| --- | --- |
| Таблица | Первичный ключ |
| City | IdCity |
| Product | IdProd |
| Order | IdOrd |
| OrdItem | IdOrd, IdProd (для выбора нескольких столбцов при установке составного ключа воспользуйтесь клавишами Shift или Ctrl) |

**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 (рассматривается в следующем разделе). 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:
   * Столбцы: Phone
   * Тип: Уникальный ключ
   * (Имя): CK\_Phone



1. Закройте конструктор таблиц с сохранением изменений.

**Задание для самостоятельной работы:** 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.

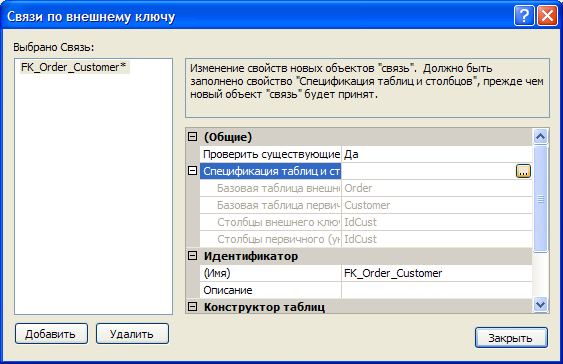
**Обеспечение целостности ссылок**

Сейчас в базе данных Sales имеются пять таблиц, которые тесно взаимосвязаны между собой и соответственно 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 отсутствия в базе данных таких записей необходимо обеспечить целостность ссылок.

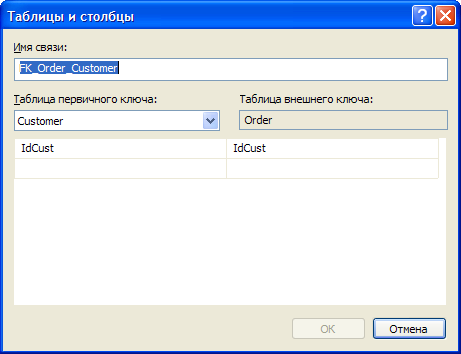
Суть обеспечения целостности ссылок очевидна из названия: данные в одной таблице, ссылающиеся на данные из другой таблицы, защищены от некорректного обновления. В терминологии SQL Server это называется *декларативной ссылочной целостностью* и достигается путем связывания первичного ключа одной из таблиц 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 (рассматривается в следующем разделе) 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…».



1. 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 Customer, а полей связи - IdCust.



1. Закройте все открывшиеся окна с сохранением изменений.

**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 следующей таблице.

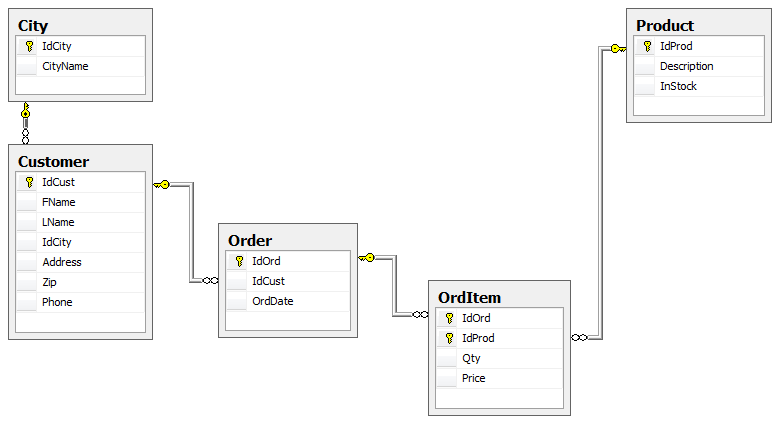
|  |  |  |
| --- | --- | --- |
| Настройка | Правило удаления | Правило обновления |
| Нет действия | Невозможно удалить в главной таблице строку, на которую есть ссылки в подчиненной | Невозможно обновить значения полей первичного ключа главной таблицы при наличии связанных записей в подчиненной |
| 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 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 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 для самостоятельной работы:** Создайте ограничение внешнего ключа FK\_OrdItem\_Order в таблице OrderItem для связи таблиц Order и OrderItem 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.

**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, щелкните на кнопке «Закрыть» и вы увидите созданную диаграмму базы данных (на рисунке представлен окончательный вид диаграммы: некоторые связи у вас могут отсутствовать).



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 ILM.
4. Расположите таблицы в канонической форме (главные таблицы выше подчиненных) в соответствии с вышеприведенным рисунком.

**Задание для самостоятельной работы:** Аналогичным образом создайте связь между таблицами Product и OrdItem по полю IdProduct. Окончательный список связей между таблицами со всеми их характеристиками представлен в следующей таблице:

|  |  |  |  |
| --- | --- | --- | --- |
| Главная таблица | Подчиненная таблица | Поле связи (внешний ключ) | Правила каскадирования |
| City | Customer | IdCity | Без действия |
| Customer | Order | IdCust | Без действия |
| Order | OrdItem | IdOrd | Каскадное удаление |
| Product | OrdItem | IdProd | Без действия |

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

* Ограничений проверки: попробуйте ввести в поле Zip (почтовый индекс) таблицы Customer нечисловые значения, а в поля InStock таблицы Product и Qty, Price таблицы OrdItem - отрицательные.
* Значений 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.
* Ограничений первичного и уникального ключа: попробуйте ввести в таблицы записи с дублирующими значениями первичного или уникального ключа.
* Ограничений 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.
* Правил каскадирования: убедитесь, что при удалении записи из таблицы Order все связанные записи из таблицы OrdItem удаляются автоматически.

