Самоучитель

Алексей Жилинский

Microsoft® SQL Server 2008

Проектирование баз данных

Описание языка SQL

Программирование баз данных

Интеграция с .NET Framework

Описание служб MS SQL Server 2008

bhv

Алексей Жилинский

Самоучитель Microsoft® SQL Server 2008

Санкт-Петербург

«БХВ-Петербург»

2009

УДК 681.3.06
ББК 32.973.26-018.2
Ж72

Жилинский А. А.

Ж72 Самоучитель Microsoft SQL Server 2008. — СПб.: БХВ-Петербург, 2009. — 240 с.: ил.

ISBN 978-5-9775-0217-7

Рассмотрены основы работы с СУБД Microsoft SQL Server 2008, начиная с вопросов установки, создания и программирования баз данных и заканчивая описанием специальных возможностей SQL Server, включая интеграцию с .NET Framework, работу с XML и использование различных служб. Большое внимание уделено программированию на языке T-SQL. Описано создание и использование курсоров, хранимых процедур и триггеров, приведены табличные и скалярные функции. Уделено внимание способам построения отчетов, вопросам безопасности и администрирования, в том числе с использованием службы SQL Server Agent. Материал книги сопровождается большим количеством примеров.

Для начинающих программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.12.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 19,35.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0217-7

Оглавление

Введение

Система управления SQL Server 2008 представляет собой последнюю разработку фирмы Microsoft в области баз данных и анализа данных для быстрого создания масштабируемых решений электронной коммерции, бизнес-приложений и хранилищ данных или для любых других целей, где вам может потребоваться надежная система управления базами данных (СУБД). Эта книга научит вас создавать эффективные базы данных на SQL Server.

SQL Server 2008 является продолжением развития SQL Server, по сравнению с предыдущими версиями, обладает множеством изменений и улучшений. Эта версия не содержит столь же много изменений, как и SQL Server 2005 по сравнению с SQL Server 2000, но новые возможности еще больше упрощают и увеличивают эффективность разработчиков баз данных.

Эта книга состоит из нескольких частей.

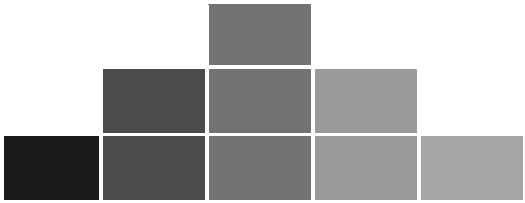
Из *части I* вы узнаете, как проектировать и создавать базы данных, как управлять их поведением. Здесь же описывается, как извлекать и изменять данные в базе данных. Знания, описанные в этой части, пригодятся вам при работе с любой базой данных, а не только с SQL Server.

В *части II* описывается программирование для баз данных. Здесь вы узнаете, как включить в вашу базу данных более сложную логику, чем простая выборка данных, как можно просто искать текст в базе данных и какие возможности вам дает интеграция с .NET Framework и XML.

Часть III посвящена администрированию сервера. В ней рассмотрена система безопасности SQL Server, а также конфигурирование и работа служб SQL Server.

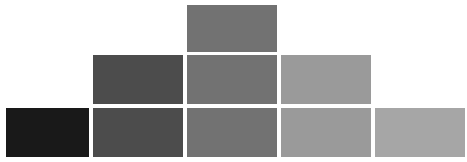
Книга снабжена множеством примеров для лучшего понимания, как работают изучаемые принципы. Также в книге описываются различия между возможностями SQL Server 2008 и его предыдущими версиями: SQL Server 2005 и SQL Server 2000. Эти версии еще некоторое время будут использоваться, и вы должны знать различия между ними.

Автор надеется, что эта книга поможет вам стать профессиональным разработчиком баз данных для SQL Server.



ЧАСТЬ I

Установка SQL Server и проектирование баз данных



ГЛАВА 1

Установка SQL Server

SQL Server 2008 представляет собой последнюю разработку фирмы Microsoft в области баз данных и предназначен для быстрого создания масштабируемых решений электронной коммерции, бизнес-приложений и хранилищ данных.

Для работы с материалом и примерами из книги вам потребуется одна из следующих версий SQL Server 2008:

- SQL Server 2008 Developer edition — это специальная версия, предназначенная для разработки баз данных;
- SQL Server 2008 Express edition — бесплатная версия SQL Server 2008, но с ограниченной функциональностью. Правда, вам будет достаточно этой версии для освоения материала книги.

Вы можете скачать любую из этих версий с сайта Microsoft **http://download.microsoft.com**.

Установка SQL Server 2008

Рассмотрим сценарий обычной установки SQL Server 2008.

1. Для запуска установки SQL Server 2008 вставьте DVD-диск в дисковод и дождитесь запуска мастера установки. Или запустите setup.exe.
2. После этого вам будет представлено лицензионное соглашение. Если вы согласитесь с ним, то мастер проверит наличие всех необходимых для установки SQL Server компонентов (например .NET Framework). Если необходимо, мастер их установит.

   В конце этого шага мастер установки будет выглядеть так, как показано на рис. 1.1. Вы сможете выбрать необходимые вам действия, и мастер установки выполнит их.
3. Выберите пункт **Installation** слева, а справа — **New SQL Server stand-alone installation**. Это запустит процесс установки SQL Server.

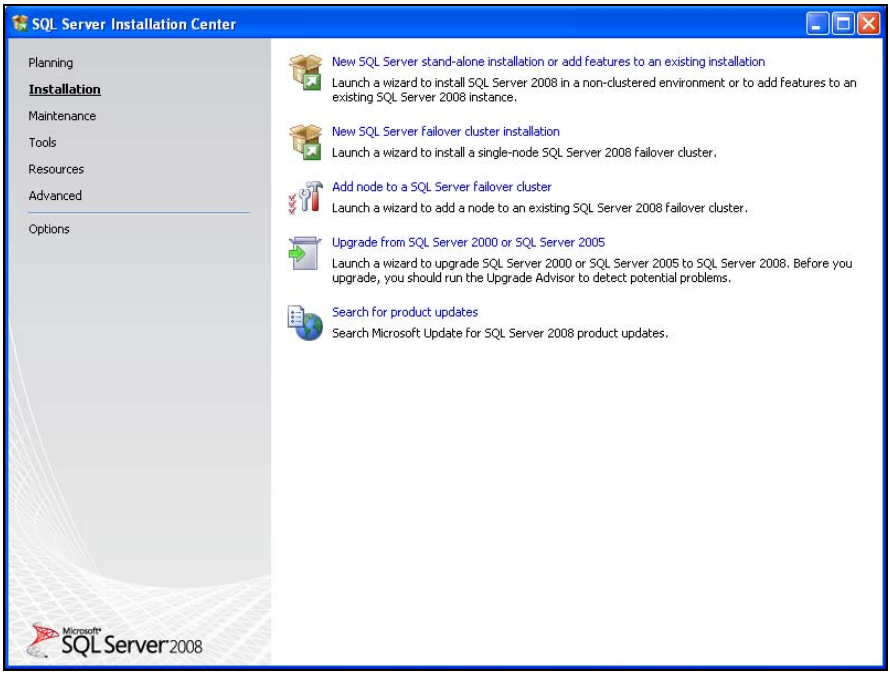


Рис. 1.1. Выбор нужного действия в мастере установки SQL Server 2008

4. Нажмите кнопку **Next**, и мастер начнет проверять систему на наличие в ней всех необходимых компонентов для работы SQL Server. Если все системные требования выполнены и ваш компьютер подходит для установки SQL Server, кнопка **Next** будет доступна для нажатия.
5. После этого вам необходимо выбрать компоненты SQL Server (рис. 1.2), которые вы хотите установить. Просто отметьте все флажки. Это приведет к установке как сервера, так и компонентов для его управления.
6. Отметьте флажок **Default instance** на следующей странице мастера и нажмите кнопку **Next**.
7. После этого вам необходимо определить, под какой учетной записью будет работать SQL Server и службы, автоматически стартующие при загрузке системы. Выберите учетные записи, под которыми будут работать службы SQL Server. Для простоты вы можете ввести имя (логин) и пароль любой учетной записи с правами администратора (рис. 1.3).
8. На следующей странице мастера вам необходимо задать, какой тип аутентификации будет использовать SQL Server. Выберите **Windows authentication mode** и нажмите кнопку **Next**.

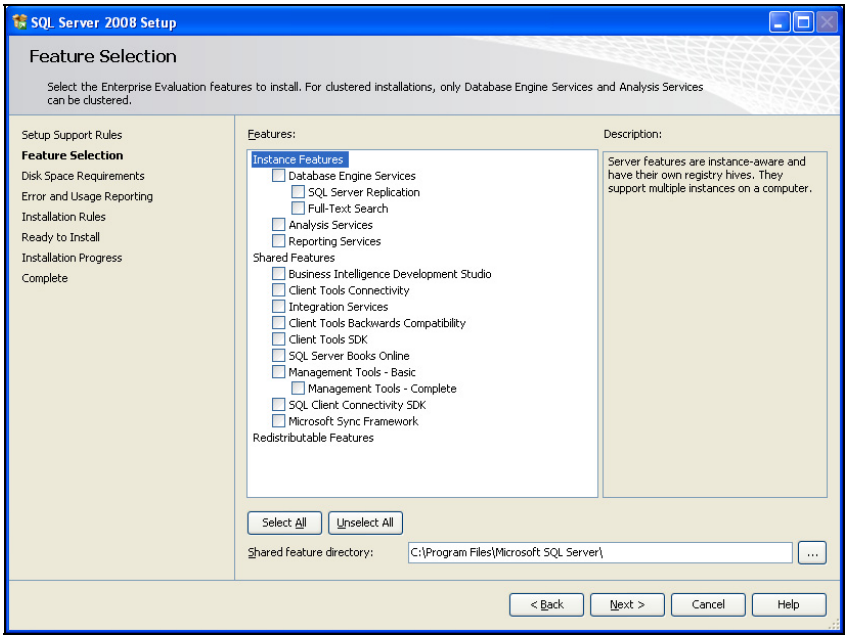


Рис. 1.2. Выбор устанавливаемых компонентов SQL Server

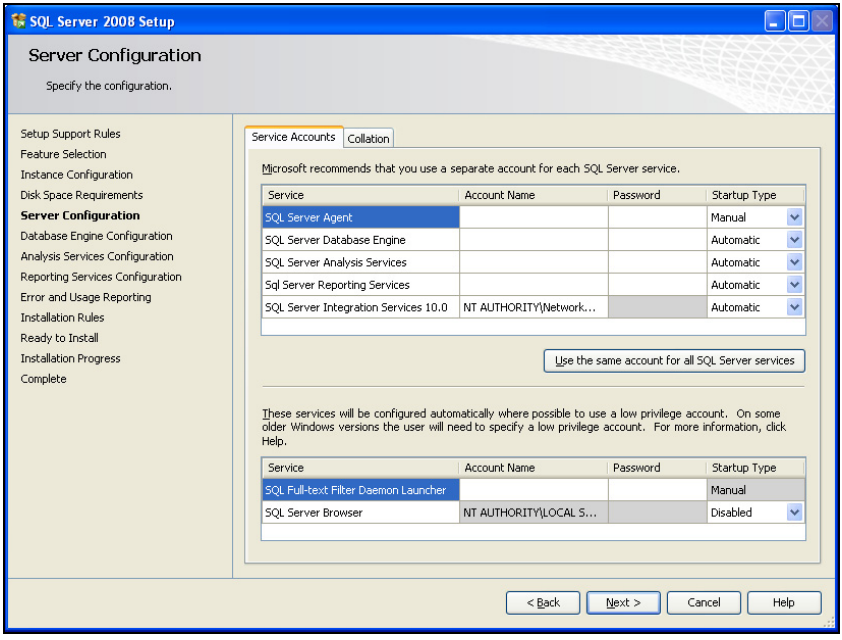


Рис. 1.3. Настройка параметров работы служб SQL Server

9. После этого появится страница настройки параметров службы отчетов. Оставьте установку конфигурации по умолчанию и нажмите кнопку **Next**. Если позже, при проверке возможности установки, мастер выдаст сообщение о невозможности установить сервер отчетов, вернитесь обратно к этой странице мастера и задайте режим установки без настройки сервера отчетов.
10. Появится страница настройки отправления сообщений об ошибках при установке SQL Server в Microsoft. Эти сообщения позволяют сделать следующие версии SQL Server более стабильными и удобными в работе. Просто нажмите кнопку **Next**.
11. После этого мастер проверит возможность установки всех выбранных компонентов на ваш компьютер. Если все хорошо, нажмите кнопку **Next** (рис. 1.4). Если мастер сообщит о невозможности установить какой-либо компонент, то устраните описанные им проблемы и попробуйте снова.
12. Теперь появится страница со списком всех компонентов, которые будут установлены. Нажмите кнопку **Install**, откроется страница мастера, на которой будет отображаться процесс установки различных компонентов SQL Server. По завершении установки нажмите кнопку **Finish**.

