МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Омский государственный технический университет»

О. Б. Малков, М. В. Девятерикова

РАБОТА С TRANSACT-SQL

Учебное текстовое электронное издание
локального распространения

Омск
Издательство ОмГТУ
2015

Сведения об издании: 1, 2

© ОмГТУ, 2015

ISBN 978-5-8149-1923-6

УДК 004.65(075)
ББК 32.973.26-018.2я73
М19

Рецензенты:
А. А. Колоколов, доктор физ.-мат. наук, профессор, зав. лабораторией дискретной оптимизации Омского филиала Института математики им. С. Л. Соболева СО РАН;

О. Н. Лучко, профессор, зав. кафедрой прикладной информатики и математики Омского государственного института сервиса

Малков, О. Б.
М19 Работа с Transact-SQL : учеб. пособие / О. Б. Малков, М. В. Девятерикова ; Минобрнауки России, ОмГТУ. – Омск : Изд-во ОмГТУ, 2015.

ISBN 978-5-8149-1923-6

Описана практическая работа с СУБД Microsoft SQL Server средствами языка Transact-SQL. Рассмотрен процесс создания и использования реляционной базы данных, включая реализацию запросов, представлений, транзакций, хранимых процедур, триггеров и курсоров.

Предназначено для студентов, обучающихся по направлениям 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 27.03.03 «Системный анализ и управление». Будет полезно студентам, изучающим дисциплины «Базы данных» и «Системы управления базами данных».

УДК 004.65(075)
ББК 32.973.26-018.2я73

Рекомендовано редакционно-издательским советом Омского государственного технического университета

ISBN 978-5-8149-1923-6
© ОмГТУ, 2015

1 электронный оптический диск

Оригинал-макет издания выполнен в Microsoft Office Word 2007
с использованием возможностей Adobe Acrobat X.

Минимальные системные требования:

- процессор Intel Pentium 1,3 ГГц и выше;
- оперативная память 256 Мб;
- свободное место на жестком диске 260 Мб;
- операционная система Microsoft Windows XP/Vista/7;
- разрешение экрана 1024×576 и выше;
- акустическая система не требуется;
- дополнительные программные средства Adobe Acrobat Reader 5.0 и выше.

№ госрегистрации ##########

Редактор *О. В. Маер*
Компьютерная верстка *О. Н. Савостеевой*
Сводный темплан 2015 г.
Подписано к использованию 30.01.15.
Объем 2,4 Мб.

Издательство ОмГТУ.
644050, г. Омск, пр. Мира, 11; т. 23-02-12
Эл. почта: info@omgtu.ru

ПРЕДИСЛОВИЕ

Microsoft SQL Server (далее – SQL Server) – популярная система управления реляционными базами данных размером от персональных до крупных баз данных масштаба предприятия. В этом сегменте рынка она успешно конкурирует с другими СУБД. Операторы языка Transact-SQL позволяют создавать, изменять и удалять объекты базы данных и саму базу данных; создавать представления, хранимые процедуры и триггеры; добавлять, модифицировать, удалять и искать данные.

В ходе изучения дисциплины и выполнения практических работ студент на примере учебной базы данных *Book* последовательно проходит путь от создания базы данных и заполнения таблиц до работы с транзакциями и программируемыми объектами базы данных.

В учебном пособии рассмотрен процесс создания и использования реляционной базы данных средствами языка Transact-SQL на СУБД Microsoft SQL Server, описаны способы обеспечения целостности и безопасности данных. Состоит из двух частей: теоретической и практической. Теоретическая часть содержит общие сведения по дисциплине, практическая часть включает десять практических занятий, в каждом из которых кратко представлен теоретический материал по рассматриваемой теме и приведены конкретные примеры.

Практикум реализован на версии SQL Server 2012 Express – мощной и надежной бесплатной СУБД, обеспечивающей функциональное хранилище данных для веб-сайтов и настольных приложений. Установочные файлы свободно загружаются со страницы www.microsoft.com (достаточен пакет SQLEXPRWT – выпуск Express с инструментами). Пакет содержит все необходимое для установки и настройки SQL Server в качестве сервера баз данных, включая полную версию интегрированной среды SQL Server 2012 Management Studio для доступа, настройки, управления, администрирования и разработки.

По результатам практической работы оформляется отчет в виде пояснительной записки объемом 20–25 страниц формата А4. Отчет должен содержать номер и тему занятия, задания и результаты их выполнения. Текст готовится с помощью текстового процессора Microsoft Word. Размеры полей страницы, мм: левое – 25, правое – 10, верхнее – 20, нижнее – 25. Размер шрифта – 14 пунктов. Тип шрифта – Times New Roman. Величина абзацного отступа – 10 мм. Межстрочный интервал – одинарный. Каждый раздел записки начинается с новой страницы, номера страниц располагаются внизу страницы по центру. Рисунки и таблицы нумеруются последовательно арабскими цифрами.

Полученные знания, умения и навыки могут быть использованы при создании приложений баз данных для автоматизированных информационных систем, разрабатываемых в рамках выпускной квалификационной работы.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

База данных (БД) – совокупность специальным образом организованных данных, при наличии минимальной избыточности, допускающей их оптимальное использование одним или несколькими пользователями (приложениями). Наибольшее распространение получили *реляционные базы данных*, в которых данные представлены в виде двумерных таблиц (отношений).

Система управления базами данных (*СУБД)* – программное обеспечение, предназначенное для ведения баз данных. Ведение включает создание баз данных, поддержание в актуальном состоянии и поиск данных по запросам пользователей.

SQL (Structured Query Language) – структурированный язык запросов, являющийся стандартом языка работы с реляционными базами данных.

Язык SQL стандартизован Американским национальным институтом стандартов (ANSI) и Международной организацией по стандартизации (ISO). Производители СУБД изменяют и дополняют этот язык, формируя диалекты SQL. Основа диалекта совместима со стандартом, дополнения в разных СУБД различны.

Язык SQL не является процедурным языком программирования, поэтому вводимые расширения являются процедурными. Наиболее известные – Transact-SQL (SQL Server) и PL/SQL (Oracle Database).

Transact-SQL расширил язык SQL, включив:

- управляющие операторы;
- локальные и глобальные переменные;
- хранимые процедуры, триггеры и представления;
- дополнительные функции (математические, обработки строк, дат и т. п.);
- поддержку аутентификации Microsoft Windows.

Со временем под Transact-SQL стали понимать не только процедурное расширение SQL, но и весь *диалект* стандарта SQL, используемый в SQL Server.

Упростить управление сервером позволяет утилита SQL Server Management Studio, предоставляющая удобный визуальный интерфейс. Однако профессиональное использование SQL Server невозможно без владения языком Transact-SQL.

Все приложения, взаимодействующие с SQL Server, независимо от их реализации и пользовательского интерфейса, отправляют серверу инструкции

Transact-SQL. Поэтому предпочтительнее использовать непосредственно операторы Transact-SQL и скрипты (сценарии), включающие эти операторы.

В новых версиях SQL Server графические утилиты могут измениться, тогда как операторы Transact-SQL существенных изменений не претерпят. Язык Transact-SQL необходимо знать и для тестирования производительности сервера. Администратор должен уметь выполнять запросы и анализировать скорость их работы.

В составе Transact-SQL выделяют следующие части:

- язык определения данных (DDL – Data Definition Language);
- язык манипулирования данными (DML – Data Manipulation Language);
- язык управления доступом к данным (DCL – Data Control Language);
- язык управления транзакциями (TCL – Transaction Control Language);
- язык хранимых процедур и триггеров или процедурное расширение SQL (Stored Procedures and Triggers Language).

DDL применяется для работы с объектами баз данных, с метаданными. Для действий с метаданными используются следующие группы операторов:

- *CREATE* – операторы, создающие саму базу данных и объекты базы данных (таблицы, пользовательские типы данных, индексы, хранимые процедуры, триггеры, роли и др.);
- *DROP* – операторы, удаляющие ранее созданные объекты базы данных, а также и саму базу данных;
- *ALTER* – операторы, позволяющие изменить существующие в базе данных объекты и её характеристики.

DML используется для работы с собственно данными, позволяет создавать, изменять и удалять данные. В DML есть и некоторые другие операторы.

Для работы с данными используются четыре основных оператора:

- *INSERT* – добавление данных;
- *UPDATE* – изменение существующих данных;
- *DELETE* – удаление данных;
- *SELECT* – выборка (поиск) данных.

DCL содержит операторы, назначающие, отменяющие и удаляющие полномочия доступа к объектам базы данных для пользователей и ролей:

- *GRANT* – предоставление полномочий доступа к защищаемому объекту;
- *DENY* – отмена полномочия;
- *REVOKE* – удаление полномочия.

TCL включает операторы, осуществляющие запуск, подтверждение, откат или создание точки сохранения транзакции:

- *BEGIN TRANSACTION* – старт транзакции;

- *COMMIT TRANSACTION* и *COMMIT WORK* – фиксация транзакции;
- *ROLLBACK TRANSACTION* и *ROLLBACK WORK* – откат транзакции;
- *SAVE TRANSACTION* – создание точки сохранения.

Язык хранимых процедур и триггеров содержит операторы, обеспечивающие процедурные средства обработки данных.

Синтаксические определения языка SQL задают с помощью специальной металингвистической символики – Бэкуса – Науэра формул (БНФ).

Прописные буквы используются для записи зарезервированных слов и должны указываться в операторах так, как это показано в шаблоне. Строчные буквы употребляются для записи слов, определяемых пользователем.

Применяемые в нотации БНФ символы и их обозначения показаны в табл. 1:

Таблица 1

Символ	Обозначение
::=	Равно по определению
|	Необходимость выбора одного из нескольких приведенных значений
< … >	Описанная с помощью метаязыка структура языка
{ … }	Обязательный выбор некоторой конструкции из списка
[…]	Необязательный выбор некоторой конструкции из списка
[, … n]	Необязательная возможность повторения некоторой конструкции из списка от нуля до нескольких раз

Далее рассмотрим основы архитектуры сервера SQL SERVER.

2. АРХИТЕКТУРА SQL SERVER

Экземпляры сервера. Основным компонентом SQL Server является Database Engine – ядро СУБД, главная служба хранения, обработки и обеспечения безопасности данных, репликации, полнотекстового поиска, средств управления реляционными и XML-данными. На одном компьютере можно инсталлировать и использовать несколько *экземпляров SQL Server* (Database Engine), в т. ч. разных версий.

Экземпляр – сервер баз данных, который не разделяет использование системных и пользовательских баз данных с другими экземплярами (серверами), выполняющимися на этом же компьютере. *Текущий экземпляр сервера* – сервер

БД, с которым выполняется соединение в одной из клиентских программ (утилита командной строки *sqlcmd* или программа Management Studio).

Каждый экземпляр полностью независим от других с точки зрения безопасности и управляемых данных. При этом экземпляры совместно используют физические ресурсы сервера (центральный процессор, оперативную и дисковую память).

Один из установленных экземпляров может быть задан как экземпляр *по умолчанию (default)*, остальные должны быть *именованными (named) экземплярами*. Тип экземпляра задается в процессе установки, и изменить выбор нельзя. Можно сконфигурировать экземпляры так, что экземпляра по умолчанию вообще не будет.

При запуске и останове экземпляра используется его имя. Имя компьютера задает только имя экземпляра по умолчанию. Для соединения с именованным экземпляром следует указать имя компьютера (IP-адрес), за которым следует обратный слэш и имя экземпляра SQL Server.

Компоненты SQL Server, используемые всеми экземплярами:

- программная группа SQL Server;
- сервер Analysis Services;
- библиотеки разработки.

Существование одной программной группы SQL Server означает, что есть одна копия каждой утилиты (представлена значком в программной группе). Работа каждой утилиты для всех экземпляров конфигурируется на уровне компьютера.

Внимание! При инсталляции версии SQL Server Express устанавливается именованный экземпляр с именем *SQLEXPRESS*.

Если при инсталляции SQL Server не был задан автоматический запуск нужного экземпляра сервера при загрузке ОС или его выполнение было остановлено, то экземпляр следует запустить вручную одним из следующих способов:

- из командной строки (или в программе PowerShell);
- с помощью программы Configuration Manager;
- с помощью программы Management Studio.

Внимание! Если не удается установить соединение с сервером, в первую очередь проверьте, запущен ли экземпляр сервера.

Запуск экземпляра SQL Server Express из командной строки:

Пуск / Все программы / Стандартные / Командная строка / ПКМ – Запуск от имени администратора: net start "SQL Server (SQLEXPRESS)".

Останов экземпляра SQL Server Express из командной строки:

Пуск / Все программы / Стандартные / Командная строка / ПКМ – Запуск от имени администратора: net start "SQL Server (SQLEXPRESS)".

Запуск и останов экземпляра сервера в программе Configuration Manager:

Пуск / Все программы / Microsoft SQL Server 2012 / Средства настройки / Диспетчер конфигурации SQL Server.

Способы запуска экземпляра сервера:

- в главном окне программы в левой панели выбрать *Службы SQL Server*, в правой панели щелкнуть ПКМ по имени сервера и в контекстном меню выбрать команду *Запустить*;
- щелкнуть по имени сервера ПКМ и в контекстном меню выбрать элемент *Свойства*; в диалоговом окне свойств выбрать команду *Пуск*.

Способы останова экземпляра сервера:

- в главном окне программы в левой панели выбрать *Службы SQL Server*, в правой панели щелкнуть ПКМ по имени сервера и в контекстном меню выбрать команду *Остановить*;
- щелкнуть по имени сервера ПКМ и в контекстном меню выбрать элемент *Свойства*; в диалоговом окне свойств выбрать команду *Стоп*.

Информацию о запуске и останове сервера с помощью программы Management Studio можно найти в работе [2].

Пользователь подключается к экземпляру SQL Server, пройдя *аутентификацию* (проверку). Для этого администратор баз данных должен создать для него *регистрацию (login)*. Варианты аутентификации:

- *аутентификация Windows* – регистрация связывается с именем пользователя и его паролем в ОС Windows (этот вариант рекомендован Microsoft);
- *смешанный режим* – при подключении пользователь предъявляет имя регистрации и пароль. Администратор должен превратить его регистрацию в *Database User* (пользователя базы данных) для каждой БД.

Базы данных SQL Server. Каждый экземпляр SQL Server может содержать несколько баз данных, представляющих собой контейнеры объектов (таблиц, представлений, хранимых процедур и т. д.). Базы данных могут быть системными или пользовательскими.

Особенности *системных баз данных*:

- хранят системные данные и обслуживают внутренние задачи; создаются автоматически при инсталляции SQL Server и изменяются системой;

- профессиональные программисты вносят нужные изменения в базу данных *model*, которая используется при создании пользовательских баз данных.

Особенности *пользовательских баз данных*:

- создаются, изменяются и удаляются пользователями; в одном экземпляре сервера можно создать необходимое количество пользовательских баз данных;
- содержат данные для решения задач предметной области и другие объекты – хранимые процедуры, пользовательские типы данных, триггеры, функции и т. д.;
- характеристики пользовательской базы данных можно просматривать и изменять с помощью системных средств.

Любая база данных в SQL Server состоит:

- из файла (файлов) данных;
- файла (файлов) журнала транзакций.

Каждому файлу (данных или журнала транзакций) присваивается:

- имя, под которым файл известен ОС;
- логическое имя, задаваемое явно при создании базы данных и по которому к файлу можно обращаться в среде SQL Server.

К базе данных можно обратиться по ее *логическому имени*. Есть также *идентификатор базы данных* – целочисленное значение, присваиваемое системой. Функция *db_id( )* по имени базы данных находит ее идентификатор, функция *db_name( )* по идентификатору находит имя базы данных.

Системные базы данных:

- *master* – главная системная. Хранит метаданные экземпляра сервера, его конфигурацию, сведения о базах данных экземпляра, установочную информацию;
- *model* – шаблон для новых баз данных, которые создаются как копии *model*. Если нужно, чтобы в новых базах данных появлялись некоторые объекты или были настроены какие-то свойства, то следует создать такие объекты и настроить свойства в *model*. В существующие базы данных эти изменения внесены не будут;
- *tempdb* – хранит временные данные (временные таблицы, область сортировки, сведения для контроля версий строк). Уничтожается и создается как копия *model* при каждом запуске экземпляра. Объекты тестирования, которые не должны оставаться, создаются в *tempdb*;
- *msdb* – хранит данные службы SQL Server Agent, отвечающей за автоматизацию (задания, расписания, оповещения) и репликацию. Хранит данные, касающиеся средств Database Mail и Service Broker;

- *resource* – скрытая системная база данных, хранящая системные хранимые процедуры, представления, функции, доступные из любой пользовательской и системной базы данных. Не видна в списке, напрямую к ней обратиться невозможно. Облегчает внесение изменений при установке пакетов исправлений и обновлений.

Файлы данных хранят данные объектов, файлы журнала транзакций – информацию для поддержки транзакций. Файлы данных имеют страничную организацию (размер страницы – 8 Кбайт). Запись может осуществляться параллельно в несколько файлов данных, но в каждый момент времени – в один файл журнала.

При создании базы данных можно задать для каждого файла имя, местоположение, начальный и максимальный размеры, размеры автоприращения.

Файлы данных объединены в *файловые группы*. Данные распределены между файлами, принадлежащими файловой группе. Каждая база данных имеет основную файловую группу *PRIMARY*, содержащую основной файл данных (*mdf – MasterDataFile*) и системный каталог.

В эту группу можно вставить дополнительные файлы данных (*ndf – NotMasterDataFile*). Начальный размер первичного файла – не меньше 3 Мбайт. Пользовательские файловые группы содержат только дополнительные файлы данных. Можно пометить файловую группу как группу по умолчанию. Если в операторе создания объекта явно не задана файловая группа, объект создается в файловой группе по умолчанию.

Файлы журнала имеют расширение *ldf* (*LogDataFile*). Журнал транзакций может состоять из нескольких файлов, новый файл журнала добавляют при заполнении диска. Файлы журнала транзакций не объединяются в файловые группы. Размер файла журнала транзакций – не менее 512 Кбайт.

Основные характеристики базы данных:

- владелец базы данных;
- порядок сортировки;
- возможность изменения данных базы;
- состояние базы данных.

Владелец базы данных (Owner) – имеет все полномочия, может менять характеристики базы данных, удалять ее, вносить изменения в данные и метаданные. Владельцем становится создатель базы данных. Изменение владельца пользовательской базы данных выполняется оператором:

EXECUTE sp_changedbowner '<имя нового владельца>'

Имя нового владельца должно быть уже описано в системе. Получить список существующих в системе пользователей можно с помощью оператора:

EXEC sp_helplogins

Порядок сортировки (Collation) для базы данных – определяет допустимый набор символов в типах *CHAR, VARCHAR* и правила упорядочения строковых данных. Определяет способ сортировки (по внутреннему коду или в алфавитном порядке), в каком порядке будут размещаться строчные и прописные буквы, как распределяются знаки препинания и специальные символы.

Для элемента данных можно указать порядок сортировки, отличный от порядка сортировки всей базы.

Возможность изменения данных – база данных может быть в состоянии только для чтения (*read_only*) или в состоянии чтения-записи (*read_write*).

Состояние базы данных (Database State) – в каждый момент времени любая база данных находится в одном конкретном состоянии:

- *ONLINE* – база в доступном состоянии (в оперативном режиме). Можно выполнять любые действия по изменению данных и метаданных. В этом состоянии средствами ОС нельзя удалить или скопировать файлы базы на другие устройства.
- *OFFLINE* – база в недоступном состоянии (в автономном режиме). Действия с объектами базы невозможны. Средствами ОС можно удалить файлы базы или скопировать их на другой носитель.
- *RESTORING* – база недоступна. В это состояние она переводится, когда выполняется восстановление файлов данных из резервной копии.
- *RECOVERING* – база недоступна, она в процессе восстановления. После восстановления база автоматически будет переведена в оперативное состояние.
- *RECOVERY_PENDING* – база недоступна, в процессе восстановления произошла ошибка, требующая вмешательства пользователя. После исправления ошибки пользователь сам должен перевести базу в состояние *ONLINE*.
- *SUSPECT* – база недоступна. Она помечена как подозрительная и может быть поврежденной. Пользователь должен устранить ошибки.
- *EMERGENCY* – база повреждена и находится в состоянии только для чтения (*read_only*). Состояние используется для диагностики и при попытках скопировать неповрежденные данные.

Файлы данных также могут находиться в различных состояниях. Состояние файла поддерживается независимо от состояния самой БД:

- *ONLINE* – файл доступен (в оперативном режиме), с его данными можно выполнять любые действия.
- *OFFLINE* – файл недоступен (в автономном режиме). Пользователь переводит файл в это состояние, если он поврежден и требует восстановления. После восстановления файла пользователь должен перевести его в состояние *ONLINE*.

- *RESTORING* – файл в процессе восстановления. Другие действия с данными в этом файле невозможны. После завершения восстановления файл автоматически переводится системой в состояние *ONLINE.*

- *RECOVERY_PENDING* – файл автоматически переводится в это состояние, если при его восстановлении произошла ошибка. Восстановление файла отложено, пользователь должен устранить ошибку. После этого пользователь вручную переводит файл в оперативное состояние.

- *SUSPECT* – в процессе оперативного восстановления файла произошла ошибка. База данных помечается как подозрительная.

- *DEFUNCT* – файл был удален, когда не был в состоянии *ONLINE*.

Даже если при работе с базой отдельные файлы недоступны, то операции с ней возможны, если для выполнения этих операций требуются данные, содержащиеся в файлах, находящихся в оперативном режиме.

Схемы и объекты базы данных. База данных включает *схемы* (*Schema*). Именно схемы содержат *объекты базы данных,* т. е. служат контейнерами объектов (таблиц, представлений, хранимых процедур и пр.).

Физически база данных представлена в файлах данных, а логически состоит из произвольного количества схем. Каждый объект базы данных принадлежит только одной схеме. Если при создании объекта базы данных не указывается явно, какой схеме он принадлежит, то он помещается в схему по умолчанию (обычно *dbo*).

Схему можно рассматривать как *пространство имен (Name space).* Имена объектов одного типа в схеме должны быть уникальными.

Причины объединения объектов в схемы:

- повышается уровень безопасности данных. На уровне схемы можно управлять правами доступа, предоставляя права на использование объектов только указанным группам пользователей – *принципалам*;

- схема позволяет упаковать группы объектов, что облегчает манипулирование привилегиями;

- будучи пространством имен, схема используется как префикс в имени объекта (*Sales.Orders* – таблица *Orders* в схеме *Sales*).

Если при ссылке на объект имя схемы не указано, запускается процесс разрешения имени схемы, включающий проверку существования объекта в пользовательской схеме по умолчанию.

При его отсутствии – проверку существования объекта в схеме *dbo (Database Owner).* Схема *dbo* создается автоматически в каждой базе данных и используется как схема по умолчанию.

Рекомендуется всегда использовать составные имена (разрешение имени объекта требует расходов, и если в разных схемах есть одноименные объекты, можно получить не тот объект).

При создании базы данных в ней автоматически формируется не менее десяти схем. Схемы можно создавать, модифицировать и удалять, используя:

- операторы Transact-SQL;
- диалоговые средства Management Studio.

Изменение схемы заключается в перемещении объектов из одной схемы в другую. Удалять можно только схему, не содержащую объектов.

3. ВВОД И ПЕРЕДАЧА СЕРВЕРУ ОПЕРАТОРОВ TRANSACT-SQL

Программа Management Studio. Ввод и передача серверу операторов Transact-SQL реализуется с помощью:

- универсальной утилиты командной строки *sqlcmd*;
- окна ввода запросов программы Management Studio.

Программа Management Studio – мощное и удобное средство работы с базами данных в SQL Server. Все объекты базы данных рассматриваются как образующий сценарий набор операторов DDL (создать, модифицировать, удалить). Создаваемый программный код (хранимая процедура, триггер) – это *запрос*, который можно сохранить в виде текстового файла с расширением *sql* или выполнить.

Чтобы начать сеанс работы, необходимо запустить среду Management Studio и установить соединение (*Соединить*) с сервером (рис. 1).

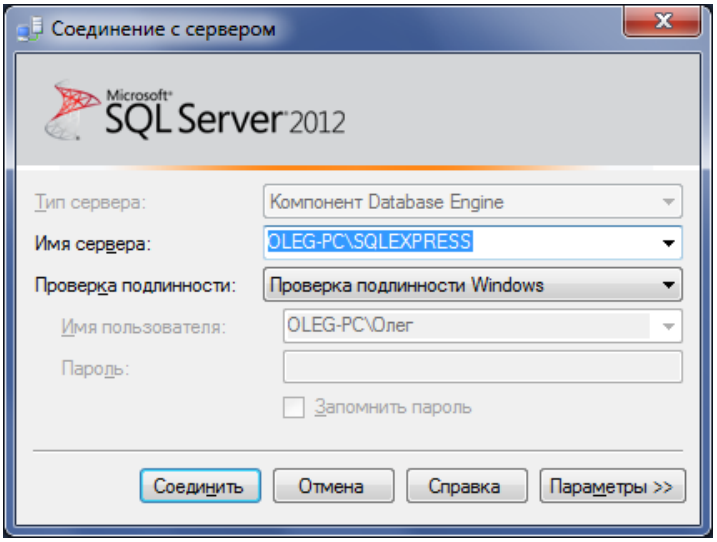


Рис. 1. Установление соединения с сервером

В окне *Соединение с сервером* нужно выбрать тип сервера, имя сервера (имя одного из установленных экземпляров сервера, если существует несколько экземпляров), вид аутентификации и ввести имя пользователя.

Если на компьютере установлен один экземпляр сервера, то в этом окне, ничего не изменяя и не выбирая, следует щелкнуть по кнопке *Соединить*. Произойдет соединение с текущим экземпляром Database Engine, и появится главное окно Management Studio (рис. 2).

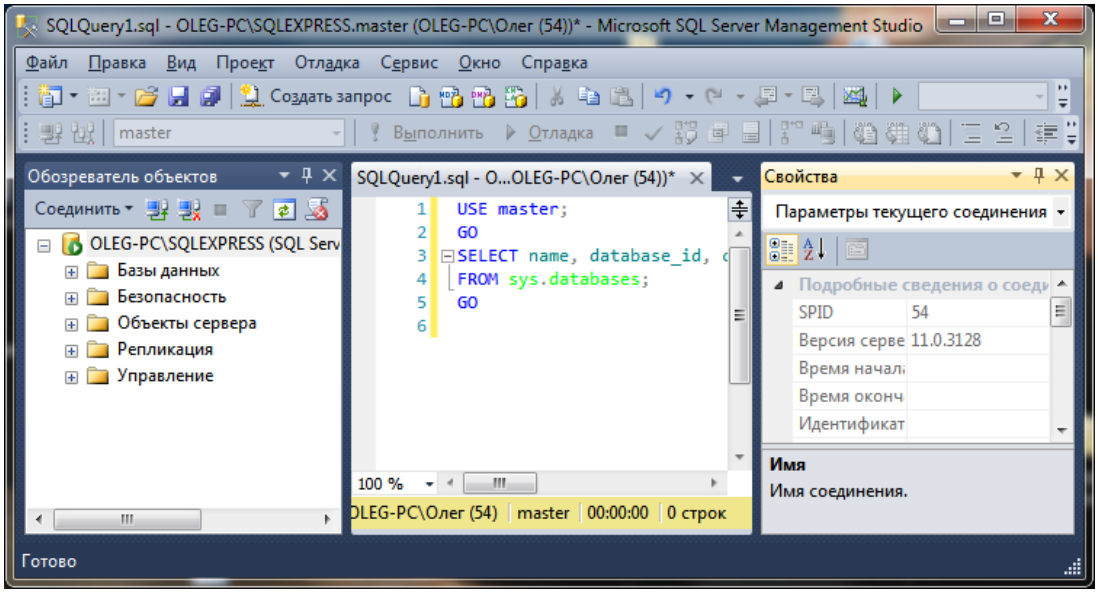


Рис. 2. Главное окно Management Studio

Если на компьютере установлено несколько экземпляров сервера, то из раскрывающегося списка *Имя сервера* следует выбрать соответствующий сервер.

В верхней части окна располагается главное меню, панель инструментов с кнопками быстрого доступа к некоторым функциям главного меню. В левой части находится *Обозреватель объектов*. В нем в виде дерева представлены все объекты экземпляра сервера. Управлять объектами можно, наведя на них курсор мыши и щелкнув ПКМ. Контекстное меню покажет допустимые действия. Если окно обозревателя объектов не отображается, следует выбрать в меню *Вид / Обозреватель объектов* или нажать клавишу <*F8*>.

Чтобы выполнить в Management Studio пакеты операторов Transact-SQL, необходимо вызвать окно выполнения запросов. Для этого на панели инструментов нужно щелкнуть по кнопке *Создать запрос*.

Для вызова этого окна можно выбрать в меню *Файл / Создать / Запрос в текущем соединении*. Наконец, для вызова окна запросов можно просто нажать клавиши <*Ctrl*>+<*N*>. В центральной части главного окна программы появится пустое окно запросов, где можно вводить, изменять и выполнять операторы Transact-SQL. Эти окно открыто на рис. 2.

Можно создать произвольное количество окон запросов, в которых могут содержаться любые операторы Transact-SQL. Эти окна создаются в виде вкладок. Переключение между вкладками выполняется щелчком мыши по их заголовкам.

Окно запросов позволяет решать и обратную задачу – в нем для каждого объекта базы данных отображается набор скриптов: создать объект, изменить объект, удалить объект, выполнить объект (для процедуры или функции) и т. д.

При вводе операторов Transact-SQL программа Management Studio выделяет цветом ключевые слова, константы, подчеркивает неправильные выражения красной волнистой линией. Хотя бывают случаи, когда красной волнистой линией подчеркнуты тексты, созданные правильно.

Для выполнения введенных операторов следует щелкнуть мышью по кнопке *Выполнить* на панели инструментов (либо выбрать в меню *Запрос / Выполнить* либо нажать клавишу *<F5>* или комбинацию клавиш *<Ctrl>+<E>*). Результат выполнения операторов Transact-SQL появится в нижней части окна запросов.

В текстовом редакторе (например, в программе *Блокнот*) можно создать файл скрипта, содержащий операторы Transact-SQL. Таким файлам принято давать расширение *sql*, хотя это не обязательно. Чтобы загрузить файл в окно запросов, нужно выбрать в меню *Файл / Открыть / Файл*. Можно нажать клавиши *<Ctrl>+<O>* или щелкнуть мышью по соответствующей кнопке панели инструментов.

Для сохранения скрипта следует выбрать в меню *Файл / Сохранить (имя скрипта)*, нажать *<Ctrl>+<S>* или щелкнуть мышью по кнопке сохранения.

Если в окне множество операторов, а выполнить нужно только некоторые из них, следует выделить требуемую группу операторов и запустить только их.

Некоторые замечания по использованию Management Studio:

- для выполнения оператора или группы операторов в окне запросов следует выделить эту группу и нажать *<F5>* (*<Ctrl>+<E>*);
- для вывода результатов в текстовом виде (как в утилите *sqlcmd*) – выбрать в меню *Запрос / Отправить результаты / В виде текста* (*<Ctrl>+<T>*);
- чтобы вернуться к выводу результатов в табличном виде – выбрать в меню *Запрос / Отправить результаты / В виде сетки* (*<Ctrl>+<D>*);
- для доступа к настройкам – выбрать в меню *Запрос / Параметры запроса…* (щелкнуть мышью по кнопке *Параметры запроса* на панели инструментов);
- в окне *Параметры запроса* задаются требуемые характеристики (например, на вкладке *Результаты* можно задать включение запроса в результат его выполнения, отображение результата выполнения запроса в отдельной таблице и т. д.);

- для отображения номеров строк в окне запроса – выбрать в меню *Сервис / Параметры*. В левой части диалогового окна раскрыть элементы *Текстовый редактор / Transact-SQL* и щелкнуть мышью по элементу *Общие*. В правой части окна в разделе *Показывать* установить флажок *Номера строк* и щелкнуть *ОК*.

Получение сведений о системных и пользовательских базах данных. Обеспечивают:

- системные представления;
- системные хранимые процедуры;
- системные функции;
- диалоговые средства Management Studio.

Системное представление *sys.databases* позволяет просмотреть список и характеристики пользовательских и системных баз данных в текущем экземпляре.

Системное представление *sys.master_files* позволяет получить список всех баз данных, файлов, входящих в состав каждой БД, а также характеристики файлов.

Системное представление *sys.database_files* позволяет получить список файлов и их характеристик текущей базы данных, указанной в операторе *USE*. Характеристики дублируют значения столбцов представления *sys.master_files.*

Системное представление *sys.filegroups* позволяет получить некоторые данные о файловых группах текущей базы данных, указанной в операторе *USE*.