**Лабораторная работа №3: Основы Transact SQL: Простые (однотабличные) выборки данных**

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 данных
* DML (Data Manipulation Language) - операторы манипуляции данными
* DCL (Data Control Language) - операторы определения доступа к данным
* TCL (Transaction Control Language) - операторы управления транзакциями

SQL является стандартизированным языком. Стандартный SQL поддерживается комитетом стандартов ANSI (Американский национальный институт стандартов), и соответственно называется ANSI SQL.

Многие разработчики СУБД расширили возможности SQL, введя в язык дополнительные операторы или инструкции. Эти расширения необходимы для выполнения дополнительных функций или для упрощения выполнения определенных операций. И хотя часто они очень полезны, эти расширения привязаны к определенной СУБД и редко поддерживаются более чем одним разработчиком. Все крупные СУБД и даже те, у которых есть собственные расширения, поддерживают ANSI SQL (в большей или меньшей степени). Отдельные же реализации носят собственные имена (PL-SQL, Transact-SQL и т.д.). Transact-SQL (T-SQL) – реализация языка SQL корпорации Microsoft, используемая, в частности, и в SQL Server.

**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**

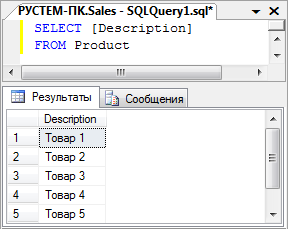
SELECT [Description]

FROM Product

В приведенном выше операторе используется оператор SELECT для выборки одного столбца под названием Description 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:

1. В контекстном меню базы Sales выберите команду «Создать запрос» или щелкните соответствующую кнопку на панели инструментов .
2. В открывшемся окне создания нового запроса введите представленные выше инструкции SQL.
3. Для запуска запроса на выполнение щелкните кнопку  на панели инструментов или нажмите клавишу F5. В нижней части экрана должны появиться результаты.



1. 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.

SELECT [Description], 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.

SELECT \*

FROM Product

**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. Взгляните на следующий пример.

SELECT IdProd, [Description], 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.

**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 количеству, а потом по названию.

SELECT IdProd, [Description], InStock

FROM Product

ORDER BY InStock, [Description]

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 Description, только если существует несколько строк с одинаковыми значениями InStock. Если никакие значения столбца InStock не совпадают, данные по столбцу Description сортироваться не будут.

**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.

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).

**Фильтрация данных**

В таблицах баз данных обычно содержится много информации и довольно редко возникает необходимость выбирать все строки таблицы. Гораздо чаще бывает нужно извлечь какую-то часть данных таблицы для каких-либо действий или отчетов. Выборка только необходимых данных включает в себя критерий поиска, также известный под названием предложение фильтрации. В операторе SELECT данные фильтруются путем указания критерия поиска в предложении WHERE. Предложение WHERE указывается сразу после названия таблицы (предложения FROM) следующим образом:

SELECT IdProd, [Description], InStock

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. SQL поддерживает весь спектр условных (логических) операций, которые приведены в следующей таблице.

|  |  |
| --- | --- |
| Операция | Описание |
| = | Равенство |
| <> | Неравенство |
| != | Неравенство |
| < | Меньше |
| <= | Меньше или равно |
| !< | Не меньше |
| > | Больше |
| >= | Больше или равно |
| !> | Не больше |
| BETWEEN | Между двумя указанными значениями |
| IS NULL | Значение NULL |

В следующем примере осуществляется выборка всех клиентов, для которых не указан контактный телефон.

SELECT FName, LName, Phone

FROM Customer

WHERE PHONE IS NULL

Для поиска диапазона значений можно использовать операцию BETWEEN. Ее синтаксис немного отличается от других операций предложения WHERE, так как для нее требуются два значения: начальное и конечное. Например, операцию BETWEEN можно использовать для поиска товаров, количество которых находится в промежутке между 5 и 10.

SELECT IdProd, [Description], InStock

FROM Product

WHERE InStock BETWEEN 5 AND 10

Для объединения в предложении WHERE нескольких условий необходимо использовать логические операторы AND и (или) OR. Оператор AND требует одновременного выполнения обоих условий. Запишем предыдущий запрос посредством объединения двух операции сравнения оператором AND.

SELECT IdProd, [Description], InStock

FROM Product

WHERE (InStock >= 5) AND (InStock <= 10)

Ключевое слово AND указывает СУБД возвращать только те строки, которые удовлетворяют всем перечисленным критериям отбора. В данном случае будут выбраны только те товары, количество которых находится в промежутке от 5 до 10.

Оператор OR указывает СУБД выбирать только те строки, которые удовлетворяют хотя бы одному из условий.

SELECT IdCity, CityName

FROM City

WHERE (CityName = 'Москва') OR (CityName = 'Казань')

Посредством этого SQL запроса из справочника городов выбираются только Москва и Казань. Ключевое слово OR указывает СУБД использовать какое-то одно условие, а не сразу два. Если бы здесь использовалось ключевое слово AND, мы бы не получили никаких данных.

Если вы внимательно рассмотрите выражение в предыдущем предложении WHERE, то заметите, что значения, с которыми сравниваются названия городов, заключены в одинарные кавычки. Одинарные кавычки используются для определения границ строки (строковой константы). При работе со строковыми константами их всегда необходимо отделять одинарными кавычками.