После установки SQL Server в меню **Пуск | Все программы | Microsoft SQL Server 2008** у вас окажется несколько новых значков, в частности, SQL Server Management Studio — утилита для создания и ведения баз данных и запросов.

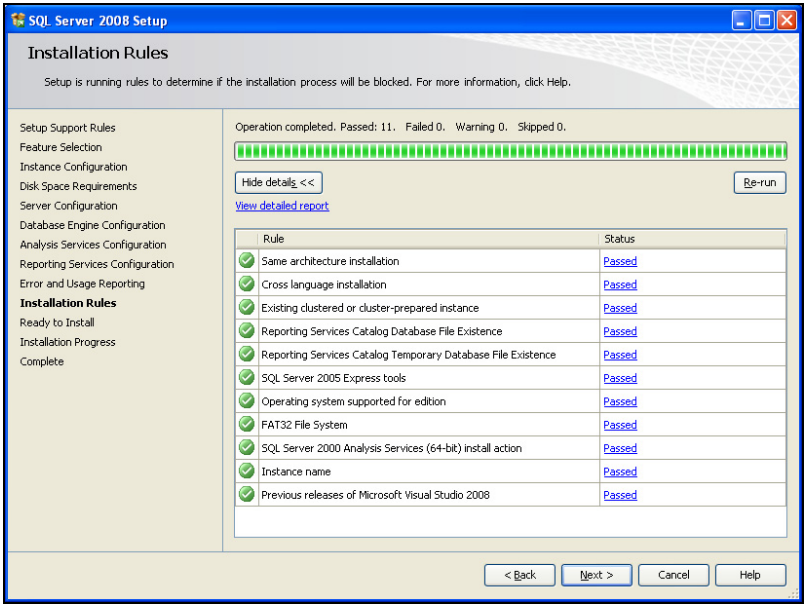
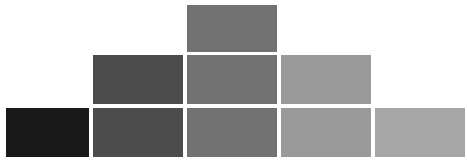


Рис. 1.4. Проверка возможности установки компонентов SQL Server



ГЛАВА 2

Введение в проектирование баз данных

Основные сведения

Термин *"реляционный"* означает "основанный на отношениях". Реляционная база данных состоит из сущностей (таблиц), находящихся в некотором отношении друг с другом. Название произошло от английского слова *relation* — отношение.

Проектирование базы данных состоит из двух основных фаз: логического и физического моделирования.

Во время логического моделирования вы собираете требования и разрабатываете модель базы данных, не зависящую от конкретной СУБД (системы управления реляционными базами данных). Это похоже на то, как если бы вы создавали чертежи вашего дома. Вы могли бы продумать и начертить все: где будет кухня, спальни, гостиная. Но это все на бумаге и в макетах.

Во время физического моделирования вы создаете модель, оптимизированную для конкретного приложения и СУБД. Именно эта модель реализуется на практике. Если вернуться к дому из предыдущего абзаца, на этом этапе вам придется строить где-нибудь дом — таскать бревна, кирпичи...

Процесс проектирования базы данных состоит из следующих этапов:

- ❒ сбор информации;
- ❒ определение сущностей;
- ❒ определение атрибутов для каждой сущности;
- ❒ определение связей между сущностями;
- ❒ нормализация;
- ❒ преобразование к физической модели;
- ❒ создание базы данных.

Первые 5 этапов образуют фазу логического проектирования, а остальные два — фазу физического моделирования.

Логическая фаза

Логическая фаза состоит из нескольких этапов. Далее они все рассмотрены.

Сбор требований

На этом этапе вам необходимо точно определить, как будет использоваться база данных и какая информация будет в ней храниться. Соберите как можно больше сведений о том, что система должна делать и чего не должна.

Определение сущностей

На этом этапе вам необходимо определить сущности, из которых будет состоять база данных.

Сущность — это объект в базе данных, в котором хранятся данные. Сущность может представлять собой нечто вещественное (дом, человек, предмет, место) или абстрактное (банковская операция, отдел компании, маршрут автобуса). В физической модели сущность называется *таблицей*.

Сущности состоят из *атрибутов* (столбцов таблицы) и *записей* (строк в таблице).

Обычно базы данных состоят из нескольких основных сущностей, связанных с большим количеством подчиненных сущностей. Основные сущности называются *независимыми*: они не зависят ни от какой-либо другой сущности. Подчиненные сущности называются *зависимыми*: для того чтобы существовала одна из них, должна существовать связанная с ней основная таблица.

На диаграммах сущности обычно представляются в виде прямоугольников. Имя сущности указывается внутри прямоугольника (рис. 2.1).

Рис. 2.1. База данных "Дома". Сущности

Любая таблица имеет следующие характеристики:

- в ней нет одинаковых строк;
- все столбцы (атрибуты) в таблице должны иметь разные имена;

- элементы в пределах одной колонки имеют одинаковый тип (строка, число, дата);
- порядок следования строк в таблице может быть произвольным.

На этом этапе вам необходимо выявить все категории информации (сущности), которые будут храниться в базе данных.

Пример базы данных с некоторыми сущностями показан на рис. 2.1.

Определение атрибутов

Атрибут представляет свойство, описывающее сущность. Атрибуты часто бывают числом, датой или текстом. Все данные, хранящиеся в атрибуте, должны иметь одинаковый тип и обладать одинаковыми свойствами.

В физической модели атрибуты называют *колонками*.

После определения сущностей необходимо определить все атрибуты этих сущностей.

На диаграммах атрибуты обычно перечисляются внутри прямоугольника сущности. На рис. 2.2 вы найдете пример базы данных "Дома", только теперь для сущностей из этой базы определены некоторые атрибуты.

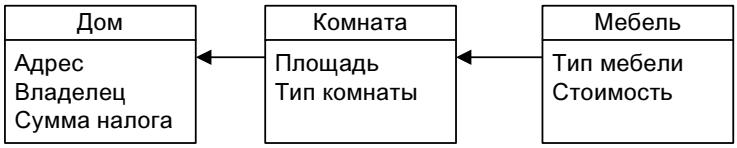


Рис. 2.2. База данных "Дома". Сущности и атрибуты

Для каждого атрибута определяется тип данных, их размер, допустимые значения и любые другие правила. К их числу относятся правила обязательности заполнения, изменяемости и уникальности.

Правило обязательности заполнения определяет, является ли атрибут обязательной частью сущности. Если атрибут является необязательной частью сущности, то он может принимать `NULL`-значение, иначе — нет.

Также вы должны определить, является ли атрибут изменяемым. Значения некоторых атрибутов не могут измениться после создания записи.

И, наконец, вам нужно определить, является ли атрибут уникальным. Если это так, то значения атрибута не могут повторяться.

Ключи

Ключом (key) называется набор атрибутов, однозначно определяющий запись. Ключи делятся на два класса: простые и составные.

Простой ключ состоит только из одного атрибута. Например, в базе "Паспорта граждан страны" номер паспорта будет простым ключом: ведь не бывает двух паспортов с одинаковым номером.

Составной ключ состоит из нескольких атрибутов. В той же базе "Паспорта граждан страны" может быть составной ключ со следующими атрибутами: фамилия, имя, отчество, дата рождения. Это — как пример, т. к. этот составной ключ, теоретически, не обеспечивает гарантированной уникальности записи.

Также существует несколько типов ключей, о которых рассказано далее.

Возможный ключ

Возможный ключ представляет собой любой набор атрибутов, однозначно идентифицирующих запись в таблице. Возможный ключ может быть простым или составным.

Каждая сущность должна иметь, по крайней мере, один возможный ключ, хотя таких ключей может быть и несколько. Ни один из атрибутов первичного ключа не может принимать неопределенное (`NULL`) значение.

Возможный ключ называется также суррогатным.

Первичные ключи

Первичным ключом называется совокупность атрибутов, однозначно идентифицирующих запись в таблице (сущности). Один из возможных ключей становится первичным ключом. На диаграммах первичные ключи часто изображаются выше основного списка атрибутов или выделяются специальными символами. Сущность на рис. 2.3 имеет как ключевые, так и обычные атрибуты.

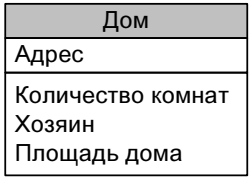


Рис. 2.3. Сущность с первичным ключом

Альтернативные ключи

Любой возможный ключ, не являющийся первичным, называется *альтернативным ключом*. Сущность может иметь несколько альтернативных ключей.

Внешние ключи

Внешним ключом называется совокупность атрибутов, ссылающихся на первичный или альтернативный ключ другой сущности. Если внешний ключ не связан с первичной сущностью, то он может содержать только неопределенные значения. Если при этом ключ является составным, то все атрибуты внешнего ключа должны быть неопределенными.

На диаграммах атрибуты, объединяемые во внешние ключи, обозначаются специальными символами. На рис. 2.4 изображены две связанные сущности (*Дома* и их *Хозяева*) и образованные ими внешние ключи (ведь один человек может владеть больше, чем одним домом).

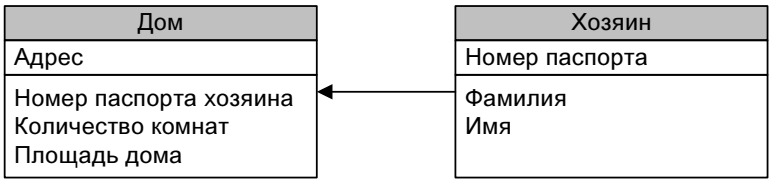


Рис. 2.4. Сущность с внешним ключом

Ключи являются логическими конструкциями, а не физическими объектами. В реляционных базах данных предусмотрены механизмы, обеспечивающие сохранение ключей.

Определение связей между сущностями

Реляционные базы данных позволяют объединять информацию, принадлежащую разным сущностям.

Отношение — это ситуация, при которой одна сущность ссылается на первичный ключ второй сущности. Как, например, сущности *Дом* и *Хозяин* на рис. 2.4.

Отношения определяются в процессе проектирования базы. Для этого следует проанализировать сущности и выявить логические связи, существующие между ними.

Тип отношения определяет количество записей сущности, связанных с записью другой сущности. Отношения делятся на три основных типа, о которых рассказано далее.

Один-к-одному

Каждой записи первой сущности соответствует только одна запись из второй сущности. А каждой записи второй сущности соответствует только одна запись из первой сущности. Например, есть две сущности: *Люди* и *Свидетельства о рождении*. И у одного человека может быть только одно свидетельство о рождении.

Один-ко-многим

Каждой записи первой сущности могут соответствовать несколько записей из второй сущности. Однако каждой записи второй сущности соответствует только одна запись из первой сущности. Например, есть две сущности: *Заказ* и *Позиция заказа*. И в одном заказе может быть много товаров.

Многие-ко-многим

Каждой записи первой сущности могут соответствовать несколько записей из второй сущности. Однако и каждой записи второй сущности может соответствовать несколько записей из первой сущности. Например, есть две сущности: *Автор* и *Книга*. Один автор может написать много книг. Но у книги может быть несколько авторов.

По критерию обязательности отношения делятся на обязательные и необязательные.

- *Обязательное отношение* означает, что для каждой записи из первой сущности непременно должны присутствовать связанные записи во второй сущности.
- *Необязательное отношение* означает, что для записи из первой сущности может и не существовать записи во второй сущности.

Нормализация

Нормализацией называется процесс удаления избыточных данных из базы данных. Каждый элемент данных должен храниться в базе в одном и только в одном экземпляре. Существует пять распространенных форм нормализации. Как правило, база данных приводится к третьей нормальной форме.

В процессе нормализации выполняются определенные действия по удалению избыточных данных. Нормализация повышает быстродействие, ускоряет сортировку и построение индекса, уменьшает количество индексов на сущность, ускоряет операции вставки и обновления.

Нормализованная база данных обычно отличается большей гибкостью. При модификации запросов или сохраняемых данных в нормализованную базу обычно приходится вносить меньше изменений, а внесение изменений имеет меньше последствий.

Первая нормальная форма

Чтобы преобразовать сущность в первую нормальную форму, следует исключить повторяющиеся группы значений и добиться того, чтобы каждый атрибут содержал только одно значение, списки значений не допускаются. Другими словами, каждый атрибут в сущности должен храниться только в одном экземпляре.

Например, на рис. 2.5 сущность *Дом* не нормализована. Она содержит несколько атрибутов для хранения данных о владельцах дома.