Для обращения к системным представлениям (как и к другим представлениям в базе данных) используется оператор *SELECT*.

Отобразим все базы данных в экземпляре сервера, используя системное представление *sys.databases*:

```
1  USE master;
2  GO
3  SELECT name,  database_id,  create_date,  collation_name
4  FROM  sys.databases;
5  GO
```

100 %

Результаты | Сообщения

	name	database_id	create_date	collation_name
1	master	1	2003-04-08 09:13:36.390	Cyrillic_General_CI_AS
2	tempdb	2	2014-12-19 06:33:03.347	Cyrillic_General_CI_AS
3	model	3	2003-04-08 09:13:36.390	Cyrillic_General_CI_AS
4	msdb	4	2012-02-10 21:02:17.770	Cyrillic_General_CI_AS
5	ReportServer$SQLEXPRESS	5	2014-12-11 09:08:12.167	Latin1_General_CI_AS_KS_WS
6	ReportServer$SQLEXPRESSTempDB	6	2014-12-11 09:08:12.980	NULL
7	Book	7	2014-12-15 07:40:56.683	Cyrillic_General_CI_AS

Отобразим сведения об одной базе данных, используя системное представление *sys.databases*:

```
1  USE master;
2  GO
3  SELECT  name ,  database_id ,  create_date ,   collation_name
4  FROM   sys.databases
5  WHERE   name = 'tempdb';
6  GO
```

100 %

Результаты | Сообщения

	name	database_id	create_date	collation_name
1	tempdb	2	2014-12-19 06:33:03.347	Cyrillic_General_CI_AS

Отобразим сведения о базах данных и их файлах с помощью системного представления *sys.master_files* (полный вывод оборван):

```
1  USE master;
2  GO
3  SELECT DB_NAME(database_id) AS 'DB Name',
4         type_desc  AS 'Descr',
5         name  AS  'File Name',
6         state_desc  AS 'State',
7         size  AS 'Size',
8         max_size  AS 'Max Size',
9         growth AS 'Growth'
10 FROM    sys.master_files
11 ORDER   BY   'DB Name';
12 GO
```

100 %

Результаты | Сообщения

	DB Name	Descr	File Name	State	Size	Max Size	Growth
1	Book	ROWS	Book_dat	ONLINE	640	-1	128
2	Book	LOG	Book_log	ONLINE	128	3840	128
3	master	ROWS	master	ONLINE	624	-1	10
4	master	LOG	mastlog	ONLINE	224	-1	10

4. ПРОГРАММИРОВАНИЕ НА TRANSACT-SQL

Язык SQL стандарта ANSI позволяет создавать и модифицировать базы данных, а также читать и изменять информацию в них. Программирование требует дополнительных возможностей.

Язык Procedural SQL, используемый для написания хранимых процедур, пользовательских функций и триггеров, является полноценным языком программирования, позволяющим выполнить любые действия по обработке данных – локальных или хранящихся в базе данных.

В языке допустимы любые действия с базой данных, кроме создания и изменения метаданных. Можно использовать операторы *INSERT, UPDATE, DELETE* и *SELECT*, можно отправлять сообщения (events), вызывать пользовательские исключения (exception). Нельзя устанавливать и разрывать связь с базой данных.

При создании процедур и триггеров выделяются заголовок и тело. Тело является блоком операторов. В теле процедур и триггеров могут присутствовать операторы присваивания, оператор ветвления и операторы циклов. Их можно использовать в обычных пакетах в командной строке и в Management Studio.

Пакеты. Пакет – последовательность операторов Transact-SQL, отправляемых клиентским приложением серверу для выполнения как единого целого. Пакет проходит синтаксический анализ, разрешение имен и оптимизацию как единый блок. В сравнении с группой одиночных операторов дает выигрыш производительности.

Внимание! Не путайте пакеты и транзакции.

Транзакция – это неделимый рабочий блок. Пакет может содержать несколько транзакций, и транзакция может быть представлена частями в разных пакетах. При отмене транзакции SQL Server удаляет те действия, которые были выполнены с момента запуска транзакции, независимо от того, где находится начало пакета.

Внимание! Не путайте пакеты и блоки.

Блок – программная единица из операторов Transact-SQL, выделенная словами *BEGIN* и *END*. Окончание пакета обозначает клиентская команда *GO*, которую предоставляют утилиты Management Studio и *sqlcmd*.

Команда *GO* не является оператором языка Transact-SQL и не может завершаться точкой с запятой, так как это служебная команда утилит *sqlcmd* и Management Studio. Выполнение пакета операторов Transact-SQL начинается после ввода *GO*.

Синтаксис команды:

GO [<количество повторений>]

Если указано количество повторений, то пакет выполняется заданное число раз. Если не указано, то пакет выполняется один раз. Не все операторы могут комбинироваться с другими операторами в одном пакете.

Пакет подвергается синтаксическому анализу как единый блок. Если синтаксическая проверка успешна, SQL Server пытается выполнить пакет, иначе весь пакет не передается SQL Server для выполнения.

Переменные локальны по отношению к пакету, в котором определены. Если сослаться на переменную, определенную в другом пакете, то выйдет ошибка.

```
1  DECLARE @i AS INT; SET @i = 10;
2  -- Удачная попытка
3  PRINT @i;
4  GO
5  -- Неудачная попытка
6  PRINT @i;
```

100 %

Сообщения

```
10
Сообщение 137, уровень 15, состояние 2, строка 2
Необходимо объявить скалярную переменную "@i".
```

Блок операторов *BEGIN … END*. Позволяет создавать программную единицу из одного или более операторов Transact-SQL, как показано ниже:

BEGIN

оператор 1

оператор 2

…

END

Все действия описываются в блоке операторов между необязательными ключевыми словами *BEGIN* и *END*. Программы могут содержать произвольное количество блоков как независимых, так и вложенных друг в друга.

Комментарии. В любое место программного кода можно поместить комментарий. Комментарии бывают одно- и многострочными:

/ Многострочный*

*комментарий */*

-- Однострочный комментарий

Переменные. SQL Server поддерживает локальные и глобальные переменные. Локальные переменные существуют только в пределах сеанса, во время которого созданы, глобальные переменные используются для получения информации о сервере в целом.

Имена *локальных переменных* должны начинаться с символа @. Они позволяют временно хранить значения данных для использования в том пакете, где были объявлены. Каждая локальная переменная должна быть определена с использованием оператора *DECLARE* (см. выше). В одном операторе можно объявить произвольное количество локальных переменных, разделяя объявле-

ния запятыми. При объявлении переменной ей можно задать начальное значение.

Для присваивания значения локальной переменной используется оператор

SET <имя переменной> = <выражение>;

Поддерживаются объявление и инициализация переменных в одном операторе:

DECLARE <имя переменной> [AS] <тип данных> [= <значение>];

Значение, присваиваемое скалярной переменной, может быть скалярным подзапросом. Объявляется переменная *@username*, и ей присваивается результат скалярного подзапроса, возвращающего имя и фамилию клиента с кодом 1:

```
1  USE Book;
2  GO
3  DECLARE @username AS VARCHAR(50);
4  SET @username = (SELECT u_name+' '+u_surname
5                   FROM Users
6                   WHERE user_ID=1);
7  SELECT @username AS Клиент;
```

100 %

Результаты | Сообщения

	Клиент
1	Александр Иванов

Оператор *SET* может обрабатывать только одну переменную в каждый момент времени, поэтому для присвоения значений нескольким атрибутам необходимы множественные операторы *SET*. SQL Server поддерживает нестандартное *SELECT*-присваивание, позволяющее в одном операторе запросить данные и присвоить разным переменным многочисленные значения, полученные из одной строки.

```
1  DECLARE @firstname AS VARCHAR(20), @lastname AS VARCHAR(20);
2  SELECT @firstname = u_name, @lastname = u_surname
3   FROM Users
4   WHERE user_ID = 1;
5   SELECT @firstname AS Имя, @lastname As Фамилия;
```

100 %

Результаты | Сообщения

	Имя	Фамилия
1	Александр	Иванов

SELECT-присваивание предсказуемо, если извлечена одна строка. Если запрос извлек несколько строк, программный код не завершается аварийно. Присваивания выполняются для каждой выбранной строки, значения переменных

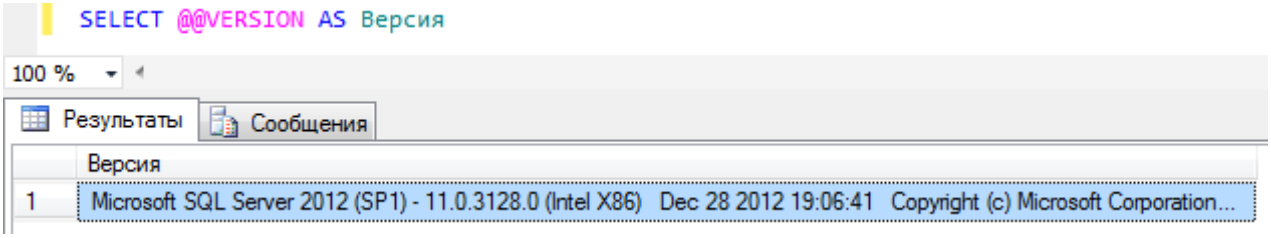
перезаписываются. Когда *SELECT*-присваивание заканчивается, значения переменных равны значениям строки, к которой SQL Server обратился последней.

Оператор *SET* безопаснее *SELECT*-присваивания, так как требует применения скалярного подзапроса для извлечения данных из таблицы. Скалярный запрос во время выполнения аварийно завершается, если возвращает более одного значения.

Глобальные переменные используются сервером для отслеживания информации уровня сервера или базы данных, относящейся к конкретному сеансу. Глобальные переменные нельзя явно присваивать или объявлять. Некоторые глобальные переменные представлены в табл. 2:

Таблица 2

Глобальная переменная	Описание
@@ROWCOUNT	Количество записей, обработанных предыдущей командой
@@ERROR	Код ошибки для последней команды SQL
@@TRANCOUNT	Уровень вложенности транзакции
@@SERVERNAME	Имя локального сервера
@@VERSION	Номер версии SQL Server, дата и тип процессора
@@IDENTITY	Последнее значение счетчика, используемое в операции вставки
@@NESTLEVEL	Количество уровней вложенности для хранимой процедуры или триггера
@@FETCH_STATUS	Статус предыдущей команды выборки для курсора
@@ SPID	Идентификатор текущего процесса



Оператор *PRINT*. Возвращает определенное пользователем сообщение длиной до 1024 байтов. В Management Studio служебные сообщения выводятся во вкладке *Сообщения (Messages)*. Сообщение представляет собой текст в апострофах.

Ветвление в программе. Осуществляется с помощью оператора *IF*:

IF <условие>

{ <оператор> | <блок операторов> }

[ELSE

{ <оператор > | <блок операторов> }];

Первый оператор (блок операторов) выполняется, если логическое выражение, стоящее за словом *IF*, возвращает значение *TRUE*. Второй оператор (блок операторов) в необязательной части *ELSE* будет выполняться, если логическое выражение возвращает *FALSE* или *UNKNOWN*. Если условие содержит оператор *SELECT*, возвращающий значение, то весь этот оператор должен быть заключен в круглые скобки. В следующем примере оператор *IF ... ELSE* проверяет среднюю цену книг:

```
1  USE Book;
2  GO
3  IF (SELECT AVG(b_price) FROM Books) > 300
4  PRINT 'Средняя цена книг больше 300 руб.'
5  ELSE
6  PRINT 'Средняя цена книг меньше 300 руб.'
```

100 %

Сообщения

```
Средняя цена книг больше 300 руб.
```

Transact-SQL использует троичную логику, поэтому если вовлечены значения *NULL*, следует ввести явную проверку значений *NULL* с помощью *IS NULL*.

Если управляемый поток включает более двух вариантов обработки, можно вкладывать элементы *IF ... ELSE* один в другой. Если в ветках *IF* или *ELSE* необходимо выполнить несколько операторов, применяется *блок BEGIN … END*.

Следующий пакет вычисляет среднее значение цены книг и сравнивает это значение с ценой книги с кодом 4. Если цена этой книги меньше вычисленного значения, то цена книги с кодом 4 увеличивается на значение *@extra_price*.

```
1  USE Book;
2  GO
3  DECLARE @avg_price MONEY, @extra_price MONEY;
4  SET @extra_price = 50;
5  SELECT @avg_price = AVG(b_price) FROM Books;
6  IF (SELECT b_price FROM Books WHERE book_ID = 4) < @avg_price
7      BEGIN
8          UPDATE Books SET b_price = b_price + @extra_price
9          WHERE book_ID = 4;
10         PRINT 'Цена книги 4 увеличена на @extra_price'
11     END
12 ELSE
13     PRINT 'Цена книги 4 не изменилась'
```

100 %

Сообщения

```
Цена книги 4 не изменилась
```

Оператор *IF EXISTS* является частным случаем оператора *IF* и позволяет узнать, существуют ли какие-либо экземпляры, определяемые условием. Он заменяет конструкцию *COUNT(*)* > 0 для проверки существования записей. При обнаружении первой совпадающей записи обработка оператора прекращается. В примере определяется, есть ли в таблице *Users* данные о клиенте с фамилией Корнеев:

```
1  USE Book;
2  GO
3  IF EXISTS(SELECT * FROM Users WHERE u_surname = 'Корнеев')
4      PRINT 'Клиент с такой фамилией существует'
5  ELSE
6      PRINT 'Такого клиента не существует'
```

100 %

Сообщения

```
Клиент с такой фамилией существует
```

Ветвление в программах может быть выполнено также при помощи оператора *CASE*, который имеет две формы. Синтаксис первой формы:

CASE <выражение>
WHEN <значение> THEN { <оператор> | <блок операторов> }
[WHEN <значение> THEN { <оператор> | <блок операторов> }] ...
END;

Выражение возвращает некоторое значение. Если в предложении *WHEN* указано именно это значение, то выполняется соответствующий оператор (блок операторов), заданный после *THEN*. Остальные предложения *WHEN* не проверяются.

Синтаксис второй формы:

CASE
WHEN <условие> THEN { <оператор> | <блок операторов> }
[WHEN <условие> THEN { <оператор> | <блок операторов> }] ...
[ELSE { <оператор> | <блок операторов> }]
END;

Здесь предложения *WHEN* не связаны друг с другом по используемым в них выражениям. Если условие имеет значение *TRUE*, то выполняется оператор (блок) после *THEN* и работа оператора *CASE* завершается. Если ни одно условие *WHEN* не возвращает значения *TRUE*, то выполняется оператор (блок) в предложении *ELSE* (если оно есть).

Конструкция *CASE*, в отличие от *IF*, может использоваться в операторе *SELECT*. Пример использования *CASE* (вывод результатов запроса оборван):

```
1  USE Book;
2  GO
3  SELECT book_ID, b_name,
4      CASE b_cat_ID
5          WHEN 1 THEN 'Программирование'
6          WHEN 2 THEN 'Интернет'
7          WHEN 3 THEN 'Базы данных'
8          WHEN 4 THEN 'Сети'
9          WHEN 5 THEN 'Мультимедиа'
10     ELSE 'Неизвестно'
11     END AS [Жанр книги],
12     b_price,
13     CASE
14         WHEN b_price < 200 THEN 'Дешевая'
15         WHEN b_price BETWEEN 200 AND 400 THEN 'Средняя'
16         WHEN b_price > 400 THEN 'Дорогая'
17     END AS [Категория книги]
18     FROM Books
```

100 %

Результаты | Сообщения

	book_ID	b_name	Жанр книги	b_price	Категория книги
1	1	Java. Руководство для начинающих	Программирование	636,00	Дорогая
2	2	Microsoft Visual Studio 2010	Программирование	472,00	Дорогая

Организация циклов. Для организации циклов используется оператор *WHILE*:

WHILE <условие>

{ <оператор> | <блок операторов> } ;

Оператор *WHILE* многократно выполняет оператор (блок операторов), пока условие, заданное после ключевого слова *WHILE*, равно *TRUE*. Когда условие становится равным *FALSE* или *UNKNOWN*, цикл завершается. В теле цикла должны быть операторы, влияющие на условие цикла.

Внутри цикла могут присутствовать операторы *BREAK* и *CONTINUE*. Оператор *BREAK* означает завершение цикла и переход на оператор, следующий за конечным словом *END*. Этот оператор не означает завершения выполнения хранимой процедуры или триггера. Оператор *CONTINUE* осуществляет переход на начало цикла, а операторы после ключевого слова *CONTINUE* игнорируются.

Следующий пример показывает использование команды *WHILE* вместе с командами *BREAK* и *CONTINUE*:

```
1  USE Book;
2  GO
3  WHILE (SELECT SUM(b_price) FROM Books) < 10000
4      BEGIN
5          UPDATE Books SET b_price = b_price * 1.1;
6          IF (SELECT MAX(b_price) FROM Books) > 1000
7              BREAK
8          ELSE
9              CONTINUE
10     END;
```

Стоимость каждой книги будет увеличиваться на 10 %, пока общая сумма не превысит 10 000 руб. При этом цикл будет остановлен, если цена одной из книг превысит 1 000 руб.

Оператор *GOTO*. Операторы внутри тела процедуры или триггера могут быть снабжены метками. Метка – идентификатор, после которого следует двоеточие:

<метка>: <оператор>;

Метки используются для того, чтобы именовать операторы, на которые может выполняться переход при использовании оператора *GOTO*:

GOTO <метка>;

Использование оператора *GOTO* позволяет по-другому организовать многократно повторяющиеся вычисления. При грамотном использовании операторов ветвления и цикла потребности в операторе перехода нет.

Оператор *WAITFOR*. Переводит запрос в состояние ожидания на некоторое время (если используется опция *DELAY*) или до наступления заданного времени (если указана опция *TIME*). Параметр *DELAY* заставляет сделать паузу заданной длины (максимальное значение 24 часа).

Если указан параметр *TIME*, процесс приостанавливается до наступления заданного времени. В обоих случаях время задается в формате *hh:mi:ss* (дату задать нельзя). Примеры использования оператора *WAITFOR*:

```
-- Пауза до 13:00
WAITFOR TIME '13:00:00'
```

100 %

Сообщения

Выполнение команд успешно завершено.

```
-- Выводить текущий список пользователей каждые 10 секунд
WHILE 1<2
    BEGIN
        WAITFOR DELAY '00:00:10'
        EXEC sp_who
    END
```

100 %

Результаты | Сообщения

	spid	ecid	status	loginame	hostname	blk	dbname	cmd	request_id
1	1	0	background	sa		0	NULL	LOG WRITER	0
2	2	0	background	sa		0	master	SIGNAL HANDLER	0
3	3	0	background	sa		0	NULL	RECOVERY WRITER	0
4	4	0	background	sa		0	NULL	LAZY WRITER	0
5	5	0	background	sa		0	NULL	LOCK MONITOR	0
6	6	0	background	sa		0	NULL	RESOURCE MONITOR	0
7	7	0	background	sa		0	NULL	XE DISPATCHER	0
8	8	0	background	sa		0	NULL	XE TIMER	0

Оператор *RETURN*. Осуществляет выход из хранимой процедуры, пользовательской функции, триггера (завершение выполнения).

RETURN [<целое>];

Хранимая процедура может возвращать целое число – код возврата. Если целое не указано, возвращается 0. Код возврата используют в вызывающей программе для анализа результатов выполнения хранимой процедуры. Пользовательская функция может возвращать значения типов, объявленных при ее описании.

Оператор *RETURN* внутри пакета осуществляет безусловный выход из пакета, после чего выполнится первый оператор, следующий сразу после пакета:

```
1  USE Book;
2  GO
3  IF EXISTS(SELECT * FROM Books WHERE b_cat_ID = 1)
4  BEGIN
5      PRINT 'Все в порядке'
6      RETURN
7  END
8  PRINT 'У вас нет книг по программированию'
```

100 %

Сообщения

Все в порядке

Оператор SET. Обычно используется для установки значения переменной (см. выше). Также с помощью этого оператора задаются некоторые параметры, определяющие реакцию сервера на некоторые условия. Синтаксис оператора:

SET <условие> { ON | OFF | значение }

Например, *SET NOCOUNT ON*.

Параметры настраиваются на уровне сеанса или на уровне хранимой процедуры. Между сеансами они не сохраняются.

Оператор *RAISERROR*. Предназначен для обработки ошибок, возникающих при выполнении SQL-кода. Передает состояния, коды и сообщения ошибок на программном уровне. Можно использовать стандартное сообщение или определить новое сообщение об ошибке непосредственно в операторе.

RAISERROR ({ код_ошибки | символьная_строка } ,

код_серьезности , состояние [, список_аргументов })

[WITH параметр]

Коды ошибок должны быть больше 50 и меньше 2 147 483 647. Незапланированные сообщения автоматически инициируют ошибку с кодом 50 000.

Номер ошибки, определенной пользователем, должен быть больше 50 000 (все номера ошибок, меньше или равные 50 000, резервируются для Database

Engine). Значение кода ошибки сохраняется в глобальной переменной *@@ERROR*.

Следующий пример вызывает незапланированное сообщение об ошибке:

```
IF NOT EXISTS (SELECT * FROM Books)
RAISERROR ('Таблица Books не заполнена. Заполните ее', 16, 1)
```

100 %

Сообщения

```
Сообщение 50000, уровень 16, состояние 1, строка 2
Таблица Books не заполнена. Заполните ее
```

Пример использования функции *@@ERROR*, возвращающей код последней ошибки, происшедшей в текущем соединении:

```
1 IF NOT EXISTS(SELECT * FROM Books)
2  RAISERROR ('Таблица Books не заполнена. Заполните ее', 16, 1);
3 IF @@ERROR <> 0
4      PRINT 'Возникла ошибка при проверке таблицы Books'
5  ELSE
6      PRINT 'Проверка прошла успешно'
7  GO
```

100 %

Сообщения

```
Сообщение 50000, уровень 16, состояние 1, строка 2
Таблица Books не заполнена. Заполните ее
Возникла ошибка при проверке таблицы Books
```

Конструкция *TRY ... CATCH*. Другим способом обработки ошибок являются перехват и обработка исключений с помощью конструкции *TRY … CATCH*.

Исключение – проблема, препятствующая продолжению выполнения программы. При наличии такой проблемы нельзя продолжать обработку, потому что нет информации, необходимой для решения проблемы. Поэтому проблема направляется другой части программы, которая обработает это исключение. Эта конструкция не может быть использована в функциях, определенных пользователем.

Синтаксис конструкции:

BEGIN TRY

{ <оператор> | <блок операторов> }

END TRY

BEGIN CATCH

{ <оператор> | <блок операторов> }

END CATCH;

Если в операторах блока *TRY* возникает ошибка работы с базой данных с кодом серьезности более 10, то управление передается операторам обработки ошибок в блоке *CATCH*. Если ошибок нет, то блок *CATCH* не выполняется. Обработанные в блоке ошибки не передаются вызывающей программе.

Преимущества обработки исключений с помощью *TRY … CATCH*:

- исключения предоставляют ясный способ обнаружения ошибок без использования хаотичного кода;
- исключения предоставляют механизм прямой сигнализации об ошибке, а не при помощи каких-то посторонних эффектов;
- исключения могут быть обнаружены программистом и проверены в процессе компиляции программного кода.

Для получения подробной информации об ошибке используются системные функции, представленные в табл. 3.

Следующий пример демонстрирует использование этих функций.

```
1  USE Book;
2  GO
3  BEGIN TRY
4      SELECT 1/0;
5  END TRY
6  BEGIN CATCH
7      SELECT ERROR_NUMBER() AS 'Number', ERROR_SEVERITY() AS 'Severity',
8             ERROR_STATE() AS 'State', ERROR_PROCEDURE() AS 'Procedure',
9             ERROR_LINE() AS 'Line', ERROR_MESSAGE() AS 'Message',
10            @@ERROR AS '@@ERROR';
11 END CATCH;
12 GO
```

100 %

Результаты | Сообщения

(Отсутствует имя столбца)

	Number	Severity	State	Procedure	Line	Message	@@ERROR
1	8134	16	1	NULL	2	Обнаружена ошибка: деление на ноль.	8134

Каждый блок *TRY ... CATCH* должен быть в одном пакете.

Таблица 3

Имя функции	Описание
ERROR_NUMBER()	Возвращает номер ошибки (целое число), если вызвана из блока *CATCH*. Иначе возвращает *NULL*
ERROR_MESSAGE()	Возвращает полный текст сообщения об ошибке (до 4000 символов), если функция вызвана из блока *CATCH*. Иначе возвращает *NULL*

Окончание табл. 3

Имя функции	Описание
ERROR_LINE()	Возвращает номер строки в триггере или процедуре, где возникла ошибка
ERROR_PROCEDURE()	Возвращает имя хранимой процедуры или триггера, где возникла ошибка. Если ошибка произошла вне процедуры или триггера, то возвращает *NULL*
ERROR_SEVERITY()	Возвращает описание уровня серьезности ошибки (целое число), если вызвана из блока *CATCH*. Иначе возвращает *NULL*
ERROR_STATE()	Возвращает код состояния ошибки (целое число), если функция вызвана из блока *CATCH*. Иначе возвращает *NULL*

В следующем примере, если при вставке записи в таблицу *Books* произойдет ошибка, будет выведено описание этой ошибки.

```
1  USE Book;
2  GO
3  BEGIN TRY
4      INSERT INTO Books VALUES(33, 'Муму', 'Тургенев И.С.', 2008, 50.00, 8, 7);
5  END TRY
6  BEGIN CATCH
7      SELECT ERROR_NUMBER() AS ErrorNumber,
8             ERROR_MESSAGE() AS ErrorMessage;
9  END CATCH;
10 GO
```

100 %

Результаты | Сообщения

	ErrorNumber	ErrorMessage
1	547	Конфликт инструкции INSERT с ограничением FOREIGN KEY "FK_Books_Catalogs". Конфликт произ...

Оператор *THROW*. Выдает исключение и передает управление блоку *CATCH* в конструкции *TRY … CATCH*. Синтаксис оператора:

THROW [<номер ошибки>, <сообщение>, <состояние>];

<номер ошибки> – номер ошибки, определенной пользователем. Находится в диапазоне от 50 000 до 2 147 483 647. Ошибки с номером менее 50 000 являются системными ошибками.

<сообщение> – строка длиной до 2048 символов.

<состояние> – число в диапазоне от 0 до 255.

ПРАКТИЧЕСКАЯ ЧАСТЬ

ПРАКТИЧЕСКАЯ РАБОТА С SQL SERVER

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 1
ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ER-ТЕХНОЛОГИИ

Теоретические сведения

Рассмотрим следующие вопросы:

- проектирование логической модели данных предметной области;
- проектирование физической модели данных предметной области.

Для заданной предметной области необходимо определить состав реляционных таблиц и логические связи между таблицами. Для каждого атрибута следует задать тип и размер данных, ограничения целостности; для каждой таблицы – первичный ключ, потенциальные и внешние ключи.

Практическая работа

Для своей предметной области необходимо:

- построить логическую модель данных в соответствии со стандартом IDEF1X;
- построить физическую модель данных;
- составить отчет.

Пример выполнения задания

Рассмотрим базу данных *Book* интернет-магазина, торгующего литературой по компьютерной тематике [3]. База данных должна поддерживать следующую информацию:

- тематические каталоги, по которым сгруппированы книги;
- предлагаемые книги (название, автор, год, цена, количество на складе);
- зарегистрированные покупатели (имя, отчество, фамилия, телефон, адрес электронной почты, статус);
- покупки, совершенные в магазине (время совершения покупки, число приобретенных экземпляров книги).

Логическая модель данных предметной области в стандарте IDEF1X представлена на рис. 3. Выделены сущности *КАТАЛОГ, КНИГА, КЛИЕНТ, ЗАКАЗ*, между которыми установлены неидентифицирующие связи мощностью один-ко-многим, определяемые спецификой предметной области.

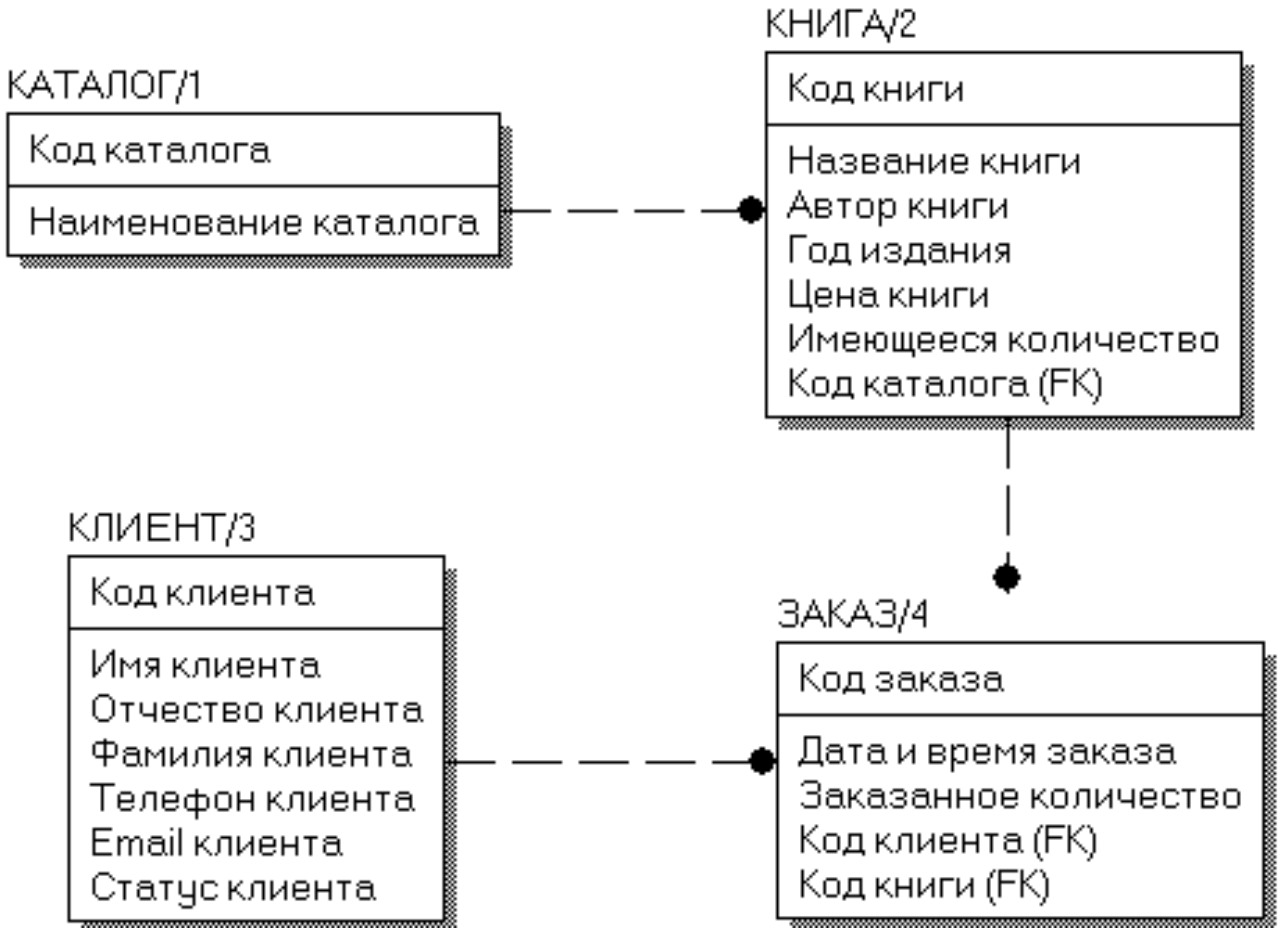


Рис. 3. Логическая модель данных предметной области

Физическая модель данных предметной области в стандарте IDEF1X для целевой СУБД MS SQL Server представлена на рис. 4.

База данных *Book* состоит из четырех таблиц:

- *Catalogs* – список тематических каталогов;
- *Books* – список предлагаемых книг;
- *Users* – список зарегистрированных пользователей магазина;
- *Orders* – список заказов (осуществленных сделок).

Таблица *Catalogs* состоит из двух полей:

- *cat_ID* – уникальный код каталога;
- *cat_name* – имя каталога.

Оба поля должны быть снабжены атрибутом *NOT NULL*, поскольку неопределенное значение для них недопустимо.

Таблица *Books* состоит из семи полей:

- *book_ID* – уникальный код книги;
- *b_name* – название книги;
- *b_author* – автор книги;
- *b_year* – год издания;

- *b_price* – цена книги;
- *b_count* – количество книг на складе;
- *b_cat_ID* – код каталога из таблицы *Catalogs*.