Предложения WHERE могут содержать любое количество логических операторов AND и OR. Комбинируя их можно создавать сложные фильтры. Однако при комбинировании ключевых слов AND и OR необходимо учитывать, что оператор AND выполняется раньше оператора OR, т.е. имеет более высокий приоритет. Изменить приоритет можно с помощью круглых скобок.

В следующем примере осуществляется выборка из таблицы клиентов всех Ивановых и Петровых, для которых не указан контактный телефон.

SELECT FName, LName, Phone

FROM Customer

WHERE (LName = 'Иванов' OR LName = 'Петров') AND PHONE IS NULL

В случае отсутствия скобок результат был бы не верным, а именно включал бы в себя всех Петровых без контактного телефона и всех Ивановых без каких либо ограничений.

Для определения входит ли сравниваемое значение в определенное заданное множество можно воспользоваться оператором IN. При этом все допустимые значения, заключенные в скобки, перечисляются через запятую. В частности предыдущий пример с использованием оператора IN может быть записан в более компактной форме.

SELECT FName, LName, Phone

FROM Customer

WHERE LName IN ('Иванов','Петров') AND PHONE IS NULL

Для отрицания какого-то условия используется логический оператор NOT. Поскольку NOT никогда не используется сам по себе (а только вместе с другими логическими операторами), его синтаксис немного отличается от синтаксиса остальных операторов. В отличие от них, NOT вставляется перед названием столбца, значения которого нужно отфильтровать, а не после. В следующем примере отбираются все клиенты, для которых имеются сведения об их контактом телефоне.

SELECT FName, LName, Phone

FROM Customer

WHERE NOT PHONE IS NULL

Для фильтрации данных по критерию соответствия определенной символьной строки заданному шаблону используется оператор LIKE. Шаблон может включать обычные символы и символы-шаблоны. Во время сравнения с шаблоном необходимо, чтобы его обычные символы в точности совпадали с символами, указанными в строке. Символы-шаблоны могут совпадать с произвольными элементами символьной строки. Использование символов-шаблонов с оператором LIKE предоставляет больше возможностей, чем использование обычных операторов сравнения. Шаблон может включать в себя следующие символы-шаблоны.

|  |  |  |
| --- | --- | --- |
| Символ-шаблон | Описание | Пример |
| % | Любое количество символов | Инструкция WHERE FName LIKE 'А%' выполняет поиск и выдает всех клиентов, имена которых начинаются на букву ‘А’. |
| \_ | Любой одиночный символ | Инструкция 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.). |
| [] | Любой символ, указанный в квадратных скобках | Инструкция 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. |
| [^] | Любой символ, кроме перечисленных в квадратных скобках | Инструкция WHERE LName LIKE 'ив[^а]%' выполняет поиск и выдает всех клиентов, фамилии которых начинаются на ‘ив’ и третья буква отличается от ‘а’. |

В следующем примере осуществляется выборка всех товаров, названия которых начинаются на букву Т.

SELECT \*

FROM Product

WHERE [Description] LIKE 'Т%'

**Создание вычисляемых полей**

Конструкция SELECT кроме имен столбцов таблиц может также включать так называемые вычисляемые поля. В отличие от всех выбранных нами ранее столбцов, вычисляемых полей на самом деле в таблицах базы данных нет. Они создаются "на лету" SQL-оператором SELECT. Рассмотрим следующий пример.

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

FROM Customer

Здесь создается вычисляемое поле, которому с помощью ключевого слова AS дан псевдоним ‘Фамилия и имя клиента’. Оно позволяет объединить (произвести конкатенацию) с помощью оператора + фамилию, пробел и имя клиента в одно поле (столбец). Псевдоним может быть задан и для обычного столбца таблицы. В частности здесь столбцу IdCust задан псевдоним ‘Номер клиента’.

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

SELECT IdProd, Qty, Price, Qty \* Price AS 'Стоимость'

FROM OrdItem

WHERE IdOrd = 1

Здесь с помощью оператора умножения \* вычисляется общая стоимость каждого товара в заказе с кодом 1 как произведение количества на цену.

**Исключение дублирующих записей**

Для исключения из результата выборки повторяющихся строк используется ключевое слово DISTINCT, которое указывается сразу после SELECT. В следующем примере осуществляется вывод всех фамилий клиентов. Даже если среди них есть однофамильцы, каждая фамилия будет выведена только один раз.

SELECT DISTINCT LName

FROM Customer

**Задание для самостоятельной работы:** Сформулируйте на языке SQL запросы на выборку следующих данных:

* Список всех заказов за определенный период времени (например, сентябрь 2010 года) отсортированный по дате заказа;
* Список всех товаров, названия которых включают слово ‘монитор’ с указанием их остатка на складе.

**Использование агрегатных функций**

В SQL определено множество встроенных функций различных категорий, среди которых особое место занимают агрегатные функции, оперирующие значениями столбцов множества строк и возвращающие одно значение. Аргументами агрегатных функций могут быть как столбцы таблиц, так и результаты выражений над ними. Агрегатные функции и сами могут включаться в другие арифметические выражения. В следующей таблице приведены наиболее часто используемые стандартные унарные агрегатные функции.

|  |  |
| --- | --- |
| **Функция** | **Возвращаемое значение** |
| COUNT | Количество значений в столбце или строк в таблице |
| SUM | Сумма |
| AVG | Среднее |
| MIN | Минимум |
| MAX | Максимум |

Общий формат унарной агрегатной функции следующий:

имя\_функции([АLL | DISTINCT] выражение)

где DISTINCT указывает, что функция должна рассматривать только различные значения аргумента, а ALL — все значения, включая повторяющиеся (этот вариант используется по умолчанию). Например, функция AVG с ключевым словом DISTINCT для строк столбца со значениями 1, 1, 1 и 3 вернет 2, а при наличии ключевого слова ALL вернет 1,5.