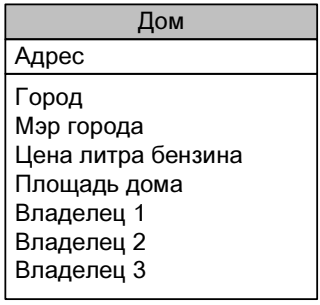


Рис. 2.5. Сущность *Дом*, не соответствующая первой нормальной форме

Для приведения сущности *Дом* в первую нормальную форму необходимо удалить повторяющиеся группы значений, т. е. удалить атрибуты *Владелец 1—3*, поместив их в отдельную сущность. Результат представлен на рис. 2.6.

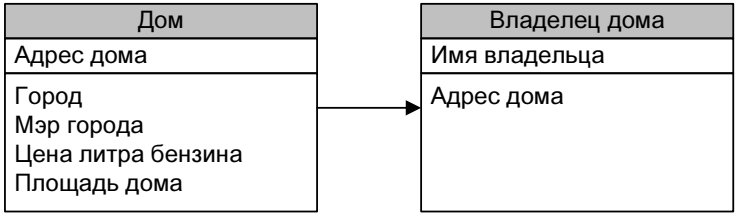


Рис. 2.6. Сущность *Дом*, приведенная к первой нормальной форме

Вторая нормальная форма

Таблица во второй нормальной форме содержит только те данные, которые к ней относятся. Значения неключевых атрибутов сущности зависят от первичного ключа. Если более точно, то атрибуты зависят от первичного ключа, от всего первичного ключа и только от первичного ключа.

Для соответствия второй нормальной форме сущности должны быть в первой нормальной форме.

Например, у сущности *Дом* на рис. 2.6 есть атрибут *Цена литра бензина*, который не имеет ничего общего с домами. Этот атрибут удаляется (или вы можете перенести его в другую сущность). А также мы переносим атрибут *Мэр* в отдельную сущность — этот атрибут зависит от города, где находится дом, а не от дома.

На рис. 2.7 изображена сущность *Дом* во второй нормальной форме.

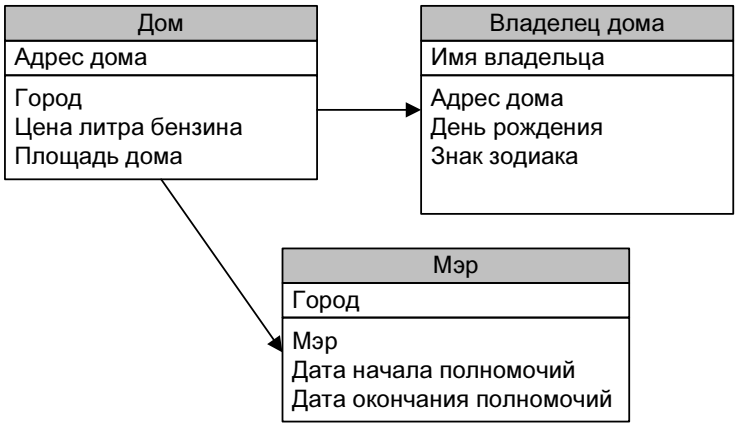


Рис. 2.7. Сущность *Дом*, приведенная ко второй нормальной форме

Третья нормальная форма

В третьей нормальной форме исключаются атрибуты, не зависящие от всего ключа. Любая сущность, находящаяся в третьей нормальной форме, находится также и во второй. Это самая распространенная форма базы данных.

В третьей нормальной форме каждый атрибут зависит от ключа, от всего ключа и ни от чего, кроме ключа.

Например, у сущности *Владелец дома* на рис. 2.7 есть атрибут *Знак зодиака*, который зависит от даты рождения владельца дома, а не от его имени (которое является ключом).

Для приведения сущности *Владелец дома* необходимо создать сущность *Знаки зодиака* и перенести туда атрибут *Знак зодиака* (рис. 2.8).

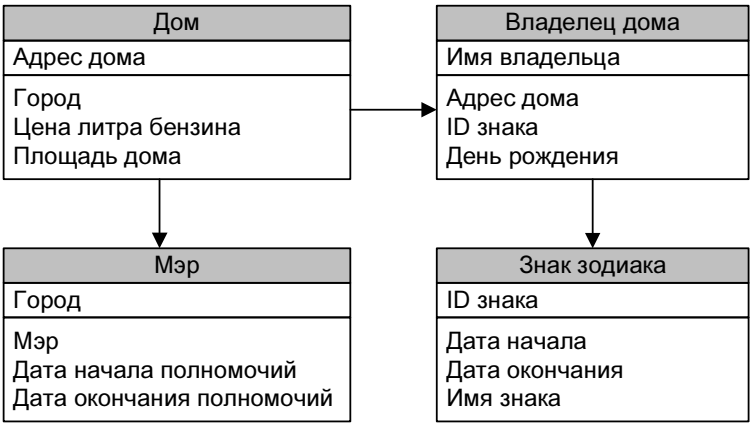


Рис. 2.8. Сущность *Владелец дома*, приведенная к третьей нормальной форме

Ограничения

Ограничения (constrains) — это правила, за соблюдением которых следит система управления базы данных. Ограничения определяют множество значений, которые можно вводить в столбец или столбцы.

Например, вы не хотите, чтобы сумма заказа в вашем очень крутом магазине была бы меньше 500 рублей. Вы просто устанавливаете ограничение на колонку *Сумма заказа*.

Хранимые процедуры

Хранимые процедуры (stored procedures) — это предварительно откомпилированные процедуры, хранящиеся в базе данных. Хранимые процедуры можно использовать для определения деловых правил, с их помощью можно осуществлять более сложные вычисления, чем с помощью одних лишь ограничений.

Хранимые процедуры могут содержать логику хода выполнения программы, а также запросы к базе данных. Они могут принимать параметры и возвращать результаты в виде таблиц или одиночных значений.

Хранимые процедуры похожи на обычные процедуры или функции в любой программе.

Примечание

Хранимые процедуры находятся в базе данных и выполняются на сервере базы данных. Как правило, они быстрее операторов SQL, поскольку хранятся в компилированном виде.

Целостность данных

Организовав данные в таблицы и определив связи между ними, можно считать, что была создана модель, правильным образом отражающая бизнес-среду. Теперь нужно обеспечить, чтобы данные, вводимые в базу, давали правильное представление о состоянии дела. Иными словами, нужно обеспечить выполнение *деловых правил* и поддержку *целостности* (integrity) базы данных.

Например, ваша компания занимается доставкой книг. Вы вряд ли примете заказ от неизвестного клиента, ведь тогда вы даже не сможете доставить заказ. Отсюда бизнес-правило: заказы принимаются только от клиентов, информация о которых есть в базе данных.

Корректность данных в реляционных базах обеспечивается набором правил. Правила целостности данных делятся на четыре категории.

- Целостность сущностей — каждая запись сущности должна обладать уникальным идентификатором и содержать данные. Ведь надо же вам как-то различать все эти записи в базе данных.
- Целостность атрибутов — каждый атрибут принимает лишь допустимые значения. Например, сумма покупки, определенно, не может быть меньше нуля.
- Ссылочная целостность — набор правил, обеспечивающих логическую согласованность первичных и внешних ключей при вставке, обновлении и удалении записей. Ссылочная целостность обеспечивает, чтобы для каждого внешнего ключа существовал соответствующий первичный ключ. Возьмем предыдущий пример с сущностями *Владелец дома* и *Дом*. Допустим, вы Вася Иванов и владеете домом. Вы сменили фамилию на Сидоров и внесли соответствующие изменения в сущность *Владелец дома*. Определенно вы бы хотели, чтобы ваш дом продолжал числиться за вами под вашим новым именем, а не принадлежал некоему Васе Иванову, которого уже не существует.
- Пользовательские правила целостности — любые правила целостности, не относящиеся ни к одной из перечисленных категорий.

Триггеры

Триггер — это аналог хранимой процедуры, который вызывается автоматически при изменении данных в таблице.

Триггеры являются мощным механизмом для поддержания целостности базы данных. Триггеры вызываются до или после изменения данных в таблице. С помощью триггеров вы можете не только отменить эти изменения, но и изменить данные в любой другой таблице.

Например, вы создаете интернет-форум, и вам необходимо сделать так, чтобы в списке форумов показывалось последнее сообщение форума. Конечно, вы можете брать сообщение из сущности *Сообщения форума*, но это увеличит сложность вашего запроса и время его выполнения. Проще добавить триггер к сущности *Сообщения форума*, который бы записывал последнее добавленное сообщение в сущность *Форумы*, в атрибут *Последнее сообщение*. Это сильно упростит запрос.

Деловые правила

Деловые правила определяют ограничения, накладываемые на данные в соответствии с требованиями бизнеса (тех, для кого вы создаете базу). Деловые правила могут состоять из набора шагов, необходимых для выполнения определенной задачи, или же они могут быть просто проверками, которые контролируют правильность введенных данных. Деловые правила могут включать правила целостности данных. В отличие от других правил, их главная цель — обеспечить правильное ведение деловых операций.

Например, в компании "Очень крутые парни" может быть так принято, что закупаются для служебных нужд только белые, синие и черные автомобили. Тогда деловое правило для атрибута *Цвет автомобиля* сущности *Служебные автомобили* будет гласить, что автомобиль может быть только белым, синим или черным.

Большинство СУБД предоставляют средства:

- для указания значений по умолчанию;
- для проверки данных перед занесением их в базу;
- для поддержания связей между таблицами;
- для обеспечения уникальности значений;
- для хранения хранимых процедур непосредственно в базе.

Все эти возможности можно применять для реализации деловых правил в базе данных.

Физическая модель

Следующим шагом, после создания логической модели, является построение физической модели. *Физическая модель* — это практическая реализация базы данных. Физическая модель определяет все объекты, которые вам предстоит реализовать.

При переходе от логической модели к физической сущности преобразуются в таблицы, а атрибуты в столбцы.

Отношения между сущностями можно преобразовать в таблицы или оставить как внешние ключи.

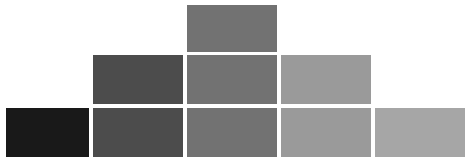
Первичные ключи преобразуются в ограничения первичных ключей. Возможные ключи — в ограничения уникальности.

Денормализация

Денормализация — это умышленное изменение структуры базы, нарушающее правила нормальных форм. Обычно это делается с целью улучшения производительности базы данных.

Теоретически, надо всегда стремиться к полностью нормализованной базе, однако на практике полная нормализация базы почти всегда означает падение производительности. Чрезмерная нормализация базы данных может привести к тому, что при каждом извлечении данных придется обращаться к нескольким таблицам. Обычно в запросе должны участвовать четыре таблицы или менее.

Стандартными приемами денормализации являются: объединение нескольких таблиц в одну, сохранение одинаковых атрибутов в нескольких таблицах, а также хранение в таблице сводных или вычисляемых данных.



ГЛАВА 3

Выборка данных

Хранение данных в базе данных принесет пользу лишь в том случае, если вы сможете получить эти данные. Для обращения к данным в Microsoft SQL Server используется язык SQL.

SQL (Structured Query Language, язык структурированных запросов) — язык, используемый для запросов, обновления и управления реляционной базы данных. Диалекты языка SQL, используемые в различных реляционных базах данных, имеют некоторые различия, но в целом схожи. Диалект SQL, который используется в Microsoft SQL Server, называется Transact-SQL (T-SQL).

В SQL выборка данных осуществляется командой `SELECT`. С ее помощью вы можете получать строки из одной или нескольких таблиц базы данных.

Структура команды *SELECT*

Команда `SELECT` состоит из следующих основных частей:

- список выборки;
- `FROM` — определяет, откуда выбирать данные;
- `WHERE` — определяет условия выборки данных;
- `GROUP BY` — определяет условия группировки данных;
- `HAVING` — определяет условия выборки данных при использовании агрегатных функций;
- `ORDER BY` — определяет условия сортировки.

Все части команды, кроме списка выборки, являются необязательными. Эти части следуют в строго определенном порядке.

После выполнения команды `SELECT` вы получаете от SQL Server *итоговый набор* (таблицу с результатами запроса). Итоговый набор состоит из трех частей: столбцов, заголовков столбцов и записей. Итоговый набор может

содержать один или несколько столбцов. Кроме того, он может содержать пустые заголовки столбцов, а также строки с данными.

Другими словами, если вы захотите получить список покупок какого-нибудь Васи Петрова, то вы получите список в виде таблицы, с колонками "имя покупателя", "сумма заказа" и другими, которые вам нужны. Также в итоговом наборе, который вы получите, будут сами имена колонок, даже если нет ни одной записи о покупках (т. е. Вася Петров никогда ничего не покупал).