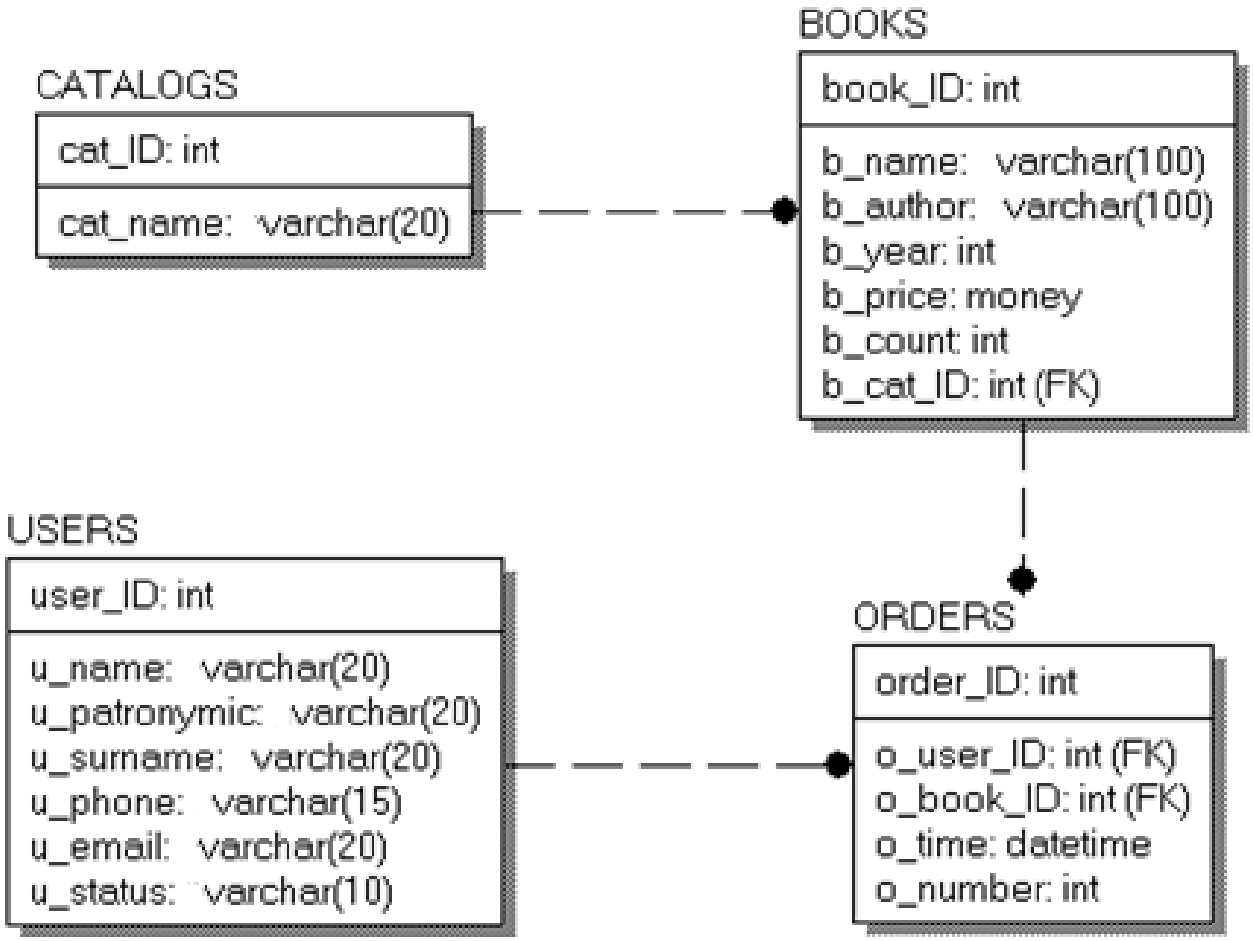


Рис. 4. Физическая модель предметной области

Поля *b_price* и *b_count* могут принимать неопределенное значение *NULL*, поскольку в момент доставки количество книг и их цена могут быть неизвестны.

Поле *b_cat_ID* устанавливает связь между таблицами *Catalogs* и *Books*. Это поле должно быть объявлено как внешний ключ (FK) с правилом каскадного удаления и обновления. Обновление таблицы *Catalogs* вызовет автоматическое обновление таблицы *Books*. Удаление каталога в таблице *Catalogs* приведет к автоматическому удалению всех записей в таблице *Books*, соответствующих каталогу.

Таблица *Users* состоит из семи полей:

- *user_ID* – уникальный код покупателя;
- *u_name* – имя покупателя;
- *u_patronymic* – отчество покупателя;

- *u_surname* – фамилия покупателя;
- *u_phone* – телефон покупателя (если есть);
- *u_email* – e-mail покупателя (если есть);
- *u_status* – статус покупателя.

Статус покупателя может принимать одно из четырех значений:

- *active* – авторизованный покупатель, который может делать покупки через Интернет;
- *passive* – неавторизованный покупатель, который осуществил процедуру регистрации, но не подтвердил ее и пока не может делать покупки;
- *lock* – заблокированный покупатель, который не может делать покупки;
- *gold* – активный покупатель с хорошей кредитной историей, которому предоставляется скидка при следующих покупках в магазине.

Поля *u_phone* и *u_email* могут принимать неопределенные значения *NULL*. Остальные поля должны получить атрибут *NOT NULL*.

Таблица *Orders* включает пять полей:

- *order_ID* – уникальный номер сделки;
- *o_user_ID* – номер пользователя из таблицы *Users*;
- *o_book_ID* – номер товарной позиции из таблицы *Books*;
- *o_time* – дата и время совершения сделки;
- *o_number* – количество приобретенных книг.

Поля таблицы *Orders* должны быть снабжены атрибутом *NOT NULL*, так как при совершении покупки вся информация должна быть занесена в таблицу.

В таблице *Orders* устанавливается связь с таблицами *Users* (за счет поля *o_user_ID*) и *Books* (за счет поля *o_book_ID*). Эти поля объявлены как внешние ключи (FK) с правилом каскадного удаления и обновления. Обновление таблиц *Users* и *Books* приведет к автоматическому обновлению таблицы *Orders*. Удаление любого пользователя в таблице *Users* приведет к автоматическому удалению всех записей в таблице *Orders*, соответствующих этому пользователю. Удаление любой книги в таблице *Books* приведет к автоматическому удалению всех записей в таблице *Orders*, ссылающихся на эту книгу.

Практическое занятие № 2
Создание новой базы данных в среде SQL Server

Теоретические сведения

Рассмотрим следующие вопросы:

- создание, изменение, выбор и удаление базы данных;
- создание, модификация и удаление схемы базы данных;
- отображение созданной базы данных и ее файлов.

Создание базы данных. Оператор создания базы данных:

CREATE DATABASE < логическое имя базы данных >

[CONTAINMENT = { NONE | PARTIAL }]

[<предложение ON> [<предложение LOG ON>]]

[COLLATE <порядок сортировки>]

[WITH <опция> [, <опция>] ...];

Обязательно только логическое имя базы данных. Остальные конструкции необязательны. В этом случае базе данных и ее файлам будут присвоены значения всех характеристик по умолчанию.

Логическое имя должно соответствовать правилам задания идентификаторов:

- в обычном идентификаторе можно использовать буквы кириллицы (если при инсталляции сервера задан порядок сортировки *Cyrillic_General_CI_AS*);
- в идентификаторах с разделителями можно использовать любые символы (при этом идентификатор заключается в квадратные скобки).

Логическое имя базы данных содержит не более 128 символов. Оно должно быть уникальным среди имен всех баз данных текущего экземпляра сервера.

Предложение *CONTAINMENT* определяет степень независимости базы данных от характеристик экземпляра сервера, в котором создается база:

- *NONE* (значение по умолчанию) – создается обычная (неавтономная) база;
- *PARTIAL* – создается частично автономная база данных. К ней можно подключаться, используя средства аутентификации, не связанные с уровнем экземпляра сервера. Такую базу данных проще перенести на другой экземпляр сервера или на другой компьютер.

Предложение *ON* задает первичный файл данных. Файл, описанный первым, и указанные за ним файлы (если есть), помещаются в файловую группу *PRIMARY*. Описания файлов заключаются в круглые скобки и отделяются запятыми.

Кроме обязательной первичной файловой группы, база данных может содержать вторичные файловые группы. Вторичные файловые группы вместе с принадлежащими им файлами данных описываются после первичной файловой группы.

Предложение *LOG ON* описывает файл (файлы) журнала транзакций. Несколько файлов журналов транзакций задают, если для одного файла не хватает места на внешнем носителе.

При создании базы данных можно не задавать файлы данных и файлы журнала транзакций (будут присвоены значения по умолчанию). Можно задать файл (файлы) данных и не описывать файлы журнала транзакций. Однако нельзя указывать файл (файлы) журнала транзакций, не описав файл (файлы) данных.

Предложение *COLLATE* позволяет задать для создаваемой базы данных порядок сортировки, отличный от установленного при инсталляции системы для экземпляра сервера. Если предложение не указано, то база данных будет иметь порядок сортировки, заданный при инсталляции системы.

Создание и отображение базы данных с характеристиками по умолчанию:

```
1  USE master;
2  GO
3  CREATE DATABASE SimpleDB;
4  GO
5  SELECT file_id AS 'ID',
6         type  AS 'Type',
7         type_desc  AS 'Description',
8         name AS 'Name',
9         state AS 'State',
10        state_desc AS 'State desc',
11        size  AS 'Size'
12 FROM  sys.master_files
13 WHERE database_id = DB_ID('SimpleDB');
14 GO
```

100 %

Результаты | Сообщения

	ID	Type	Description	Name	State	State desc	Size
1	1	0	ROWS	SimpleDB	0	ONLINE	520
2	2	1	LOG	SimpleDB_log	0	ONLINE	130

Создание базы данных с удалением существующей одноименной базы данных:

```
1  USE master;
2  GO
3  IF DB_ID('SimpleDB') IS NOT NULL
4  DROP DATABASE SimpleDB;
5  GO
6  CREATE DATABASE SimpleDB;
7  GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Если база данных *SimpleDB* существует, то функция *DB_ID( )* вернет ее идентификатор. Если такой базы данных нет, то функция *DB_ID( )* вернет *NULL*.

Создание базы данных с явным определением характеристик. Файл данных и файл журнала транзакций будут расположены на диске *D*, в каталоге *MyDatabase* (каталог должен существовать, но таких файлов в нем быть не должно).

```
1  USE  master;
2  GO
3  IF DB_ID('MyDatabase') IS NOT NULL
4      DROP DATABASE MyDatabase;
5  GO
6  CREATE DATABASE MyDatabase
7  ON PRIMARY (NAME = MyDatabase_dat,
8      FILENAME = 'D:\MyDatabase\my.mdf',
9      SIZE = 5 MB,
10 MAXSIZE = UNLIMITED,
11     FILEGROWTH = 1 MB)
12 LOG  ON  (NAME = MyDatabase_log ,
13     FILENAME = 'D:\MyDatabase\my.ldf',
14     SIZE = 2 MB,
15     MAXSIZE = 30 MB,
16     FILEGROWTH = 1 MB);
17 GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Начальный размер файла данных – 5 Мбайт, приращение – 1 Мбайт, максимальный размер не ограничен (файл растет до исчерпания дискового пространства или до 16 Гбайт). Начальный размер файла журнала транзакций – 2 Мбайта, максимальный – 30 Мбайт, приращение – 1 Мбайт.

Создание многофайловой базы данных:

```
1  USE master;
2  GO
3  IF DB_ID('Multy') IS NOT NULL
4      DROP DATABASE Multy;
5  GO
6  CREATE DATABASE Multy
7  ON
8  PRIMARY
9      (NAME = Multy1,
10     FILENAME = 'D:\Multy\Multy1.mdf'),
11     (NAME = Multy2,
12     FILENAME = 'D:\Multy\Multy2.ndf')
13 LOG ON
14     (NAME = MultyL1,
15     FILENAME = 'D:\Multy\MultyL1.ldf'),
16     (NAME = MultyL2,
17     FILENAME = 'D:\Multy\MultyL2.ldf');
18 GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Будет создана база данных, содержащая два файла данных и два файла журнала транзакций (для примера). До этого на диске *D:* должен быть создан каталог *Multy*.

Создание базы данных с двумя файловыми группами:

```
1  USE  master;
2  GO
3  IF DB_ID('MultyGroup') IS NOT NULL DROP DATABASE MultyGroup;
4  GO
5  CREATE DATABASE MultyGroup
6  ON
7  PRIMARY
8      (NAME = MultyGroup1,
9      FILENAME = 'D:\MultyGroup\MultyGroup1.mdf'),
10     (NAME = MultyGroup2,
11     FILENAME = 'D:\MultyGroup\MultyGroup2.ndf'),
12 FILEGROUP  MultyGroup2
13     (NAME = MultyGroup3,
14     FILENAME = 'D:\MultyGroup\MultyGroup3.ndf'),
15     (NAME = MultyGroup4,
16     FILENAME = 'D:\MultyGroup\MultyGroup4.ndf')
17 LOG  ON
18     (NAME = MultyGroupLog1,
19     FILENAME = 'D:\MultyGroup\MultyGroupLog1.ldf'),
20     (NAME = MultyGroupLog2,
21     FILENAME = 'D:\MultyGroup\MultyGroupLog2.ldf');
22 GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Создается база данных, в которой будет две файловые группы по два файла данных: файловая группа *PRIMARY* включает файлы данных с логическими именами *MultyGroup1* и *MultyGroup2*, файловая группа *MultyGroup2* – файлы данных с логическими именами *MultyGroup3* и *MultyGroup4*.

Отображение файлов базы данных с двумя файловыми группами:

```
1  USE MultyGroup;
2  GO
3  SELECT file_id AS 'ID',
4         type AS 'Type',
5         type_desc AS 'Description',
6         name AS 'Name',
7         state AS 'State',
8         state_desc AS 'State desc'
9  FROM sys.database_files;
10 GO
```

100 %

Результаты | Сообщения

	ID	Type	Description	Name	State	State desc
1	1	0	ROWS	MultyGroup1	0	ONLINE
2	2	1	LOG	MultyGroupLog1	0	ONLINE
3	3	0	ROWS	MultyGroup2	0	ONLINE
4	4	0	ROWS	MultyGroup3	0	ONLINE
5	5	0	ROWS	MultyGroup4	0	ONLINE
6	6	1	LOG	MultyGroupLog2	0	ONLINE

Отображение файловых групп базы данных *MultyGroup*:

```
1  USE MultyGroup;
2  GO
3  SELECT name AS 'Name',
4         type AS 'Type' ,
5         type_desc AS 'Description',
6         is_read_only AS 'Read-only'
7  FROM sys.filegroups;
8  GO
```

100 %

Результаты | Сообщения

	Name	Type	Description	Read-only
1	PRIMARY	FG	ROWS_FILEGROUP	0
2	MultyGroup2	FG	ROWS_FILEGROUP	0

Модификация базы данных. Изменять можно базу данных, с которой не соединены другие пользователи. Для изменения используется оператор:

ALTER DATABASE { <имя базы данных> | CURRENT }
{ MODIFY NAME = <новое имя базы данных>
| COLLATE <порядок сортировки>
| <характеристики файлов>
| <характеристики файловых групп>
| <характеристики базы данных> } ;

Изменения будут выполнены либо для базы данных, указанной в операторе по имени, либо для текущей базы данных, указанной в последнем операторе *USE*, если указано ключевое слово *CURRENT*.

В одном операторе можно либо переименовать базу данных, либо изменить порядок сортировки, характеристики файлов, файловых групп или всей базы данных. Несколько действий выполняются с помощью нескольких операторов *ALTER DATABASE*.

Оператор *ALTER DATABASE* имеет широкую функциональность, полную информацию о которой можно найти в работе [2].

Удаление базы данных. Удаление базы данных, созданной пользователем и существующей в текущем экземпляре сервера, реализуется оператором:

DROP DATABASE <имя базы данных> [, <имя базы данных>] ... ;

Оператор удаляет одну или более пользовательских баз данных. Нельзя удалить системную базу данных или базу данных, открытую для работы другим пользователем. При удалении базы данных она удаляется из списка баз данных экземпляра сервера. С внешних носителей физически удаляются все файлы данных и все файлы журналов транзакций этой базы данных. Физическое удаление файлов возможно, если база данных находится в состоянии *ONLINE*.

Создание схемы. Для создания схемы используется оператор:

CREATE SCHEMA <имя схемы>
[AUTORIZATION <имя владельца>] [<элемент схемы> ...];
<элемент схемы> ::=
{ <определение таблицы> | <определение представления>
| <оператор GRANT> | <оператор REVOKE> | <оператор DENY> }

Имя схемы уникально в текущей базе данных. Предложение *AUTHORIZATION* задает владельца схемы. В процессе создания схемы в ней можно сформировать объекты (таблицы, представления); можно предоставить полномочия (*GRANT*), удалить полномочия (*REVOKE*), запретить наследование полномочий (*DENY*).

Операторы, используемые при создании схемы, выполняются как единое целое. Если возникает ошибка, то действия всех операторов отменяются. Признак завершения группы операторов, создающих одну схему, – оператор *GO*.

Создание двух схем в базе данных *MyDatabase*:

```
1  USE  MyDatabase;
2  GO
3  CREATE  SCHEMA  FirstSchema;
4  GO
5  CREATE  SCHEMA  SecondSchema;
6  GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Отобразить список схем базы данных позволяет системное представление *sys.schemas*. Оно имеет три столбца: *name* – имя схемы, *schema_id* – идентификатор схемы и *principal_id* – идентификатор владельца схемы.

```
1  USE MyDatabase;
2  GO
3  SELECT name, schema_id, principal_id
4  FROM sys.schemas;
5  GO
```

100 %

Результаты | Сообщения

	name	schema_id	principal_id
1	dbo	1	1
2	guest	2	2
3	INFORMATION_SCHEMA	3	3
4	sys	4	4
5	FirstSchema	5	1
6	SecondSchema	6	1
7	db_owner	16384	16384
8	db_accessadmin	16385	16385
9	db_securityadmin	16386	16386
10	db_ddladmin	16387	16387
11	db_backupoperator	16389	16389
12	db_datareader	16390	16390
13	db_datawriter	16391	16391
14	db_denydatareader	16392	16392
15	db_denydatawriter	16393	16393

Модификация схемы. Изменение схемы (перемещение объектов между схемами одной БД) осуществляет оператор:

ALTER SCHEMA <имя схемы> TRANSFER <элемент схемы>;

Имя перемещаемого элемента из другой схемы должно состоять из имени схемы и имени объекта, которые разделены точкой.

Удаление схемы. Удаление пустой схемы выполняется оператором:

DROP SCHEMA <имя схемы>;

Практическая работа

При выполнении задания необходимо для своей предметной области:

- создать новую базу данных;
- составить отчет.

Пример выполнения задания

Результатом выполнения задания является скрипт создания базы данных *Book*:

```
1  /*  Создание базы данных Book */
2  USE master;
3  GO
4  IF DB_ID('Book') IS NOT NULL
5      DROP DATABASE Book;
6  GO
7  CREATE DATABASE Book
8  ON  PRIMARY (NAME = Book_dat, FILENAME = 'D:\Book\book.mdf',
9          SIZE = 5 MB, MAXSIZE = UNLIMITED,
10         FILEGROWTH = 1 MB)
11 LOG ON (NAME = Book_log, FILENAME = 'D:\Book\book.ldf',
12         SIZE = 1 MB, MAXSIZE = 30 MB,
13         FILEGROWTH = 1 MB)
14 COLLATE Cyrillic_General_CI_AS;
15 GO
```

Скрипт создает обычную (неавтономную) базу данных с логическим именем *Book*. Файл данных и файл журнала транзакций располагаются на диске *D:* в каталоге *Book*. Каталог с именем *Book* должен существовать к моменту выполнения скрипта, а файлов *book.mdf* и *book.ldf* в указанном каталоге не должно быть.

База данных *Book* содержит только первичную файловую группу. Для файла данных установлен начальный размер 5 Мбайт, приращение – 1 Мбайт, максимальный размер не ограничен, т. е. файл может расти до исчерпания объема дискового пространства или до 16 Гбайт.

Начальный размер файла журнала транзакций – 1 Мбайт, максимальный – 30 Мбайт, а квант увеличения размера – 1 Мбайт. Для базы данных задан порядок сортировки *Cyrillic_General_CI_AS*.

Практическое занятие № 3
Создание и связывание таблиц базы данных в среде SQL Server

Теоретические сведения

Рассмотрим следующие вопросы:

- создание таблиц с помощью оператора *CREATE TABLE*;
- изменение структуры таблиц с помощью оператора *ALTER TABLE*;
- удаление таблиц с помощью оператора *DROP TABLE*.

Таблицы, содержащие данные предметной области, являются важнейшими объектами реляционной базы данных. Структура таблицы задается набором *столбцов*. В таблице могут быть описаны *ограничения* – некоторые дополнительные характеристики одного или группы столбцов.

Создание таблицы. Таблицы создаются оператором *CREATE TABLE*. Создавать таблицы может любой пользователь, имеющий на это права или обладающий ролью владельца базы либо системного администратора. Синтаксис оператора:

CREATE TABLE
[[<имя базы данных>.] <имя схемы>.] <имя таблицы>
[AS FileTable]
[(<определение столбца>
[, <определение столбца>
| , <определение вычисляемого столбца>
| , <определение набора столбцов>
| , <ограничение таблицы>] …)]
[ON (<схема секционирования> (<разделяющий ключ>)
| <файловая группа> | “default”)]
[TEXTIMAGE_ON (<файловая группа> | “default”)]
[FILESTREAM_ON (<схема секционирования>
| <файловая группа> | “default”)]
[WITH (<параметр таблицы> [, ... n]) ;

Имя обычной таблицы содержит до 128 символов. Если имя базы данных не указано – таблица создается в текущей базе данных. Если имя схемы не указано, то таблица создается в схеме по умолчанию (обычно – *dbo*). Имя таблицы должно быть уникальным среди имен таблиц данной схемы.

После идентификатора таблицы в круглых скобках перечисляются: список столбцов, список вычисляемых столбцов и ограничения таблицы. Предложение *AS FileTable* означает, что создается файловая таблица. Предложение *ON* указывает, как будут размещаться строки таблицы.

Предложение *TEXTIMAGE_ON* указывает, как будут размещаться столбцы с типами данных *TEXT, NTEXT, IMAGE, XML, VARCHAR(MAX), NVARCHAR(MAX), VARBINARY(MAX)* и с пользовательскими типами данных.

Предложение *FILESTREAM_ON* указывает файловую группу или схему секционирования для хранения данных файловых потоков. Предложение *WITH* указывает параметры таблицы, задающие режим сжатия данных и характеристики файловых таблиц.

Таблица имеет до 1024 столбцов. Синтаксис определения столбца таблицы:

< определение столбца > ::=
<имя столбца> < тип данных>
[FILESTREAM]
[COLLATE <порядок сортировки>]
[NULL | NOT NULL]
[DEFAULT <выражение> | [IDENTITY [(<начало> , <приращение>)]]]
[ROWGUIDCOL]
[< ограничение столбца > [, ... n]]
[SPARSE]

Для столбца определяются имя и тип данных. Имена столбцов имеют длину до 128 символов и уникальны в пределах таблицы. Тип данных может быть системным или пользовательским. Системные типы перечислены в приложении.

Ключевые слова *NULL* и *NOT NULL* определяют допустимость для столбца неопределенных значений. По умолчанию устанавливается атрибут *NOT NULL*.

Ключевое слово *FILESTREAM* указывается только для типа *VARBINARY(MAX).* Значения столбца будут храниться не в базе данных, а в контейнере данных в файловой системе Windows.

Предложение *COLLATE* позволяет для строкового столбца задать порядок сортировки, отличный от порядка, установленного для базы данных.

Предложение *DEFAULT* позволяет задать для столбца значение по умолчанию. Оно будет помещено в столбец, если при вводе строки не было явно указано значение столбца. Значение по умолчанию нельзя задавать для столбца *IDENTITY*.

Предложение *IDENTITY* означает, что это автоинкрементный столбец (столбец счетчика, столбец идентификатора). Такие столбцы автоматически генерируют уникальные значения с некоторым приращением (1, 2, 3 ... или 1, 4, 7, 10 ...). Используются как искусственные первичные ключи таблиц. Столбец счетчика в таблице только один, он не может содержать значения *NULL* и дол-

жен иметь числовой тип данных (*INT, SMALLINT, TINYINT, BIGINT, NUMERIC, DECIMAL*). По умолчанию действует начальное значение 1 и приращение 1.

Например, если столбец счетчика *author_ID* объявлен с типом *TINYINT*, то добавить больше 255 авторов нельзя.

CREATE TABLE Authors (
author_ID TINYINT IDENTITY,
name VARCHAR(50) NOT NULL) ;

Столбец счетчика не гарантирует уникальности значений, особенно если база размещена на нескольких серверах.

Ключевое слово *ROWGUIDCOL* означает, что столбец является столбцом идентификаторов GUID. Он должен иметь тип *UNIQUEIDENTIFIER*. Обеспечивает глобальную уникальность значений для всех компьютеров сети.

Пример таблицы со столбцом идентификаторов GUID:

CREATE TABLE Authors (
author_ID UNIQUEIDENTIFIER NOT NULL ROWGUIDCOL
DEFAULT (NEWID( )),
name VARCHAR(50) NOT NULL) ;

В примере использована функция *NEWID( )*, создающая новый идентификатор при вставке новой записи.

Ключевое слово *SPARSE* указывает, что столбец является разреженным. Используется для экономии внешней памяти, если столбец имеет большое количество значений *NULL*. Нельзя использовать для типов данных *TEXT, NTEXT, IMAGE, TIMESTAMP, GEOMETRY, GEOGRAPHY* и пользовательских типов данных. Такой столбец не может иметь атрибутов *NOT NULL* и *FILESTREAM*.

Ограничения. Накладывают определенные условия на вводимые данные на уровне столбца или на уровне таблицы. Есть четыре вида ограничений:

- первичный ключ (*PRIMARY KEY*);
- уникальный ключ (*UNIQUE*);
- внешний ключ (*FOREIGN KEY*);
- ограничение проверки (*CHECK*).

Синтаксис ограничения на уровне столбца и на уровне таблицы:

< ограничение столбца или ограничение таблицы > ::=

[CONSTRAINT <имя ограничения>]

{ <первичный ключ> | <уникальный ключ> | <внешний ключ> | < CHECK> }

Предложение *CONSTRAINT* дает имя ограничению. Рекомендуется присваивать ограничениям понятные имена, иначе система присвоит свои.

Ограничение первичного ключа *PRIMARY KEY* требует, чтобы содержимое столбца было уникальным. Первичный ключ в таблице один. Он может состоять из одного или нескольких столбцов, для которых следует указать *NOT NULL*.

Ограничение уникального ключа *UNIQUE* также требует, чтобы содержимое столбца было уникальным. Уникальных ключей в таблице может быть несколько, а столбцы уникального ключа могут иметь значение *NULL*.

Синтаксис ограничения первичного ключа на уровне столбца:

< первичный ключ (столбец) > ::=
PRIMARY KEY
[CLUSTERED | NONCLUSTERED]
[WITH (<параметр индекса> [, <параметр индекса>] ...)]

Синтаксис ограничения первичного ключа на уровне таблицы:

< первичный ключ (таблица) > ::=
PRIMARY KEY
[CLUSTERED | NONCLUSTERED]
(<столбец> [ASC | DESK] [, <столбец> [ASC | DESK]] …)
[WITH (<параметр индекса> [, <параметр индекса>] ...)]

Любое ограничение на уровне таблицы может относиться к одному столбцу.

Синтаксис ограничения уникального ключа отличается от синтаксиса ограничения первичного ключа названием вида ключа (*UNIQUE* вместо *PRIMARY KEY).*

Для первичного и уникального ключа система автоматически создает поддерживающий ограничение индекс, которому будет присвоено имя, указанное в *CONSTRAINT* или сгенерированное системой. Можно указать тип индекса – кластерный (*CLUSTERED*) или некластерный (*NONCLUSTERED*). Для первичного ключа по умолчанию создается кластерный индекс, для уникального ключа – некластерный.

Объявление ограничения первичного ключа на уровне столбца:

```
CREATE TABLE Authors (
   author_ID  INT  NOT NULL  CONSTRAINT  author_pk  PRIMARY  KEY,
   name  VARCHAR(50)  NOT  NULL  );
```

Указывать ключевое слово *CONSTRAINT* с именем ограничения необязательно (если имя не указано, система присвоит его сама).

```
CREATE TABLE Authors (
   author_ID  INT  NOT NULL  PRIMARY  KEY,
   name  VARCHAR(50)  NOT  NULL  );
```

Объявление первичного ключа из нескольких столбцов на уровне таблицы (предполагается, что не может быть авторов с одинаковым именем и фамилией):

CREATE TABLE Authors (
first_name VARCHAR(50) NOT NULL,
last_name VARCHAR(50) NOT NULL,
CONSTRAINT author_pk PRIMARY KEY (first_name, last_name));

То же самое без задания имени ограничения:

CREATE TABLE Authors (
first_name VARCHAR(50) NOT NULL,
last_name VARCHAR(50) NOT NULL,
PRIMARY KEY (first_name, last_name));

Ограничение внешнего ключа *FOREIGN KEY* позволяет обеспечить целостность данных дочерней таблицы. Система обращается к первичному или уникальному ключу родительской таблицы и проверяет совпадение данных в столбцах.

Синтаксис ограничения внешнего ключа на уровне столбца:

< внешний ключ (столбец) > ::=
[FOREIGN KEY]
REFERENCES [<схема>.] <таблица> [(<столбец>)]
[ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]
[ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]

Синтаксис ограничения внешнего ключа на уровне таблицы:

< внешний ключ (таблица) > ::=
FOREIGN KEY (<столбец> [, ... n])
REFERENCES [<схема>.] <таблица> [(<столбец> [, ... n])]
[ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]
[ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]

В обоих случаях для родительской таблицы можно не указывать список столбцов, если ссылка осуществляется только на первичный ключ этой таблицы.

Внимание! Структура и типы данных столбцов внешнего ключа дочерней таблицы должны полностью совпадать со структурой и типами данных первичного или уникального ключа родительской таблицы.

Предложения *ON DELETE* и *ON UPDATE* определяют, что произойдет со строками дочерней таблицы при удалении связанной строки родительской таблицы или при изменении значения любого столбца в ключе родительской таблицы.

Возможные варианты:

- *NO ACTION* – нет действий (по умолчанию); нарушается ссылочная целостность базы данных и необходимо каким-либо образом реагировать;
- *CASCADE* – удаление всех связанных строк дочерней таблицы (изменение значений реквизитов внешнего ключа всех связанных строк дочерней таблицы);
- *SET NULL* – столбцы внешнего ключа всех связанных строк дочерней таблицы устанавливаются в *NULL* (строки дочерней таблицы остаются в базе);
- *SET DEFAULT* – столбцы внешнего ключа дочерней таблицы устанавливаются в значение по умолчанию.

Пример ограничения внешнего ключа, заданного на уровне столбца. Создается таблица *Books*, связанная по внешнему ключу с таблицей *Authors*.

```
CREATE TABLE Books (
    book_name  VARCHAR(250)  NOT NULL,
    author_ID  INT  NOT NULL
    CONSTRAINT author_fk  REFERENCES Authors (author_ID)   ) ;
```

Это же ограничение, объявленное на уровне таблицы:

```
CREATE TABLE Books (
    book_name  VARCHAR(250)  NOT NULL,
    author_ID  INT  NOT NULL,
    CONSTRAINT author_fk  FOREIGN KEY (author_ID)
            REFERENCES Authors (author_ID)
                ON DELETE  CASCADE ON UPDATE  NO ACTION   );
```

Внимание! Ограничения внешнего ключа из нескольких столбцов должны объявляться на уровне таблицы.

Ограничение проверки *CHECK* дает возможность проверять данные на соответствие бизнес-правилам предметной области. Кроме того, ограничения проверки позволяют сравнивать данные из различных столбцов – при условии, что столбцы находятся в одной таблице, а ограничение объявлено на уровне таблицы.

Синтаксис ограничения проверки на уровне столбца и на уровне таблицы:

< ограничение CHECK > ::=

CHECK (<логическое выражение>)

Логическое выражение определяет условия, которым должны удовлетворять вставляемые или обновляемые данные. В ограничении на уровне столбца логическое выражение относится к текущему столбцу. В ограничении на уров-

не таблицы логическое выражение может содержать имена других столбцов таблицы.

В примере создается таблица *Authors*, хранящая сведения об авторах, имена которых начинаются с буквы *A* (ограничение *last_name* определено на уровне столбца).