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

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

* COUNT(\*)
* COUNT([DISTINCT | ALL] выражение)

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

**Запрос:** Количество видов продукции, информация о которых имеется в базе данных.

SELECT COUNT(\*) AS 'Количество видов продукции'

FROM Product

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

**Запрос:** Количество различных имен, содержащихся в таблице Customer.

SELECT COUNT(DISTINCT FNAME)

FROM Customer

Использование остальных унарных агрегатных функции аналогично COUNT за тем исключением, что для функций MIN и MAX использование ключевых слов DISTINCT и ALL не имеет смысла. С функциями COUNT, MAX и MIN кроме числовых могут использоваться и символьные поля. Если аргумент агрегатной функции не содержит значений, функция COUNT возвращает 0, а все остальные - значение NULL.

**Запрос:** Дата последнего заказа до 1 сентября 2010 года.

SELECT MAX(OrdDate)

FROM [Order]

WHERE OrdDate<'1.09.2010'

**Задание для самостоятельной работы:** Сформулируйте на языке SQL запросы на выборку следующих данных:

* Суммарная стоимость всех заказов;
* Количество различных городов, содержащихся в таблице Customer.

**Запросы с группировкой строк**

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 оно может быть:

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

Самым простым вариантом использования фразы GROUP BY является группировка по значениям одного столбца.

**Запрос:** Количество клиентов по городам.

SELECT IdCity, COUNT(\*) AS 'Кол-во клиентов'

FROM Customer

GROUP BY IdCity

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.

**Запрос:** Количество клиентов по городам с фамилией ‘Иванов’.

SELECT IdCity, COUNT(\*) AS 'Кол-во клиентов'

FROM Customer

WHERE LName = 'Иванов'

GROUP BY IdCity

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

**Запрос:** Количество клиентов по каждой фамилии и имени.

SELECT LName, FName, COUNT(\*)

FROM Customer

GROUP BY LName, FName

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.

**Запрос:** Список городов, количество клиентов из которых больше 10.

SELECT IdCity

FROM Customer

GROUP BY IdCity

HAVING COUNT(\*)>10

**Задание для самостоятельной работы:** Сформулируйте на языке SQL запросы на выборку следующих данных:

* Список всех заказов с указанием их суммарной стоимости;
* Список клиентов, которые за заданный период (например, сентябрь 2010 года) совершили более 3 заказов.

**Лабораторная работа №4: Основы Transact SQL: Сложные (многотабличные запросы)**

В SQL сложные запросы являются комбинацией простых SQL-запросов. Каждый простой запрос в качестве ответа возвращает набор записей (таблицу), а комбинация простых запросов возвращает результат тех или иных операций над ответами на простые запросы.

В SQL сложные запросы получаются из других запросов следующими способами:

* вложением SQL-выражения запроса в SQL-выражение другого запроса. Первый из них называют подзапросом, а второй — внешним или основным запросом;
* применением к SQL-запросам операторов объединения и соединения наборов записей, возвращаемых запросами. Эти операторы называют теоретико-множественными или реляционными.

**Подзапросы**

Подзапрос — это запрос на выборку данных, вложенный в другой запрос. Подзапрос всегда заключается в круглые скобки и выполняется до содержащего выражения. Внешний запрос, содержащий подзапрос, если только он сам не является подзапросом, не обязательно должен начинаться с оператора SELECT. В свою очередь, подзапрос может содержать другой подзапрос и т. д. При этом сначала выполняется подзапрос, имеющий самый глубокий уровень вложения, затем содержащий его подзапрос и т. д. Часто, но не всегда, внешний запрос обращается к одной таблице, а подзапрос — к другой. На практике именно этот случай наиболее интересен.

**Простые подзапросы**

Простые подзапросы характеризуются тем, что они формально никак не связаны с содержащими их внешними запросами. Это обстоятельство позволяет сначала выполнить подзапрос, результат которого затем используется для выполнения внешнего запроса. Кроме простых подзапросов, существуют еще и связанные (коррелированные) подзапросы, которые будут рассмотрены в следующем разделе.

Рассматривая простые подзапросы, следует выделить три частных случая:

* подзапросы, возвращающие единственное значение;
* подзапросы, возвращающие список значений из одного столбца таблицы;
* подзапросы, возвращающие набор записей.

Тип возвращаемой подзапросом таблицы определяет, как можно ее использовать и какие операторы можно применять в содержащем выражении для взаимодействия с этой таблицей. По завершении выполнения содержащего выражения таблицы, возвращенные любым подзапросом, выгружаются из памяти. Таким образом, подзапрос действует как временная таблица, областью видимости которой является выражение (т. е. после завершения выполнения выражения сервер высвобождает всю память, отведенную под результаты подзапроса).

**Подзапросы, возвращающие единственное значение**

Допустим, из таблицы Customer требуется выбрать данные обо всех клиентах из Казани. Это можно сделать с помощью следующего запроса.