Имена таблиц и столбцов могут быть настроены так, чтобы в них различался или не различался регистр символов. В ключевых словах SQL регистр символов не имеет значения. В SQL Server имена могут заключаться в квадратные скобки, например, `[имя_объекта]`. Приведем две одинаковых команды `SELECT`, с квадратными скобками и без них:

```
SELECT [id], [НазваниеТовара] FROM [Покупки]
select id, НазваниеТовара from [Покупки]
```

Далее будет подробно рассмотрена каждая часть команды `SELECT`.

Список выборки

В списке выборки вы указываете, какие столбцы включаются в итоговый набор команды `SELECT`. Столбцы возвращаются в порядке их перечисления в списке выборки.

Например:

```
SELECT Колонка1, Колонка2, Колонка3
```

При выполнении команды `SELECT` возвращается таблица результатов (итоговый набор). Таблица может содержать константы, столбцы, функции, выражения и производные столбцы.

Например, следующая команда возвращает константу `'константа'`, столбец базы данных с именем `Column1`, функцию `@@SERVERNAME` и производный столбец `Column2 + Column3`:

```
SELECT 'константа', Column1, @@SERVERNAME, Column2 + Column3 FROM MyTabe
```

Чтобы список выборки содержал все столбцы таблицы, можно либо перечислить имена всех столбцов, либо включить в список выборки символ звездочки (`*`). Если список выборки содержит звездочку, возвращаются все столбцы всех таблиц, участвующих в запросе. Символ `*` может указываться наряду с ограничениями, столбцами и производными столбцами.

Следующий пример демонстрирует использование звездочки в команде `SELECT`:

```
SELECT * FROM MyTable
SELECT *, @@SERVERNAME, Column2 + Column3 FROM MyTable
```

По умолчанию команда `SELECT` возвращает все записи, включая дубликаты. Чтобы явно запросить все записи, поместите ключевое слово `ALL` перед списком выборки:

```
SELECT ALL 'константа', Column1, @@SERVERNAME, Column2 + Column3 FROM
MyTable
```

Без ключевого слова `ALL` всегда можно обойтись. Оно поддерживается только для того, чтобы T-SQL соответствовал стандарту ANSI.

Чтобы команда `SELECT` возвращала только уникальные записи, поставьте перед списком выборки ключевое слово `DISTINCT`. Ключевое слово `DISTINCT` (или `ALL`, если оно используется) должно находиться сразу же после ключевого слова `SELECT`:

```
SELECT DISTINCT 'константа', Column1, @@SERVERNAME, Column2 + Column3
FROM MyTable
```

Команда `SELECT` позволяет присвоить значения переменным или получить итоговый набор. В одной команде `SELECT` нельзя сделать и то, и другое. Все переменные в SQL имеют префикс `@`. Например, если `@a` — целая переменная, то следующая команда `SELECT` будет верна:

```
SELECT @a = 5
```

Однако следующая команда `SELECT` недопустима:

```
SELECT @a = 5, id FROM authors
```

При выполнении команды `SELECT` в итоговый набор для каждого возвращаемого столбца включается заголовок по умолчанию, совпадающий с его именем. Значения констант, функции и производные столбцы имеют пустые заголовки. Так в предыдущем примере для константы, функции `@@SERVERNAME` и `Column2 + Column3` был бы создан пустой заголовок, а для столбца `Column1` — заголовок `Column1`.

Вы можете изменить заголовки столбцов и назначить им псевдонимы в списке выборки. В SQL Server существуют три формы назначения заголовков для итогового набора. Во всех трех примерах далее заголовок колонки `Column1` — `MyColumn`:

```
MyColumn = Column1
Column1 MyColumn
Column1 AS MyColumn
```

Если заголовок столбца содержит какие-либо специальные символы (например, пробелы), его необходимо заключить в кавычки.

> ***Примечание***
>
> Microsoft SQL Server позволяет использовать в командах SQL как апострофы ('), так и кавычки ("). В стандарте ANSI для имен объектов и заголовков столбцов со специальными символами используются кавычки, а для всего остального — апострофы. Лучше следовать этим правилам.

Секция *FROM*

Секция `FROM` определяет источники данных для выборки. Вы можете обращаться к данным из таблиц, представлений или производных таблиц.

Без секции `FROM` команда `SELECT` может лишь присваивать значения переменным, возвращать переменные или выполнять функции. Приведем пример выборки данных из двух таблиц:

```
SELECT * FROM MyTable
```

Эта команда означает, что вы хотите выбрать все столбцы из таблицы `MyTable`. Для упрощения команд SQL можно воспользоваться псевдонимами таблиц.

Псевдонимы таблиц

Если имя таблицы длинное или трудное для запоминания, вы можете определить для нее *псевдоним*. Псевдоним (alias) представляет собой более короткое и легко запоминающееся имя таблицы. Если для таблицы задается псевдоним, то вы уже не сможете ссылаться на нее по имени в командах `SELECT`, если только таблица не упоминается в секции `FROM` несколько раз:

```
SELECT M.Column, MyTable.Column2 FROM MyTable M, MyTable
```

В этом примере первому экземпляру `MyTable` назначается псевдоним `M`. Секция `FROM` также содержит `MyTable` без псевдонима. Столбец `Column` берется из экземпляра-псевдонима `M`, а столбец `Column2` из обычного экземпляра `MyTable`. Следующий пример демонстрирует применение псевдонимов для сокращения длинного имени таблицы:

```
SELECT M.Column1 FROM MyVeryVeryVeryLongTableName AS M
```

В этом примере `MyVeryVeryVeryLongTableName` — таблица, а `M` — псевдоним.

Псевдонимы также применяются при создании производных таблиц. Производная таблица — это таблица, возникающая тогда, когда секция `FROM` вместо имени таблицы из базы данных содержит команду `SELECT`. В этом случае

данные производной таблицы состоят из данных, которые возвращаются командой SELECT. Если вы используете производную таблицу, ей необходимо назначить псевдоним:

```
SELECT * FROM (SELECT * FROM CustomTable) C
```

В этом примере создается производная таблица с псевдонимом C.

Псевдонимы таблиц и столбцов упрощают запись объектов, используемых в запросе. Полные имена объектов часто делают команды SQL излишне громоздкими.

Уточнение имен объектов

Уточненное имя таблицы записывается в форме:

ИмяСервера.ИмяБазы.ИмяВладельца.ИмяТаблицы

Здесь:

- *ИмяСервера* — по умолчанию текущий сервер;
- *ИмяБазы* — по умолчанию текущая база данных;
- *ИмяВладельца* — по умолчанию сначала проверяется, принадлежит ли таблица текущему пользователю; если не принадлежит, то по умолчанию используется владелец базы данных;
- *ИмяТаблицы* — имя таблицы; должно быть задано всегда.

Если какая-либо часть имени таблицы определена (например, имя сервера), то и все остальные части имени, находящиеся справа, тоже должны быть либо определены, либо должен использоваться заполнитель (. .).

Например:

```
pubs.dbo.books
pubs..books
otherserver.yourdb.superUser.TheTable
```

В первом примере pubs — имя базы данных, dbo — владелец, а books — имя таблицы.

Во втором примере pubs — база данных, владелец таблицы — текущий пользователь или владелец базы данных, а таблица — books.

В третьем примере используется другой сервер с именем otherserver, база данных yourdb, владелец superUser и таблица TheTable.

При использовании псевдонима таблицы уточненное имя столбца записывается в форме:

ПсевдонимТаблицы.ИмяСтолбца

Если псевдоним таблицы не используется, уточненное имя столбца записывается в форме:

ИмяСервера.ИмяБазы.ИмяВладельца.ИмяТаблицы.ИмяСтолбца

Если имя столбца задается без уточнения, оно должно быть уникальным для всех таблиц секции `FROM`, поскольку в противном случае SQL Server не сможет определить, из какой таблицы следует брать столбец. При указании уточненного имени уточняющий префикс столбца должен точно совпадать с уточненным именем таблицы.

При использовании псевдонима уточненное имя столбца записывается по следующей форме:

ПсевдонимТаблицы.ИмяСтолбца

Как и прежде, если псевдоним таблицы не указан, имя столбца должно быть уникальным.

В следующем фрагменте приведены примеры уточнения имен столбцов:

```
Column1
books.title
pubs.dbo. books.title
```

Первый пример ссылается на столбец с именем `Column1`. Второй пример ссылается на столбец `title` из таблицы `books`, а третий пример — на столбец `title` из таблицы `books`, принадлежащей владельцу базы, из базы данных `pubs`.

Секция *TOP*

Секция `TOP` позволяет при выполнении запроса вернуть только определенное число записей. Например:

```
SELECT TOP 15 * FROM [Книги]
```

Этот запрос вернет первые 15 записей из таблицы `Книги`. Секция `TOP` должна идти сразу после `SELECT`, до перечисления списка возвращаемых колонок. Если в запросе есть ключевые слова `ALL` или `DISTINCT`, то `TOP` идет после них. Аргументом секции `TOP` может быть число, переменная или выражение.

Например, переменная `@LastRowIndex` в следующем выражении определяет, сколько записей вернет выражение:

```
SELECT TOP @LastRowIndex * FROM [Книги]
```

Также вы можете вернуть определенный процент первых записей, используя ключевое слово `PERCENT`. Следующий пример при `@LastRowIndex=25` вернет первые 25% записей таблицы:

```
SELECT TOP @LastRowIndex PERCENT * FROM [Книги]
```

Также, в качестве аргумента секции `TOP`, вы можете использовать выражение. В этом случае выражение необходимо взять в скобки. Например:

```
SELECT TOP (@ЧислоНужныхКниг + 10) * FROM [Книги]
SELECT TOP (SELECT ЧислоОболтусовУчеников FROM Школа) * FROM [Книги]
```

Первый запрос возвращает определенное число книг, с небольшим запасом в 10 книг. Второй запрос возвращает столько книг, сколько оболтусов-учеников в школе.

> ***Особенности SQL Server 2000***
>
> В SQL Server 2000 вы не можете использовать переменную или выражение в секции `TOP`, только число-константу.
>
> Это ограничение можно обойти следующим способом:
>
> ```
> SET ROWCOUNT 15
> SELECT TOP @LastRowIndex * FROM [Книги]
> SET ROWCOUNT 0
> ```
>
> Здесь `SET ROWCOUNT 15` устанавливает, что запрос вернет только первые 15 записей, `SET ROWCOUNT 0` отменяет это ограничение.
>
> Но лучше не прибегать к таким ухищрениям в более поздних версиях SQL Server.

Секция *WHERE*

Секция `WHERE` определяет критерий отбора записей, включаемых в итоговый набор. Возвращаются только те записи, которые соответствуют условиям секции `WHERE`.

Секция `WHERE` представляет собой комбинацию одного или больше *предикатов* (предикат — это выражение, которое возвращает `TRUE`, `FALSE` или `Unknown`).

Пример предиката для получения списка автомобилей, у которых 8 или больше колес:

```
КоличествоКолес >= 8
```

Если использовать этот предикат в следующем запросе, вы получите список владельцев этих суперавтомобилей:

```
SELECT ИмяВладельца FROM [ВладельцыАвтомобилей] WHERE КоличествоКолес >= 8
```

Предикаты в секции `WHERE` соединены между собой логическими операторами `AND`, `OR` и `NOT`. Ключевое слово `NOT` реализует логическое отрицание условий.

В секции WHERE вы можете сравнивать столбцы, переменные, константы, функции и любые другие предикаты. Сравниваемые объекты не обязаны присутствовать в списке выборки. Выражения группируются с помощью скобок, что позволяет управлять порядком сравнений. Количество предикатов в секции WHERE не ограничено.

Вот пример более сложного выражения для выбора 8-колесных автомобилей с двумя двигателями:

```
SELECT ИмяВладельца FROM [ВладельцыАвтомобилей]
  WHERE КоличествоКолес >= 8 AND КоличествоДвигателей = 2
```

Для сравнения двух выражений используются операторы сравнения. Стандартные операторы сравнения перечислены в табл. 3.1.

Таблица 3.1. *Операторы сравнения*

Оператор	Значение
=	Равно
<>	Не равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
!>	Не больше
!<	Не меньше

Приведем еще несколько примеров команд:

```
SELECT title, price FROM books WHERE price = 19.99
SELECT title, price FROM books WHERE SORT(price) !< 19.99
SELECT title, price FROM books WHERE NOT ((price > 21.00) AND (price < 30.00))
```

Первая команда SELECT возвращает лишь те записи, в которых столбец price равен 19.99. Во втором примере для price вызывается функция SORT и возвращаются только те записи, для которых полученный результат не меньше 19.99. Последняя команда возвращает лишь те записи, для которых столбец price не входит в интервал от 21.00 до 30.00.

В секции WHERE могут использоваться различные другие секции.