```
CREATE TABLE  Authors (
   author_ID  INT  NOT NULL  CONSTRAINT author_pk  PRIMARY KEY,
   first_name  VARCHAR(50)  NOT NULL,
   last_name  VARCHAR(50)  NOT NULL
  CONSTRAINT last_name_ck  CHECK (last_name  LIKE  'A%' )  );
```

Это же ограничение, объявленное на уровне таблицы:

```
CREATE TABLE  Authors (
   author_ID  INT  NOT NULL  CONSTRAINT author_pk  PRIMARY KEY,
   first_name  VARCHAR(50)  NOT NULL,
   last_name  VARCHAR(50)  NOT NULL,
   CONSTRAINT last_name_ck  CHECK (last_name  LIKE  'A%' ) );
```

Как именованные ограничения можно оформить задание для столбца значения по умолчанию и свойства *NOT NULL*. Однако эти ограничения могут быть заданы только на уровне столбца.

Пример определения значения по умолчанию для столбца *Гражданство* как ограничения на уровне столбца:

```
CREATE  TABLE  Authors  (
      author_ID  INT  NOT NULL  CONSTRAINT  author_pk   PRIMARY  KEY,
      [Имя]  VARCHAR(50)  NOT NULL,
      [Фамилия]  VARCHAR(50)  NOT NULL,
      [Гражданство]  VARCHAR(50)  NOT NULL  CONSTRAINT  cit_def
      DEFAULT  'Россия' );
```

Пример задания свойства *NOT NULL* для столбца *Фамилия* как ограничения на уровне столбца:

```
CREATE TABLE Authors (
      author_ID   INT  NOT NULL  CONSTRAINT  author_pk  PRIMARY KEY,
      [Имя]  VARCHAR(50)  NOT NULL,
      [Фамилия]  VARCHAR(50)  CONSTRAINT fam_nn  NOT NULL,
      [Гражданство]  VARCHAR(50)  NOT NULL  );
```

Вычисляемые столбцы. Это столбцы, значения которых вычисляются с использованием функций, констант или значений других столбцов таблицы. Значения вычисляемых столбцов могут храниться в таблице (*PERSISTED*), а могут вычисляться при отображении таблицы.

Синтаксис вычисляемого столбца:

< вычисляемый столбец > ::=

< имя столбца > AS < выражение >

[PERSISTED [NOT NULL]]

[< ограничение столбца >]

Конструкция *<ограничение столбца>* рассмотрена выше. Единственное отличие – в определении внешнего ключа для вычисляемого столбца. При удалении связанной строки родительской таблицы можно задать только *NO ACTION* и *CASCADE*, при изменении – только *NO ACTION*.

Пример таблицы с не хранимым в базе вычисляемым столбцом:

CREATE TABLE book_price (

book_ID INT NOT NULL CONSTRAINT book_price_pk PRIMARY KEY,

[Оптовая цена] SMALLMONEY NOT NULL,

[Розничная цена] SMALLMONEY NOT NULL,

[Надбавка] AS ([Розничная цена] - [Оптовая цена]));

Эта же таблица с постоянно хранимым в базе данных вычисляемым столбцом:

CREATE TABLE book_price (

book_ID INT NOT NULL CONSTRAINT book_price_pk PRIMARY KEY,

[Оптовая цена] SMALLMONEY NOT NULL,

[Розничная цена] SMALLMONEY NOT NULL,

[Надбавка] AS ([Розничная цена] - [Оптовая цена])

PERSISTED NOT NULL);

Набор столбцов. Если в таблице есть разреженные (*SPARSE*) столбцы, то можно добавить столбец, называемый *набором столбцов*. Такой столбец будет вычисляемым, причем данные не хранятся в базе, а создаются при отображении значения столбца в XML-представлении.

Временные таблицы. Оператор *CREATE TABLE* позволяет создавать временные таблицы (локальные или глобальные). Локальные таблицы видны только в сеансе, в котором они созданы. С глобальными таблицами могут работать другие процессы, выполняемые параллельно с процессом, в котором создана таблица.

Отличия временных таблиц от обычных:

- имя локальной временной таблицы должно начинаться с символа #, глобальной временной таблицы – с символа ##;
- длина имени временной таблицы ограничивается 116 символами;
- временные таблицы автоматически удаляются при завершении сеанса;
- временные таблицы используются так, как будто они входят в текущую базу данных, но в действительности данные хранятся в базе *TEMPDB*.

Создание локальной временной таблицы *#local1*:

CREATE TABLE #local1 (a INT NULL, b VARCHAR(25) NULL);

Создание глобальной временной таблицы *##global1*:

CREATE TABLE ##global1 (a INT NULL, b VARCHAR(25) NULL);

Изменение структуры таблиц. Модифицировать структуру существующей таблицы позволяет оператор *ALTER TABLE*. Этот оператор является очень гибким и имеет большое количество ключевых слов.

Удаление таблиц. Удалить таблицу может ее владелец с помощью оператора

DROP TABLE [[<имя базы данных>.] < имя схемы>.]
< имя таблицы> [, ... n] ;

Нельзя удалить таблицу с внешними зависимостями (на таблицу ссылается хранимая процедура, представление или она является родительской таблицей в связи). Перед удалением таблицы необходимо удалить эти ограничения.

Удаление обычной таблицы:

DROP TABLE Authors;

Удаление глобальной временной таблицы:

DROP TABLE ##global1;

Практическая работа

При выполнении задания необходимо для своей предметной области:

- создать таблицы, определить поля таблиц и тип полей;
- определить связи между таблицами и ограничения целостности;
- составить отчет.

Пример выполнения задания

Результатом выполнения работы являются скрипты создания таблиц. При написании скрипта следует учитывать, что в первую очередь создаются родительские таблицы, а уже потом – дочерние.

```
1   /*  Создание таблицы Catalogs */
2   USE Book;
3   GO
4  IF EXISTS (SELECT * FROM sys.tables   WHERE NAME = 'Catalogs')
5   DROP TABLE Catalogs;
6  CREATE TABLE Catalogs
7      (cat_ID INT  NOT NULL,
8       cat_name VARCHAR(20) NOT NULL,
9       CONSTRAINT PK_Catalogs PRIMARY KEY (cat_ID) ) ;
10  GO
11
```

```
12  /*  Создание таблицы Books */
13  USE Book;
14  GO
15  IF EXISTS (SELECT * FROM sys.tables  WHERE NAME = 'Books')
16  DROP TABLE Books;
17  CREATE TABLE Books
18  (  book_ID int  NOT NULL,
19  b_name VARCHAR(100) NOT NULL,
20  b_author VARCHAR(100) NOT NULL,
21  b_year INT  NOT NULL,
22  b_price MONEY  NOT NULL DEFAULT 0.00,
23  b_count INT  NOT NULL DEFAULT 0,
24  b_cat_ID INT  NOT NULL,
25          CONSTRAINT PK_Books PRIMARY KEY (book_ID),
26      CONSTRAINT FK_Books_Catalogs
27  FOREIGN KEY (b_cat_ID) REFERENCES Catalogs (cat_ID)
28  ON UPDATE  CASCADE  ON DELETE  CASCADE );
29  GO
30
31  /*  Создание таблицы Users */
32  USE Book;
33  GO
34  IF EXISTS (SELECT * FROM  sys.tables  WHERE NAME = 'Users')
35  DROP TABLE Users;
36  CREATE TABLE Users
37      (  user_ID  INT   NOT NULL,
38         u_name VARCHAR(20)  NOT NULL,
39         u_patronymic VARCHAR(20)  NOT NULL,
40         u_surname VARCHAR(20)  NOT NULL,
41         u_phone VARCHAR(15),
42         u_email VARCHAR(20),
43         u_status VARCHAR(10)  NOT NULL,
44         CONSTRAINT  PK_Users  PRIMARY KEY (user_ID),
45         CONSTRAINT CH_Users CHECK(u_status IN ('gold', 'lock', 'passive', 'active')) );
46  GO
47
48  /*  Создание таблицы Orders */
49  USE Book;
50  GO
51  IF EXISTS (SELECT * FROM  sys.tables  WHERE NAME = 'Orders')
52  DROP TABLE Orders;
53  CREATE TABLE Orders
54      (order_ID INT NOT NULL,
55       o_user_ID INT NOT NULL,
56       o_book_ID   INT  NOT NULL,
57       o_time  DATETIME NOT NULL,
58       o_number INT NOT NULL DEFAULT 0,
59       CONSTRAINT PK_Orders PRIMARY KEY (order_ID),
60       CONSTRAINT FK_Orders_Books
61       FOREIGN KEY (o_book_ID) REFERENCES Books (book_ID)
62       ON UPDATE CASCADE ON DELETE CASCADE ,
63       CONSTRAINT FK_Orders_Users
64       FOREIGN KEY (o_user_ID) REFERENCES Users(user_ID)
65       ON UPDATE CASCADE ON DELETE CASCADE);
66  GO
```

Практическое занятие № 4
Вставка, удаление и обновление данных

Теоретические сведения

Рассмотрим следующие вопросы:

- вставка данных с помощью оператора *INSERT*;
- обновление данных с помощью оператора *UPDATE;*
- удаление данных операторами *DELETE* и *TRUNCATE TABLE*.

После создания таблиц разработчик должен заполнить таблицы данными. Изменять содержимое таблиц позволяют операторы *INSERT, UPDATE* и *DELETE*, работающие с таблицами и представлениями.

Вставка данных. Для добавления в таблицу или представление одной или нескольких строк используется оператор *INSERT*. Строки, помещаемые в таблицу, могут быть заданы четырьмя способами:

1) с перечислением значений в предложении *VALUES*:

INSERT [TOP (<выражение>) [PERCENT]]
[INTO] [[<база данных>.] <схема>.] { <таблица> | <представление> }
[(<список столбцов>)] [<предложение OUTPUT>]
{ VALUES ({ DEFAULT | NULL | <выражение> } [, ... n])
[, ({ DEFAULT | NULL | <выражение> } [, ... n])] …

2) с присваиванием значений по умолчанию всем столбцам:

INSERT [TOP (<выражение>) [PERCENT]]
[INTO] [[<база данных>.] <схема>.] { <таблица> | <представление> }
[(<список столбцов>)] [<предложение OUTPUT>]
DEFAULT VALUES

3) с использованием оператора *SELECT*, возвращающего одну или несколько строк какой-либо таблицы:

INSERT [TOP (<выражение>) [PERCENT]]
[INTO] [[<база данных>.] <схема>.] { <таблица> | <представление> }
[(<список столбцов>)] [<предложение OUTPUT>]
<оператор SELECT>

4) с вызовом хранимой процедуры оператором *EXECUTE*:

INSERT [TOP (<выражение>) [PERCENT]]
[INTO] [[<база данных>.] <схема>.] { <таблица> | <представление> }

[(<список столбцов>)] [<предложение OUTPUT>]
<оператор EXECUTE>

Предложение *TOP* указывает количество (процент) добавляемых строк. После ключевого слова *INTO* записывают, куда именно добавляются строки. В операторе можно указать список столбцов, которым присваиваются значения. Порядок столбцов в списке может отличаться от порядка столбцов в таблице. Однако в списке должны присутствовать все столбцы, для которых запрещены значения *NULL* и которые не имеют значений по умолчанию. При отсутствии списка столбцов порядок значений соответствует порядку столбцов в таблице.

Предложение *OUTPUT* позволяет поместить все добавленные оператором строки в локальную переменную типа *TABLE* или в заданную таблицу базы данных, обычную или временную.

При использовании предложения *VALUES* можно одним оператором добавлять множество строк, указывая через запятую списки значений в скобках.

Каждому столбцу в списке значений соответствует выражение (константа, функция, их сочетание), ключевое слово *DEFAULT* или *NULL*. Нельзя использовать подзапросы, столбцы, агрегатные функции. Присваиваемое значение неявно преобразуется к типу данных столбца, или разработчиком выполняется явное преобразование.

Если используется *DEFAULT*, то при вставке данных столбцу будет присвоено значение по умолчанию. При этом если *DEFAULT* находится в списке значений, оно относится к соответствующему столбцу. Если *DEFAULT* находится перед началом списка, оно относится ко всем столбцам, тогда список столбцов должен быть пустым. В этом случае слово *DEFAULT* обрабатывается следующим образом:

- если у столбца есть значение по умолчанию, оно будет занесено в столбец;
- если значения по умолчанию нет, будет присвоено значение *NULL*;
- если столбец не допускает *NULL*, сервер выдаст сообщение об ошибке.

Вставка данных в таблицу со столбцом идентификаторов показана в примере. Первые два оператора *INSERT* разрешают формирование значений идентификаторов для новых строк. Затем переопределяется свойство *IDENTITY* столбца с помощью инструкции *SET IDENTITY_INSERT* и вставляется явно заданное значение.

```
1  USE Work;
2  GO
3  IF OBJECT_ID ('dbo.T1', 'U') IS NOT NULL
4      DROP TABLE dbo.T1;
5  GO
6  CREATE TABLE dbo.T1 ( col_1 INT IDENTITY, col_2 VARCHAR(30));
7  GO
8  INSERT T1 VALUES ('Столбец 1');
9  INSERT T1 (col_2) VALUES ('Столбец 2');
10 GO
11 SET IDENTITY_INSERT T1 ON;
12 GO
13 INSERT INTO T1 (col_1, col_2)
14     VALUES (-99, 'Явное значение идентификатора');
15 GO
16 SELECT col_1, col_2
17 FROM T1;
18 GO
```

Результаты | Сообщения

	col_1	col_2
1	1	Столбец 1
2	2	Столбец 2
3	-99	Явное значение идентификатора

При вставке данных в столбцы с типом данных *UNIQUEIDENTIFIER* для получения глобально-уникального идентификатора, присваиваемого столбцу, можно воспользоваться функцией *NEWID()*, как показано в примере:

```
1  USE Work;
2  GO
3  IF OBJECT_ID ('dbo.T2', 'U') IS NOT NULL
4      DROP TABLE dbo.T2;
5  GO
6  CREATE TABLE dbo.T2
7  (col_1 INT IDENTITY, col_2 UNIQUEIDENTIFIER);
8  GO
9  INSERT INTO dbo.T2 (col_2)
10     VALUES (NEWID());
11 INSERT INTO dbo.T2 DEFAULT VALUES;
12 GO
13 SELECT col_1, col_2
14 FROM dbo.T2;
15 GO
```

100 %

Результаты | Сообщения

	col_1	col_2
1	1	06C8660D-20C1-46C2-8AFA-F5AAE1B5F650
2	2	NULL

В отличие от столбцов идентификаторов, для столбцов с типом данных *UNIQUEIDENTIFIER* автоматически значения не создаются (см. действие второго оператора *INSERT*).

Многострочный оператор *INSERT* совпадает по форме с однострочным оператором, но после *VALUES* добавляются через запятую несколько списков значений в круглых скобках.

Примеры использования многострочного оператора *INSERT* для заполнения учебной БД *Book* см. в пункте «Пример выполнения работы».

Кроме того, для вставки нескольких записей одновременно можно использовать форму оператора *INSERT*, где список значений заменяется оператором *SELECT* или вызовом хранимой процедуры.

Оператор *SELECT*, являющийся частью оператора *INSERT*, может не возвратить ни одной записи, что ошибкой не является. Если хотя бы одна вставленная запись нарушит ограничение таблицы, весь оператор *INSERT* не выполняется.

Можно использовать *SELECT* для выборки записей из таблицы, в которую вставляются данные (следите за сохранением уникальности индексов).

В следующем примере создается временная таблица *#Temp1*, в которую помещаются значения общей стоимости книг в каждом каталоге (без учета количества экземпляров).

```
1   USE Book;
2   GO
3   CREATE TABLE #Temp1(cat_ID INT, price MONEY);
4   GO
5   INSERT INTO #Temp1(cat_ID, price)
6       SELECT b_cat_ID, SUM(b_price)
7       FROM Books
8       GROUP BY b_cat_ID;
9   GO
10  SELECT cat_ID, price
11  FROM #Temp1;
12  GO
```

100 %

Результаты | Сообщения

	cat_ID	price
1	1	6923,80
2	2	2094,80
3	3	1484,00
4	4	1970,70
5	5	2591,90

Вставляемые в таблицу записи можно получить, вызвав хранимую процедуру, возвращающую данные с помощью оператора *SELECT*. Хранимые процедуры в операторе *INSERT* подчиняются следующим правилам:

- хранимая процедура, используемая для создания нескольких записей, может выполнять обновление и вставку в других таблицах;
- процедура может извлекать несколько итоговых наборов, выполняя несколько операторов *SELECT*;
- полученные данные должны соответствовать столбцам в списке;
- если хранимая процедура не возвращает ни одной записи, это не ошибка;
- если хотя бы одна вставляемая запись нарушает ограничение таблицы, то оператор *INSERT* завершается неудачей.

Пусть в базе данных имеется хранимая процедура *Catalog_Select*, осуществляющая выборку из таблицы *Catalogs*. Выполним вставку получаемой выборки во временную таблицу *#Temp3*.

```
1  USE Book;
2  GO
3  CREATE TABLE #Temp3(id INT, name VARCHAR(20));
4  GO
5  INSERT INTO #Temp3(id, name)
6      EXEC Catalog_Select;
7  GO
8  SELECT id, name
9  FROM #Temp3;
10 GO
```

Результаты | Сообщения

	id	name
1	1	Программирование
2	2	Интернет
3	3	Базы данных
4	4	Сети
5	5	Мультимедиа

Для ограничения количества вставляемых записей используют секцию *TOP*. В следующем примере будут вставлены только первые две записи из выборки:

```
1  USE Book;
2  GO
3  CREATE TABLE #Temp5(id INT, name VARCHAR(20));
4  GO
5  INSERT TOP(2) #Temp5(id, name)
6      EXEC Catalog_Select;
7  GO
8  SELECT id,name
9  FROM #Temp5;
10 GO
```

100 %

Результаты | Сообщения

	id	name
1	1	Программирование
2	2	Интернет

Оператор *SELECT … INTO* позволяет создать новую таблицу и заполнить ее данными. Его применение ограничено, как правило, созданием временных таблиц.

SELECT <список выборки> INTO <новая таблица> FROM <список таблиц>
[секция WHERE] [секция GROUP BY] [секция HAVING]
[секция ORDER BY]

В следующем примере с помощью оператора *SELECT … INTO* копируются все записи и столбцы из таблицы *Users* в новую временную таблицу *#Temp6*.

```
1  USE Book;
2  GO
3  SELECT * INTO #Temp6 FROM Users;
4  GO
5  SELECT * FROM #Temp6;
6  GO
```

100 %

Результаты | Сообщения

	user_ID	u_name	u_patronymic	u_surname	u_phone	u_email	u_status
1	1	Александр	Валерьевич	Иванов	58-98-78	ivanov@email.ru	active
2	2	Сергей	Иванович	Лосев	90-57-77	losev@email.ru	passive
3	3	Нина	Сергеевна	Грачева	95-66-61	gracheva@email.ru	active
4	4	Максим	Петрович	Кузнецов	NULL	kuznetsov@email.ru	active
5	5	Анатолий	Юрьевич	Петров	NULL	NULL	lock
6	6	Александр	Александрович	Корнеев	89-78-36	korneev@email.ru	gold

Оператор *SELECT … INTO* создает новую таблицу, содержащую результаты выборки. Новая таблица состоит из столбцов, присутствующих в списке выборки. Эта таблица может не принадлежать текущей базе данных. Она не должна существовать перед выполнением оператора. Не допускается выполнение *SELECT … INTO* внутри транзакции.

Если оператор *SELECT* не возвращает записей, будет создана пустая таблица. Свойства счетчика наследуются, если *SELECT* не содержит объединения, агрегатной функции, секции *GROUP BY, UNION*, нескольких столбцов счетчика в списке выборки или столбца, используемого в выражениях.

В новой таблице не создаются ограничения и индексы, а вычисляемые столбцы становятся обычными (не вычисляемыми) столбцами.

Обновление данных. Изменение значений полей в существующих записях обеспечивает оператор *UPDATE*:

UPDATE [TOP (<выражение>) [PERCENT]]
[[<база данных>.] <схема>.] { <таблица> | <представление> }

SET { <изменяемое значение> [, ... n] }
[<предложение OUTPUT>]
[FROM <таблица-источник> [, ... n]]
[WHERE <условия>]
<изменяемое значение> ::=
{ <столбец> = { <выражение> | DEFAULT | NULL }
| @<переменная> [= <столбец>] <операция> <выражение>] [, ... n] }

После ключевого слова *UPDATE* указывается изменяемая таблица. Это может быть также представление, ссылающееся на таблицу текущего экземпляра сервера. В предложении *SET* указывается, какие столбцы обновляются, и устанавливаются их новые значения. Необязательное условие *WHERE* позволяет задать критерий отбора строк (обновляться будут строки, удовлетворяющие условию).

В предложении *TOP* указывается количество либо процент (*PERCENT*) строк, изменяемых в операторе. В предложении *SET* допустимы разные варианты указания столбцов и их новых значений.

Обычно после имени столбца и знака равенства записывается константа, выражение, ключевое слово *DEFAULT* (значение по умолчанию) или *NULL* (неопределенное значение). Выражение может быть сложным, причем это может быть оператор *SELECT* (подзапрос, возвращающий записи). Такой оператор должен быть заключен в круглые скобки. Нельзя непосредственно присвоить столбцу агрегатную функцию. Однако допускается присваивание подзапросов, возвращающих агрегатные функции.

В другом варианте сначала столбцу изменяемой строки присваивается значение, полученное в результате применения указанной операции к выражению, затем это же значение присваивается локальной переменной. Операции:

- = – обычное присваивание;
- += – сложение значения, полученного при вычислении выражения, с существующим значением столбца (результат присваивается переменной и столбцу);
- –= – вычитание и присваивание;
- *= – умножение и присваивание;
- /= – деление и присваивание;
- %= – получение остатка от деления и присваивание;
- &= – побитовая операция *И* и присваивание;
- ^= – побитовая операция исключающее *ИЛИ* и присваивание;
- |= – побитовая операция *ИЛИ* и присваивание.

Если в конструкции не указано имя столбца, то значение присваивается только локальной переменной.

Предложение *OUTPUT* возвращает измененные строки таблицы. В предложении *FROM* задается список таблиц, столбцы из которых могут использоваться в выражениях. В предложении *WHERE* задаются условия выбора обновляемых строк таблицы. Секции *FROM* и *WHERE* команды *UPDATE* подчиняются тем же правилам, что и соответствующие секции команды *SELECT*.

Запрос, изменяющий в таблице *Catalogs* название каталога «Сети» на «Компьютерные сети». При выполнении обновления изменяется значение переменной *@code*:

```
1  USE Book;
2  GO
3  DECLARE @code INT
4  UPDATE Catalogs
5  SET cat_name = 'Компьютерные сети', @code = 1
6  WHERE cat_name = 'Сети';
7  SELECT @code;
8  GO
```

100 %

Результаты | Сообщения

	(Отсутствует имя столбца)
1	1

Обновлять можно всю таблицу. Пусть требуется уменьшить на 5 % цену на все книги. Для этого следует старую цену в рублях умножить на 0,95.

```
1  USE Book;
2  GO
3  UPDATE Books
4  SET b_price = b_price*0.95
5  GO
```

Запрос, устанавливающий статус *'gold'* всем клиентам интернет-магазина, имевшим статус *'active'*. Для определения списка обновляемых записей используется подзапрос *SELECT*:

```
1  USE Book;
2  GO
3  UPDATE Users SET u_status = 'gold'
4  WHERE user_ID IN
5  (SELECT user_ID FROM Users
6  WHERE u_status = 'active');
7  GO
```

Если имя обновляемой таблицы должно использоваться в секции *WHERE* или в подзапросе, обычно таблице назначается псевдоним.

Использование секции *TOP* позволяет ограничить количество изменяемых записей. При этом обновляемые записи не сортируются в каком-либо порядке (необходимую сортировку можно предусмотреть в подзапросе). Например, в таблице *Books* для первых трех самых дешевых товарных позиций следует уменьшить количество книг на складе на единицу, а цену – на 5 %.

```
1  USE Book;
2  GO
3  UPDATE TOP (3) Books
4  SET b_price = b_price*0.95, b_count = b_count - 1
5  WHERE book_ID IN
6  (SELECT TOP (30) book_ID FROM Books
7  ORDER BY b_price);
8  GO
```

При выполнении *UPDATE* нельзя изменять столбцы счетчика. Если какая-либо из модифицированных записей нарушает правило или ограничение таблицы, весь оператор *UPDATE* не выполняется.

Удаление данных. Для удаления одной или более строк из таблицы или представления предусмотрены операторы *DELETE* и *TRUNCATE TABLE*.

Синтаксис оператора *DELETE*:

DELETE [TOP (<выражение>) [PERCENT]] [FROM]
[[<база данных>.] <схема>.] { <таблица> | <представление> }
[<предложение OUTPUT>]
[FROM <таблица-источник> [, ... n]]
[WHERE <условия>]

Назначение и использование параметров оператора *DELETE* аналогично соответствующим параметрам оператора *UPDATE*. Оператор удаляет из таблицы или представления записи, удовлетворяющие условию.

В следующем примере из таблицы *Catalogs* удаляются записи, имеющие значение первичного ключа *catalog_ID* больше двух.

```
1  USE Book;
2  GO
3  DELETE FROM Catalogs
4  WHERE cat_ID > 2;
5  GO
```

Если в операторе отсутствует условие *WHERE*, удаляются все строки таблицы.

С помощью секции *TOP* можно ограничить количество удаляемых строк. При этом сортировка удаляемых строк не определяется. Например, следующий оператор удалит только первые 4 книги из каталога «Сети»:

```
1  USE Book;
2  GO
3  DELETE TOP (4) FROM Books
4  WHERE b_cat_ID = 4;
5  GO
```

Оператор *TRUNCATE TABLE* полностью очищает таблицу и не допускает условного удаления. Он аналогичен оператору *DELETE* без условия *WHERE* и ограничения *TOP*. Удаление происходит быстрее. Синтаксис оператора:

TRUNCATE TABLE

[[<база данных>.] <схема>.] { <таблица> }

Необходимо учитывать следующее. Если на очищаемую таблицу ссылаются внешние ключи других таблиц, то попытка удаления будет завершена с ошибкой. Удаления не произойдет, даже если в дочерней таблице нет строк.

Практическая работа

При выполнении задания необходимо для своей предметной области:

- заполнить согласованными данными таблицы базы данных;
- при необходимости исправить введенную информацию;
- составить отчет.

Пример выполнения задания

Заполнение таблицы *Catalogs* с использованием одного оператора *INSERT* для каждой строки таблицы. Оператор *SET NOCOUNT ON* подавляет сообщения о количестве обработанных каждой командой строк.

```
1  /*  Заполнение таблицы Catalogs  */
2  USE Book;
3  GO
4  SET NOCOUNT ON;
5  DELETE FROM Catalogs;
6  INSERT INTO Catalogs(cat_ID, cat_name) VALUES (1, 'Программирование');
7  INSERT INTO Catalogs(cat_ID, cat_name) VALUES (2, 'Интернет');
8  INSERT INTO Catalogs(cat_ID, cat_name) VALUES (3, 'Базы данных');
9  INSERT INTO Catalogs(cat_ID, cat_name) VALUES (4, 'Сети');
10 INSERT INTO Catalogs(cat_ID, cat_name) VALUES (5, 'Мультимедиа');
11 GO
```

Заполнение таблицы *Users* с использованием одного оператора *INSERT* для добавления всех строк таблицы. Имена столбцов таблицы в операторе не указаны.

```
1  /*  Заполнение таблицы Users */
2  USE Book;
3  GO
4  SET NOCOUNT ON;
5  DELETE FROM Users;
6  INSERT INTO Users VALUES
7  (1, 'Александр', 'Валерьевич', 'Иванов', '58-98-78', 'ivanov@email.ru', 'active'),
8  (2, 'Сергей', 'Иванович', 'Лосев', '90-57-77', 'losev@email.ru', 'passive'),
9  (3, 'Нина', 'Сергеевна', 'Грачева', '95-66-61', 'gracheva@email.ru', 'active'),
10 (4, 'Максим', 'Петрович', 'Кузнецов', NULL, 'kuznetsov@email.ru', 'active'),
11 (5, 'Анатолий', 'Юрьевич', 'Петров', NULL, NULL, 'lock'),
12 (6, 'Александр', 'Александрович', 'Корнеев', '89-78-36', 'korneev@email.ru', 'gold');
13 GO
```

Заполнение таблицы *Books* с использованием одного оператора *INSERT* для добавления всех строк таблицы.

```
1  /*  Заполнение таблицы Books */
2  USE Book;
3  GO
4  SET NOCOUNT ON;
5  DELETE  FROM  Books;
6  INSERT INTO Books(book_ID, b_name, b_author, b_year, b_price, b_count, b_cat_ID) VALUES
7  (1, 'Java. Руководство для начинающих', 'Шилдт Г.', 2012, 636.00, 10, 1),
8  (2, 'Microsoft Visual Studio 2010', 'Голощапов А.Л.', 2011, 472.00, 2, 1),
9  (3, 'Язык программирования C++. Лекции и упражнения', 'Прата С.', 2012, 1674.00, 4, 1),
10 (4, 'Язык программирования C# 5.0 и платформа .NET 4.5', 'Троелсен Э.', 2013, 1672.00, 1, 1),
11 (5,'Язык C#. Базовый курс', 'Подбельский В.В.', 2011, 534.00, 6, 1),
12 (6, 'Delphi. Программирование для Windows, OS X, iOS и Android', 'Осипов Д.', 2014, 440.00, 6, 1),
13 (7, 'PHP, MySQL, XML. Программирование для  интернета', 'Бенкен Е.С.', 2011, 239.00, 5, 1),
14 (8, 'Совершенный код', 'Макконнелл С.', 2014, 814.80, 1, 1),
15 (9, 'C#. Программирование на языке высокого уровня', 'Павловская Т.А.', 2013, 442.00, 12, 1),
16 (10, 'Новейшая энциклопедия. Компьютер и Internet 2014', 'Леонтьев В.П.', 2014, 468.00, 4, 2),
17 (11, 'Internet: поиск информации и продвижение сайтов', 'Байков В.Д.', 2012, 395.00, 2, 2),
18 (12, 'Internet у вас дома', 'Березин С.', 2012, 578.80, 6, 2),
19 (13, 'Интернет. Быстрый старт', 'Шапошников И.В.', 2012, 395.00, 4, 2),
20 (14, 'Популярные интернет-браузеры', 'Маринин С.А.', 2007, 82.00, 6, 2),
21 (15, 'Анонимность и безопасность в Интернете', 'Колисниченко Д.Н.', 2012, 176.00, 5, 2),
22 (16, 'Основы использования и проектирования баз данных', 'Илюшечкин В.М.', 2010, 355.00, 2, 3),
23 (17, 'SQL для чайников',   'Тейлор А.Г.', 2012, 400.00, 6, 3),
24 (18, 'Раскрытие тайн SQL', 'Оппель Э.', 2007, 200.00, 3, 3),
25 (19, 'Microsoft SQL Server 2012', 'Бондарь А.Г.', 2013, 529.00, 6, 3),
26 (20, 'Локальные сети. Модернизация и поиск неисправностей', 'Поляк-Брагинский А.В.',
27         2012, 456.40, 6, 4),
28 (21, 'Компьютерные сети и сетевые технологии', 'Кузьменко Н.Г.', 2013, 421.70, 4, 4),
29 (22, 'Практическое руководство системного администратора', 'Кенин А.М.', 2013, 383.10, 5, 4),
30 (23, 'TCP/IP. Архитектура. Протоколы. Реализация', 'Фейт С.', 2014, 464.00, 3, 4),
31 (24, 'Технологии современных сетей Ethernet', 'Смирнова Е.В.' , 2012, 245.50, 8, 4),
32 (25, 'Цифровая обработка 2D- и 3D-изображений', 'Красильников Н.Н.', 2011, 558.40, 20,5),
33 (26, 'Самоучитель Adobe Illustrator CS6', 'Тучкевич Е.И.', 2014, 353.00, 15, 5),
34 (27, 'Photoshop для начинающих', 'Заика А.', 2013, 148.00, 10, 5),
35 (28, 'Звуковой дизайн в видеоиграх', 'Деникин А.А.', 2012, 888.00, 20, 5),
36 (29, 'Слушаем музыку и смотрим фильмы на ПК', 'Дронов В.А.', 2011, 252.50, 12, 5),
37 (30, 'Звукозапись на компьютере',  'Петелин Р.Ю.', 2010, 392.00, 8, 5);
38 GO
```

Заполнение таблицы *Orders* с использованием одного оператора *INSERT* для каждой строки таблицы. Имена столбцов таблицы в операторах не указаны.