SELECT \*

FROM Customer

WHERE IdCity = (SELECT idCity FROM City WHERE CityName = 'Казань')

В данном запросе сначала выполняется подзапрос (SELECT idCity FROM City WHERE CityName = 'Казань'). Он возвращает единственное значение (а не набор записей, поскольку по полю City организовано ограничение уникальности) – уникальный идентификатор города Казань. Если сказать точнее, то данный подзапрос возвращает единственную запись, содержащую единственное поле. Далее выполняется внешний запрос, который выводит все столбцы таблицы Customer и записи, в которых значение столбца IdCity равно значению, полученному с помощью подзапроса. Таким образом, сначала выполняется подзапрос, а затем внешний запрос, использующий результат подзапроса.

**Задание для самостоятельной работы:** По аналогии с предыдущим примером сформулируйте запрос, возвращающий все заказы, в которых содержится заданный товар (по названию товара).

**Подзапросы, возвращающие список значений из одного столбца таблицы**

Подзапрос, вообще говоря, может возвращать несколько записей. 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 (или ANY) (некоторый).

Рассмотрим общий случай использования запросов с кванторами ALL и SOME. Пусть имеются две таблицы: T1, содержащая как минимум столбец A, и T2, содержащая, по крайней мере, один столбец B. Тогда запрос с квантором ALL можно сформулировать следующим образом:

SELECT A FROM T1  
WHERE A оператор\_сравнения ALL (SELECT B FROM T2)

Здесь оператор\_сравнения обозначает любой допустимый оператор сравнения. Данный запрос должен вернуть список всех тех значений столбца A, для которых оператор сравнения истинен для всех значений столбца B.

Запрос с квантором SOME, очевидно, имеет аналогичную структуру. Он должен вернуть список всех тех значений столбца A, для которых оператор сравнения истинен хотя бы для какого-нибудь одного значения столбца B.

**Запрос:** Список всех клиентов, проживающих в городах Казань или Елабуга.

SELECT \*

FROM Customer

WHERE IdCity = SOME(SELECT IdCity FROM City WHERE CityName IN ('Казань', 'Елабуга'))

Предыдущий запрос может быть также реализован и с использованием оператора IN, который рассматривался в разделе “Фильтрация данных”.

SELECT \*

FROM Customer

WHERE IdCity IN (SELECT IdCity FROM City WHERE CityName IN ('Казань', 'Елабуга'))

Напомним — он проверяет вхождение элемента во множество, в качестве элемента может выступать имя столбца или скалярное выражение, а в качестве множества — явно заданный список значений или подзапрос. Использование подзапроса в качестве второго операнда IN также как и кванторы позволяет избежать ограничения на единственность значения, возвращаемого подзапросом.

С помощью оператора IN можно проверять не только наличие значения в наборе значений, но и его отсутствие. Делается это добавлением оператора отрицания NOT. Вот другой вариант предыдущего запроса:

SELECT \*

FROM Customer

WHERE IdCity NOT IN (SELECT IdCity FROM City WHERE CityName IN ('Казань', 'Елабуга'))

Этот запрос возвращает всех клиентов, кроме тех которые проживают в городах Казань и Елабуга.

Аналогичный запрос с использование квантора ALL:

SELECT \*

FROM Customer

WHERE IdCity != ALL(SELECT IdCity FROM City WHERE CityName IN ('Казань', 'Елабуга'))

**Задание для самостоятельной работы:** Cформулируйте запрос, возвращающий список всех клиентов (с указанием фамилии и имени), совершивших заказ за определенный период времени.

**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.

**Связанные (коррелированные) подзапросы**

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.

SELECT \*

FROM Customer c

WHERE 1 < (SELECT COUNT(\*) FROM [Order] r WHERE r.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 с.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 большее 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.

SELECT IdProd, [Description]

FROM Product p

WHERE EXISTS (SELECT \* FROM OrdItem oi WHERE oi.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.

SELECT IdProd, [Description]

FROM Product p

WHERE NOT EXISTS (SELECT \* FROM OrdItem oi WHERE oi.IdProd = p.IdProd)

**Задание для самостоятельной работы:** Cформулируйте запрос, возвращающий список всех заказов с суммарной стоимость более заданной величины.

**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, например, таблица Customer содержит столбец IdCity, который для каждой строки клиента содержит значение первичного ключа того города, в котором проживает данный клиент. Наличие внешних ключей является основой для инициирования поиска по многим таблицам.

Одна из наиболее важных особенностей предложения SELECT — это способность использования связей между различными таблицами, а также вывода содержащейся в них информации. Операция, которая приводит к соединению из двух таблиц всех пар строк, для которых выполняется заданное условие, называется соединением таблиц.

**Соединение таблиц во фразе WHERE по равенству значений столбцов различных таблиц**

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. Так, например, мы можем соединить таблицы городов и клиентов по условию City.IdCity = Customer.IdCity. В таком варианте мы соединяем таблицы осмысленно, так как 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:

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

Эти два варианта, а также их комбинация, характерны для любого вида соединения, а не только по равенству. Рассмотрим следующие примеры.

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.

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 предложение SELECT это было не обязательно, так как 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. В частности предыдущий запрос с использование синонимов для таблиц можно записать более компактным образом.

SELECT k.FName, k.LName, c.CityName

FROM Customer k, City c

WHERE k.IdCity = c.IdCity

Следующий запрос отбирает всех клиентов из Казани с фамилией Иванов

SELECT K.IdCust, k.FName

FROM Customer k, City c

WHERE k.IdCity = c.IdCity AND k.LName = 'Иванов' AND c.CityName = 'Казань'

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

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

SELECT DISTINCT c.IdCust, c.FName, c.LName

FROM Customer c, [Order] o, OrdItem oi

WHERE c.IdCust = o.IdCust AND o.IdOrd = oi.IdOrd AND oi.IdProd = 1

Сформулируем общую процедуру составления многотабличного запроса.