Секция *IN*

Секция IN проверяет, принадлежит ли столбец набору (списку) значений. Команду IN всегда можно переписать в виде группы условий, объединяемых оператором OR. Далее приведены два примера использования секций IN:

```
SELECT title, price FROM books WHERE price IN (19.99, 21.00, 25.00, 29.99)
SELECT title, price FROM books WHERE price NOT IN (19.99, 21.00, 25.00, 29.99)
```

Первый пример возвращает записи, у которых столбец price равен 19.99, 21.00, 25.00 или 29.99. Во втором примере возвращаются те записи, у которых значение price отлично от 19.99, 21.00, 25.00 или 29.99.

Секция *LIKE*

Секция LIKE выполняет поиск по шаблону для символьных строк. Она используется в ситуациях, когда вас не интересует точное совпадение. Например, вы не знаете точного написания слова. Шаблон может содержать обычные и универсальные символы. В табл. 3.2 перечислены универсальные символы шаблонов и их смысл.

Таблица 3.2. *Универсальные символы в шаблонах*

Универсальный символ	Описание
%	Ноль и более символов
_ (подчеркивание)	Ровно один символ в строковом выражении
[*символы*]	Один символ, определяемый выражением в скобках. В скобках указывается интервал ([L-Z]) или список конкретных символов ([AbKl2M])
[^*символы*]	Похож на предыдущий, однако ищет строки, не содержащие символы из заданного списка или интервала

Универсальные символы можно комбинировать в шаблоне. Для поиска символов, используемых в качестве универсальных, применяется служебный символ, определяемый ключевым словом ESCAPE. Стандартного служебного символа не существует.

Следующие примеры демонстрируют применение секции LIKE:

```
-- найти все значения Column1, содержащие два и более символа
Column1 LIKE '_%_'
-- найти все значения Column1 с открывающей квадратной скобкой
Column1 LIKE '%\[%' ESCAPE '\'
-- найти различные варианты написания имени Smith
Column1 LIKE '[Ss]m[iy]th'
```

Первый пример ищет строку, которая содержит, по крайней мере, два символа. Во втором примере символ \ назначается служебным и используется для определения символа [. В результате секция LIKE находит все значения Column1, содержащие символ [. Последний пример ищет варианты написания слова "smith", начинающиеся со строчной или прописной буквы s и содержащие гласную i или y.

Секция *BETWEEN*

Секция BETWEEN определяет верхнюю и нижнюю границы интервала. Она применяется в случаях, когда вам известны минимальное и максимальное значения искомой величины. Если нижняя граница больше верхней, не возвращается ни одна запись. Выражения должны принадлежать к типам данных, неявно преобразуемым друг в друга, или их явное преобразование должно выполняться в команде SELECT (преобразование типов данных мы рассмотрим позже). Пример использования секции BETWEEN продемонстрирован в следующем фрагменте:

```
SELECT title, price FROM books WHERE price BETWEEN 10 AND 20
```

Команда находит все записи, у которых столбец price больше или равен 10 и меньше или равен 20.

Ключевое слово *NOT*

Ключевое слово NOT может включаться в любое сравнение секции WHERE. В этом случае проверяется условие, противоположное указанному. Например:

```
SELECT title, price FROM books WHERE price NOT BETWEEN 10 AND 20
```

Команда возвращает все записи, у которых значение price либо меньше 10, либо больше 20.

Проверка *NULL*

Для проверки, является ли значение колонки NULL или нет, используйте выражение IS NULL:

```
SELECT * FROM books WHERE notes IS NULL
```

> ***Внимание!***
>
> Не путайте неопределенное значение NULL с текстовой строкой "NULL". Пустая текстовая строка ("") тоже не является NULL-значением.

Также вы можете получить все строки, определенная колонка в которых не равна NULL:

```
SELECT * FROM books WHERE notes IS NOT NULL
```

> ***Особенности SQL Server 2000***
>
> В SQL Server 2000 вы можете узнать, содержит ли колонка NULL, используя выражение типа MyColumn=NULL.
>
> В более поздних версиях SQL Server можно использовать только выражение MyColumn IS NULL. Выражение MyColumn=NULL вернет Unknown.

Секция *ORDER BY*

При выборке данные возвращаются в порядке их нахождения в SQL Server. Ключевое слово ORDER BY позволяет определить порядок возвращения записей. Например, следующее выражение возвращает список автомобилей, отсортированный по номерам:

```
SELECT Цвет, Номер, Марка FROM Автомобили ORDER BY Номер
```

Столбцы, определяющие порядок записей, могут указываться с помощью заголовков столбцов, имен столбцов (даже если они отсутствуют в списке выборки) и целых чисел, определяющих порядок столбцов в списке выборки.

Вы можете определять порядок сортировки (по возрастанию или по убыванию), используя ключевые слова ASC и DESC. По умолчанию данные сортируются по возрастанию (ASC). Ключевые слова ASC и DESC относятся лишь к одному столбцу.

Также итоговый набор можно сортировать по нескольким столбцам.

Например, следующее выражение возвращает список автомобилей, отсортированный по номерам в обратном порядке, а потом отсортированный по марке автомобиля:

```
SELECT Цвет, Номер, Марка FROM Автомобили ORDER BY Номер DESC, Марка
```

Секция *GROUP BY*

Секция GROUP BY используется в сочетании с *агрегатными функциями* для получения обобщающих данных. Агрегатная функция обрабатывает данные нескольких записей и вычисляет обобщающий результат, возвращаемый как часть списка выборки.

При использовании GROUP BY все столбцы итогового набора либо являются агрегатными функциями, либо включаются в секцию GROUP BY.

Например:

```
SELECT OrderID, SUM([СтоимостьПозиции]) AS [СуммаЗаказа]
  FROM [Продажи]
  GROUP BY OrderID
  ORDER BY OrderID
```

В этом примере в итоговом наборе возвращается `ID` заказа и его сумма, которая получается путем суммирования стоимости всех позиций заказа. Здесь используется агрегатная функция `SUM`, которая возвращает сумму всех значений в колонке.

Агрегатные функции не могут работать со значениями типа `NULL`, за исключением функции `count(*)`.

Список всех агрегатных функций приведен в табл. 3.3.

Таблица 3.3. *Агрегатные функции*

Функция	Описание
Count([ALL | DISTINCT] *выражение*)	Вычисляет количество значений в столбце, отличных от NULL
Count(*)	Вычисляет количество выбранных записей. Эта функция учитывает NULL-значения
SUM([ALL | DISTINCT] *выражение*)	Вычисляет общую сумму в числовом столбце
AVG([ALL | DISTINCT] *выражение*)	Вычисляет среднее арифметическое разных значений в столбце
MAX(*выражение*)	Определяет максимум среди выбранных значений
MIN(*выражение*)	Определяет минимум среди выбранных значений
STDEV(*выражение*)	Вычисляет статистическое стандартное отклонение для всех значений выражения
STDEVP(*выражение*)	Возвращает смешанную оценку стандартного отклонения (квадратный корень от значения, возвращаемого функцией VARP) для всех значений столбца
VAR(*выражение*)	Возвращает несмешанную оценку дисперсии величин для всех значений выражения

Таблица 3.3 *(окончание)*

Функция	Описание
VARPD(*выражение*)	Возвращает смешанную оценку дисперсии величин для всех значений выражения
CHECKSUM(* | *выражение* [, ...*n*])	Возвращает контрольную сумму для строки таблицы или списка выражений. Эта функция появилась только в SQL Server 2005
CHECKSUM_AGG([ALL | DISTINCT] *выражение*)	Возвращает контрольную сумму значения выражения. Эта функция появилась только в SQL Server 2005

Секция GROUP BY имеет следующий синтаксис:

```
GROUP BY [ALL] список_группировки [WITH CUBE|ROLLUP]
```

Ключевое слово *ALL*

При использовании ключевого слова ALL в результате выборки будут включены все группы, независимо от того, соответствуют ли связанные с ними данные условиям выборки в разделе WHERE или нет. В строках, которые не соответствуют условиям выборки, во всех столбцах, кроме столбцов, по которым выполняется группировка, будут выведены значения NULL.

Следующий пример возвращает ID только тех продуктов, которых было заказано больше 10:

```
SELECT ProductID, AVG(Price) AS 'Средняя цена'
FROM [Заказы]
WHERE КоличествоЗаказов > 10
GROUP BY ProductID

ProductID   Средняя цена
----------- --------------------
708         19.2862
709         5.3243
711         19.2155
```

А этот пример возвращает ID всех продуктов, но для тех из них, которых было заказано меньше 10, возвращается NULL (как для продукта с ID 710):

```
SELECT ProductID, AVG(Price) AS 'Средняя цена'
FROM [Заказы]
WHERE КоличествоЗаказов > 10
```

```
GROUP BY ALL ProductID

ProductID   Средняя цена
----------- --------------------
708         19.2862
709         5.3243
710         NULL
711         19.2155
```

Ключевые слова *CUBE* и *ROLLUP*

Ключевые слова CUBE и ROLLUP предназначены для выполнения многомерного анализа данных в одной команде SQL. Они не могут одновременно присутствовать в командах SQL.

С помощью этой конструкции выполняется сверхагрегирование данных. Обычные функции агрегирования применяются к строкам группы и возвращают единственное значение для всей группы. При использовании ключевых слов CUBE или ROLLUP происходит вычисление функции агрегирования для значений, возвращаемых для каждой колонки.

CUBE создает итоговый набор, который показывает сводные показатели для всех комбинаций значений в выбранных колонках.

Каждое сочетание значений столбцов, входящих в секцию GROUP BY, порождает совокупность групп. Для каждого неповторяющегося значения столбца из списка группировки CUBE создает дополнительную запись, в которой столбцу присвоено значение NULL. Это значение NULL представляет все значения данного столбца.

Например, есть таблица с именем Table1:

```
Column 1    Column 2                                   Column 3
----------- ------------------------------------------ -----------
1           abc                                        1
1           def                                        2
1           ghi                                        3
1           abc                                        4
2           def                                        5
2           ghi                                        6
```

Далее представлен запрос с WITH CUBE и его результат:

```
SELECT [Column 1], [Column 2], SUM([Column 3])
FROM Table1
GROUP BY [Column 1], [Column 2] WITH CUBE
```

Результат запроса:

```
Column 1    Column 2                                  Column 3
----------- ----------------------------------------- -----------
1           abc                                       5
1           def                                       2
1           ghi                                       3
1           NULL                                      10
2           def                                       5
2           ghi                                       6
2           NULL                                      11
NULL        NULL                                      21
NULL        abc                                       5
NULL        def                                       7
NULL        ghi                                       9
```

`ROLLUP` создает итоговый набор, который демонстрирует сводные показатели для иерархии значений в выбранных колонках. `ROLLUP` не возвращает все возможные комбинации значений. Группировка `ROLLUP` образуется из столбцов, расположенных справа от значения текущего столбца.

```
SELECT [Column 1], [Column 2], SUM ([Column 3])
FROM Table1
GROUP BY [Column 1], [Column 2] WITH ROLLUP
```

Результат запроса:

```
Column 1    Column 2                                  Column 3
----------- ----------------------------------------- -----------
1           abc                                       5
1           def                                       2
1           ghi                                       3
1           NULL                                      10
2           def                                       5
2           ghi                                       6
2           NULL                                      11
NULL        NULL                                      21
```

Секция *HAVING*

Секция `HAVING` похожа на `WHERE`, однако она может использоваться только в сочетании с секцией `GROUP BY`. Секция `HAVING` содержит только те столбцы, которые присутствуют в списке выборки. В ней можно использовать агрегат-

ные функции, отсутствующие в списке выборки. Следующий пример демонстрирует применение секции HAVING:

```
SELECT [Дата], SUM([Сумма продажи])
FROM [Продажи]
GROUP BY [Дата]
HAVING SUM([Сумма продажи]) > 1000
```

Данный пример возвращает сумму продаж по дням, но только для тех дней, в которые сумма продаж была больше 1000.

Главное отличие между секциями HAVING и WHERE заключается в моменте их выполнения. Секция WHERE выполняется во время построения результата, а секция HAVING — после того, как будет создан промежуточный набор. Вот почему агрегатные функции могут использоваться в секции HAVING, но не в секции WHERE.

Секция *COMPUTE*

Секция COMPUTE позволяет включить в итоговый набор запроса основные и сводные данные. COMPUTE выполняет агрегатные функции для вычисления общих и промежуточных итогов. Пример использования:

```
SELECT CustomerID, OrderDate, SubTotal
FROM SalesOrderHeader
ORDER BY OrderDate
COMPUTE SUM(SubTotal)
```

Этот запрос сначала возвращает все данные, а потом сумму всех заказов. Вы также можете использовать COMPUTE BY для группировки результатов вычисления по столбцам, при этом все столбцы из этого списка должны присутствовать в секции ORDER BY в том же порядке. Столбцы нельзя пропускать, однако вы не обязаны использовать все столбцы из секции ORDER BY. Например, если в список секции ORDER BY входят столбцы LastName, FirstName и Zip, то секция COMPUTE BY может содержать LastName, или LastName и FirstName, или LastName, FirstName и Zip. Например:

```
SELECT SalesPersonID, CustomerID, OrderDate, SubTotal
FROM Sales.SalesOrderHeader
ORDER BY SalesPersonID, OrderDate
COMPUTE SUM(SubTotal) BY SalesPersonID
```

Выражения в секциях COMPUTE и COMPUTE BY должны совпадать с выражениями списка выборки. Например, если в COMPUTE используется выражение price*2, оно также должно присутствовать в выражении SELECT.