```
1  /*  Заполнение таблицы Orders */
2  USE Book;
3  GO
4  SET NOCOUNT ON;
5  DELETE FROM Orders;
6  INSERT INTO Orders VALUES (1, 3, 8, '04/01/2014 10:39:38',  1);
7  INSERT INTO Orders VALUES (2, 6, 10,'10/02/2014 09:40:29', 2);
8  INSERT INTO Orders VALUES (3, 1, 20,'18/02/2014 13:41:05', 4);
9  INSERT INTO Orders VALUES (4, 4, 20,'10/03/2014 18:20:00', 1);
10 INSERT INTO Orders VALUES (5, 3, 20,'17/03/2014 19:15:36', 1);
11 GO
```

Практическое занятие № 5
Создание запросов на выборку

Теоретические сведения

Рассмотрим следующие вопросы:

- выборка данных из таблиц с помощью оператора *SELECT*;
- использование в запросах операторов и встроенных функций SQL Server;
- создание вложенных запросов.

Для извлечения строк из одной или нескольких таблиц базы данных (без их изменения) используется оператор *SELECT*. Результирующий набор записей возвращается в виде таблицы, его можно использовать для создания новой таблицы.

Оператор *SELECT* может быть представлен в общей форме:

SELECT [ALL | DISTINCT] [TOP (<количество>)] <список выборки>
FROM <источники>
[WHERE <условия>]
[GROUP BY <группа> [HAVING <групповые_условия>]]
[ORDER BY <список полей>];

Все части оператора, кроме списка выборки, являются необязательными, но если они есть, то должны указываться в том порядке, в каком приведены.

Базовый запрос *SELECT*. Содержит две части:

- *SELECT [ALL | DISTINCT]* – разделенные запятыми поля, которые должны быть возвращены в списке результатов. Если необходимо выбрать все поля таблицы (представления), то вместо имен может быть поставлена звездочка (*). Ключевое слово *DISTINCT* позволяет исключить дубликаты в результатах;

• *FROM* – таблицы (представления), из которых выбираются данные. Вместо настоящего имени таблицы или представления может использоваться синоним (псевдоним для таблицы или представления).

Список столбцов в операторе *SELECT* используют, чтобы изменить порядок следования столбцов в результирующей таблице или выбрать часть столбцов:

```
1  USE Book;
2  GO
3  SELECT cat_name, cat_ID FROM Catalogs;
4  GO
```

100 %

Результаты | Сообщения

	cat_name	cat_ID
1	Программирование	1
2	Интернет	2
3	Базы данных	3
4	Сети	4
5	Мультимедиа	5

Пример использования ключевого слова *DISTINCT*:

```
1  USE Book;
2  GO
3  SELECT DISTINCT u_status FROM Users;
4  GO
```

100 %

Результаты | Сообщения

	u_status
1	active
2	gold
3	lock
4	passive

Можно указывать в запросе *псевдоним поля* после ключевого слова *AS*:

```
1  USE Book;
2  GO
3  SELECT cat_name AS [Код каталога], cat_ID AS [Наименование каталога]
4  FROM Catalogs;
5  GO
```

100 %

Результаты | Сообщения

	Код каталога	Наименование каталога
1	Программирование	1
2	Интернет	2
3	Базы данных	3
4	Сети	4
5	Мультимедиа	5

Псевдоним не сохраняется после выполнения запроса и не может использоваться в большинстве других частей запроса. Допускается пропустить слово *AS*, оставляя пробел между именем и псевдонимом поля.

Сортировка результатов (*ORDER BY*). Чтобы отсортировать значения по одному из столбцов, необходимо в секции *ORDER BY* указать этот столбец:

```
1  USE Book;
2  GO
3  SELECT * FROM Orders
4  ORDER BY o_user_ID;
5  GO
```

100 %

Результаты | Сообщения

	order_ID	o_user_ID	o_book_ID	o_time	o_number
1	3	1	20	2014-02-18 13:41:05.000	4
2	1	3	8	2014-01-04 10:39:38.000	1
3	5	3	20	2014-03-17 19:15:36.000	1
4	4	4	20	2014-03-10 18:20:00.000	1
5	2	6	10	2014-02-10 09:40:29.000	2

По умолчанию сортировка производится по возрастанию, но если после имени поля добавить ключевое слово *ASC* (по возрастанию) или *DESC* (по убыванию), то сортировка осуществится соответствующим образом.

```
1  USE Book;
2  GO
3  SELECT o_time FROM Orders
4  ORDER BY o_time DESC;
5  GO
```

100 %

Результаты | Сообщения

	o_time
1	2014-03-17 19:15:36.000
2	2014-03-10 18:20:00.000
3	2014-02-18 13:41:05.000
4	2014-02-10 09:40:29.000
5	2014-01-04 10:39:38.000

Сортировка задействует много ресурсов, поэтому большинство СУБД использует индексы и обращается к алгоритму сортировки в крайнем случае. После *ORDER BY* можно использовать псевдонимы, но в этом случае индексы использовать нельзя (они ссылаются на имена полей). Обработчик запросов по-

лучит большую гибкость, если после *ORDER BY* будут указаны только имена полей.

Сортировку записей можно производить по нескольким столбцам (их указывают после слов *ORDER BY* через запятую). Поля, указанные в *ORDER BY*, не обязательно должны быть включены в список результатов запроса (лучше включить).

Целесообразно расположить поля, по которым производится сортировка, первыми в результатах запроса и в том же порядке в списке *ORDER BY*:

```
1  USE Book;
2  GO
3  SELECT o_user_ID, order_ID, o_book_ID, o_time, o_number FROM Orders
4  ORDER BY o_user_ID, order_ID;
5  GO
```

100 %

Результаты | Сообщения

	o_user_ID	order_ID	o_book_ID	o_time	o_number
1	1	3	20	2014-02-18 13:41:05.000	4
2	3	1	8	2014-01-04 10:39:38.000	1
3	3	5	20	2014-03-17 19:15:36.000	1
4	4	4	20	2014-03-10 18:20:00.000	1
5	6	2	10	2014-02-10 09:40:29.000	2

Ограничение выборки (*TOP*). Секция *TOP* позволяет при выполнении запроса вернуть определенное число записей.

Следующий запрос извлекает первые 5 записей таблицы *Books*, отсортированных в обратном порядке по полю *b_count*:

```
1  USE Book;
2  GO
3  SELECT TOP 5 b_count, book_ID FROM Books
4  ORDER BY b_count DESC;
5  GO
```

100 %

Результаты | Сообщения

	b_count	book_ID
1	20	25
2	20	28
3	15	26
4	12	9
5	12	29

Можно вернуть определенный процент первых записей, используя ключевое слово *PERCENT*. Запрос возвращает первые 10 % записей (3 книги):

```
1  USE Book;
2  GO
3  SELECT TOP 10 PERCENT b_count, book_ID FROM Books
4  ORDER BY b_count DESC;
5  GO
```

100 %

Результаты | Сообщения

	b_count	book_ID
1	20	25
2	20	28
3	15	26

Аргументом секции *TOP* может быть переменная или выражение в скобках. В следующем примере используется переменная *@RowCount*:

```
1  USE Book;
2  GO
3  DECLARE @RowCount INT;
4  SET @RowCount = 4;
5  SELECT TOP (@RowCount) b_count, book_ID FROM Books
6  ORDER BY b_count DESC;
7  GO
```

100 %

Результаты | Сообщения

	b_count	book_ID
1	20	25
2	20	28
3	15	26
4	12	29

Фильтрация строк (*WHERE*). Для выбора записей, удовлетворяющих определенным критериям поиска, можно использовать конструкцию *WHERE*:

```
1  USE Book;
2  GO
3  SELECT user_ID, u_surname FROM Users
4  WHERE u_status = 'active';
5  GO
```

100 %

Результаты | Сообщения

	user_ID	u_surname
1	1	Иванов
2	3	Грачева
3	4	Кузнецов

Группировка записей *(GROUP BY).* Конструкция *GROUP BY* позволяет группировать извлекаемые строки. Полезна в сочетании с агрегирующими функциями (табл. 4), которые вычисляют одно значение для каждой создаваемой группы. Допускается использование агрегирующих функций в запросах без группировки:

Таблица 4

Функция	Описание
AVG ([DISTINCT] столбец)	Вычисляет среднее значение столбца или выражения. Слово *DISTINCT* позволяет обрабатывать только уникальные значения столбца
COUNT( )	Подсчитывает число значений в столбце. Форма *COUNT (столбец)* возвращает число записей, для которых значение *столбца* отлично от *NULL*. Форма *COUNT(*)* возвращает общее число строк в таблице независимо от того, принимает какой-либо столбец значение *NULL* или нет. Форма *COUNT([DISTINCT] столбец)* позволяет подсчитывать только уникальные значения столбца
MAX ([DISTINCT] столбец)	Находит максимальное значение в столбце. Слово *DISTINCT* позволяет обрабатывать только уникальные значения столбца
MIN ([DISTINCT] столбец)	Находит минимальное значение в столбце. Слово *DISTINCT* позволяет обрабатывать только уникальные значения столбца
SUM ([DISTINCT] столбец)	Находит общую сумму значений в столбце. Слово *DISTINCT* позволяет обрабатывать только уникальные значения столбца

Функция *AVG* используется внутри функции *ROUND* для получения результата в рублях и копейках. Определение средней цены книги:

```
1  USE Book;
2  GO
3  SELECT ROUND(AVG(b_price), 2) AS avg_price
4  FROM Books;
5  GO
```

100 %

Результаты | Сообщения

	avg_price
1	502,17

Определение числа строк в таблице *Books*:

```
1  USE Book;
2  GO
3  SELECT COUNT(*) AS num_books
4  FROM Books;
5  GO
```

100 %

Результаты | Сообщения

	num_books
1	30

Определение количества разных статусов пользователей:

```
USE Book;
GO
SELECT COUNT(DISTINCT(u_status)) AS num_status
FROM Users;
GO
```

100 %

Результаты | Сообщения

	num_status
1	4

Определение размера самого длинного и самого короткого названия книги, из числа имеющихся в интернет-магазине:

```
USE Book;
GO
SELECT MIN(LEN(b_name)) AS min_len,
       MAX(LEN(b_name)) AS max_len
FROM Books;
GO
```

100 %

Результаты | Сообщения

	min_len	max_len
1	15	57

Использование функций в конструкции *WHERE* приведет к ошибке. Попробуем извлечь из таблицы *Catalogs* запись с максимальным значением поля *cat_ID*:

```
USE Book;
GO
SELECT * FROM Catalogs
WHERE cat_ID = MAX(cat_ID);
GO
```

100 %

Сообщения

```
Сообщение 147, уровень 15, состояние 1, строка 2
Статистическое выражение не может использоваться в предложении WHERE,
если оно не содержится во вложенном запросе предложения HAVING или в
списке выбора, и столбец, подвергаемый статистической обработке, не
является внешней ссылкой.
```

Решение задачи следует искать в использовании конструкции *ORDER BY*:

```
USE Book;
GO
SELECT TOP 1 * FROM Catalogs
ORDER BY cat_ID DESC;
GO
```

100 %

Результаты | Сообщения

	cat_ID	cat_name
1	5	Мультимедиа

Запрос, перечисляющий все каталоги с указанием количества наименований книг в каждом каталоге:

```
1  USE Book;
2  GO
3  SELECT b_cat_ID AS catalog, COUNT(*) AS count
4  FROM Books
5  GROUP BY b_cat_ID;
6  GO
```

100 %

Результаты | Сообщения

	catalog	count
1	1	9
2	2	6
3	3	4
4	4	5
5	5	6

СУБД сортирует строки, выбранные запросом, по столбцам, перечисленным после *GROUP BY* (если не будет указан другой порядок в секции *ORDER BY*).

> Внимание! Когда запрос включает агрегирующую функцию, каждое поле в результатах запроса должно включаться либо в агрегирующую функцию, либо в выражение *GROUP BY*.

Чтобы ограничить выборку по результату функции (например, выбрать каталоги, где число товарных позиций больше 5), вместо *WHERE* используется *HAVING*:

```
1  USE Book;
2  GO
3  SELECT b_cat_ID, COUNT(b_cat_ID) AS total
4  FROM Books
5  GROUP BY b_cat_ID HAVING COUNT(b_cat_ID) > 5
6  ORDER BY b_cat_ID;
7  GO
```

100 %

Результаты | Сообщения

	b_cat_ID	total
1	1	9
2	2	6
3	5	6

В случае использования ключевого слова *HAVING* сначала происходит группировка таблицы и лишь затем выборка с применением условия. Допускается использование условия *HAVING* без группировки *GROUP BY*.

Использование функций. Для решения задач при выборке удобны встроенные функции SQL Server. Каждая функция имеет уникальное имя и может иметь несколько аргументов (перечисляются через запятую в скобках). Если аргументы отсутствуют, скобки все равно следует указывать. Пробелы между именем функции и скобками недопустимы. Число доступных для использования функций велико, в приложениях приведены наиболее полезные из них.

Оператор *SELECT* может использоваться вообще без таблиц. В такой форме *SELECT* применяют как калькулятор:

```
1  SELECT 2+3 AS [Сумма 2+3];
2  GO
```

100 %

Результаты | Сообщения

	Сумма 2+3
1	5

Возможность выполнять математические расчеты на уровне *SELECT* позволяет проводить финансовый анализ значений таблиц и отображать полученные результаты в отчетах. В выражениях можно использовать скобки, чтобы изменять порядок вычислений.

Операторы. Под операторами подразумеваются конструкции языка, которые производят преобразование данных (операндов).

В SQL Server используются три типа операторов:

- арифметические;
- операторы сравнения;
- логические.

Арифметические операторы. Обычные арифметические операторы: сложение (+), вычитание (–), умножение (*), деление (/) и взятие остатка от деления (%).

Операторы сравнения. Используются для сравнения двух значений и возвращают значение *TRUE* или *FALSE*. Сравниваемые значения могут быть константами, значениями полей базы данных или их комбинацией. В проверках после *WHERE* можно использовать следующие операторы сравнения: равно (=), не равно (!=, <>), меньше (<), больше (>), меньше или равно, не больше (<=, !>), больше или равно, не меньше (>=, !<). Если один или оба сравниваемых операнда имеют значение *NULL*, то результатом сравнения будет *NULL*.

Логические операторы. SQL Server поддерживает все обычные логические операции, которые можно применять к выражениям, возвращающим логические значения: дизъюнкцию (оператор *OR*), конъюнкцию (оператор *AND*), отрицание (оператор *NOT*).

Получим список книг из каталога «Программирование», цена на которые меньше 500 руб.

```
1  USE Book;
2  GO
3  SELECT * FROM Books
4  WHERE b_cat_ID = 1 AND b_price < 500;
5  GO
```

100 %

Результаты | Сообщения

	book_ID	b_name	b_author	b_year	b_price	b_count	b_cat_ID
1	2	Microsoft Visual Studio 2010	Голощапов А.Л.	2011	472,00	2	1
2	6	Delphi. Программирование для Windows, OS X, iOS и...	Осипов Д.	2014	440,00	6	1
3	7	PHP, MySQL, XML. Программирование для интернета	Бенкен Е.С.	2011	239,00	5	1
4	9	C#. Программирование на языке высокого уровня	Павловская Т.А.	2013	442,00	12	1

Если операторы *AND* и *OR* встречаются в одном запросе, следует для контроля использовать скобки. Условия в скобках всегда возвращают значение первыми.

Логический оператор *IS NULL* используется для того, чтобы определить, является данное значение значением *NULL* или нет. Следующий запрос выдает список клиентов, у которых отсутствуют контактные телефоны:

```
1  USE Book;
2  GO
3  SELECT * FROM Users
4  WHERE u_phone IS NULL;
5  GO
```

100 %

Результаты | Сообщения

	user_ID	u_name	u_patronymic	u_surname	u_phone	u_email	u_status
1	4	Максим	Петрович	Кузнецов	NULL	kuznetsov@email.ru	active
2	5	Анатолий	Юрьевич	Петров	NULL	NULL	lock

Логический оператор *BETWEEN … AND* используется для того, чтобы определить, попадает ли данное значение в заданный интервал. Интервал задается минимальным и максимальным значениями, причем минимальное и максимальное значения включаются в интервал. Следующий запрос выдает список заказов, сделанных в марте 2014 года:

```
1  USE Book;
2  GO
3  SELECT * FROM Orders
4  WHERE o_time BETWEEN '01/03/2014' AND '31/03/2014';
5  GO
```

100 %

Результаты | Сообщения

	order_ID	o_user_ID	o_book_ID	o_time	o_number
1	4	4	20	2014-03-10 18:20:00.000	1
2	5	3	20	2014-03-17 19:15:36.000	1

Логический оператор *LIKE* используется для сравнения строки с маской, возвращая значение *TRUE*, если значение строки подходит под маску, и *FALSE*, если не подходит. В маске можно использовать специальные символы:

- подчеркивание (_) – может использоваться как маска позиции (этот символ совпадает с любым символом, находящимся в заданной позиции);
- процент (%) – может использоваться как маска, независимая от позиции (символ совпадает с любым количеством любых символов).

Следующий запрос выдает список книг, названия которых содержат союз «и»:

```
1  USE Book;
2  GO
3  SELECT * FROM Books
4  WHERE b_name LIKE '% и %';
5  GO
```

100 %

Результаты | Сообщения

	book_ID	b_name	b_author	b_year	b_price	b_count	b_cat_ID
1	3	Язык программирования C++. Лекции и упражнения	Прата С.	2012	1674,00	4	1
2	4	Язык программирования C# 5.0 и платформа .NET 4.5	Троелсен Э.	2013	1672,00	1	1
3	6	Delphi. Программирование для Windows, OS X, iOS и A...	Осипов Д.	2014	440,00	6	1
4	10	Новейшая энциклопедия. Компьютер и Internet 2014	Леонтьев В.П.	2014	468,00	4	2
5	11	Internet: поиск информации и продвижение сайтов	Байков В.Д.	2012	395,00	2	2
6	15	Анонимность и безопасность в Интернете	Колисниченко Д.Н.	2012	176,00	5	2
7	16	Основы использования и проектирования баз данных	Илюшечкин В.М.	2010	355,00	2	3
8	20	Локальные сети. Модернизация и поиск неисправнос...	Поляк-Брагинский А.В.	2012	456,40	6	4
9	21	Компьютерные сети и сетевые технологии	Кузьменко Н.Г.	2013	421,70	4	4
10	25	Цифровая обработка 2D- и 3D-изображений	Красильников Н.Н.	2011	558,40	20	5
11	29	Слушаем музыку и смотрим фильмы на ПК	Дронов В.А.	2011	252,50	12	5

Логический оператор *IN* используется для проверки условия, принадлежит ли значение заданному множеству. В качестве множества может использоваться список литеральных значений (выражений) или подзапрос. Следующий запрос выдает список книг, входящих в каталоги «Интернет» и «Сети»:

```
1  USE Book;
2  GO
3  SELECT * FROM Books
4  WHERE b_cat_ID IN (2, 4);
5  GO
```

100 %

Результаты | Сообщения

	book_ID	b_name	b_author	b_year	b_price	b_count	b_cat_ID
1	10	Новейшая энциклопедия. Компьютер и Internet 2014	Леонтьев В.П.	2014	468,00	4	2
2	11	Internet: поиск информации и продвижение сайтов	Байков В.Д.	2012	395,00	2	2
3	12	Internet у вас дома	Березин С.	2012	578,80	6	2
4	13	Интернет. Быстрый старт	Шапошников И.В.	2012	395,00	4	2
5	14	Популярные интернет-браузеры	Маринин С.А.	2007	82,00	6	2
6	15	Анонимность и безопасность в Интернете	Колисниченко Д.Н.	2012	176,00	5	2
7	20	Локальные сети. Модернизация и поиск неисправност...	Поляк-Брагинский А.В.	2012	456,40	6	4
8	21	Компьютерные сети и сетевые технологии	Кузьменко Н.Г.	2013	421,70	4	4
9	22	Практическое руководство системного администратора	Кенин А.М.	2013	383,10	5	4
10	23	TCP/IP. Архитектура. Протоколы. Реализация	Фейт С.	2014	464,00	3	4
11	24	Технологии современных сетей Ethernet	Смирнова Е.В.	2012	245,50	8	4

Логический оператор *EXISTS* используется для проверки наличия строк во вложенном выборе. Если в списке результатов нет, устанавливается значение *FALSE*. Оператор *EXISTS* чаще всего используется с более сложной формой вложенного запроса, называемой коррелированным вложенным запросом, когда данные из внешнего запроса сравниваются со строками во вложенном запросе.

Существуют также *операторы составных запросов*. Их используют, когда нужно выполнять несколько запросов и объединять результаты в одном списке.

Оператор *UNION* добавляет строки списка результатов одного запроса к списку результатов другого, исключая повторяющиеся значения. Операция разрешена, если оба запроса совместимы для соединения, то есть содержат одинаковое количество полей и типы данных соответствующих полей совпадают.

Вывести список всех клиентов, имеющих статус *active* и статус *gold*:

```
1  USE Book;
2  GO
3  SELECT user_ID, u_name, u_surname FROM Users
4  WHERE u_status = 'active'
5  UNION
6  SELECT user_ID, u_name, u_surname FROM Users
7  WHERE u_status = 'gold'
8  GO
```

100 %

Результаты | Сообщения

	user_ID	u_name	u_surname
1	1	Александр	Иванов
2	3	Нина	Грачева
3	4	Максим	Кузнецов
4	6	Александр	Корнеев

Оператор *UNION ALL* работает так же, как *UNION*, но не исключает повторяющиеся значения из списка результатов.

Оператор *INTERSECT* находит значения, входящие в результаты как одного, так и другого запроса (пересечение значений). Следующий запрос выдает список клиентов со статусом *active*, которые делали заказы в марте 2014 года:

```
1  USE Book;
2  GO
3  SELECT DISTINCT user_ID, u_name, u_surname FROM Users, Orders
4  WHERE user_ID = o_user_ID AND o_time BETWEEN '01/03/2014' AND '31/03/2014'
5  INTERSECT
6  SELECT user_ID, u_name, u_surname FROM Users
7  WHERE u_status = 'active'
8  GO
```

100 %

Результаты | Сообщения

	user_ID	u_name	u_surname
1	3	Нина	Грачева
2	4	Максим	Кузнецов

Оператор *EXCEPT* возвращает отличия между двумя списками результатов путем выдачи значений первого списка, отсутствующих во втором. Следующий запрос выдает список клиентов со статусом *active*, которые не делали заказы в январе 2014 года:

```
1  USE Book;
2  GO
3  SELECT user_ID, u_name, u_surname FROM Users
4  WHERE u_status = 'active'
5  EXCEPT
6  SELECT DISTINCT user_ID, u_name, u_surname FROM Users, Orders
7  WHERE user_ID = o_user_ID AND o_time BETWEEN '01/01/2014' AND '31/01/2014'
8  GO
```

100 %

Результаты | Сообщения

	user_ID	u_name	u_surname
1	1	Александр	Иванов
2	4	Максим	Кузнецов

Запросы, которые обращаются одновременно к нескольким таблицам, называются многотабличными, или сложными, запросами.

Абсолютные ссылки на базы данных и таблицы. В запросе можно прямо указывать необходимые базу данных и таблицу. Например, можно представить ссылку на столбец *u_surname* из таблицы *Users* в виде *Users.u_surname*. Можно добавить имя схемы, которой принадлежит таблица.

Использование сложных запросов дает возможность избежать двусмысленности при указании источника необходимой информации.

Соединения. Соединение – операция реляционной БД, комбинирующая поля из одной или более таблиц в едином результате запроса. Соединение происходит, когда после *FROM* в запросе *SELECT* перечисляется более чем одно имя таблицы. При этом должен быть указан способ объединения таблиц – имена полей, которые должны быть совмещены.

Эквисоединение (внутреннее соединение) – соединение, в котором одно или более полей одной таблицы (обычно внешний ключ) совмещается с аналогичными полями другой таблицы (обычно первичный ключ), при условии равенства.

Есть два способа указать совмещаемые поля:

- с помощью ключевого слова *WHERE*;
- с помощью ключевого слова *JOIN*.

В условии *WHERE* сравнивается одно поле с другим. Если поля обеих таблиц имеют одинаковые имена, следует конкретизировать имена полей. Это относится и к остальному тексту запроса, включая список после *SELECT*. Простейшая форма конкретизации – имя таблицы с точкой, предшествующее имени поля.

В качестве примера соединения двух таблиц рассмотрим запрос, извлекающий из базы данных *Book* номера заказов вместе с фамилиями покупателей:

```
1  USE Book;
2  GO
3  SELECT Orders.order_ID, Users.u_surname
4  FROM Orders, Users
5  WHERE Orders.o_user_ID = Users.user_ID
6  ORDER BY Orders.order_ID;
7  GO
```

100 %

Результаты | Сообщения

	order_ID	u_surname
1	1	Грачева
2	2	Корнеев
3	3	Иванов
4	4	Кузнецов
5	5	Грачева

Объединение нескольких таблиц аналогично объединению двух таблиц. Например, необходимо выяснить, какому каталогу принадлежит товарная позиция из заказа, сделанного 10 февраля 2014 г. в 09:40:29:

```
1  USE Book;
2  GO
3  SELECT Catalogs.cat_name
4  FROM Catalogs, Books, Orders
5  WHERE Orders.o_time = '10/02/2014 09:40:29'
6  AND Catalogs.cat_ID = Books.b_cat_ID
7  AND Orders.o_book_ID = Books.book_ID;
8  GO
```

100 %

Результаты | Сообщения

	cat_name
1	Интернет

Допустимы *псевдонимы таблиц*. Они работают как псевдонимы полей, за исключением того, что ключевое слово *AS* не указывается (оставьте пробел между именем таблицы и псевдонимом в списке после *FROM*).

```
1  USE Book;
2  GO
3  SELECT O.order_ID, U.u_surname
4  FROM Orders O, Users U
5  WHERE O.o_user_ID = U.user_ID
6  ORDER BY O.order_ID;
7  GO
```

100 %

Результаты | Сообщения

	order_ID	u_surname
1	1	Грачева
2	2	Корнеев
3	3	Иванов
4	4	Кузнецов
5	5	Грачева

Для псевдонимов таблиц чаще всего применяют последовательные заглавные латинские буквы (*A, B, C* и т. д.). Если таблице присвоен псевдоним, то он должен использоваться вместо имени таблицы во всем тексте запроса.

Ключевое слово *JOIN* появилось в стандарте SQL-92. Оно объединяет список таблиц после *FROM* и условия соединения. Вместо *WHERE* используется параметр *ON*. Вместо ключевого слова *JOIN* можно использовать *INNER JOIN*.

```
1  USE Book;
2  GO
3  SELECT Orders.order_ID, Users.u_surname
4  FROM Orders JOIN Users
5  ON Orders.o_user_ID = Users.user_ID;
6  GO
```

100 %

Результаты | Сообщения

	order_ID	u_surname
1	1	Грачева
2	2	Корнеев
3	3	Иванов
4	4	Кузнецов
5	5	Грачева

Пример запроса, выдающего число товарных позиций в каталогах:

```
1  USE Book;
2  GO
3  SELECT Books.b_cat_ID AS [Код каталога],
4  COUNT(Books.book_ID) AS [Число позиций]
5  FROM Catalogs JOIN Books
6  ON Catalogs.cat_ID = Books.b_cat_ID
7  GROUP BY Books.b_cat_ID;
8  GO
```

100 %

Результаты | Сообщения

	Код каталога	Число позиций
1	1	9
2	2	6
3	3	4
4	4	5
5	5	6

Внешние соединения. Соединения, которые были рассмотрены выше, являются внутренними. В список результатов отбирались только строки, для которых было найдено совпадение в объединяемых таблицах. Однако в некоторых случаях требуется включить в результаты запроса несовпавшие строки.

Внешнее соединение включает несовпавшие строки как минимум одной из таблиц в результаты запроса. Соответственно, все значения, выбранные из другой таблицы, где соответствующая строка не найдена, являются неопределенными. Есть три основных типа внешних соединений:

- *левое внешнее соединение* – возвращает все строки в таблице, упомянутой слева от *JOIN*, совмещенные с любыми строками правой таблицы, которые имеют совпадения;
- *правое внешнее соединение* – возвращает все строки в таблице, упомянутой справа от *JOIN*, совмещенные с любыми строками левой таблицы, которые имеют совпадения;
- *полное внешнее соединение* – возвращает все строки из обеих таблиц. Полное внешнее соединение объединяет каждую строку одной таблицы с *NULL* или несколькими совпадающими строками другой таблицы.

Общий синтаксис внешнего соединения:

<таблица> {RIGHT | LEFT | FULL} [OUTER] JOIN <таблица> { ON условие | USING (имя_поля [, имя_поля]) }

Запрос, выдающий список клиентов и список заказов, сделанных клиентами:

```
1  USE Book;
2  GO
3  SELECT U.user_ID, U.u_surname, O.order_ID
4  FROM Users U JOIN Orders O
5  ON U.user_ID = O.o_user_ID
6  ORDER BY U.user_ID, O.order_ID;
7  GO
```

100 %

Результаты | Сообщения

	user_ID	u_surname	order_ID
1	1	Иванов	3
2	3	Грачева	1
3	3	Грачева	5
4	4	Кузнецов	4
5	6	Корнеев	2

Этот запрос не отражает наличие клиентов, не делавших заказы. Выходом является использование левого объединения (таблица *Users* должна быть левой). Отметим в списке результатов строки со значениями *NULL* – это клиенты, у которых нет заказов.

```
1  USE Book;
2  GO
3  SELECT U.user_ID, U.u_surname, O.order_ID
4  FROM Users U LEFT OUTER JOIN Orders O
5  ON U.user_ID = O.o_user_ID
6  ORDER BY U.user_ID, O.order_ID;
7  GO
```

100 %

Результаты | Сообщения

	user_ID	u_surname	order_ID
1	1	Иванов	3
2	2	Лосев	NULL
3	3	Грачева	1
4	3	Грачева	5
5	4	Кузнецов	4
6	5	Петров	NULL
7	6	Корнеев	2

Предыдущий запрос, переписанный в качестве правого внешнего запроса с условием *ON* (таблица *Users* должна быть правой):

```
1  USE Book;
2  GO
3  SELECT U.user_ID, U.u_surname, O.order_ID
4  FROM  Orders O RIGHT OUTER JOIN Users U
5  ON U.user_ID = O.o_user_ID
6  ORDER BY U.user_ID, O.order_ID;
7  GO
```

100 %

Результаты | Сообщения

	user_ID	u_surname	order_ID
1	1	Иванов	3
2	2	Посев	NULL
3	3	Грачева	1
4	3	Грачева	5
5	4	Кузнецов	4
6	5	Петров	NULL
7	6	Корнеев	2

Предыдущий запрос, переписанный в виде полного внешнего запроса, даст такой же результат, как и правое внешнее соединение (каждый заказ должен иметь своего клиента). Если бы существовали абстрактные заказы, полное внешнее соединение позволило бы увидеть клиентов, которые не делали заказы, и заказы, не принадлежащие клиентам.

Список клиентов, которые не сделали ни одного заказа. В запросе используется тот факт, что несовпадающие строки возвращаются как пустые значения. Поскольку значение поля *order_ID* в списке результатов всегда будет пустым, нет смысла его выбирать и отображать:

```
1  USE Book;
2  GO
3  SELECT U.user_ID, U.u_surname
4  FROM  Orders O RIGHT OUTER JOIN Users U
5  ON U.user_ID = O.o_user_ID
6  WHERE O.order_ID IS NULL;
7  GO
```

100 %

Результаты | Сообщения

	user_ID	u_surname
1	2	Посев
2	5	Петров

Рефлексивные соединения (самообъединение таблиц). Можно соединить таблицу саму с собой (когда интересуют связи между строками одной таблицы). Чаще всего рефлексивные соединения используются для отношений, первичный и внешний ключ которых находятся в одной таблице (рекурсивная связь). В базе данных *Book* таких отношений нет, поэтому рассмотрим упрощенный пример.

Пусть нужно выяснить, какие книги есть в каталоге, содержащем книгу с названием «Раскрытие тайн SQL». Необходимо найти в таблице *Books* номер каталога (*b_cat_ID*) с этой книгой, а затем посмотреть в таблице *Books* книги каталога. В этом запросе для таблицы *Books* определены два разных псевдонима (две отдельных таблицы *B1* и *B2*, которые должны содержать одни и те же данные). После этого они объединяются, как любые другие таблицы. Сначала ищется строка в таблице *B1*, а затем в таблице *B2* – строки с тем же значением номера каталога.