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

**Задание для самостоятельной работы:** Cформулируйте запрос, возвращающий список товаров в заданном заказе (по заданному IdOrd). Результат должен включать следующие поля: название товара, цена, количество, стоимость.

**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}

**Внутреннее соединение**

В операторе JOIN внутреннее соединение указывается ключевым словом INNER (впрочем, его можно опустить, так как 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.

**Запрос:** 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.

**Запрос:** Список всех клиентов из Казани с фамилией Иванов

SELECT K.IdCust, k.FName

FROM Customer k INNER JOIN

City c ON k.IdCity = c.IdCity

WHERE k.LName = 'Иванов' AND c.CityName = 'Казань'

**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 таблицей, 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

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

Если в данном 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). Обратите внимание, что таблица City соединяется не с таблицей, а с подзапросом, которому задан псевдоним a.

**Задание для самостоятельной работы:** Сформулируйте на языке SQL запросы на выборку следующих данных (с использование оператора JOIN для соединения таблиц):

* список всех товаров, которые когда-либо заказывал заданный клиент;
* список всех клиентов, не имеющих ни одного заказа.

**Множественные операции**

Множественные операции выполняются над наборами записей (таблицами), полученными в результате запросов, и, в свою очередь, возвращают таблицу.

В стандарте SQL множественные операции имеют следующий синтаксис:

запрос {UNION | INTERSECT | EXCEPT} [DISTINCT | ALL] запрос

где запрос является предложением SELECT. Отличаются эти операции тем, какие строки возвращенных запросами таблиц отбираются в новую результирующую таблицу:

* UNION — все строки таблиц, возвращенных обоими запросами;
* INTERSECT — только те строки, которые имеются в таблицах обоих запросов;
* EXCEPT — только те строки таблицы первого запроса, которых нет среди строк таблицы второго запроса.

Запросы, содержащие множественные операторы, называются составными.

Стандарт SQL определяет следующие правила относительно повторяющихся строк в таблицах.

* базовые таблицы не могут содержать повторяющихся строк (это принципиальное требование реляционной модели данных);
* результирующие таблицы запросов могут содержать повторяющиеся строки, если это не запрещено ключевым словом DISTINCT во фразе SELECT;
* результирующие таблицы множественных операций не могут содержать повторяющихся строк, если это не разрешено ключевым словом ALL.

Таким образом, именно ключевые слова ALL и DISTINCT указывают, допускаются ли в результирующей таблице повторяющиеся строки. В запросах при отсутствии явного указания предполагается использование ключевого слова ALL, а во множественных операциях — DISTINCT.

Таблицы, используемые в качестве операндов множественной операции, должны быть совместимы. Под этим подразумевается следующее:

* обе таблицы должны иметь одинаковое количество столбцов;
* соответствующие пары столбцов должны быть одинаковых или совместимых типов.

**Объединение наборов записей**

Нередко требуется объединить записи двух или более таблиц с похожими структурами в одну таблицу. Иначе говоря, к набору записей, возвращаемому одним запросом, требуется добавить записи, возвращаемые другим запросом. Для этого служит оператор UNION (объединение):

3anpoc1 UNION Запрос2;

При этом в результатной таблице остаются только отличающиеся записи. Чтобы сохранить в ней все записи, после оператора UNION следует написать ключевое слово ALL.

К базе данных Sales сложно сформулировать осмысленный запрос с объединением, который бы имел какую-либо практическую ценность. Поэтому в качестве примера рассмотрим объединение результатов выполнения запросов, возвращающих просто константные значения.

**Запрос:** Объединение с исключением дублирующих строк

SELECT 1, 'Один'

UNION

SELECT 1, 'Один'

UNION

SELECT 2, 'Два'

**Запрос:** Объединение с сохранением дубликатов

SELECT 1, 'Один'

UNION ALL

SELECT 1, 'Один'

UNION ALL

SELECT 2, 'Два'

**Пересечение наборов записей**

Пересечение двух наборов записей осуществляется с помощью оператора INTERSECT (пересечение), возвращающего таблицу, записи в которой содержатся одновременно в двух наборах:

Запрос1 INTERSECT Запрос2;

(SELECT 1, 'Один'

UNION

SELECT 2, 'Два'

UNION

SELECT 3, 'Три')

INTERSECT

(SELECT 1, 'Один'

UNION

SELECT 2, 'Два'

UNION

SELECT 4, 'Четыре')

**Разность наборов записей**

Для получения записей, содержащихся в одном наборе и отсутствующих в другом, служит оператор EXCEPT(за исключением):

Запрос1 ЕХCЕРТ Запрос2;

(SELECT 1, 'Один'

UNION

SELECT 2, 'Два'

UNION

SELECT 3, 'Три')

EXCEPT

(SELECT 1, 'Один'

UNION

SELECT 2, 'Два'

UNION

SELECT 4, 'Четыре')

**Лабораторная работа №5: Основы Transact SQL: Добавление, изменение и удаление данных в таблицах**

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.

**Добавление новых записей**

Для вставки записей в таблицу используется оператор INSERT, который имеет несколько форм:

* INSERT INTO имяТаблицы VALUES (списокЗначений)

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 имяТаблицы (списокСтолбцов) VALUES (списокЗначений)

вставляет запись в указанную таблицу и вводит в заданные столбцы значения из указанного списка. При этом в первый столбец из списокСтолбцов вводится первое значение из списокЗначений, 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.