В секции COMPUTE нельзя использовать псевдонимы столбцов.

Может показаться, что вы используете агрегатную функцию для агрегатной функции, что обычно не разрешается. Следующий фрагмент демонстрирует применение секции COMPUTE:

```
SELECT type, SUM(price)
FROM books
GROUP BY type
COMPUTE AVG(SUM(price))
```

Команда возвращает type и SUM(price) для значений type в таблице books. Она создает завершающую запись, для которой выполняется секция COMPUTE. Эта запись содержит среднее арифметическое всех сумм price.

Команда *UNION*

С помощью ключевого слова UNION вы можете связать данные из двух и более итоговых наборов в один. UNION связывает воедино данные, полученные от нескольких команд SELECT. Команда UNION имеет следующий синтаксис:

```
Команда SELECT
UNION [ALL]
Команда SELECT
```

Все списки выборки в UNION должны содержать одинаковое число столбцов, при этом соответствующие столбцы должны принадлежать к одному типу данных (или преобразовываться к нему, явно или неявно), иначе SQL Server выдаст сообщение об ошибке. Неявные преобразования типов подчиняются правилам приоритетов. Первая команда SELECT в конструкции UNION определяет заголовки столбцов результата.

В каждой команде SELECT могут присутствовать секции GROUP BY или HAVING. Они относятся только к конкретной команде SELECT — вы не сможете применить их к союзу в целом.

Если вы используете секции ORDER BY или COMPUTE, они могут задаваться только в последней команде SELECT союза. Содержимое ORDER BY и COMPUTE относится ко всей конструкции UNION. Например:

```
SELECT [Дата], [Сумма продаж] FROM [Продажи отдела 1]
UNION
SELECT [Дата], [Общая выручка продаж] FROM [Оказанные услуги]
```

По умолчанию UNION возвращает только различающиеся записи. Ключевое слово ALL говорит о том, что возвращаться должны все записи без исключения. В одной команде можно смешивать UNION и UNION ALL.

Объединение таблиц

Довольно часто данные, которые требуется извлечь, находятся в нескольких таблицах. Конструкция `JOIN` позволяет создать объединение, т. е. выбрать данные из многих таблиц, представлений или производных таблиц.

Объединения делятся на три типа: ортогональные, внутренние и внешние.

Ортогональные объединения

Ортогональное объединение возвращает все записи из всех объединяемых таблиц. Общее количество возвращаемых записей равно произведению количества записей в каждой из таблиц. Предположим, в объединении участвуют три таблицы. Первая таблица возвращает 30 000 записей, а остальные — по 1000 записей каждая. Общее число записей, возвращаемых ортогональным объединением, будет равно 30 000×1000×1000, или 30 000 000 000 записей.

Конечно, при организации ортогональных объединений следует быть очень осторожными.

Для ортогонального объединения используются ключевые слова `CROSS JOIN`. Примеры ортогональных объединений:

```
SELECT p.ProjectID, u.Name AS UserName
FROM DoWork.Projects p
CROSS JOIN DoWork.Users u

SELECT p.ProjectID, u.Name AS UserName
FROM DoWork.Projects p, DoWork.Users u
```

Первая команда создает стандартное ортогональное объединение таблиц `Employee` и `Department` в стандарте ANSI. Вторая команда делает то же самое, но с применением расширений T-SQL.

> ***Внимание!***
>
> При разработке баз данных используйте объединения в стандарте ANSI. И тот и другой подход будут работать и в SQL Server 2000, и в более поздних версиях SQL Server, но компания Microsoft объявила, что в будущем поддержка расширений T-SQL будет прекращена.

Внутренние объединения

Внутреннее объединение является самым распространенным видом объединений. В нем используется оператор сравнения, ограничивающий число возвращаемых записей. Во внутреннем объединении вы задаете общие столбцы

(называемые общими ключами), используемые для связи таблиц. Пример внутреннего объединения в стандарте ANSI:

```
SELECT *
FROM DoWork.Projects AS p
    INNER JOIN DoWork.Users AS u ON p.UserID = u.UserID
```

После ключевого слова `ON` идет выражение с условием объединения. Это условие может содержать несколько частей, объединенных ключевыми словами `OR` и `AND`.

Пример внутреннего объединения, с помощью расширения T-SQL:

```
SELECT *
FROM DoWork.Projects AS p, DoWork.Users AS u
WHERE p.UserID = u.UserID
```

Внешние объединения

Внешние объединения тоже используются для связывания таблиц, однако результат возвращается только в том случае, когда условия объединения не выполняются. Внешние объединения делятся на три типа: левые, правые и полные. Вы можете осуществлять объединения как в стандарте ANSI, так и используя расширения T-SQL.

Левое внешнее объединение

Левое внешнее объединение возвращает все записи левой таблицы объединения даже в том случае, если записи в правой таблице не существуют. Если условие объединения не выполняется, столбцы правой таблицы заполняются неопределенными значениями `NULL`. Пример левого внешнего объединения в стандарте ANSI:

```
SELECT p.Name, pr.ProductReviewID
FROM Production.Product p
LEFT OUTER JOIN Production.ProductReview pr
    ON p.ProductID = pr.ProductID
```

Пример левого внешнего объединения с помощью расширения T-SQL:

```
SELECT p.Name, pr.ProductReviewID
FROM Production.Product p, Production.ProductReview pr
WHERE p.ProductID *= pr.ProductID
```

Команда возвращает все записи левой таблицы (`Products`) даже в том случае, если соответствующие записи правой таблицы (`ProductReview`) не существуют.

Правое внешнее объединение

Правое внешнее объединение возвращает все записи правой таблицы объединения даже в том случае, если записи в левой таблице не существуют. Если условие объединения не выполняется, столбцы правой таблицы заполняются неопределенными значениями NULL. Пример правого внешнего объединения в стандарте ANSI:

```
SELECT st.Name AS Territory, sp.SalesPersonID
FROM Sales.SalesTerritory st
RIGHT OUTER JOIN Sales.SalesPerson sp
    ON st.TerritoryID = sp.TerritoryID
```

Пример правого внешнего объединения с помощью расширения T-SQL:

```
SELECT st.Name AS Territory, sp.SalesPersonID
FROM Sales.SalesTerritory st, Sales.SalesPerson sp
WHERE st.TerritoryID =* sp.TerritoryID
```

Команда возвращает все записи правой таблицы (SalesPerson) даже в том случае, если соответствующие записи левой таблицы (SalesTerritory) не существуют.

Полное внешнее объединение

Полное внешнее объединение возвращает все записи левой и правой таблиц даже в том случае, если условие не выполняется. Там, где между таблицами находится совпадение, все столбцы заполняются соответствующими значениями. При отсутствии совпадения возвращаются значения одной таблицы и NULL — для столбцов другой таблицы.

Полное внешнее объединение можно задать только в стандарте ANSI. При попытке воспользоваться оператором *=* SQL Server выдаст сообщение об ошибке.

Пример полного внешнего объединения:

```
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
FULL OUTER JOIN Sales.SalesOrderDetail sod
    ON p.ProductID = sod.ProductID
```

Подзапросы

Подзапрос — это команда SELECT, создающая новый запрос, находясь в другом запросе.

В подзапросы могут передаваться столбцы основного запроса. Они используются для ограничения записей, возвращаемых подзапросом. Если таблица

принадлежит не внешнему запросу, а вложенному подзапросу, то столбцы этой таблицы могут использоваться только в подзапросе. Смотрите пример:

```
SELECT ProductID,
  (SELECT AVG (Price) FROM ProductPrices pp WHERE pp. ProductID =
p.ProductID)
  AS [Средняя цена]
FROM Products p
```

Подзапросы не могут содержать секций `ORDER BY`, `COMPUTE` или `FOR BROWSE`. Они могут присутствовать в секциях `FROM` в качестве производных таблиц, а также в секциях `WHERE` и `HAVING`.

Если подзапрос находится в секции `WHERE` или `HAVING`, он может использоваться в сочетании с ключевыми словами `EXIST`, `ANY`, `SOME`, `ALL` или `IN`.

Синтаксис команды *SELECT*

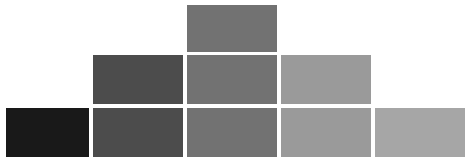
Формальный синтаксис команды `SELECT` приведен в листинге 3.1. А формальный синтаксис секции `WHERE` представлен в листинге 3.2.

Листинг 3.1. Синтаксис команды `SELECT`

```
SELECT [ALL | DISTINCT]
    [TOP значение [PERCENT] [WITH TIES]]
    список_выборки
    [INTO имя_новой_таблицы]
    [FROM {<имя_таблицы>} [, ...n]]
    [WHERE <условия_поиска>]
    [GROUP BY [ALL] условие_группировки [, ...n]
    [WITH {CUBE | ROLLUP}]
   ]
    [HAVING <условия_поиска>]
    [ORDER BY {условия_сортировки | список_выборки [ASC | DESC]}
  [, ...n]]
    [COMPUTE
  {{AVG | COUNT | MAX | MIN | SUM} (выражение)} [, ...n]
  [BY имя_столбца [, ...n]]
  [{UNION [ALL] | EXCEPT | INTERSECT}
        <query specification> | (<query expression>) [, ...n]]
```

Листинг 3.2. Синтаксис секции `WHERE`

```
[WHERE <условие_поиска>]

<условие_поиска> ::=
    {[NOT] <предикат> | ( <условие_поиска> )}
    [{AND | OR} [NOT] {<предикат> | ( <условие_поиска> )}]
[,...n]
<предикат> ::=
    {выражение {= | < > | ! = | > | > = | ! > | < | < = | ! <} выражение
    | строковое_выражение [NOT] LIKE строковое_выражение
  [ESCAPE 'escape_character']
    | выражение [NOT] BETWEEN выражение AND выражение
    | выражение IS [NOT] NULL
    | CONTAINS
    ({столбец | *} , '<содержимое_условия_поиска>')
    | FREETEXT ({столбец | *} , 'строка')
    | expression [NOT] IN (подзапрос | выражение [, ...n])
    | expression {= | < > | ! = | > | > = | ! > | < | < = | ! <}
  {ALL | SOME | ANY} (подзапрос)
    | EXISTS (подзапрос)}
```



ГЛАВА 4

Модификация данных

Данные в базу данных надо как-то добавлять и обновлять. Для этого используются команды `INSERT` и `UPDATE`.

INSERT

Команда `INSERT` создает новые записи в таблице, а не изменяет существующие записи. Модификация существующих данных при выполнении команды `INSERT` возможна лишь в результате срабатывания триггера.

Команда `INSERT` имеет четыре основных формы. В первых двух формах задаются значения столбцов одной записи. Две другие формы позволяют добавить несколько записей, основанных на результатах команды `SELECT` или хранимой процедуры. Синтаксис всех четырех форм `INSERT` выглядит следующим образом:

```
INSERT [INTO] [таблица\представление] (список_столбцов)
    VALUES ([DEFAULT | выражение_константа][, ... n])

INSERT [INTO] [таблица\представление] (список_столбцов)
    DEFAULT VALUES

INSERT [INTO] [таблица\представление] (список_столбцов)
    команда_select

INSERT [INTO] [таблица\представление] (список_столбцов)
    команда_execute
```

Списки столбцов

Список столбцов представляет собой перечень имен столбцов, разделенных запятыми и заключенных в круглые скобки:

```
INSERT INTO [Список телефонов] ([Имя человека], [Номер телефона])
VALUES ('Вася Иванов', '095-555-00-03')
```

В этом примере в таблицу Список телефонов, в столбцы Имя человека и Номер телефона, вставляется новая запись с телефоном Иванова Васи.

Порядок столбцов в списке может отличаться от порядка столбцов в таблице. Тем не менее, в списке столбцов команды INSERT должны присутствовать все столбцы, для которых запрещены значения типа NULL и которые не имеют значений по умолчанию.

Если в команде INSERT отсутствует список столбцов, она выполняется так, словно список содержит все столбцы в том же порядке, что и в таблице. Это означает, что при отсутствии списка столбцов порядок значений соответствует порядку столбцов в таблице.