```
1  USE Book;
2  GO
3  SELECT B2.b_name
4  FROM  Books B1 JOIN Books B2
5  ON B1.b_name = 'Раскрытие тайн SQL'
6  AND B1.b_cat_ID = B2.b_cat_ID;
7  GO
```

100 %

Результаты | Сообщения

	b_name
1	Основы использования и проектирования баз данных
2	SQL для чайников
3	Раскрытие тайн SQL
4	Microsoft SQL Server 2012

Прочие соединения. Большинство соединений являются эквисоединениями. Однако иногда необходимо другое условие для сопоставления строк. Использование других соединений может привести к проблемам с производительностью (СУБД не будет использовать индексы, если условный оператор отличен от «=»).

Один из примеров – поиск дубликатов в таблице. Добавим в таблицу *Users* еще одного клиента с фамилией Иванов (потом его нужно удалить):

```
1  USE Book;
2  GO
3  INSERT INTO Users VALUES (7, 'Иван', 'Иванович', 'Иванов', '11-11-11', NULL, 'passive');
4  GO
```

Запрос для поиска дубликатов фамилий в таблице *Users*. Сравнение на неравенство (<>) поля *user_ID* предотвращает совпадение строки с самой собой:

```
1  USE Book;
2  GO
3  SELECT A.user_ID AS [Первый], B.user_ID AS [Второй]
4  FROM  Users A JOIN Users B
5  ON A.user_ID <> B.user_ID AND A.u_surname = B.u_surname;
6  GO
```

100 %

Результаты | Сообщения

	Первый	Второй
1	1	7
2	7	1

Перекрестное соединение. Это – стандартный синтаксис декартова произведения. Запрос, соединяющий таблицы *Orders* и Users в декартово произведение, можно записать следующим образом:

```
1  USE Book;
2  GO
3  SELECT O.order_ID, U.user_ID, U.u_surname
4  FROM  Orders O CROSS JOIN Users U
5  ORDER BY O.order_ID;
6  GO
```

Результатом являются 30 записей, представляющих все возможные комбинации заказов и клиентов (вывод не показан).

Вложенные запросы. Позволяют использовать возвращаемый результат в другом запросе. Обычно помещаются после *WHERE* в качестве способа ограничения строк, возвращаемых в списке результатов внешнего запроса.

В качестве вложенного запроса всегда выступает *SELECT*-запрос. Внешним запросом может быть запрос с участием любого SQL-оператора: *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *CREATE TABLE* и др. Вложенные запросы должны быть заключены в скобки.

Некоррелированный вложенный запрос – вложенный запрос, в котором внутренний запрос не обращается к содержащему его внешнему. Т. е. сначала можно выполнить внутренний запрос, а затем использовать полученный список результатов во внешнем запросе.

Пусть требуется вывести названия и цены товарных позиций из таблицы *Books* для каталога «Базы данных» таблицы *Catalogs*:

```
1  USE Book;
2  GO
3  SELECT b_name AS [Название книги], b_price AS [Цена]
4  FROM  Books
5  WHERE b_cat_ID = (SELECT cat_ID FROM Catalogs
6  WHERE cat_name = 'Базы данных')
7  ORDER BY b_price;
8  GO
```

100 %

Результаты | Сообщения

	Название книги	Цена
1	Раскрытие тайн SQL	200,00
2	Основы использования и проектирования баз данных	355,00
3	SQL для чайников	400,00
4	Microsoft SQL Server 2012	529,00

Аналогичный результат можно получить при помощи многотабличного запроса, но имеется ряд задач, которые решаются только при помощи вложенных запросов. Вложенный запрос может применяться не только с условием *WHERE*, но и в конструкциях *DISTINCT*, *GROUP BY*, *ORDER BY* и т. д. Различают вложенные запросы, возвращающие:

- одно значение;
- несколько строк.

В первом случае вложенный запрос возвращает скалярное значение или литерал, которое используется во внешнем запросе (подставляет результат на место своего выполнения). Например, необходимо определить название каталога, содержащего самую дорогую товарную позицию:

```
1  USE Book;
2  GO
3  SELECT cat_name AS [Каталог] FROM Catalogs
4  WHERE cat_ID = (SELECT b_cat_ID FROM Books
5  WHERE b_price = (SELECT MAX(b_price) FROM Books));
6  GO
```

100 %

Результаты | Сообщения

	Каталог
1	Программирование

Чаще всего вложенные запросы используются в операциях сравнения в условиях, которые задаются ключевыми словами *WHERE*, *HAVING* или *ON*.

Ключевое слово *ANY* может применяться с использованием любого оператора сравнения. Используется логика *ИЛИ*, т. е. достаточно, чтобы срабатывало хотя бы одно из многих условий. Запрос вида *WHERE X > ANY (SELECT Y …)* можно интерпретировать как «где *X* больше хотя бы одного выбранного *Y*». Соответственно, запрос вида *WHERE X < ANY (SELECT Y …)* интерпретируется как «где *X* меньше хотя бы одного выбранного *Y*».

Рассмотрим запрос, возвращающий имена и фамилии покупателей, совершивших хотя бы одну покупку:

```
1  USE Book;
2  GO
3  SELECT u_name AS [Имя], u_surname AS [Фамилия] FROM Users
4  WHERE user_ID = ANY (SELECT o_user_ID FROM Orders);
5  GO
```

100 %

Результаты | Сообщения

	Имя	Фамилия
1	Александр	Иванов
2	Нина	Грачева
3	Максим	Кузнецов
4	Александр	Корнеев

Ключевое слово *ALL* также может применяться с использованием любого оператора сравнения, но при этом используется логика *И*, т. е. должны срабатывать все условия. Запрос вида *WHERE X > ALL (SELECT Y …)* интерпретируется как «где *X* больше любого выбранного *Y*». Запрос вида *WHERE X < ALL (SELECT Y …)* интерпретируется как «где *X* меньше, чем все выбранные *Y*».

Рассмотрим запрос, возвращающий все товарные позиции, цена которых превышает среднюю цену каждого из каталогов:

```
1  USE Book;
2  GO
3  SELECT b_name AS [Название книги], b_price AS [Цена]
4  FROM  Books
5  WHERE b_price > ALL (SELECT AVG(b_price) FROM Books
6  GROUP BY b_cat_ID)
7  ORDER BY b_price;
8  GO
```

100 %

Результаты | Сообщения

	Название книги	Цена
1	Совершенный код	814,80
2	Звуковой дизайн в видеоиграх	888,00
3	Язык программирования C# 5.0 и платформа .NET 4.5	1672,00
4	Язык программирования C++. Лекции и упражнения	1674,00

Результирующая таблица, возвращаемая вложенным запросом, может не содержать ни одной строки. Для проверки этого факта могут использоваться ключевые слова *EXISTS* и *NOT EXISTS*.

Запрос, формирующий список покупателей, совершивших хотя бы одну покупку, можно переписать следующим образом:

```
1  USE Book;
2  GO
3  SELECT u_name AS [Имя], u_surname AS [Фамилия] FROM Users
4  WHERE EXISTS (SELECT * FROM Orders
5  WHERE Orders.o_user_ID = Users.user_ID);
6  GO
```

100 %

Результаты | Сообщения

	Имя	Фамилия
1	Александр	Иванов
2	Нина	Грачева
3	Максим	Кузнецов
4	Александр	Корнеев

Коррелированный вложенный запрос – вложенный запрос, в котором внутренний запрос обращается к значениям, полученным с помощью внешнего.

Такие запросы менее эффективны, чем некоррелированные, так как внутренний запрос должен выполняться для каждой строки, возвращенной внешним запросом (в случае некоррелированного вложенного запроса внутренний запрос выполняется один раз).

Фирма решила предоставить скидку всем клиентам, сделавшим в Интернет-магазине больше одного заказа. Необходимо найти таких клиентов:

```
1  USE Book;
2  GO
3  SELECT user_ID FROM Users U
4  WHERE 1 <
5  (SELECT COUNT(O.o_user_ID)
6  FROM Orders O
7  WHERE U.user_ID = O.o_user_ID);
8  GO
```

100 %

Результаты | Сообщения

	user_ID
1	3

Обратите внимание на псевдонимы таблиц во внешнем и внутреннем запросах и их использование после ключевого слова *WHERE* во внутреннем запросе. Это отличительный признак коррелированного вложенного запроса.

Внешний запрос возвращает список, содержащий *user_ID* из таблицы *Users*. Каждое возвращенное значение передается внутреннему запросу, который находит число заказов клиента. Если число заказов больше 1, условие *WHERE* возвращает значение *TRUE*, и соответствующее значение *user_ID* добавляется в список результатов запроса.

Практическая работа

При выполнении задания необходимо для своей предметной области:

- построить два простых запроса на выборку с использованием операторов и функций SQL Server;
- построить многотабличный запрос на выборку с использованием какого-либо соединения;
- построить запрос на выборку, содержащий вложенный запрос;
- составить отчет.

Пример выполнения задания

Создадим простой запрос на выборку к таблице *Books*, который выводит максимальную и минимальную цены товарных позиций, присваивая им соответственно псевдонимы *maximum* и *minimum*:

```
1  USE Book;
2  GO
3  SELECT MAX(b_price) AS maximum, MIN(b_price) AS minimum
4  FROM Books;
5  GO
```

100 %

Результаты | Сообщения

	maximum	minimum
1	1674,00	82,00

Создадим простой запрос на выборку к таблице *Books*, который выводит среднее значение цены книг для каждого каталога:

```
1  USE Book;
2  GO
3  SELECT b_cat_ID AS [Каталог], AVG(b_price) AS [Средняя цена]
4  FROM Books
5  GROUP BY b_cat_ID;
6  GO
```

100 %

Результаты | Сообщения

	Каталог	Средняя цена
1	1	769,3111
2	2	349,1333
3	3	371,00
4	4	394,14
5	5	431,9833

Создадим многотабличный запрос на выборку, который выводит коды покупателей магазина, сделавших менее двух покупок:

```
1  USE Book;
2  GO
3  SELECT Users.user_ID, COUNT(Orders.order_ID) AS total
4  FROM Users LEFT JOIN Orders
5  ON Users.user_ID = Orders.o_user_ID
6  GROUP BY Users.user_ID
7  HAVING COUNT(Orders.order_ID) < 2
8  ORDER BY total DESC, Users.user_ID ASC;
9  GO
```

100 %

Результаты | Сообщения

	user_ID	total
1	1	1
2	4	1
3	6	1
4	2	0
5	5	0

Создадим запрос на выборку с вложенным запросом, выводящим перечень книг, которые не заказывались покупателями (вывод оборван):

```
1  USE Book;
2  GO
3  SELECT book_ID, b_price, b_name FROM Books
4  WHERE NOT EXISTS ( SELECT * FROM Orders
5      WHERE Orders.o_book_ID = Books.book_ID);
6  GO
```

100 %

Результаты | Сообщения

	book_ID	b_price	b_name
1	1	636,00	Java. Руководство для начинающих
2	2	472,00	Microsoft Visual Studio 2010

Практическое занятие № 6
Представления

Теоретические сведения

Рассмотрим следующие вопросы:

- создание представлений с помощью оператора *CREATE VIEW*;
- модификация представлений с помощью оператора *ALTER VIEW*;
- удаление представлений с помощью оператора *DROP VIEW*.

Представление (*View*) – это запрос на выборку из одной или нескольких таблиц, которому присваивается уникальное имя и который можно сохранять или удалять из базы данных как хранимую процедуру.

Фактически представление является предварительно проанализированным оператором *SELECT*, который может интерпретироваться как таблица. Представление является логической, а не физической таблицей, однако в большинстве случаев работает как обычная таблица.

Достоинства представлений:

- *безопасность* – каждый пользователь имеет доступ к небольшому числу представлений, содержащих информацию, которую ему позволено знать;
- *простота запросов* – с помощью представления можно извлечь данные из нескольких таблиц и представить их как одну таблицу (запрос ко многим таблицам заменяется однотабличным запросом к представлению);
- *простота структуры* – представления позволяют создать для каждого пользователя собственную структуру базы данных (отображаются только те данные, которые ему нужны);
- *защита от изменений* – таблицы могут изменяться и переименовываться; представления дают возможность создавать виртуальные таблицы со старыми именами и структурой, позволяя избежать модификации приложений.

Недостатки представлений:

- *производительность* – представления создают видимость существования таблицы, и серверу приходится преобразовывать запрос к представлению в запрос к исходным таблицам (простой запрос к представлению становится сложным объединением);
- *ограничение на обновление* – когда пользователь пытается обновить строки представления, серверу необходимо обновить строки в исходных таблицах; это возможно только для простых представлений, сложные представления доступны только для выборки.

Поэтому не следует везде применять представления вместо исходных таблиц.

Представления могут создаваться только в той базе данных, которую вы используете, хотя к представлениям можно обращаться и из других БД – для этого необходимо ввести полностью определенное имя представления.

Создание представлений. Осуществляется следующим оператором:

CREATE VIEW [<имя_схемы>.] <имя_представления>
[(<столбец> [, ... n])]
[WITH < атрибут_представления> [, ... n]]
AS <оператор_select>
[WITH CHECK OPTION] ;

<атрибут_представления> ::=

{ [ENCRYPTION] [SCHEMABINDING] [VIEW_METADATA] }

Оператор создает представление *<имя_представления>* со столбцами, перечисленными в списке столбцов *<столбец> [, ... n]*, на основании запроса *<оператор_select>*. Если в операторе не заданы явно новые имена столбцов, то они наследуются из таблиц базы данных.

Параметры оператора управляют доступом к представлению со стороны пользователей. Так, например, при создании представления с параметром *ENCRYPT* никто, кроме его создателя, не сможет просмотреть его SQL-код.

Представления не могут содержать конструкции:

- *SELECT INTO*;
- *COMPUTE*;
- *ORDER BY* (но может использоваться при выборке из представления);
- ссылки на временные таблицы и табличные переменные.

Создадим представление *sum_orders_view*, которое дублирует таблицу *Orders* базы данных *Book*, но при этом вычисляет для каждого заказа общую сумму:

```
1  USE Book;
2  GO
3  CREATE VIEW sum_orders_view
4  AS
5  SELECT o.order_ID, o.o_user_ID, o.o_book_ID, o.o_time,
6         o.o_number, o.o_number * b.b_price AS [Сумма заказа]
7  FROM Orders o JOIN Books b
8  ON o.o_book_ID = b.book_ID;
9  GO
```

Можно выполнить запрос на выборку к представлению *sum_orders_view*:

```
1  USE Book;
2  GO
3  SELECT * FROM sum_orders_view;
4  GO
```

100 %

Результаты | Сообщения

	order_ID	o_user_ID	o_book_ID	o_time	o_number	Сумма заказа
1	1	3	8	2014-01-04 10:39:38.000	1	814,80
2	2	6	10	2014-02-10 09:40:29.000	2	936,00
3	3	1	20	2014-02-18 13:41:05.000	4	1825,60
4	4	4	20	2014-03-10 18:20:00.000	1	456,40
5	5	3	20	2014-03-17 19:15:36.000	1	456,40

При создании представления можно явно указать список столбцов, изменить их названия и порядок следования, например:

```
1  USE Book;
2  GO
3  CREATE VIEW cat_view_2
4  (Каталог, [Код каталога])
5  AS
6  SELECT cat_name, cat_ID FROM Catalogs;
```

Запрос на выборку к представлению *cat_view_2:*

```
1  USE Book;
2  GO
3  SELECT * FROM cat_view_2;
4
```

100 %

Результаты | Сообщения

	Каталог	Код каталога
1	Программирование	1
2	Интернет	2
3	Базы данных	3
4	Сети	4
5	Мультимедиа	5

Представления, не содержащие дополнительных столбцов, называются *вертикальными представлениями.* Они применяются для ограничения доступа пользователей к столбцам таблицы.

Горизонтальные представления делают видимыми только те строки, с которыми работают пользователи.

Чтобы в интернет-магазине каждый менеджер видел только те товарные позиции, за которые отвечает, можно создать представления для менеджеров. Учетные записи менеджеров следует лишить привилегии доступа к таблице и разрешить просматривать только свои представления.

Создадим представление *manager_view* для менеджера, работающего с каталогами «Интернет» и «Сети», используя секцию *WHERE* запроса в представлении:

```
1  USE Book;
2  GO
3  CREATE VIEW manager_view
4  AS
5  SELECT * FROM Books
6  WHERE b_cat_ID IN (SELECT cat_ID
7  FROM Catalogs
8  WHERE cat_name = 'Интернет' OR cat_name = 'Сети');
```

Запрос на выборку к представлению *manager_view* позволяет получить книги, входящие в указанные каталоги:

```
1  USE Book;
2  GO
3  SELECT b_name, b_price, b_count
4  FROM manager_view;
5  GO
```

100 %

Результаты | Сообщения

	b_name	b_price	b_count
1	Новейшая энциклопедия. Компьютер и Internet 2014	468,00	4
2	Internet: поиск информации и продвижение сайтов	395,00	2
3	Internet у вас дома	578,80	6
4	Интернет. Быстрый старт	395,00	4
5	Популярные интернет-браузеры	82,00	6
6	Анонимность и безопасность в Интернете	176,00	5
7	Локальные сети. Модернизация и поиск неисправностей	456,40	6
8	Компьютерные сети и сетевые технологии	421,70	4
9	Практическое руководство системного администратора	383,10	5
10	TCP/IP. Архитектура. Протоколы. Реализация	464,00	3
11	Технологии современных сетей Ethernet	245,50	8

Можно вставлять, удалять и обновлять данные в представлениях с помощью операторов *INSERT*, *UPDATE*, *DELETE*, соответственно будут изменены записи базовой таблицы. Однако если представление создано с параметром *WITH CHECK OPTION*, то вставка и обновление записей через представление возможны, если эти записи могут быть получены выборкой из представления.

Например, представление *active_users_view* возвращает список клиентов, имеющих статус *'active'*:

```
1   USE Book;
2   GO
3   CREATE VIEW active_users_view
4   AS
5   SELECT user_ID AS [Код], u_name AS [Имя], u_patronymic AS [Отчество],
6       u_surname AS [Фамилия], u_phone AS [Телефон], u_email AS [E-mail],
7       u_status AS [Статус]
8   FROM Users
9   WHERE u_status = 'active'
10  WITH CHECK OPTION;
11  GO
```

Из этого представления можно осуществлять выборку данных:

```
1   USE Book;
2   GO
3   SELECT * FROM active_users_view;
4   GO
```

100 %

Результаты | Сообщения

	Код	Имя	Отчество	Фамилия	Телефон	E-mail	Статус
1	1	Александр	Валерьевич	Иванов	58-98-78	ivanov@email.ru	active
2	3	Нина	Сергеевна	Грачева	95-66-61	gracheva@email.ru	active
3	4	Максим	Петрович	Кузнецов	NULL	kuznetsov@email.ru	active

Следующий оператор удаляет соответствующие записи из базовой таблицы *Users* через представление *active_users_view*:

```
1  USE Book;
2  GO
3  DELETE active_users_view
4  WHERE [Код] = 1;
5  GO
```

Однако попытка вставить данные в представление *active_users_view* с параметром *WITH CHECK OPTION* окончится неудачей, если значение поля *u_status* вставляемой записи отлично от *'active'*.

```
1  USE Book;
2  GO
3  INSERT INTO active_users_view VALUES (8, 'Анна', 'Михайловна', 'Шустова', '12-31-49', NULL, 'passive');
4  GO
```

100 %

Сообщения

```
Сообщение 550, уровень 16, состояние 1, строка 1
Ошибка при попытке вставки или обновления, поскольку целевое представление либо указывает WITH CHECK OPTION,
либо охватывает представление, которое указывает WITH CHECK OPTION, а одна или несколько строк, получающиеся
при операции, не определены в рамках ограничения CHECK OPTION.
Выполнение данной инструкции было прервано.
```

Если представление содержит столбцы из нескольких таблиц, то нельзя удалить записи из представления или обновить столбцы нескольких таблиц в одном операторе *DELETE* или *UPDATE*. То есть один оператор может модифицировать лишь одну из базовых таблиц. Эта проблема может быть решена поочередным обновлением таблиц.

Вставка в представлениях возможна лишь при условии, что все столбцы базовой таблицы, не включенные в представление, допускают значения *NULL*, или для них определены значения по умолчанию. Если представление содержит объединение, вставка допускается при условии, что все вставляемые столбцы принадлежат одной базовой таблице.

Нельзя обновлять, удалять или вставлять данные в представлении с секцией *DISTINCT*, поскольку *DISTINCT* дает временную рабочую таблицу и вместо настоящей базовой таблицы будет модифицироваться временная таблица.

Модификация представлений. Для модификации представлений используется оператор *ALTER VIEW*, имеющий следующий синтаксис:

ALTER VIEW [<имя_схемы>.] <имя_представления>
[(<столбец> [, ... n])]
[WITH < атрибут_представления> [, ... n]]
AS <оператор_select>
[WITH CHECK OPTION] ;

<атрибут_представления> ::=

{ [ENCRYPTION] [SCHEMABINDING] [VIEW_METADATA] }

Если представление было создано с параметрами *WITH ENCRYPTION* или *WITH CHECK OPTION*, то эти параметры остаются в модифицированном представлении, если они присутствуют в команде *ALTER VIEW*. Можно случайно отменить параметры *WITH ENCRYPTION* или *WITH CHECK OPTION*, если не включить их в оператор *ALTER VIEW*.

Удаление представлений. Выполняется с помощью оператора *DROP VIEW*, имеющего следующий синтаксис:

DROP VIEW [<схема>]. <имя_представления>

[, [<схема>]. <имя_представления>] … ;

Оператор позволяет удалить одно или несколько представлений. При удалении представления его определение удаляется из БД. Команду *DROP* нельзя отменить.

Практическая работа

При выполнении задания необходимо для своей предметной области:

- создать два представления и включить их в базу данных;
- сформировать запрос к одному из представлений;
- составить отчет.

Пример выполнения задания

Создадим вертикальное представление *list_user_view*, которое будет отображать фамилию и инициалы покупателей, скрывая другие поля.

```
1  USE Book;
2  GO
3  CREATE VIEW list_user_view
4  AS
5  SELECT CONCAT(u_surname, ' ',
6  SUBSTRING(u_name, 1, 1), '.',
7  SUBSTRING(u_patronymic, 1, 1), '.')
8  AS [ФИО]
9  FROM Users;
10 GO
```

```
1  USE Book;
2  GO
3  SELECT * FROM list_user_view
4  ORDER BY [ФИО];
5  GO
```

100 %

Результаты | Сообщения

	ФИО
1	Грачева Н.С.
2	Иванов А.В.
3	Корнеев А.А.
4	Кузнецов М.П.
5	Лосев С.И.
6	Петров А.Ю.

Создадим представление *price_view*, отображающее общую стоимость книг по каждому каталогу, и обратимся к этому представлению:

```
1  USE Book;
2  GO
3  CREATE VIEW price_view
4  AS
5  SELECT b_cat_ID, SUM(b_price * b_count) AS price
6  FROM Books
7  GROUP BY b_cat_ID;
8  GO
```

```
1  USE Book;
2  GO
3  SELECT * FROM price_view;
4  GO
```

100 %

Результаты | Сообщения

	b_cat_ID	price
1	1	28829,80
2	2	9086,80
3	3	6884,00
4	4	9696,70
5	5	41869,00

Сформируем запрос к таблице *Catalogs* и представлению *price_view:*

```
1  USE Book;
2  GO
3  SELECT Catalogs.cat_name, price_view.price
4  FROM price_view, Catalogs
5  WHERE price_view.b_cat_ID = Catalogs.cat_ID
6  ORDER BY Catalogs.cat_ID;
7  GO
```

100 %

Результаты | Сообщения

	cat_name	price
1	Программирование	28829,80
2	Интернет	9086,80
3	Базы данных	6884,00
4	Сети	9696,70

Запрос к представлению *price_view* (получение минимального и максимального значений общей стоимости книг в каталогах и общей стоимости всех книг):

```
1  USE Book;
2  GO
3  SELECT MIN(price) AS MIN,
4      MAX(price) AS MAX, SUM(price) AS SUM
5  FROM price_view;
6  GO
```

100 %

Результаты | Сообщения

	MIN	MAX	SUM
1	6884,00	41869,00	96366,30

Практическое занятие № 7
Хранимые процедуры

Теоретические сведения

Рассмотрим следующие вопросы:

– создание, изменение и удаление хранимых процедур;

– использование хранимых процедур.

Хранимые процедуры – предварительно откомпилированные процедуры, написанные на Transact-SQL и находящиеся в базе данных. Выполняются быстрее обычного Transact-SQL, поскольку хранятся в откомпилированном виде. Помогают изолировать пользователей от базовых табличных структур, скрыть особенности реализации каких-либо возможностей.

Хранимая процедура создается в текущей базе данных в указанной схеме. К ней могут обращаться другие процедуры, триггеры, клиентские программы. Процедура может получать входные параметры и возвращать выходные параметры. Хранимая процедура выполняется на сервере, что уменьшает сетевой трафик.

Минимально необходим для базы данных набор хранимых процедур, обеспечивающий выборку, вставку, модификацию и удаление данных для всех таблиц. Использование этих хранимых процедур позволяет:

- существенно ограничить возможности обычного пользователя в плане нанесения ущерба базе данных;
- обеспечить эффективный код приложения, работающего с базой данных.

Создание хранимой процедуры. Реализуется оператором:

CREATE PROCEDURE [<имя схемы>.] <имя процедуры>
[<параметр> [, <параметр>] ...]
[WITH <опция процедуры> [, <опция процедуры>] ...]
AS BEGIN { <оператор> | <блок операторов> } END ;

<параметр> ::= <имя> [<имя схемы>.]<тип данных>
[VARYING] [= <значение по умолчанию] [OUTPUT] [READONLY]
<опция процедуры>::= { ENCRIPTION | RECOMPILE | EXECUTE AS <контекст> }

Имя процедуры должно быть уникальным в схеме. Можно создать временную хранимую процедуру. *Локальная временная процедура* видна только в текущем соединении. Ее имя должно начинаться с символа #. *Глобальная временная процедура* видна всем соединениям. Имя такой процедуры должно начинаться с двух символов ##. Временные хранимые процедуры находятся в базе данных *tempdb*.

Если имя схемы не указано, то хранимая процедура создается в схеме базы данных по умолчанию.

Для процедуры можно указать параметры (до 2100). Их описания отделяются друг от друга запятыми. Имя параметра начинается с символа @. Должен быть указан тип данных параметра (системный или пользовательский). Ключевое слово *VARYING* указывает, что в качестве выходного параметра будет использоваться результирующий набор (применяется к типу данных *CURSOR*).

Для параметра можно указать значение по умолчанию, которое принимается, если при вызове процедуры значение для параметра не было задано. Значение по умолчанию помещается после знака равенства (=).

Ключевое слово *OUTPUT* (можно *OUT*) означает, что параметр является выходным. Его значение может быть использовано вызвавшей программой.

Ключевое слово *READONLY* означает, что значение параметра не может быть изменено внутри хранимой процедуры.

После ключевого слова *WITH* можно указать параметры процедуры:

- *ENCRIPTION* – система кодирует исходный текст процедуры. Другие пользователи не смогут просмотреть текст этой процедуры.
- *RECOMPILE* – при каждом вызове процедуры система будет перекомпилировать текст. Указывают, если произошли изменения в составе выбираемых данных, изменилась структура таблицы, появились новые индексы, чтобы оптимизатор запросов сформировал более эффективный способ выборки данных.
- *EXECUTE AS* – контекст безопасности вызова процедуры, т. е. определение тех, кто может вызывать данную хранимую процедуру.

Между необязательными ключевыми словами *BEGIN* и *END* помещаются операторы, осуществляющие все действия хранимой процедуры.

Процедуры не могут содержать команд *CREATE TABLE*, *ALTER TABLE*, *DROP TABLE*, *CREATE INDEX*, *DROP INDEX*, *TRUNCATE TABLE*, *UPDATE STATISTIC*.

Список и характеристики хранимых процедур можно просмотреть в Management Studio, раскрыв базу данных в *Обозревателе объектов*, а в ней – папки *Программирование* и *Хранимые процедуры*.

Можно просмотреть параметры процедуры (папка *Параметры*), вывести ее текст, щелкнув ПКМ по имени процедуры и выбрав в меню *Создать скрипт для хранимой процедуры | Используя CREATE | Новое окно редактора запросов*. В окне появится текст создания процедуры.

Изменение хранимой процедуры. Предполагает полное пересоздание существующей процедуры. Выполняется оператором *ALTER PROCEDURE*, синтаксис которого повторяет синтаксис оператора создания хранимой процедуры:

ALTER PROCEDURE [<имя схемы>.] <имя процедуры>
[<параметр> [, <параметр>] ...]
[WITH <опция процедуры> [, <опция процедуры>] ...]
AS BEGIN { <оператор> | <блок операторов> } END ;

Удаление хранимых процедур. Выполняется оператором:

DROP PROCEDURE [<имя схемы>.] <имя процедуры>
[, <имя схемы>.] <имя процедуры>] ... ;

В одном операторе можно удалить несколько хранимых процедур, разделяя их имена запятыми.

Использование хранимых процедур. Для вызова хранимой процедуры используется оператор *EXECUTE* (можно *EXEC*). В нем задается имя вызываемой процедуры и список значений передаваемых процедуре параметров. Если параметр является выходным, то он задается в виде имени локальной переменной, за которым должно идти ключевое слово *OUTPUT (OUT)*.

Если процедура возвращает целочисленное значение с помощью оператора *RETURN*, то в операторе *EXECUTE* перед ее именем записывают имя локальной переменной, которой будет присвоено возвращаемое значение, и знак равенства.

Рассмотрим пример хранимой процедуры, которую можно использовать для получения уникального значения первичного или уникального ключа в таблице:

```
1   USE Book;
2   GO
3   IF OBJECT_ID('proc_users', 'P') IS NOT NULL
4       DROP PROCEDURE proc_users;
5   GO
6   CREATE PROCEDURE proc_users
7       @cod INT OUTPUT
8   AS
9   BEGIN
10      SET @cod = (SELECT MAX(user_ID) FROM Users);
11  END
12  GO
```

100 %

Сообщения

Выполнение команд успешно завершено.

Вначале проверяем, существует ли процедура *proc_users* и при необходимости удаляем ее. Затем создаем саму процедуру, в которой описываем выходной целочисленный параметр *@cod*. Процедура присваивает выходному параметру максимальное существующее значение первичного ключа в таблице *Users*.

Создание хранимой процедуры должно быть единственным оператором в пакете. Проверка существования процедуры в БД должна завершаться командой *GO*. После нее начинается новый пакет.

Обращение к процедуре можно выполнить следующим образом:

```
1  USE Book;
2  GO
3  DECLARE @new_cod AS INT;
4  EXEC proc_users @new_cod OUTPUT;
5  PRINT @new_cod
6  GO
```

100 %

Сообщения

```
6
```

В этом примере мы использовали оператор *PRINT*, отображающий значение локальной переменной.

Следующая хранимая процедура выводит различные сообщения, в зависимости от пола клиента. Процедура имеет один параметр – код клиента. В ней проводится анализ имени, по результатам которого формируется обращение.

```
1  USE Book;
2  GO
3  IF OBJECT_ID('MyProc', 'P') IS NOT NULL
4      DROP PROCEDURE MyProc;
5  GO
6  CREATE PROCEDURE MyProc
7      @user_ID INT
8  AS
9  BEGIN
10     DECLARE @user AS VARCHAR(50), @FirstName VARCHAR(20),
11         @LastName VARCHAR(20);
12     SELECT @FirstName = u_name, @LastName = u_surname
13     FROM Users
14     WHERE user_ID = @user_ID;
15     SET @user =  @FirstName + ' ' + @LastName +'!'
16     IF (@FirstName LIKE '%а') OR (@FirstName LIKE '%я') OR
17        (@FirstName LIKE '%ь')
18         PRINT 'Дорогая '+ @user
19     ELSE
20         PRINT 'Дорогой ' + @user;
21 END
22 GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Примеры вызова процедуры:

```
1  USE Book;
2  GO
3  EXEC MyProc 1;
4  EXEC MyProc 2;
5  EXEC MyProc 3;
6  GO
```

100 %

Сообщения

```
Дорогой Александр Иванов!
Дорогой Сергей Лосев!
Дорогая Нина Грачева!
```

При вызове процедуры должны передаваться все параметры, для которых не определены значения по умолчанию. Параметры передаются как по имени (*имя = значение*), так и по позиции в списке. Если параметры передаются в том же порядке, как они следуют в хранимой процедуре, то можно не указывать их имена. Но если некоторый параметр передается по имени, все последующие параметры в списке тоже должны передаваться по имени.