INSERT INTO City(CityName)

VALUES('Калуга')

Обратите внимание, что столбец IdCity не задается, поскольку он является счетчиком и заполняется СУБД автоматически.

* INSERT INTO имяТаблицы (списокСтолбцов) 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.

INSERT INTO City(CityName)

SELECT 'Уфа'

UNION

SELECT 'Волгоград'

**Удаление записей**

Для удаления записей из таблицы применяется оператор DELETE:

DELETE FROM имяТаблицы 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':

DELETE FROM Customer

WHERE LName = 'Иванов'

Если таблица содержатся сведения о нескольких клиентах с фамилией Иванов, то все они будут удалены.

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 Москва, при этом уникальный идентификатор города возвращается с помощью подзапроса.

DELETE FROM Customer

WHERE IdCity IN (SELECT IdCity FROM City WHERE CityName = 'Москва')

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

DELETE FROM Customer

FROM Customer k INNER JOIN

City c ON k.IdCity = c.IdCity AND c.CityName = 'Москва'

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

SELECT \*

FROM Customer k INNER JOIN

City c ON k.IdCity = c.IdCity AND c.CityName = 'Москва'

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.

DELETE FROM Product

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

**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 имяТаблицы  
SET имяСтолбца = 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 имяТаблицы  
SET имяСтолбца1 = значение1, имяСтолбца2 = значение2, ... , имяСтолбцаN = значениеN  
WHERE условие;

Например, следующий запрос изменяет фамилию и имя клиента с кодом 5.

UPDATE Customer

SET FName='Иван', LName='Иванов'

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 необязательно. Если обновляемый объект встречается в предложении FROM несколько раз, одна и только одна ссылка на этот объект 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 Price, просто умножить их на 0,95. Однако эти изменения должны быть выполнены, только если суммарная стоимость заказа превышает 1000. Таким образом, в качестве критерия обновления записей в таблице OrdItem может быть задан запрос возвращающий список всех заказов с суммарной стоимостью более 1000.

UPDATE OrdItem

SET Price = Price \* 0.95

FROM OrdItem o INNER JOIN

(SELECT IdOrd

FROM OrdItem

GROUP BY IdOrd

HAVING SUM(Qty\*Price) > 1000) a ON o.IdOrd = a.IdOrd

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. Например, перед выполнением приведенного ранее запроса на обновление данных не помешает выполнить соответствующий запрос на выборку данных.

SELECT \*

FROM OrdItem o INNER JOIN

(SELECT IdOrd

FROM OrdItem

GROUP BY IdOrd

HAVING SUM(Qty\*Price) > 1000) a ON o.IdOrd = a.IdOrd

**Задание для самостоятельной работы:** Сформулируйте на языке SQL запрос имитирующий поступление на склад новой партий определенного товара (Обновление столбца InStock в таблице Product).

**Лабораторная работа №6: Представления**

Представления – это именованные запросы на выборку данных (инструкции SELECT на языке T-SQL), хранящиеся в базе данных. В запросах представления можно использовать, так же как и таблицы, независимо от сложности их инструкций SELECT. Подобно таблицам представления также состоят из полей и записей. Однако в отличие от таблиц они не содержат каких-либо данных (за исключением материализованных (индексированных) представлений). 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).

• 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 позволяет упрощать 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.

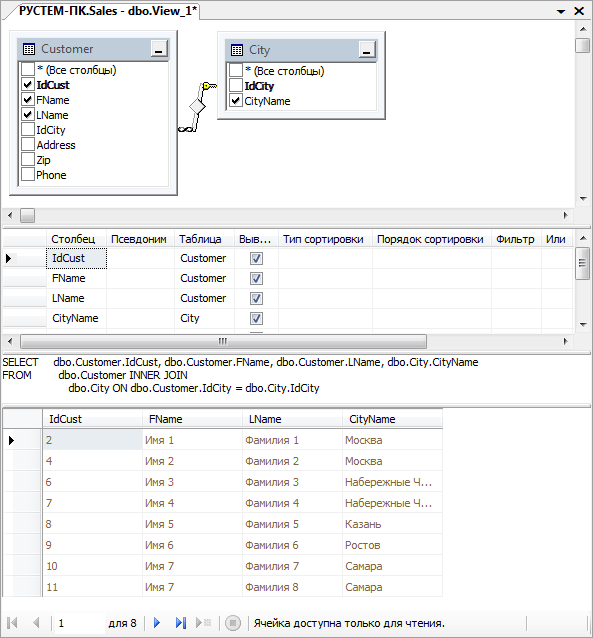
**Создание представлений в Management Studio**

В утилите SQL Server Management Studio представления можно создавать, редактировать, выполнять и вставлять в другие запросы.

Поскольку представление является ничем иным, как сохраненной инструкцией SELECT, его создание начинается с проектирования этой инструкции. Инструкция SELECT, если она является корректной, может быть вырезана и вставлена в представление практически из любого инструмента.

В утилите Management Studio представления перечислены в собственном узле в каждой базе данных.

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



Конструктор запросов утилиты Management Studio способен одновременно отображать множество панелей, выбранных на панели инструментов .