Вставка отдельной записи

Команда INSERT чаще всего применяется для вставки в таблицу отдельной записи. Обычно при этом используется ключевое слово VALUES, за которым следует список значений. Он содержит значения, присваиваемые соответствующим столбцам.

В следующих примерах продемонстрированы различные варианты вставки записи в таблицу с именем MyTable, состоящую из двух столбцов (a и b).

Следующая команда вставляет новую запись в таблицу MyTable. Столбцу a присваивается значение 'a', а столбцу b — значение 1234:

```
INSERT MyTable VALUES ('a', 1234)
```

В следующей команде использовано необязательное ключевое слово INTO, не влияющее на выполнение команды INSERT. Список столбцов состоит из одного столбца a:

```
INSERT INTO MyTable (a) VALUES ('ccc')
```

При наличии ключевых слов DEFAULT VALUES список столбцов не допускается. Каждому столбцу MyTable присваивается значение по умолчанию:

```
INSERT MyTable DEFAULT VALUES
```

В следующем примере список столбцов тоже не используется. Столбцу a присваивается значение по умолчанию, а столбцу b — значение 5:

```
INSERT MyTable VALUES (DEFAULT, 5)
```

Далее столбец b стоит на первом месте в списке, а столбец a — на втором. Столбцу b присваивается значение, полученное при вызове функции @@ROWCOUNT, а столбцу a — значение, полученное при вызове функции USER_NAME():

```
INSERT MyTable (b, a) VALUES (@@ROWCOUNT, USER_NAME())
```

Каждому столбцу соответствует выражение в списке значений. Это выражение может представлять собой константу, функцию, любое сочетание констант и функций или ключевое слово DEFAULT. В списке значений не допускается использование подзапросов, столбцов или агрегатных функций. Присваиваемое значение либо неявно преобразуется к типу данных столбца, либо вы должны выполнить явное преобразование.

Иногда при вставке данных в таблицу требуется, чтобы SQL Server присвоил столбцу значение по умолчанию. Для этого используется ключевое слово DEFAULT. Существуют два возможных применения ключевого слова DEFAULT в команде INSERT. Если DEFAULT используется в списке значений, оно относится к соответствующему столбцу списка. Если ключевое слово DEFAULT находится перед началом списка, оно относится ко всем столбцам таблицы, в этом случае список столбцов должен быть пустым.

Ключевое слово DEFAULT обрабатывается следующим образом:

- если у столбца имеется значение по умолчанию, оно заносится в столбец;
- если у столбца нет значения по умолчанию, присваивается значение типа NULL;
- если столбец не допускает значений типа NULL, SQL Server выдает сообщение об ошибке.

UNIQUEIDENTIFIER

UNIQUEIDENTIFIER — это тип данных в SQL Server для хранения глобально-уникальных идентификаторов (GUID). Значения столбцов UNIQUEIDENTIFIER представляют собой шестнадцатеричные числа, состоящие из 36 цифр в формате xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. Присутствие дефисов обязательно.

При вставке данных в столбцы с типом данных UNIQUEIDENTIFIER необходимо учитывать особенности поведения этих столбцов. Для получения глобально-уникального идентификатора, присваиваемого столбцу, можно воспользоваться функцией NEWID(). Следующий фрагмент показывает, как происходит вставка данных в таблицу со столбцом типа UNIQUEIDENTIFIER:

```
INSERT INTO [Проекты DoWork.ru]
    (ProjectID, [Название проекта], [Описание])
    VALUES (NEWID(), 'Создать сайт', 'Сайт должен быть красивым')
```

Вставка нескольких записей

Команда INSERT позволяет вставить в таблицу несколько записей одновременно. Для этого список значений заменяется командой SELECT или хранимой процедурой.

Использование команды *SELECT*

Большинство команд `SELECT` может использоваться в командах `INSERT` для вставки нескольких записей в таблицу. Если команда `SELECT` является частью команды `INSERT`, она не может содержать секций `COMPUTE` или `COMPUTE BY`.

Допускается применение команды `SELECT` для выборки записей из таблицы, в которую вы собираетесь вставлять данные, но при этом необходимо проследить за сохранением уникальности индексов, или операция вставки завершится неудачей.

Команда `SELECT`, являющаяся частью `INSERT`, может не возвратить ни одной записи. Это не является ошибкой.

А если хотя бы одна вставленная запись нарушает ограничение таблицы, вся команда `INSERT` не может быть выполнена.

Пример команды `INSERT` с командой `SELECT`:

```
INSERT MyTable (a, b) SELECT c, SUM(d) FROM MyTable2 GROUP BY c
```

Команда вставляет в таблицу `MyTable` каждую запись, возвращаемую командой `SELECT`.

Использование хранимых процедур

Записи, включаемые в таблицу, могут быть получены с помощью хранимой процедуры. При этом допускается использование любой хранимой процедуры, которая возвращает данные с помощью команд `SELECT` или `READTEXT`. Хранимые процедуры в команде `INSERT` подчиняются следующим правилам:

- если процедура производит выборку данных из текстового столбца с помощью команды `READTEXT`, при каждом вызове `READTEXT` возвращается не более 1024 Кбайт данных;
- хранимая процедура, используемая для создания нескольких записей, также может выполнять обновление и вставку в других таблицах;
- выполняя несколько команд `SELECT`, процедура может извлекать несколько итоговых наборов;
- полученные данные должны соответствовать типу и количеству столбцов в списке;
- если хранимая процедура не возвращает ни одной записи, это не является ошибкой;
- если хотя бы одна вставляемая запись нарушает ограничение таблицы, вся команда `INSERT` завершается неудачей.

Приведем пример вставки записей с помощью хранимой процедуры:

```
INSERT MyTable (a, b) EXECUTE MyProcedure
```

Процедура MyProcedure должна возвращать один или несколько итоговых наборов. Каждый итоговый набор содержит два значения, вставляемые в столбцы a и b таблицы MyTable.

Вы можете использовать секцию TOP, чтобы ограничить количество вставляемых записей. Например, следующая команда обновит только первые 10 записей из запроса:

```
INSERT TOP 10 MyTable (a, b)
    SELECT c, SUM(d) FROM MyTable2 GROUP BY c
```

> ***Особенности SQL Server 2000***
>
> В SQL Server 2000 вы не можете использовать секцию TOP с командой INSERT. Эта возможность появилась только в SQL Server 2005.

SELECT INTO

Команда SELECT с параметром INTO позволяет одновременно создать таблицу и заполнить ее данными. Чтобы воспользоваться этим средством заполнения таблицы, необходимо установить параметр базы данных SELECT INTO.

> ***Внимание!***
>
> Как правило, команда SELECT INTO не используется при создании рабочих баз данных. Но если команда SELECT INTO все же применяется при создании рабочих баз данных, обычно ее использование ограничивается созданием временных таблиц.

Команда SELECT с параметром INTO имеет следующий синтаксис:

```
SELECT список_выборки INTO имя_таблицы FROM список_таблиц
    [секция_where] [секция_groupby] [секция_having]
    [секция_orderby]
```

Список выборки может быть любым допустимым списком выборки, а имя таблицы — именем любой таблицы, не существующей в данный момент.

Пример использования SELECT с секцией INTO:

```
SELECT *  INTO NewTable FROM Authors
```

Команда копирует все записи и столбцы из таблицы Authors в новую таблицу.

Команда SELECT INTO создает новую таблицу, содержащую результаты выборки. Новая таблица состоит из столбцов, присутствующих в списке выборки. В команде могут использоваться любые секции SELECT, за исключением COMPUTE и COMPUTE BY. Присутствие секции FROM обязательно. Каждый столбец в списке выборки должен иметь уникальное непустое имя. Если два столбца имеют одинаковые имена, одному из них необходимо назначить

псевдоним. Псевдоним столбца может быть назначен любому выражению или функции. Кроме того, не допускается выполнение SELECT INTO внутри транзакции.

Таблица, создаваемая командой SELECT INTO, не должна существовать перед выполнением этой команды.

Секции WHERE и HAVING ограничивают количество вставляемых записей.

Если команда SELECT не возвращает ни одной записи, создается пустая таблица. При выполнении команды SELECT INTO свойства счетчика наследуются, если только 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.

Для выполнения команды SELECT INTO необходимо иметь привилегии создания таблиц для базы данных, в которую осуществляется выборка, а также привилегии чтения данных для таблицы, из которой выбираются данные. База данных, в которой создается таблица, должна иметь установленный параметр SELECT INTO/BULKCOPY.

Таблица, в которую записываются данные, не обязана принадлежать текущей базе данных. Вы можете использовать любое полностью определенное имя таблицы, не существующей в данный момент. Кроме того, таблица может быть временной.

UPDATE

Команда UPDATE обновляет существующие записи в существующей таблице и может обновлять любое количество записей, начиная с нуля.

Команда UPDATE обновляет один или несколько столбцов из таблицы, указанной после ключевого слова UPDATE. SQL Server позволяет модифицировать данные и задавать значения для переменной в рамках одной команды. Имена столбцов и переменных разделяются запятыми.

Команда UPDATE имеет следующий синтаксис:

```
UPDATE  [<таблица_или_представление>]
    SET [имя_столбца] = (выражение | DEFAULT | NULL
        | @переменная = выражение
        | @переменная = имя_столбца = выражение)  [, ... n]
   [FROM таблица-источник]
   [WHERE <условия_поиска>]
```

Пример команды UPDATE, которая переименовывает всех Саш в Вась в таблице Люди:

```
UPDATE [Люди] SET [Имя]='Вася' WHERE [Имя]='Саша'
```

Следующая команда меняет значение переменной @a при выполнении обновления:

```
DECLARE @a int
UPDATE [Люди] SET [Имя]='Вася', @a=105 WHERE [Имя]='Саша'
```

Следующая команда использует подзапрос для определения списка обновляемых имен:

```
UPDATE [Люди] SET [Имя]='Вася'
    WHERE [Имя] IN (SELECT [Имя] FROM [Черный список имен])
```

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

При выполнении обновлений действуют следующие правила:

- столбцам могут присваиваться выражения;
- в выражениях могут использоваться столбцы таблицы, заданной в секции FROM;
- нельзя непосредственно присвоить столбцу агрегатную функцию. Однако допускается присваивание подзапросов, возвращающих агрегатные функции;
- если за именем столбца следует ключевое слово DEFAULT, столбцу присваивается значение по умолчанию.

Секция FROM команды UPDATE подчиняется тем же правилам, что и секция FROM команды SELECT. В нее могут входить объединения и псевдонимы столбцов. Секция WHERE подчиняется тем же правилам, что и секция WHERE команды SELECT. Например, следующий запрос меняет имена всех людей, у кого на счете в банке больше 1000, на "Вася":

```
UPDATE [Люди]
SET [Имя]='Вася'
FROM [Люди] AS l, [Банковские счета] AS b
WHERE l.[ИНН]= b.[ИНН] AND b.[Сумма]>1000
```

Но того же можно было добиться, используя объединение ANSI:

```
UPDATE [Люди]
SET [Имя]='Вася'
FROM [Банковские счета] AS b JOIN [Люди] AS l ON l.[ИНН]= b.[ИНН]
WHERE b.[Сумма]>1000
```

Используя секцию TOP, вы можете ограничить количество обновляемых записей. При этом обновляемые записи не сортируются в каком-либо порядке. Например, следующая команда обновит только первые 10 записей:

```
UPDATE TOP 10 [Люди]
SET [Имя]='Вася'
FROM [Банковские счета] AS b JOIN [Люди] AS l ON l.[ИНН]= b.[ИНН]
WHERE b.[Сумма]>1000
```

> ***Особенности SQL Server 2000***
>
> В SQL Server 2000 вы не можете использовать секцию TOP с командой UPDATE. Эта возможность появилась только в SQL Server 2005.

При выполнении UPDATE нельзя изменять столбцы счетчика. Если какая-либо из модифицированных записей нарушает правило или ограничение таблицы, вся команда UPDATE не может быть выполнена.

Вы не можете изменять кластерный индекс для нескольких записей и модифицировать столбцы типов TEXT, IMAGE или типов данных, использующих символы Unicode в одной команде.

Команда *DELETE*

Для удаления данных служит команда DELETE. Назначение и использование параметров команды DELETE аналогично одноименным параметрам команд UPDATE или SELECT.

Пример команды DELETE:

```
DELETE FROM [Люди] WHERE [ИНН]='12345678'
    OR [Номер Паспорта]='5000 254561'
```

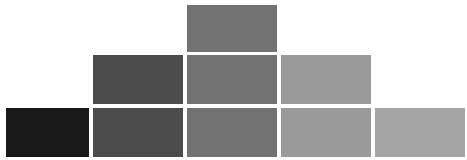
Команда DELETE имеет следующий синтаксис:

```
DELETE
   [TOP (выражение) [PERCENT]]
   [FROM таблица-источник [, ...n ]]
   [WHERE <условия_поиска>]
```

Используя секцию TOP, вы можете ограничить количество удаляемых записей. При этом не определяется сортировка удаляемых записей. Например, следующая команда обновит только первые 10 записей:

```
DELETE TOP 10 FROM [Люди] WHERE [ИНН] LIKE '123%'
```

> ***Особенности SQL Server 2000***
>
> В SQL Server 2000 вы не можете использовать секцию TOP с командой DELETE. Эта возможность появилась только в SQL Server 2005.