Использование ключевого слова *EXECUTE* обязательно, если вызов хранимой процедуры не является первой командой пакета.

```
1  USE Book;
2  GO
3  MyProc 6;
4  GO
```

100 %

Сообщения

```
Дорогой Александр Корнеев!
```

Практическая работа

При выполнении задания необходимо для своей предметной области:

- написать несколько хранимых процедур и включить их в базу данных;
- составить отчет.

Пример выполнения задания

Создадим хранимые процедуры для работы с таблицей *Orders* (выборка, вставка, модификация и удаление данных).

1. Выборка данных (процедура *Order_Select*).

```
1  USE Book;
2  GO
3  IF OBJECT_ID( 'Order_Select', 'P') IS NOT NULL
4  DROP PROCEDURE Order_Select;
5  GO
6  CREATE PROCEDURE Order_Select (
7      @order_ID INT = NULL,
8      @user_ID INT = NULL,
9      @book_ID INT = NULL )
10 AS
11 BEGIN
12     SET NOCOUNT ON;
13     SELECT
14         Orders.order_ID,
15         Orders.o_user_ID,
16         Users.u_name + ' ' + Users.u_surname AS 'user_full_name',
17         Orders.o_book_ID,
18         Books.b_name,
19         Orders.o_time,
20         Orders.o_number
21     FROM  Orders
22     LEFT OUTER JOIN Users ON Users.user_ID = Orders.o_user_ID
23     LEFT OUTER JOIN Books ON Books.book_ID = Orders.o_book_ID
24     WHERE
25         ( @order_ID IS NULL OR Orders.order_ID = @order_ID ) AND
26         ( @user_ID IS NULL OR Orders.o_user_ID = @user_ID )  AND
27         ( @book_ID IS NULL OR Orders.o_book_ID = @book_ID )
28 END
29 GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Процедура *Order_Select* использует базовую таблицу *Orders* и связанные таблицы *Users* и *Books*. Три параметра процедуры – *@order_ID*, *@user_ID* и *@book_ID* обеспечивают достаточно широкие возможности по выборке данных.

Вызов процедуры *Order_Select* с использованием значений всех трех параметров по умолчанию (*NULL*) позволяет получить список всех заказов магазина.

```
1  USE Book;
2  GO
3  EXEC Order_Select;
4  GO
```

Вызов процедуры *Order_Select* с указанием *@order_ID* = 1 (значения остальных параметров по умолчанию – *NULL*) позволяет получить информацию о заказе с кодом 1.

```
1   USE Book;
2   GO
3   EXEC Order_Select 1, DEFAULT, DEFAULT;
4   GO
```

Вызов процедуры *Order_Select* с указанием *@user_ID* = 3 (значения остальных параметров по умолчанию – *NULL*) позволяет получить список всех заказов пользователя с кодом 3.

```
1   USE Book;
2   GO
3   EXEC Order_Select  DEFAULT, 3, DEFAULT;
4   GO
```

Вызов процедуры *Order_Select* с указанием *@book_ID* = 20 (значения остальных параметров по умолчанию – *NULL*) позволяет получить список всех заказов, содержащих книгу с кодом 20.

```
1   USE Book;
2   GO
3   EXEC Order_Select  DEFAULT, DEFAULT, 20;
4   GO
```

Вызов процедуры *Order_Select* с указанием *@user_ID* = 3 и *@book_ID* = 20 (значение параметра *@order_ID* по умолчанию – NULL) позволяет получить список заказов пользователя с кодом 3, содержащих книгу с кодом 20.

```
1   USE Book;
2   GO
3   EXEC Order_Select  DEFAULT, 3, 20;
4   GO
```

2. Вставка данных (процедура *Order_Insert*).

```
1   USE Book;
2   GO
3   IF OBJECT_ID('Order_Insert','P') IS NOT NULL
4   DROP PROCEDURE Order_Insert;
5   GO
```

```
6  CREATE PROCEDURE Order_Insert (
7      @order_ID INT,
8      @user_ID  INT,
9      @book_ID  INT,
10     @o_time  DATETIME,
11     @o_number INT )
12 AS
13 BEGIN
14     SET NOCOUNT ON;
15     INSERT INTO Orders (
16         order_ID, o_user_ID, o_book_ID,
17         o_time, o_number )
18     VALUES (
19         @order_ID, @user_ID, @book_ID ,
20         @o_time, @o_number);
21     RETURN @@ERROR
22 END
23 GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Вызов процедуры *Order_Insert* с указанием значений параметров *@order_ID*, *@user_ID*, *@book_ID*, *@o_time*, *@o_number* позволяет ввести новую строку в таблицу *Orders*.

```
1 USE Book;
2 GO
3 EXECUTE Book.dbo.Order_Insert 6, 1, 3, '2014-04-01 17:36:12', 1
4 GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

3. Обновление данных (процедура *Order_Update*).

```
1  USE Book;
2  GO
3  IF OBJECT_ID('Order_Update','P') IS NOT NULL
4  DROP PROCEDURE Order_Update;
5  GO
6  CREATE PROCEDURE Order_Update (
7      @order_ID  INT,
8      @user_ID  INT,
9      @book_ID  INT,
10     @o_time  DATETIME,
11 @o_number INT )
12 AS
13 BEGIN
14     SET NOCOUNT ON;
15     UPDATE Orders
```

```
16      SET
17          order_ID = @order_ID, o_user_ID = @user_ID,
18          o_book_ID = @book_ID, o_time = @o_time,
19          o_number = @o_number
20      WHERE order_ID = @order_ID;
21      RETURN @@ERROR
22  END
23  GO
```

Вызов процедуры *Order_Update* с указанием значений параметров *@order_ID*, *@user_ID*, *@book_ID*, *@o_time*, *@o_number* изменяет строку в таблице *Orders*.

```
1  USE Book;
2  GO
3  EXECUTE Order_Update 6, 1, 3, '2014-04-01 17:36:12', 2;
4  GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

4. Удаление данных (процедура *Order_Delete*).

```
1   USE Book;
2   GO
3   IF OBJECT_ID ('Order_Delete','P') IS NOT NULL
4   DROP PROCEDURE Order_Delete;
5   GO
6   CREATE PROCEDURE Order_Delete (@order_ID INT = NULL)
7   AS
8   BEGIN
9       SET NOCOUNT ON;
10      DELETE FROM Orders
11      WHERE order_ID = @order_ID;
12      RETURN @@ERROR
13  END
14  GO
```

100 %

Сообщения

```
Выполнение команд успешно завершено.
```

Вызов процедуры *Order_Delete* с указанием значения параметра *@order_ID* позволяет удалить строку в таблице *Orders*. Выводимое значение глобальной переменной *@@ERROR* дает возможность убедиться в правильности выполнения запроса.

```
1  USE Book;
2  GO
3  EXECUTE Order_Delete 6;
4  PRINT @@ERROR;
5  GO
```

100 %

Сообщения

```
0
```

Для остальных таблиц хранимые процедуры пишутся аналогично.

Практическое занятие № 8
Курсоры

Теоретические сведения

Рассмотрим следующие вопросы:

- извлечение данных с помощью курсора;
- изменение и удаление записей с помощью курсора.

SQL-запросы, извлекающие данные, работают с *результирующими* наборами строк. Возвращаемые строки соответствуют условию отбора, указанному в операторе SQL, их может быть нуль или больше. При этом нельзя получить первую строку, следующую строку или предыдущие десять строк. Это объясняется особенностями функционирования реляционной СУБД.

Язык SQL позволяет извлекать строки, однако для принятия решений на основе содержимого полей лучше использовать приложения, написанные на процедурных языках программирования. А здесь возникает проблема. В языке SQL операции производятся *одновременно с данными всех строк таблицы*, а процедурные языки обрабатывают данные *построчно*. Поэтому, даже если запрос к реляционной базе данных вернет несколько записей, приложение за один раз сможет обработать только одну запись.

Решить указанные проблемы позволяют курсоры. *Курсор (CURrent Set Of Records)* представляет собой запрос к базе данных, хранящийся на сервере СУБД. Это – результирующее множество, выборка, полученная в результате выполнения оператора *SELECT*. После того как курсор сохранен, приложения могут прокручивать данные в прямом или обратном направлении по одной или несколько строк.

Курсоры позволяют приложениям выполнять действия с отдельными записями результирующего набора, вместо обработки этого набора в целом. Поместив код SQL в цикл, можно, строка за строкой, обработать всю таблицу. Курсоры незаменимы, если требуется выбрать строки из таблицы, проверить их со-

держимое, а также выполнить различные операции на основании содержимого полей.

Основные типы курсоров в Transact-SQL:

- *динамические курсоры* – изменения данных отображаются при перемещении курсора. Используют минимум ресурсов сервера за пределами базы данных;
- *статические курсоры* – изолированы от изменений данных. Сервер сохраняет весь итоговый набор курсора при первой выборке;
- *ключевые курсоры* – отображают большую часть изменяемых данных. При открытии курсора сервер запоминает ключи возвращаемых записей. Новые записи не включаются в курсор, а при обращении к записи, удаленной другим пользователем, происходит ошибка.

Работу с курсором можно разделить на несколько этапов.

- Прежде чем курсор может быть использован, его следует объявить (создать). При этом выборка данных не производится, а определяется используемый оператор *SELECT* и задаются некоторые параметры курсора.
- После объявления курсор может быть открыт для использования. При этом производится выборка данных согласно заданному оператору *SELECT*.
- После того как курсор заполнен данными, из него могут быть извлечены (прочитаны) необходимые строки.
- По окончании работы курсор должен быть закрыт и, возможно, должны быть освобождены занятые им ресурсы.

После того как курсор объявлен, его можно открывать и закрывать столько раз, сколько необходимо. Если курсор открыт, извлекать из него строки можно произвольное число раз.

Создание курсора. Синтаксис оператора:

DECLARE имя_курсора CURSOR [LOCAL | GLOBAL]
[FORWARD_ONLY | SCROLL]
[STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
FOR команда_select
[FOR UPDATE [OF список_столбцов]]

Опции оператора создания курсора:

- *LOCAL* – локальный курсор (видим в пределах пакета, хранимой процедуры или пользовательской функции);
- *GLOBAL* – глобальный курсор, существующий до закрытия соединения (в текущем соединении можно ссылаться на него из любого модуля);

- *FORWARD_ONLY* – последовательный курсор (выборка данных осуществляется только в направлении от первой строки к последней);
- *SCROLL* – прокручиваемый курсор (можно обращаться к нему в любом порядке и направлении);
- *STATIC* – статический курсор (содержимое курсора формируется один раз и далее не обновляется);
- *KEYSET* – ключевой курсор (набор ключей, идентифицирующих строки полного результирующего набора курсора);
- *DYNAMIC* – динамический курсор (каждый раз при обращении к нему происходит выборка из таблиц);
- *FAST_FORWARD* – курсор «только для чтения» (оптимизирован для быстрого доступа к данным). Опцию нельзя использовать с опциями *FORWARD_ONLY* и *OPTIMISTIC*;
- *SCROLL_LOCKS* – обновления и удаления, производимые над строками курсора, будут гарантированно успешны. Не совместима с *FAST_FORWARD*;
- *OPTIMISTIC* – в курсоре запрещено изменение или удаление строк. Для отслеживания изменений сервер использует столбец *timestamp*. Если столбец отсутствует, то сервер генерирует контрольные суммы строк;
- *команда_select* – стандартный запрос на выборку;
- *список_столбцов* – указываются обновляемые столбцы курсора. Если конструкция без столбцов, то можно обновлять все столбцы (если нет *READ_ONLY*).

Открытие курсора. Наполняет курсор данными и готовит его к выборке записей. Все данные курсора хранятся в одной из временных таблиц БД *tempbd*.

OPEN { { [GLOBAL] имя_курсора } | переменная }

Здесь: *GLOBAL* – указывается для глобального курсора; *имя_курсора* – имя курсора; *переменная* – переменная, в которой хранится имя курсора.

Чтение из курсора. Считывание записей курсора осуществляется оператором *FETCH*. Для выборки записей необходимо предварительно открыть курсор.

Синтаксис оператора:

FETCH [направление FROM]
{ { [GLOBAL] имя_курсора } | переменная }
[INTO список_переменных]

Если при объявлении курсора указано ключевое слово *SCROLL*, параметр *направление* может принимать следующие значения:

- *FIRST* – курсор перемещается к первой записи итогового набора;

- *LAST* – курсор перемещается к последней записи итогового набора;
- *NEXT* – курсор перемещается на одну запись вперед в текущем наборе (действие по умолчанию при выборке);
- *PRIOR* – курсор перемещается на одну запись назад в текущем наборе;
- *ABSOLUTE n /-n* – если $n > 0$, курсор перемещается к n-й записи от начала. Если $n < 0$, курсор перемещается к последней записи и отсчитывает n записей в обратном направлении. При $n = 0$ записи не возвращаются;
- *RELATIVE n /-n* – если $n > 0$, курсор перемещается на n записей вперед от текущей позиции. Если $n < 0$, курсор перемещается на n записей назад. При $n=0$ возвращается текущая запись.

При использовании параметров *ABSOLUTE* и *RELATIVE* курсор в качестве аргументов может получать целые переменные или параметры хранимых процедур.

Закрытие курсора. Оператор *CLOSE* переводит курсор в неактивное состояние и фактически удаляет его итоговый набор. Курсор можно открыть повторно, при этом все его указатели сбрасываются. Синтаксис оператора:

CLOSE { { [GLOBAL] имя_курсора } | переменная }

Освобождение курсора. При освобождении курсора удаляется вся информация о нем. Допускается освобождение курсора, даже если в нем остались незавершенные транзакции. Синтаксис оператора:

DEALLOCATE { { [GLOBAL] имя_курсора } | переменная }

После выполнения оператора имя курсора может быть использовано другим оператором *DECLARE CURSOR*. Если эта ссылка на курсор была последней, оператор освобождает всю память, занятую структурой курсора. Чтобы заново открыть курсор, необходимо повторно объявить его.

Контроль достижения конца курсора. Для контроля достижения конца курсора используют функцию *@@FETCH_STATUS*. Функция возвращает:

- 0, если выборка завершилась успешно;
- -1, если выборка завершилась неудачно из-за попытки выборки строки, находящейся за пределами курсора;
- -2, если выборка завершилась неудачно из-за попытки обращения к удаленной или измененной после открытия курсора строке.

Таким образом, работа с курсорами похожа на работу с файлами – сначала открытие курсора, затем чтение и после закрытие.

С учетом изложенного работа с курсорами происходит по следующему алгоритму:

1. При помощи оператора *DECLARE … CURSOR FOR* связывается имя курсора с выполняемым запросом.

2. Оператор *OPEN* выполняет запрос, связанный с курсором, и устанавливает курсор перед первой записью результирующей таблицы.

3. Оператор *FETCH* помещает курсор на первую запись результирующей таблицы и извлекает данные из записи в локальные переменные пакета или хранимой процедуры. Повторный вызов оператора *FETCH* приводит к перемещению курсора к следующей записи, и так до тех пор, пока записи в результирующей таблице не будут исчерпаны. Эту операцию удобно осуществлять в цикле.

4. Оператор *CLOSE* прекращает доступ к результирующей таблице и ликвидирует связь между курсором и результирующей таблицей.

Изменение и удаление записей с помощью курсора. Для выполнения изменений с помощью курсора необходимо выполнить следующий оператор:

UPDATE имя_таблицы SET { имя_столбца =
{ DEFAULT | NULL | выражение } } [, ... n]
WHERE CURRENT OF
{ { [GLOBAL] имя_курсора } | @имя_переменной курсора }

За одну операцию могут быть изменены несколько столбцов текущей строки курсора, но все они должны принадлежать одной таблице.

Для удаления данных посредством курсора используется команда *DELETE* в следующем формате:

DELETE имя_таблицы WHERE CURRENT OF
{ { [GLOBAL] имя_курсора } | @имя_переменной курсора }

В результате будет удалена строка, установленная текущей в курсоре.

Пример использования курсора для модификации данных. Разработать прокручиваемый курсор для клиентов. Если фамилия клиента начинается на букву *К*, удалить клиента с такой фамилией. В первой записи курсора заменить первую цифру в номере телефона на 1. После выполнения запроса необходимо восстановить первоначальное заполнение таблиц БД.

```
1  USE Book;
2  GO
3  DECLARE @id INT, @name VARCHAR(20), @surname VARCHAR(20), @phone VARCHAR(15);
4  DECLARE user_cursor CURSOR GLOBAL SCROLL
5  KEYSET FOR
6      SELECT user_ID, u_name, u_surname, u_phone
7      FROM Users
8  FOR UPDATE;
9  OPEN user_cursor;
10 FETCH NEXT FROM user_cursor
11     INTO @id, @name, @surname, @phone;
12 WHILE @@FETCH_STATUS = 0
13 BEGIN
```

```
14      IF @surname LIKE 'K%'
15      DELETE Users WHERE CURRENT OF user_cursor;
16      FETCH NEXT FROM user_cursor
17          INTO @id, @name, @surname, @phone;
18  END;
19  FETCH ABSOLUTE 1 FROM user_cursor
20      INTO @id, @name, @surname, @phone;
21  UPDATE Users
22  SET u_phone = '1' + RIGHT(@phone, LEN(@phone)-1)
23      WHERE CURRENT OF user_cursor;
24  CLOSE user_cursor;
25  DEALLOCATE user_cursor;
26  SELECT * FROM Users;
```

100 %

Результаты | Сообщения

	user_ID	u_name	u_patronymic	u_surname	u_phone	u_email	u_status
1	1	Александр	Валерьевич	Иванов	18-98-78	ivanov@email.ru	active
2	2	Сергей	Иванович	Лосев	90-57-77	losev@email.ru	passive
3	3	Нина	Сергеевна	Грачева	95-66-61	gracheva@email.ru	active
4	5	Анатолий	Юрьевич	Петров	NULL	NULL	lock

Практическая работа

При выполнении задания необходимо для своей предметной области:

- решить две задачи с помощью курсора: в первом случае предусмотреть использование курсора в простом пакете, во втором – включить курсор в хранимую процедуру;
- составить отчет.

Пример выполнения задания

Получим список клиентов интернет-магазина, имеющих статус *'active'*. Задача решается путем использования простого оператора *SELECT*, однако никакие дальнейшие действия с полями результирующего набора невозможны.

```
1  USE Book;
2  GO
3  BEGIN
4  SELECT u_name, u_surname
5  FROM Users
6  WHERE u_status = 'active'
7  END;
```

100 %

Результаты | Сообщения

	u_name	u_surname
1	Александр	Иванов
2	Нина	Грачева
3	Максим	Кузнецов

Если использовать курсор, то можно проверять содержимое записей, а также выполнять различные операции с использованием содержимого полей

```
1  USE Book;
2  GO
3  -- Определяем переменные
4  DECLARE @name VARCHAR(20), @surname VARCHAR(20),
5          @message VARCHAR (50);
6  PRINT 'Список клиентов, имеющих статус active:';
7  -- Объявляем курсор
8  DECLARE cur CURSOR LOCAL FOR
9      SELECT u_name, u_surname
10     FROM Users
11     WHERE u_status = 'active';
12 -- Открываем курсор
13 OPEN cur;
14 -- Переход на начало курсора
15 FETCH NEXT FROM cur INTO @name, @surname;
16 -- Двигаемся по курсору и печатаем строки
17 WHILE @@FETCH_STATUS = 0
18 BEGIN
19     SELECT @message = @name + ' ' + @surname;
20     PRINT @message;
21 -- Переход к следующему клиенту
22     FETCH NEXT FROM cur INTO @name, @surname;
23 END;
24 -- Закрываем курсор
25 CLOSE cur;
26 -- Удаляем курсор
27 DEALLOCATE cur;
28 PRINT @name;
```

100 %

Сообщения

```
Список клиентов, имеющих статус active:
Александр Иванов
Нина Грачева
Максим Кузнецов
Максим
```

Создадим хранимую процедуру, записывающую в новую таблицу *February* все заказы, сделанные в феврале 2014 г. Предварительно необходимо создать пустую таблицу *February* со структурой, аналогичной таблице *Orders*.

```
1 USE Book;
2 CREATE TABLE February(
3     f_order_ID INT NOT NULL,
4     f_o_user_ID INT NOT NULL,
5     f_o_book_ID INT NOT NULL,
6     f_o_time DATETIME NOT NULL,
7     f_o_number INT NOT NULL,
8     PRIMARY KEY(f_order_ID));
9 GO
```

Хранимая процедура *ord_febr ( )* использует курсор *cur*, который в цикле читает данные из таблицы *orders* и добавляет их в таблицу *February*.

```
1  USE Book;
2  GO
3  IF OBJECT_ID('ord_febr', 'P') IS NOT NULL
4      DROP PROCEDURE ord_febr;
5  GO
6  CREATE PROCEDURE ord_febr
7  AS
8  BEGIN
9      DECLARE @order_ID INT, @o_user_ID INT, @o_book_ID INT,
10         @o_time DATETIME, @o_number INT;
11     DECLARE cur CURSOR LOCAL FOR
12         SELECT * FROM Orders
13         WHERE o_time BETWEEN '01/02/2014' AND '28/02/2014';
14     OPEN cur;
15     FETCH NEXT FROM cur
16     INTO @order_ID, @o_user_ID, @o_book_ID, @o_time, @o_number;
17     WHILE @@FETCH_STATUS = 0
18         BEGIN
19             INSERT INTO February VALUES
20                (@order_ID, @o_user_ID, @o_book_ID,  @o_time, @o_number);
21             FETCH NEXT FROM cur INTO @order_ID, @o_user_ID, @o_book_ID,
22                 @o_time, @o_number;
23         END;
24     CLOSE cur;
25     DEALLOCATE cur;
26 END;
27 GO
```

Процедура вызывается оператором *EXECUTE*. Для просмотра результата выполнения процедуры используем полную выборку из таблицы *February*.

```
1  USE Book;
2  GO
3  EXECUTE ord_febr;
4  SELECT * FROM February;
5  GO
```

100 %

Результаты | Сообщения

	f_order_ID	f_o_user_ID	f_o_book_ID	f_o_time	f_o_number
1	2	6	10	2014-02-10 09:40:29.000	2
2	3	1	20	2014-02-18 13:41:05.000	4

Практическое занятие № 9
Транзакции

Рассмотрим следующие вопросы:

– транзакции и их свойства;

– режимы выполнения транзакций в SQL Server.

Теоретические сведения

Транзакция – один или несколько операторов SQL, которые либо успешно завершаются как единое целое, либо отменяются как единое целое. *Фиксация* транзакции обеспечивает запись на диск изменений, сделанных в процессе выполнения транзакции. *Откат* транзакции обеспечивает аннулирование всех изменений, сделанных операторами в теле транзакции.

Транзакции являются одной из самых важных концепций, используемых для построения надежных баз данных. Именно транзакции обеспечивают целостность данных в любых ситуациях.

Транзакция является логической единицей работы и обладает следующими свойствами:

- *атомарность* – транзакция должна быть выполнена в целом или не выполнена вообще (все изменения данных рассматриваются как единый блок);
- *согласованность* – в результате транзакции данные переходят из одного согласованного состояния в другое согласованное состояние;
- *изолированность* – конкурирующие транзакции обрабатываются изолированно друг от друга (однако пользователи видят параллельное выполнение);
- *устойчивость* – после фиксации изменений данные не могут быть возвращены в состояние, в котором они были до начала транзакции.

Разработчик из смысла обработки данных сам определяет, какая последовательность операций составляет транзакцию.

Режимы выполнения транзакций в SQL Server:

- *автоматическая фиксация* – каждый оператор SQL фиксируется по завершении (каждый запрос SQL – отдельная транзакция). При успешном завершении оператора база данных переходит в новое устойчивое состояние;
- *явные транзакции* – позволяет объединить несколько команд SQL в одну транзакцию, используется явная установка границ транзакций;
- *неявные транзакции* – транзакции начинаются без оператора старта транзакций. Режим устанавливается изменением значения параметра сеанса.

По умолчанию SQL Server работает в режиме автоматической фиксации транзакций, т. е. как только выполняется оператор обновления данных, модифицирующий таблицу, изменения тут же сохраняются на диске. Чтобы объединить в транзакцию несколько операторов, следует использовать

операторы управления транзакциями или включить режим неявных транзакций оператором

SET IMPLICIT_TRANSACTIONS { ON | OFF }

Присвоение параметру сеанса *IMPLICIT_TRANSACTIONS* значения *ON* переводит соединение в режим неявных транзакций. Значение *OFF* возвращает соединение в режим с автоматической фиксацией транзакций.

Определение текущего значения параметра *IMPLICIT_TRANSACTIONS*:

```
1  USE BestDatabase;
2  GO
3  DECLARE @IMPLICIT_TRANSACTIONS VARCHAR(3) = 'OFF';
4  IF ((2 & @@OPTIONS) = 2)
5      SET @IMPLICIT_TRANSACTIONS = 'ON';
6  SELECT @IMPLICIT_TRANSACTIONS AS IMPLICIT_TRANSACTIONS;
```

100 %

Результаты | Сообщения

	IMPLICIT_TRANSACTIONS
1	OFF

Операторы управления транзакциями в режиме явных транзакций:

- *BEGIN TRAN[SACTION] [имя транзакции | @переменная]* – определяет начало транзакции;
- *COMMIT TRAN[SACTION] [имя транзакции | @переменная]* – информирует сервер о завершении текущей транзакции и делает изменения постоянными;
- *COMMIT WORK* – работает аналогично *COMMIT TRAN*. Соответствует синтаксису ANSI-92;
- *ROLLBACK TRAN[SACTION] [имя_транзакции | имя_точки_сохранения | @переменная]* – заставляет сервер отменить все изменения данных от последней определенной точки сохранения или от начала транзакции;
- *ROLLBACK WORK* – работает аналогично *ROLLBACK TRAN*. Соответствует синтаксису ANSI-92;
- *SAVE TRAN[SACTION] [имя_точки_сохранения | @переменная]* – устанавливает промежуточную точку, к которой может выполняться откат (обеспечивает возможность частичного отката). При этом транзакция остается активной.

Имена транзакций и точек сохранения можно задавать обычным образом или при помощи ссылок на строковые локальные переменные. Имена строковых локальных переменных должны начинаться с символа @. Вместо ключевого слова *TRANSACTION* можно использовать сокращение *TRAN*.

Пример установки границ транзакции (явный режим).

```
1  USE BestDataBase;
2  GO
3  IF OBJECT_ID(N'dbo.t1', N'U') IS NOT NULL
4      DROP TABLE dbo.t1;
5  GO
6  CREATE TABLE dbo.t1(a INT);
7  GO
8  BEGIN TRAN;
9      INSERT dbo.t1(a) VALUES(1);
10     INSERT dbo.t1(a) VALUES(2);
11 COMMIT TRAN;
12 SELECT * FROM dbo.t1;
```

100 %

Результаты | Сообщения

	a
1	1
2	2

Если соединение находится в режиме неявных транзакций и не участвует в транзакции, выполнение любого из следующих операторов начнет транзакцию: *ALTER TABLE*, *BEGIN TRANSACTION*, *CREATE*, *DELETE*, *DROP*, *FETCH*, *GRANT*, *INSERT*, *OPEN*, *REVOKE*, *SELECT*, *TRUNCATE TABLE*, *UPDATE*.

Если соединение уже начало транзакцию, то эти операторы не начинают новую транзакцию.

Пример установки границ транзакции (режим неявных транзакций).

```
1  SET IMPLICIT_TRANSACTIONS ON;
2  USE BestDataBase;
3  GO
4  IF OBJECT_ID(N'dbo.t1', N'U') IS NOT NULL
5      DROP TABLE dbo.t1;
6  GO
7  CREATE TABLE dbo.t1(a INT);
8  GO
9  INSERT dbo.t1(a) VALUES(1);
10 INSERT dbo.t1(a) VALUES(2);
11 COMMIT TRAN;
12 INSERT dbo.t1(a) VALUES(3);
13 ROLLBACK TRAN;
14 SELECT * FROM dbo.t1;
```

100 %

Результаты | Сообщения

	a
1	1
2	2

Переменные, используемые при работе с транзакциями:

- *@@TRANCOUNT* – количество вложенных запущенных на выполнение и не подтвержденных или отмененных в данном соединении транзакций;
- *@@ERROR* – позволяет проанализировать успешность выполнения операторов обращения к БД. При успешном выполнении оператора имеет значение 0. Очищается при начале выполнения любого другого оператора.

Каждый оператор *BEGIN TRAN* увеличивает значение *@@TRANCOUNT* (число транзакций) на 1. При этом оператор *COMMIT* уменьшает количество транзакций на 1. Оператор *ROLLBACK* очищает значение переменной *@@TRANCOUNT*, так как все активные транзакции отменены.

Оператор *SAVE TRANSACTION point1* устанавливает именованную точку сохранения с именем *point1*. Оператор *ROLLBACK TRANSACTION point1* откатывает транзакцию к состоянию, в котором находилась база данных на момент установки именованной точки. Все точки сохранения транзакций удаляются, если выполняются операторы *COMMIT* или *ROLLBACK* без указания имени точки сохранения.

Рассмотрим пример, показывающий, как обрабатываются ошибки в пакете и как выполняется отмена действия всех команд в группе при появлении ошибок. Пример основан на ссылочной целостности данных в таблицах *Catalogs* и *Books*.

Первый и второй операторы *INSERT* выполняются успешно. Выполнение третьего оператора приводит к нарушению ссылочной целостности данных (ссылка на несуществующий каталог).

Поскольку все три оператора *INSERT* объединены в транзакцию внутри блока *TRY*, выдается исключение, и обработчик исключений запускает блок *CATCH*, который отменяет действие всех операторов и выводит соответствующее сообщение. В итоге содержимое таблицы *Books* не изменяется.

```
USE book;
BEGIN TRY
  BEGIN TRANSACTION
    INSERT INTO Books
    VALUES (31,'Звуковая подсистема компьютера','Иванов Г.И.', 2007, 315.00, 20, 5)
    INSERT INTO Books
    VALUES (32,'Технологии DVD','Петров А.К.', 2005, 132.00, 15, 5)
    -- ошибка ссылочной целостности - несуществующий каталог
    INSERT INTO Books VALUES (33,'Муму', 'Тургенев И.С.', 2008, 50.00, 8, 7)
    COMMIT TRANSACTION
    PRINT 'Транзакция зафиксирована'
    END TRY
BEGIN CATCH
    ROLLBACK
    PRINT 'Транзакция отменена'
END CATCH;
```

100 %

Сообщения

```
(строк обработано: 1)

(строк обработано: 1)

(строк обработано: 0)
Транзакция отменена
```

Практическая работа

При выполнении задания необходимо для своей предметной области:

- создать транзакцию с точкой сохранения в режиме явных транзакций, произвести ее откат и фиксацию;
- создать транзакцию с точкой сохранения в режиме неявных транзакций, произвести ее откат и фиксацию;
- составить отчет.

Пример выполнения задания

Объединим в явную транзакцию несколько операций по добавлению в таблицу *Catalogs* новых каталогов, создадим точку сохранения, добавим еще один каталог, а затем произведем откат транзакции к точке сохранения и проверим, какие записи добавились в таблицу (потом их следует удалить).

```
1  USE Book;
2  GO
3  SET NOCOUNT ON;
4  BEGIN TRAN my_tran
5  INSERT Catalogs VALUES(6, 'Аппаратура');
6  INSERT Catalogs VALUES(7, 'Безопасность');
7  PRINT @@TRANCOUNT;
8  SAVE TRAN point_one;
9  INSERT Catalogs VALUES(8, 'Web-дизайн');
10 ROLLBACK TRAN point_one;
11 COMMIT TRAN my_tran;
12 SELECT * FROM Catalogs;
```

100 %

Результаты | Сообщения

	cat_ID	cat_name
1	1	Программирование
2	2	Интернет
3	3	Базы данных
4	4	Сети
5	5	Мультимедиа
6	6	Аппаратура
7	7	Безопасность

100 %

Результаты | Сообщения

1

Решим эту же задачу с использованием неявных транзакций.

```
1  SET IMPLICIT_TRANSACTIONS ON;
2  SET NOCOUNT ON;
3  USE Book;
4  GO
5  INSERT Catalogs VALUES(6, 'Аппаратура');
6  INSERT Catalogs VALUES(7, 'Безопасность');
7  PRINT @@TRANCOUNT;
8  SAVE TRAN point_one;
9  INSERT Catalogs VALUES(8, 'Web-дизайн');
10 ROLLBACK TRAN point_one;
11 COMMIT TRAN;
12 PRINT @@TRANCOUNT;
13 SELECT * FROM Catalogs;
```

100 %

Результаты | Сообщения

	cat_ID	cat_name
1	1	Программирование
2	2	Интернет
3	3	Базы данных
4	4	Сети
5	5	Мультимедиа
6	6	Аппаратура
7	7	Безопасность

100 %

Результаты | Сообщения

1
0

Практическое занятие № 10
Триггеры

Теоретические сведения

Рассмотрим следующие вопросы:

- создание триггеров с помощью оператора *CREATE TRIGGER*;
- изменение триггеров с помощью оператора *ALTER TRIGGER*;
- удаление триггеров с помощью оператора *DROP TRIGGER*;
- включение и выключение триггеров.