* *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 условий)*. На этой панели перечисляются отображаемые, фильтруемые и сортируемые столбцы.
* *Панель SQL*. На этой панели можно в текстовом виде увидеть и отредактировать формируемую инструкцию SELECT.
* *Панель результатов*. Когда запрос выполняется с помощью действия «Выполнить код SQL» , на этой панели отображаются его результаты. Если результаты запроса остаются нетронутыми долгое время, Management Studio запрашивает у пользователя разрешение закрыть подключение к серверу.

Фактический код SQL отображается и редактируется в панели SQL кода. Столбцы в представление можно добавлять в панелях диаграммы, сетки и SQL кода. Функция добавления таблиц доступна в контекстном меню, а также на панели инструментов. Здесь можно добавлять таблицы, другие представления, синонимы и табличные функции.

Таблицы и другие представления могут быть добавлены посредством перетаскивания их на панель диаграммы из окна «Обозреватель объектов» или с помощью команды «Добавить таблицу»  на панели инструментов или в контекстном меню.

Функция добавления производных таблиц  способна добавить в предложение FROM представления в качестве источника данных подзапрос. Код SQL этого подзапроса можно ввести вручную на панели SQL.

Кнопка «Проверить синтаксис SQL»  позволяет проверить синтаксис введенных инструкций SQL. В то же время она не проверяет имена таблиц, столбцов и представлений в инструкции 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, выделяя название и выбирая в контекстном меню команду «Проект».

Для тестирования инструкции SELECT представления в конструкторе запросов щелкните на кнопке «Выполнить код SQL»  или нажмите 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\_Customer, возвращающее список клиентов с указанием города проживания, программным путем, в окне запросов должна быть выполнена следующая команда.

CREATE VIEW [dbo].[v\_Customer]

AS

SELECT dbo.Customer.IdCust, dbo.Customer.FName, dbo.Customer.LName, dbo.City.CityName

FROM dbo.Customer INNER JOIN

dbo.City ON dbo.Customer.IdCity = dbo.City.IdCity

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  
измененная\_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\_Customer и упорядочивает их по полям LName и FName. Предложение ORDER BY не является частью представления v\_Customer, а применяется к нему с помощью вызова инструкции SQL:

SELECT IdCust, FName, LName, CityName

FROM dbo.v\_Customer

ORDER BY LName, FName

**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\_Customer

**Задание для самостоятельной работы:** Создайте представление возвращающее список заказов с указанием имени клиента и количества товаров в каждом заказе. Таким образом, результат должен включать следующие атрибуты: IdOrd, OrdDate, IdCust, FName, LName, Количество видов товаров в заказе.

**Лабораторная работа №7: 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 С. 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:

SELECT CityName – извлекаемые столбцы

FROM City – исходная таблица

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 С 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 С является то, что многострочные запросы в них можно выполнять, даже не раскомментируя.

**Пакеты T-SQL**

Запросом называют одну инструкцию T-SQL, а пакетом — их набор. Вся последовательность инструкций пакета отправляется серверу из клиентских приложений как одна цельная единица.

SQL Server рассматривает весь пакет как рабочую единицу. Наличие ошибки хотя бы в одной инструкции приведет к невозможности выполнения всего пакета. В то же время грамматический разбор не проверяет имена объектов и схем, так как сама схема может измениться в процессе выполнения инструкции.

Файл сценария SQL и окно анализатора запросов (Query Analyzer) может содержать несколько пакетов. В данном случае все пакеты разделяют ключевые слова терминаторов. По умолчанию этим ключевым словом является GO, и оно должно быть единственным в строке. Все другие символы (даже комментарии) нейтрализуют разделитель пакета.

**Отладка T-SQL**

Когда редактор SQL обнаруживает ошибку, он отображает ее характер и номер строки в пакете. Дважды щелкнув на ошибке, можно сразу же переместиться к соответствующей строке.

В утилиту Management Studio версии SQL Server 2005 не включен отладчик языка T-SQL, — он присутствует в пакете Visual Studio.

SQL Server предлагает несколько команд, облегчающих отладку пакетов. В частности, команда PRINT отправляет сообщение без генерации результирующего набора данных. Команду PRINT можно использовать для отслеживания хода выполнения пакета. Когда анализатор запросов находится в режиме сетки, выполните следующий пакет:

SELECT CityName

FROM City

WHERE IdCity = 1;

PRINT 'Контрольная точка';

Результирующий набор данных отобразится в сетке и будет состоять из одной строки. В то же время во вкладке «Сообщения» отобразится следующий результат:

(строк обработано: 1)

Контрольная точка

**Переменные**

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

DECLARE @Имя\_Переменной Тип\_Данных [,

@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. В частности 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 и 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 Product

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 - 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 - 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 инструкция I  
[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 останова выполнения пакета, а следовательно, хранимой процедуры и триггера (рассматриваются в следующих лабораторных занятиях).

**Лабораторная работа №8: Хранимые процедуры**

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, и 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 Туре> <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 необязательно.

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, так как 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, начиная с 01.01.2010.

CREATE PROCEDURE spr\_getOrders

AS

SELECT IdOrd, IdCust, OrdDate

FROM [Order]

WHERE (OrdDate >= '01.01.2010')

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 2010 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 '01.01.2010', '01.07.2010'

EXEC spr\_getOrders

@dateBegin = '01.01.2010',

@dateEnd = '01.07.2010'

EXEC spr\_getOrders '01.01.2010', @dateEnd = '01.07.2010'

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 являются необязательным, поскольку для него указано значение по умолчанию в виде нуля, которое и будет использовано, в случае 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 процедуре, 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 для самостоятельной работы:** Создайте хранимые процедуры, реализующие следующие действия:

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