Триггер – программный объект базы данных, выполняемый на стороне сервера. Вызывается автоматически при наступлении определенного события. Это хранимая процедура, привязанная к событию в базе данных. В отличие от обычных хранимых процедур триггеры не могут вызываться напрямую и не принимают параметры. Триггеры могут создаваться для представлений.

Триггер срабатывает, когда определенное действие с базой данных выполняет приложение, хранимая процедура или триггер (другой или тот же самый).

Триггеры решают следующие задачи:

- обеспечивают ссылочную целостность, особенно в тех случаях, когда ее критерии слишком сложны для определения на уровне таблиц;
- автоматически генерируют производные значения столбцов;
- обеспечивают проверку правильности значений атрибутов;
- предотвращают неверные транзакции;
- реализуют сложные процедуры авторизации;
- реализуют сложные бизнес-правила;
- обеспечивают выбор сложных значений по умолчанию (обычно допускается только одно значение по умолчанию для каждого столбца);
- собирают статистику доступа к таблицам базы данных.

Триггер – лучшее средство обеспечения низкоуровневой целостности данных, однако он работает медленнее ограничений. При этом триггеры могут делать то, что не могут делать ограничения *CHECK*:

- ссылаться на поля других таблиц;
- объявлять индивидуальные сообщения об ошибках (ограничения и значения по умолчанию генерируют только стандартные системные ошибки);
- содержать ненормализованные данные;
- сравнивать состояние данных до и после изменения и т. д.

Триггеры могут срабатывать при каждом обращении к таблице, увеличивая нагрузку на сервер. Поэтому необходимо придерживаться следующих рекомендаций:

- если вместо триггера можно создать одно или несколько ограничений *CHECK*, выполняющих ту же работу, то следует использовать ограничения;
- если итоговые величины можно вычислять в нерабочее время, лучше прибегнуть к этому способу, а не использовать триггеры, срабатывающие при каждой транзакции.

Триггеры срабатывают после проверки всех ограничений *CHECK* и внешних ключей (ограничения работают быстрее; если быстрая проверка даст отрицательный результат, то более сложные проверки в триггере уже не нужны).

В структуре триггера выделяют *заголовок* (разный для разных типов триггеров) и *тело* (следует после служебного слова *AS*). Тело триггера – оператор или блок операторов, который необходимо выполнить при возникновении определенного события. Внутри блока допустимы все специфичные для хранимых процедур операторы и конструкции:

- другие блоки *BEGIN ... END*;
- условные операторы и операторы цикла;
- объявления локальных переменных и задание их значений;
- именованные условия и обработчики ошибок.

В SQL Server есть три вида триггеров:

- *языка манипулирования данными DML* – связаны с событиями обработки данных (вставка, обновление, удаление);
- *языка описания данных DDL* – связаны с событиями определения данных (создание таблиц);
- *входа в систему* – запускаются при соединении пользователя с экземпляром сервера; часто используются для проверки полномочий пользователей.

Триггер DML. Оператор создания имеет следующий вид:

CREATE TRIGGER [<имя схемы>.] <имя триггера>
ON { <таблица> | <представление> }
[WITH <опция триггера> [, <опция триггера>] ...]
{ FOR | AFTER | INSTEAD OF }
<операция> [, <операция>] ...
AS BEGIN { <оператор> | <блок операторов> } END;
<опция триггера> ::= { ENCRIPTION | EXECUTE AS <контекст> }
<операция> ::= INSERT | UPDATE | DELETE

Имя триггера не может быть больше 128 символов и должно быть уникально в данной схеме. После ключевого слова *ON* указывается имя таблицы или представления. После ключевого слова *WITH* могут быть заданы опции:

- *ENCRIPTION* – текст триггера должен кодироваться при помещении в БД;
- *EXECUTE AS* – определяет контекст безопасности обращения к триггеру.

SQL Server поддерживает два вида DML-триггеров:

- последующие (*FOR* или *AFTER*);
- замещающие (*INSTEAD OF*).

Последующий триггер запускается *после* завершения связанного с ним события и может быть определен только для постоянных таблиц. Такой триггер может вызывать себя рекурсивно (глубина вложенности не более 32).

Замещающий триггер запускается *вместо* связанного с ним события и может определяться для постоянных таблиц и представлений. Действия в операторе, неявно вызвавшем триггер, не выполняются, а будут выполнены действия триггера. Если в замещающем триггере выполняется та же операция, по которой триггер запущен на выполнение, то дополнительный вызов триггера не произойдет.

Для одной операции в одной таблице может быть задано несколько триггеров *INSTEAD OF* и несколько триггеров *AFTER*. Вначале запускаются триггеры *INSTEAD OF*, затем триггеры *AFTER*.

Триггеры DML создают для реагирования на следующие операции:

- добавление данных (*INSERT*);
- изменение (*UPDATE*);
- удаление (*DELETE*).

При выполнении операции усечения таблицы (*TRUNCATE TABLE*) триггер вызываться не будет.

В триггерах недопустимы следующие операторы Transact-SQL: *ALTER, CREATE, DROP, LOAD/RESTORE DATABASE, LOAD/RESTORE LOG, DISK RESIZE, DISK INIT, RECONFIGURE*.

В триггерах DML можно обращаться к логическим таблицам *DELETED* и *INSERTED*. Первая таблица содержит копии строк, удаляемых вызвавшим триггер оператором. Вторая таблица содержит копии строк, добавляемых в основную таблицу. Таблицы *DELETED* и *INSERTED* являются временными и располагаются в оперативной памяти.

При срабатывании триггера, поставленного на вставку записей:

1) пользователь выполняет оператор *INSERT* для добавления записей;
2) сервер сохраняет информацию о запросе в журнале транзакций;

3) сервер вызывает триггер;

4) сервер подтверждает изменения и физически изменяет данные.

При вызове триггера изменений в базе данных еще не произошло. Однако в теле триггера можно увидеть добавляемые записи в виде таблицы *INSERTED*.

Действия по изменению данных записываются в журнал транзакций. Эту информацию прочитать нельзя, но можно использовать данные таблицы *INSERTED*, которая имеет ту же структуру, что и таблица, на которую установлен триггер.

При срабатывании триггера, поставленного на удаление записей:

1) пользователь выполняет оператор *DELETE* для удаления записей;

2) сервер сохраняет информацию о запросе в журнале транзакций;

3) сервер вызывает триггер;

4) сервер подтверждает изменения и физически изменяет данные.

Удаляемые строки помещаются в таблицу *DELETED*, при этом:

- когда строки добавляются в таблицу *DELETED*, они еще существуют в таблице базы данных;
- для таблицы *DELETED* выделяется память, поэтому она всегда в кэше;
- триггер удаления не сработает на операцию *TRUNCATE TABLE* (очистка таблицы), поскольку эта операция не заносится в журнал и не удаляет строк.

Создадим триггер DML для базы данных *Book*, который срабатывает после вставки записей в таблицу *Orders*, ограничивая число заказываемых товаров до 1:

```
1  USE Book;
2  GO
3  CREATE TRIGGER restrict_count ON Orders
4  AFTER INSERT
5  AS
6  UPDATE Orders
7  SET o_number = 1
8  FROM inserted
9  WHERE Orders.order_ID = inserted.order_ID;
10 GO
```

Протестируем работу триггера, пытаясь добавить в заказ 10 экземпляров одной книги. Заказ добавляется с количеством экземпляров 1.

```
1  USE Book;
2  GO
3  INSERT INTO Orders VALUES (14, 3, 7, '17/03/2014 19:15:36', 10);
4  GO
```

```
100 %
Сообщения
(строк обработано: 1)
(строк обработано: 1)
```

Триггер может отменить действие оператора, вызвав откат транзакции с помощью команды *ROLLBACK TRAN*. Триггер выполняется в контексте транзакции, под управлением которой выполняется программа, инициировавшая вызов триггера. Поэтому тело триггера может содержать оператор *ROLLBACK TRAN*, даже если отсутствует оператор *BEGIN TRAN* (создается неявная транзакция).

Создадим в базе данных *Book* триггер DML, выполняющий откат транзакции:

```
1  USE Book;
2  GO
3  CREATE TRIGGER update_users ON Users
4  FOR UPDATE
5  AS
6  ROLLBACK TRANSACTION;
7  GO
```

Теперь попробуем изменить данные в таблице *Users*, чтобы сработал триггер:

```
1  USE Book;
2  GO
3  UPDATE Users SET u_name = 'Иван';
4  GO
```

```
100 %
Сообщения
Сообщение 3609, уровень 16, состояние 1, строка 1
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

При попытке изменить содержимое поля *u_name* для всех записей таблицы *Users* сработал триггер с откатом транзакции и данные не изменились.

При срабатывании триггера часто нужно узнать, обновлялся столбец или нет. Конструкция *IF UPDATE* позволяет определить, выполнялась ли вставка или обновление для конкретного столбца. Результат проверки будет равен *TRUE*, если в столбец было вставлено значение, отличное от *NULL*, или если столбец был включен в секцию *SET* оператора *UPDATE*.

Создадим в базе данных *Book* триггер DML, не допускающий, чтобы цена книги была меньше 30 рублей. Проверку модификации столбцов таблицы *Books* осуществим конструкцией *IF UPDATE*.

```
1  USE Book;
2  GO
3  CREATE TRIGGER price_30 ON Books
4  FOR INSERT, UPDATE
5  AS
6  IF UPDATE (b_price) AND
7      EXISTS(SELECT * FROM inserted WHERE b_price<30)
8  BEGIN
9      PRINT 'Недопустимое значение цены книги';
10     ROLLBACK TRAN;
11     RETURN;
12 END
13 PRINT 'Допустимая цена книги корректна';
14 RETURN;
15 GO
```

Для тестирования триггера *price_30* выполним запрос:

```
1  USE Book;
2  GO
3  INSERT INTO Books VALUES(40, 'Приложения баз данных', 'Петров А.В.', 2006, 25.00, 10, 3);
4  GO
```

100 %

Сообщения

```
Недопустимое значение цены книги
Сообщение 3609, уровень 16, состояние 1, строка 1
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

При выполнении оператора *ROLLBACK TRAN* все изменения в триггере и изменения, которые стали причиной срабатывания триггера, откатываются. Необходимо учесть следующее:

- при срабатывании оператора *ROLLBACK TRAN* содержимое транзакции откатывается, однако операторы, следующие за *ROLLBACK TRAN* (если они есть), выполняются. Для предотвращения этого используют оператор *RETURN*;
- если триггер откатывает транзакцию, определенную пользователем, то выполнение *ROLLBACK TRAN* откатывает все операции, начиная с внешней команды *BEGIN TRAN*, завершает вычисления в триггере и прерывает пакет. Последующие операторы пакета не выполняются;
- в триггере можно установить точку сохранения и вернуться к ней. Откат будет выполнен только для команд триггера, находящихся за точкой сохранения. Транзакция останется активной, а выполнение пакета продолжится;
- следует минимизировать использование *ROLLBACK TRAN* в триггерах. Откат транзакции создает дополнительное задание, что снижает производительность.

Команда *SAVE TRAN* позволяет избежать отката всей транзакции в триггере. Откат транзакции ограничивается точкой сохранения с именем. В этом случае необходимо немедленно выполнить команду *RAISERROR* или *RETURN* и передать код ошибки вызывающему процессу. Вызывающий процесс проверяет код ошибки и предпринимает необходимые действия: откат транзакции, откат до точки сохранения, игнорирование ошибки и закрепление модификации данных.

```
1  USE Book;
2  GO
3  CREATE TRIGGER price_30 ON Books
4  FOR INSERT, UPDATE
5  AS
6  SAVE TRAN point_one
7  IF UPDATE (b_price) AND
8      EXISTS (SELECT * FROM inserted WHERE b_price < 30)
9  BEGIN
10     ROLLBACK TRAN point_one;
11     RAISERROR ('Недопустимое значение цены книги', 16, 1)
12     RETURN
13 END
14 PRINT 'Допустимая цена книги корректна';
15 RETURN
16 GO
```

Пример использования триггера *price_30*:

```
1  USE Book;
2  GO
3  BEGIN TRAN myTran
4  INSERT Books VALUES (31,'SQL. Карманный справочник','Генник Дж.',2012,25.00,5,3);
5  IF @@ERROR <> 0
6  BEGIN
7      PRINT 'Ошибка при вставке записи'
8      ROLLBACK TRAN myTran
9      RETURN
10 END
11 INSERT Books VALUES (32,'SQL за 10 минут','Форта Бен',2015, 168.00, 10, 3);
12 IF @@ERROR <> 0
13 BEGIN
14     PRINT 'Ошибка при вставке записи'
15     ROLLBACK TRAN myTran
16     RETURN
17 END
18 COMMIT TRAN
19 GO
```

100 %

Сообщения

```
Сообщение 50000, уровень 16, состояние 1, процедура price_30, строка 9
Недопустимое значение цены книги
(строк обработано: 1)
Ошибка при вставке записи
```

В примере вставляются записи в таблицу *Books*, для которой определен триггер *price_30*. После каждой вставки проверяется значение *@@ERROR*, чтобы узнать, произошел ли в триггере откат, и если произошел – откатывается вся транзакция.

Запретим интернет-магазину обслуживать человека по имени Иван Иванович Иванов. Такой запрет можно реализовать с помощью следующего триггера:

```
1   USE Book;
2   GO
3   CREATE TRIGGER insert_users ON Users
4   FOR INSERT
5   AS
6   DECLARE @user_name AS VARCHAR(100);
7   SET @user_name = (SELECT u_name+' '+u_patronymic+' '+u_surname
8       FROM inserted);
9   IF @user_name = 'Иван Иванович Иванов'
10  BEGIN
11      PRINT 'Ошибка!'
12      ROLLBACK TRANSACTION
13  END;
14  GO
```

Попробуем добавить клиента с таким именем в таблицу *Users*:

```
1  USE Book;
2  INSERT INTO Users VALUES(7, 'Иван', 'Иванович', 'Иванов', NULL, NULL, 'passive');
3  GO
```

100 %

Сообщения

```
Ошибка!
Сообщение 3609, уровень 16, состояние 1, строка 2
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

Создадим в базе данных *Book* триггер DML, запрещающий удаление пользователя с именем Александр.

```
1   USE Book;
2   GO
3   CREATE TRIGGER delete_Users ON dbo.Users
4   FOR DELETE
5   AS
6   IF EXISTS (SELECT * FROM deleted WHERE u_name='Александр')
7   BEGIN
8       PRINT  'Ошибка,нельзя удалять этого пользователя!'
9       ROLLBACK TRANSACTION
10  END
11  GO
```

Попытаемся удалить запись, соответствующую пользователю с кодом 1 (Александр Валерьевич Иванов). В ответ будет выдано сообщение об ошибке.

```
1 USE Book;
2 DELETE FROM Users WHERE user_ID = 1;
3 GO
```

100 %

Сообщения

```
Ошибка,нельзя удалять этого пользователя!
Сообщение 3609, уровень 16, состояние 1, строка 2
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

Если попытаться удалить несколько записей, то не будет удалена ни одна из них, несмотря на то, что запрет действует только на имя Александр, а пользователя с именем Сергей (код 2) удалять можно. Отменяется вся транзакция.

```
1 USE Book;
2 DELETE FROM Users WHERE user_ID = 1 OR user_ID = 2;
3 GO
```

100 %

Сообщения

```
Ошибка,нельзя удалять этого пользователя!
Сообщение 3609, уровень 16, состояние 1, строка 2
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

Для таблиц и представлений можно указать триггер *INSTEAD OF*, действия которого выполняются вместо события, инициировавшего запуск триггера. При этом оператор, запущенный пользователем, только генерирует событие. Реальное обновление данных происходит с помощью операторов триггера.

Каждая таблица или представление ограничены одним триггером *INSTEAD OF* на каждое событие. Нельзя создавать триггеры *INSTEAD OF* на представление, у которого включена опция *CHECK OPTIONS*.

Триггер DDL. Оператор создания имеет следующий вид:

CREATE TRIGGER <имя триггера> ON { ALL SERVER | DATABASE }
[WITH <опция триггера> [, <опция триггера>] ...]
{ FOR | AFTER } { <событие> | <группа событий> }
[{ <событие> | <группа событий> }] ...
AS BEGIN { <оператор> | <блок операторов> } END;
<опция триггера> ::= { ENCRIPTION | EXECUTE AS <контекст> }

После ключевого слова *ON* указывается область действия триггера:

- *DATABASE* – триггер сработает при наступлении одного из перечисленных далее событий текущей базы данных;
- *ALL SERVER* – триггер сработает при наступлении события для любой базы данных текущего экземпляра сервера.

После ключевого слова *FOR* или *AFTER* указываются одиночные события или группы похожих событий, при которых вызывается триггер.

Все задаваемые события и их группы связаны с операторами DDL:

- *одиночные события* – выполнение операций создания, изменения или удаления отдельных объектов БД (таблиц, индексов, пользовательских типов и др.);
- *группы событий* – связаны с различными операциями по отношению к одному типу объектов БД (например, группа событий, относящаяся к операциям создания, изменения и удаления таблиц).

В качестве примера рассмотрим следующий DDL-триггер уровня базы данных.

```
1  USE Book;
2  GO
3  CREATE TRIGGER prevent_drop_triggers
4  ON DATABASE FOR DROP_TRIGGER
5  AS
6  PRINT 'вы должны отключить prevent_drop_triggers для удаления триггера';
7  ROLLBACK
8  GO
```

```
1  USE Book;
2  GO
3  DROP TRIGGER price_30
4  GO
```

```
100 %
Сообщения
вы должны отключить prevent_drop_triggers для удаления триггера
Сообщение 3609, уровень 16, состояние 2, строка 1
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

Триггер входа в систему. Оператор создания имеет следующий вид:

CREATE TRIGGER <имя триггера> ON ALL SERVER
[WITH <опция триггера> [, <опция триггера>] ...]
{ FOR | AFTER } LOGON
AS BEGIN { <оператор> | <блок операторов> } END;
<опция триггера> ::= { ENCRIPTION | EXECUTE AS <контекст> }

Триггер входа запускается при соединении пользователя с экземпляром сервера. Используется для дополнительной проверки полномочий пользователей.

Модификация триггеров. Осуществляется операторами *ALTER TRIGGER*, которые похожи на соответствующие операторы *CREATE TRIGGER*, где вместо слова *CREATE* используется слово *ALTER*.

Удаление триггеров. Операторы удаления триггеров:

DROP TRIGGER [<имя схемы>.] <имя триггера>
[, |<имя схемы>.] <имя триггера>] ... ;
DROP TRIGGER <имя триггера> [, <имя триггера>] ...
ON { DATABASE | ALL SERVER } ;
DROP TRIGGER <имя триггера> [, <имя триггера>] ... ON ALL SERVER ;

При удалении таблицы удаляются также все связанные с ней триггеры.

Включение и выключение триггеров. Осуществляется операторами:

ENABLE TRIGGER { [<имя схемы>.] <имя триггера> [, ... n] | ALL }
ON { <имя объекта> | DATABASE | ALL SERVER } ;
DISABLE TRIGGER { [<имя схемы>.] <имя триггера> [, ... n] | ALL }
ON { <имя объекта> | DATABASE | ALL SERVER } ;

Здесь *<имя объекта>* – имя связанной с триггером таблицы или представления; *ALL* – означает, что будут отключены все триггеры в области действия *ON*.

По умолчанию при создании триггер включается. При отключении триггер продолжает существовать как объект текущей базы данных. Отключенные триггеры можно повторно включать. При включении триггера он не создается повторно.

Практическая работа

При выполнении задания необходимо для своей предметной области:

- написать два триггера (можно ограничиться DML-триггерами);
- составить отчет.

Пример выполнения задания

Создадим для базы данных *Book* DML-триггер, который при добавлении новой записи в таблицу *Orders* автоматически вычитает число заказанных книг из их общего количества в таблице *Books*.

```
1  USE Book;
2  GO
3  CREATE TRIGGER real_quantity ON Orders
4  AFTER INSERT
5  AS
6  UPDATE Books
7  SET b_count = b_count - inserted.o_number
8  FROM inserted
9  WHERE Books.book_ID = inserted.o_book_ID;
10 GO
```

Создадим для базы данных *Book* DML-триггер, который при добавлении новых записей в таблицу *Users* преобразует полные имена и отчества покупателей в инициалы.

```
1  USE Book;
2  GO
3  CREATE TRIGGER restrict_user ON Users
4  AFTER INSERT
5  AS
6  UPDATE Users
7  SET u_name = LEFT(inserted.u_name, 1)
8  FROM inserted
9  WHERE Users.user_ID = inserted.user_ID;
10 UPDATE Users
11 SET u_patronymic = LEFT(inserted.u_patronymic, 1)
12 FROM inserted
13 WHERE Users.user_ID = inserted.user_ID;
14 GO
```

ЗАКЛЮЧЕНИЕ

Изучение процедурного расширения языка SQL – Transact-SQL является первым шагом в освоении одной из наиболее известных и популярных систем управления реляционными базами данных – Microsoft SQL Server. Эта СУБД позволяет эффективно работать с базами данных размером от персональных до корпоративных и успешно конкурирует с другими СУБД.

Флагманские версии Microsoft SQL Server поддерживают все этапы последовательной обработки данных:

- оперативную обработку транзакций (On-Line Transactional Processing);
- извлечение данных из OLTP-систем и загрузку их в хранилище;
- хранение в хранилище данных (Data Warehouse);
- оперативную аналитическую обработку (On-Line Analytical Processing);
- интеллектуальный анализ (добычу) данных (Data Mining).

Для начального знакомства с Microsoft SQL Server идеальной является свободно распространяемая версия Express, которую любой желающий может скачать с сайта корпорации Microsoft.

Выбор Express-версии в качестве программной платформы практикума естественным образом ограничил рамки учебного пособия рассмотрением функций системы оперативной обработки транзакций. Основная задача OLTP-систем – ввод, обновление и удаление данных в реляционной базе данных. Управление этими процессами и формирование запросов осуществляется на языке Transact-SQL.

Но даже при таких ограничениях в учебное пособие вошло далеко не все из того, что можно было бы написать о языке Transact-SQL и СУБД Microsoft SQL Server Express. Например, не рассматриваются конструкции, позволяющие назначать, отменять и удалять полномочия доступа пользователей к объектам базы данных, системные и пользовательские функции, сериализация транзакций, оптимизация запросов.

Поэтому в конце учебного пособия приведен библиографический список, который может быть полезен тем, кто желает продолжить изучение возможностей языка Transact-SQL и СУБД Microsoft SQL Server.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Бен-Ган, И. Microsoft SQL Server 2008. Основы Transact-SQL : пер. с англ. / Ицик Бен-Ган. – СПб. : БХВ-Петербург, 2009. – 432 с.

2. Бондарь, А. Г. Microsoft SQL Server 2012 / А. Г. Бондарь. – СПб. : БХВ-Петербург, 2013. – 608 с.

3. Малков, О. Б. Работа с СУБД MySQL : учеб. пособие / О. Б. Малков, М. В. Девятерикова. – Омск : Изд-во ОмГТУ, 2010. – 84 с.

4. Малков, О. Б. Базы данных: метод. указания к выполнению лаб. работ / О. Б. Малков, Е. Т. Гегечкори. – Омск : Изд-во ОмГТУ, 2007. – 112 с.

5. Петкович, Д. Microsoft SQL Server 2012. Руководство для начинающих : пер. с англ. / Д. Петкович. – СПб. : БХВ-Петербург, 2014. – 816 с.

6. Фленов, М. Е. Transact-SQL / М. Е. Фленов. – СПб. : БХВ-Петербург, 2006. – 576 с.

ПРИЛОЖЕНИЕ

Типы данных в SQL Server

Тип данных – главная характеристика элемента данных (столбца таблицы в базе данных, параметра в хранимой процедуре или локальной переменной). SQL Server поддерживает следующие типы данных:

- числовые;
- символьные;
- дата и время;
- двоичные строки;
- пространственные;
- другие (не входящие в перечисленные категории).

Числовые типы используются для хранения чисел и имеют два подтипа:

- точные числовые;
- приближенные числовые.

К точным числовым типам (табл. П.1) относят логический тип *BIT*, целый тип *INT* и его вариации, а также денежный тип *MONEY* и его вариации:

Таблица П.1

Тип данных	Размер (байт)	Диапазон значений
BIT	1	0, 1, *NULL*
TINYINT	1	От 0 до 255 (от 0 до $2^8 – 1$)
SMALLINT	2	От –32 768 до 32 767 (от $–2^{15}$ до $2^{15} – 1$)
INT (INTEGER)	4	От – 2 147 683 648 до 2 147 683 647 (от $– 2^{31}$ до $2^{31} – 1$)
BIGINT	8	От $–2^{63}$ до $2^{63} – 1$
NUMERIC [(p[, s])] *DECIMAL [(p[, s])]*	5, 9, 13, 17	Десятичный тип. *p* – полное количество десятичных знаков до и после запятой, *s* – количество разрядов после запятой. Диапазон от $–10^{38} + 1$ до $10^{38} – 1$
SMALLMONEY	2	От –214 748.3648 до 214 748.3647
MONEY	4	От –922 337 203 685 477.5808 до 922 337 203 685 477.5807

К приближенным числовым типам (табл. П.2) относятся:

- *FLOAT* – представление чисел с плавающей запятой с двойной точностью;
- *REAL* – представление чисел с плавающей запятой с обычной точностью.

Таблица П.2

Тип данных	Размер (байт)	Диапазон значений
FLOAT [(n)]	4 или 8 (в зависимости от *n*)	*n* – количество разрядов. отводимых для хранения мантиссы. От $-1.79 \cdot 10^{308}$ до $-2.23 \cdot 10^{-308}$, 0 и от $2.23 \cdot 10^{-308}$ до $1.79 \cdot 10^{38}$
REAL	4	Соответствует *FLOAT(24)*. Диапазон от $-3.40 \cdot 10^{38}$ до $-1.18 \cdot 10^{-38}$, 0 и от $1.18 \cdot 10^{-38}$ до $3.40 \cdot 10^{38}$

Символьные типы имеют два подтипа:

- обычные символьные строки (табл. П.3);
- символьные строки в Unicode (табл. П.4):

Таблица П.3

Тип данных	Описание
CHAR[(n)]	Строка фиксированной длины. Максимальная длина n – 8000 символов. Если помещаемая строка короче n, она дополняется справа пробелами. Размер хранения составляет n байт
VARCHAR [(n | MAX)]	Строка переменной длины. Максимальное значение n – 8000. Если помещаемая строка короче n, то она не дополняется пробелами. Размер хранения – фактическая длина введенных данных плюс 2 байта. MAX указывает, что максимальный размер для хранения равен $2^{31}-1$ байт (2 ГБ). Формат VARCHAR(MAX) рекомендован вместо типа TEXT для хранения длинных текстовых строк
TEXT	Символьные данные переменной длины. Максимальная длина 2 147 483 647 символа (2 ГБ). Тип объявлен устаревшим

Таблица П.4

Тип данных	Описание
NCHAR[(n)]	То же, что и CHAR, но все символы хранятся в формате Unicode (на каждый символ отводится два байта)
NVARCHAR [(n | MAX)]	То же, что и VARCHAR, но все символы хранятся в формате Unicode (на каждый символ отводится два байта). Формат NVARCHAR(MAX) используется вместо типа NTEXT для хранения длинных текстовых строк
NTEXT	Символьные данные переменной длины в кодировке Unicode. Максимальная длина 1 073 741 823 символа. Тип объявлен устаревшим

Типы даты и времени приведены в табл. П.5:

Таблица П.5

Тип данных	Размер (байт)	Диапазон значений
DATE	3	Даты от 01.01.0001 г. до 31.12.9999 г. Формат отображения по умолчанию: гггг-мм-дд
TIME[(n)]	3, 4, 5	Время от 00:00:00.0000000 до 23:59:59.9999999. n – масштаб (от 0 до 7), количество сотен наносекунд (по умолчанию – 7)
DATETIME	8	Дата и время. Дата от 01.01.1753 г. до 31.12.9999 г. Время в диапазоне от 00:00:00 до 23:59:59.997
SMALL-DATETIME	4	Дата и время. Дата от 01.01.1900 г. до 06.06.2079 г. Время в диапазоне от 00:00:00 до 23:59:59
DATETIME2[(n)]	6, 7, 8	Дата и время. Дата от 01.01.0001 г. до 31.12.9999 г. Время в диапазоне от 00:00:00.0000000 до 23:59:59.9999999
DATETIME-OFFSET[(m, n)]	8, 9, 10	Дата и время. Дата от 01.01.0001 г. до 31.12.9999 г. Время в диапазоне от 00:00:00.0000000 до 23:59:59.9999999. m – точность (от 26 до 34), n – масштаб (от 0 до 7). Содержит смещение часового пояса (от -14:00 до +14:00)

Внимание! Разделителями между номером дня, номером месяца и годом может быть точка, знак минус или косая черта.

Для задания формата даты используется команда:
SET DATEFORMAT { mdy | dmy | ymd | myd | dym } ,

Внимание! Время представляется в 24-часовом формате: *чч:мм:сс.nnnnnn*. Если в литерале задается дата и время, то время отделяется от даты пробелом.

Двоичные строки предназначены для хранения произвольных двоичных данных. Есть три типа таких данных: *BINARY, VARBINARY, IMAGE* (табл. П.6):

Таблица П.6

Тип данных	Описание
IMAGE	Двоичные данные размером до 2 Гбайт. Тип объявлен устаревшим. Вместо него рекомендован формат *VARBINARY(MAX)*
BINARY [(n)]	Двоичные данные фиксированной длины *n* байт (от 1 до 8000). Размер при хранении составляет *n* байт
VARBINARY [(n | MAX)]	Двоичные данные переменной длины *n* байт (от 1 до 8000). Размер хранения – фактическая длина данных плюс 2 байта. *MAX* указывает, что максимальный размер при хранении составляет 2^{31} – 1 байт

Эти типы данных могут использоваться для хранения текстов в различных форматах, изображений, звука, видео, выполняемых файлов.

Внимание! Двоичная константа начинается с символов *0x*, за которыми следуют шестнадцатеричные цифры от 0 до 9 и буквы от *A* до *F* в любом регистре.

Пространственные типы данных *GEOMETRY* и *GEOGRAPHY* позволяют создавать геометрические фигуры (точки, линии, многоугольники). С их помощью можно задать фигуру, ее внешний вид и местоположение в некоторой системе координат. Данные этих типов хранятся в базе как поток двоичных данных.

Тип данных *GEOMETRY* применяется к плоским фигурам. Тип *GEOGRAPHY* данных используется для задания фигур, объектов и определения расстояний в условиях поверхности Земли (приплюснутый сфероид).

К другим типам данных традиционно отнесены все, не вошедшие в перечисленные выше категории – *TIMESTAMP, SQL_VARIANT, UNIQUEIDENTIFIER, HIERARCHYID, CURSOR, TABLE, XML* (табл. П.7):

Таблица П.7

Тип данных	Описание
TIMESTAMP	Метка времени. Размер хранения – 8 байт. Гарантируется уникальность величины в пределах одной базы данных. При внесении любых изменений в таблицу SQL Server увеличивает значение *TIMESTAMP* на 1. Тип не приемлем в качестве первичного ключа. Объявлен устаревшим
SQL_VARIANT	Позволяет хранить в переменной или столбце другие типы данных (числовые, строковые, пространственные, XML и др.)
UNIQUEIDENTIFIER	Уникальный глобальный идентификатор (GUID). Размер хранения – 16 байт. Для генерации идентификаторов используется время в миллисекундах, номера сетевых плат и другие параметры компьютера
HIERARCHYID	Используется для представления данных, имеющих иерархическую, древовидную структуру. Позволяет описывать отношения между данными в виде отношений родитель-потомок
CURSOR	Хранит указатель на набор данных, т. е. на множество строк, полученных из таблиц базы данных в результате запроса к базе. Не может быть использован для описания столбцов таблицы
TABLE	Позволяет временно хранить в табличном виде результаты обработки данных. Может быть использован в хранимых процедурах, функциях и обычных пакетах. Не может быть использован для описания столбцов таблицы
XML	Позволяет хранить данные в виде документов XML