ГЛАВА 5

Работа с таблицами

Реляционная база данных состоит из сущностей (таблиц), находящихся в некотором отношении друг с другом. Таблицу проще всего описать в виде списка. База данных представляет собой набор таблиц, а таблицы являются наборами столбцов.

Создание базы данных начинается с создания таблиц. В процессе проектирования таблиц (и определения их столбцов) вам необходимо выбрать тип данных, которые будут в них храниться, а также допустимы ли неопределенные значения (`NULL`). Другими словами, является ли значение столбца обязательным или необязательным. Разрешая неопределенные значения, вы тем самым делаете столбец необязательным.

Создание таблиц

Таблицы создаются командой `CREATE TABLE`. Создавать таблицы может любой пользователь, имеющий на это права или обладающий ролью владельца базы или системного администратора. Синтаксис команды `CREATE TABLE` показан в листинге 5.1.

Листинг 5.1. Синтаксис команды `CREATE TABLE`

```
CREATE TABLE имя_таблицы
   ({ <определение_колонки> | <ограничение_таблицы> } [, ...n])

<определение_колонки> ::=
   { имя_колонки тип_данных }
   [ { DEFAULT выражение_константа | [ IDENTITY [ (начало, приращение) ]
] } ]
   [ ROWGUIDCOL ]
```

```
    [ <ограничения_колонки> [... n] ]

<ограничения_колонки> ::=
    [ CONSTRAINT имя_ограничения ]
    { [ NULL | NOT NULL ]
       | [ PRIMARY KEY | UNIQUE ]
       | REFERENCES ссылка_таблица [ (ссылка_столбец) ]
       [ ON DELETE { CASCADE | NO ACTION } ]
       [ ON UPDATE { CASCADE | NO ACTION } ]
    }
<ограничение_таблицы> ::=
    [ CONSTRAINT имя_ограничения ]
    { [ { PRIMARY KEY | UNIQUE }
        { (колонка [, ...n ]) }
       ]
      | FOREIGN KEY
      (колонка [, ...n ])
       REFERENCES ссылка_таблица [ (ссылка_столбец [, ...n ])]
       [ ON DELETE { CASCADE | NO ACTION } ]
       [ ON UPDATE { CASCADE | NO ACTION } ]
    }
```

Пример команды, которая создает таблицу:

```
CREATE TABLE Authors (
    AuthorID int not null,
    LastName varchar(50) not null,
    FirstName varchar(50) null
)
```

В приведенном примере создается таблица с именем `Authors`, предназначенная для хранения списка авторов. Таблица состоит из столбцов фамилий `LastName` и имен `FirstName` авторов. Также у каждого автора есть идентификационный код. Структура таблицы показана в табл. 5.1.

Таблица 5.1. *Таблица* `Authors` *с данными*

AuthorID	FirstName	LastName
1	Александр	Пушкин
2	Михаил	Лермонтов
3	NULL	Есенин

Создание таблиц в SQL Server Management Studio

Вы можете легко и просто создавать таблицы для вашей базы, используя SQL Server Management Studio. Для этого вам необходимо щелкнуть правой кнопкой мыши на списке таблиц вашей базы данных и выбрать пункт **New Table**.

Появится конструктор таблицы, как на рис. 5.1, где вы можете добавлять новые колонки и определять их свойства, используя редакторы свойств.

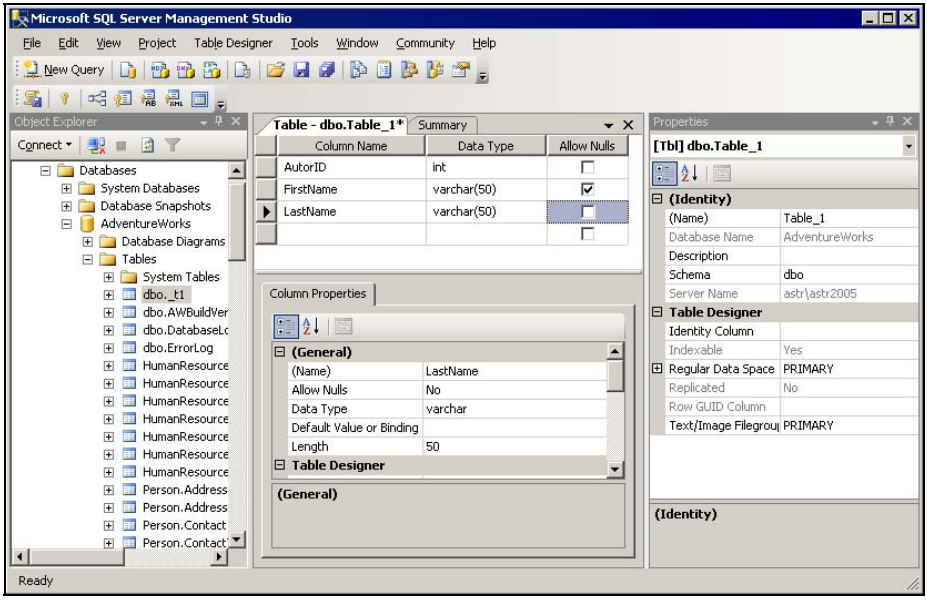


Рис. 5.1. Создание таблицы в SQL Server Management Studio

Закончив создание или редактирование таблицы, вы можете нажать кнопку **Save** и сохранить все изменения в текущей базе. Или можете выбрать пункт меню **Table Designer | Generate Change Script** и получить скрипт с SQL-кодом для создания или изменения таблицы (рис. 5.2). При этом подходе вы можете использовать скрипт не только для изменения текущей базы, но и создавать скрипты для обновления других баз, что бывает очень часто необходимо для больших баз данных.

Для получения данных, сохранения изменений SQL Server Management Studio создает SQL-код, который передается на SQL Server. Используя пункт меню **Generate Change Script**, вы можете получить этот код.

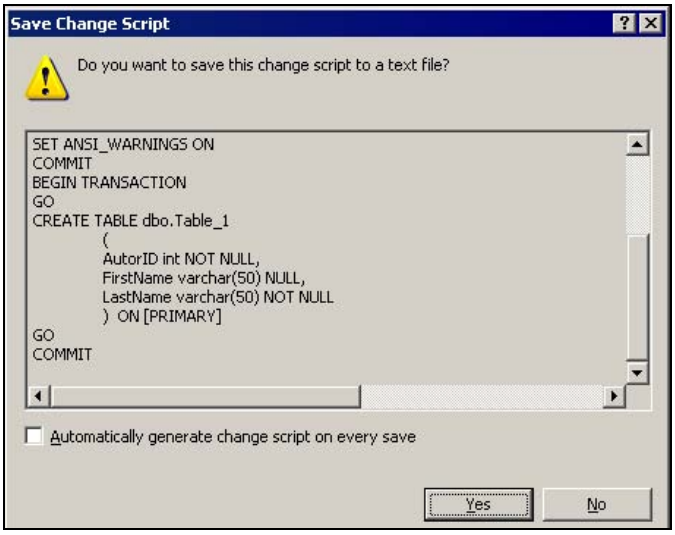


Рис. 5.2. Сгенерированный SQL-скрипт для новой таблицы

Также вы можете проводить занимательные эксперименты, внося изменения в базу данных и смотря, какой SQL-код приводит к ним.

Имена таблиц

Имя таблицы должно быть не больше 128 символов. Они должны быть уникальными по отношению к владельцу: две таблицы с одинаковыми именами не могут принадлежать одному владельцу. Но у другого пользователя может быть таблица с тем же именем.

Столбцы

Таблица может содержать до 1024 столбцов. Для каждого столбца определяются, как минимум, имя и тип данных. Имена столбцов имеют длину до 128 символов и являются уникальными в пределах таблицы. Имена столбцов соответствуют обычным правилам выбора имен идентификаторов и состоят из букв, цифр и служебных символов (пробел, `@` и `#`).

Типы данных

В табл. 5.2 перечислены основные типы данных, используемые в SQL Server.

При выборе типа данных столбца будьте внимательны. Например, для хранения данных типа `int` необходимо больше места, чем для `tinyint`. Следова-

тельно, при создании столбца целых величин следует определить их максимальные значения. Если целое данное в столбце никогда не превышает 10, логичнее выбрать `tinyint`.

Таблица 5.2. *Типы данных в SQL Server*

Тип данных	Описание
`bit`	Целое значение 0 или 1. Используется для хранения логических значений `TRUE` или `FALSE`
`bigint`	Длинное целое. Для хранения данного требуется 8 байтов. Диапазон значений определяется промежутком [-2^{63}; $2^{63}-1$]
`int`	Обычное целое. Для хранения данного требуется 4 байта. Диапазон значений определяется промежутком [-2^{31}; $2^{31}-1$]. Для SQL Server этот целый тип считается основным. Это означает, что многие функции неявно преобразуют другие целочисленные типы к `int`
`smallint`	Короткое целое. Для хранения данного требуется 2 байта: [-2^{15}; $2^{15}-1$]
`tinyint`	Крошечное целое. Для хранения данного требуется 1 байт. Диапазон значений [0; 255]
`decimal(`*p*`[, `*s*`])]` `numeric[(`*p*`[, `*s*`])]`	Десятичный тип. *p* — полное количество десятичных знаков до и после запятой, *s* — количество разрядов после запятой. Диапазон значений [$-10^{38}+1$; $10^{38}-1$]
`money`	Денежный тип. Для хранения данного требуется 8 байтов. Диапазон значений [–922 337 203 685 477,5808; 922 337 203 685 477,5807]
`smallmoney`	Укороченный денежный тип. Для хранения требуется 4 байта. Диапазон значений [–214 748,3648; 214 748,3647]
`float[(`*n*`)]`	Число с плавающей запятой. Диапазон значений [–1,79E+308; 1,79E+308]
`real`	Число с плавающей запятой. Тип `real` соответствует `float(24)`. Диапазон значений [$-3{,}40\times10^{38}$; $3{,}40\times10^{38}$]
`datetime`	Тип для хранения даты и времени. Для хранения данного требуется 8 байтов. Диапазон значений от 01.01.1753 до 31.12.9999
`smalldatetime`	Тип для хранения даты и времени. Для хранения требуется 4 байта. Диапазон значений от 01.01.1900 до 06.06.2079
`date`	Этот тип данных появился в SQL Server 2008 и предназначен для хранения только даты. Для хранения данного требуется 3 байта

Таблица 5.2 *(продолжение)*

Тип данных	Описание
`time`	Этот тип данных появился в SQL Server 2008 и предназначен для хранения только времени. Для хранения данного требуется 3—5 байтов
`datetimeoffset`	Этот тип данных появился в SQL Server 2008 и хранит дату и время со смещениями "+" или "–". Например, используется для хранения даты с учетом часового пояса. Для хранения данного требуется 8—10 байтов
`datetime2`	Этот тип данных появился в SQL Server 2008. В нем хранится дата и время в диапазоне от 1 января 0001 года до 31 декабря 9999 года. Для хранения данного требуется 6—8 байтов
`timestamp`	Пометка времени. Для хранения используется 8 байтов. SQL Server гарантирует уникальность данной величины в пределах одной базы данных. При внесении любых изменений в таблицу SQL Server увеличивает значение `timestamp` на 1. Данный тип не приемлем в качестве первичного ключа
`uniqueidentifier`	Уникальный глобальный идентификатор (GUID). Для хранения используется 16 байтов. Для генерации таких идентификаторов используется время в миллисекундах, номера сетевых плат, а также другие параметры компьютера
`char(`*n*`)`	Символьный тип. Строка фиксированной длины. Максимальная длина *n* — 8000 символов. Если помещаемая строка короче *n*, она справа дополняется пробелами
`varchar(`*n*`)`	Символьный тип. Строка переменной длины. Максимальное значение *n* — 8000. Если помещаемая строка короче *n*, то она не дополняется справа пробелами. То есть, если длины хранимых строк разные, лучше использовать `varchar`, а не `char`. Данный тип имеет еще один возможный формат — `varchar(`*max*`)`. Этот формат используется вместо типа `text` для хранения текстовых строк, длина которых 2^{31}–1 байтов
`nchar(`*n*`)`	То же, что и `char`, но все символы хранятся в формате Unicode, т. е. на каждый символ отводится 2 байта
`nvarchar(`*n*`)`	То же, что и `varchar`, но все символы хранятся в формате Unicode, т. е. на каждый символ отводится 2 байта. Начиная с SQL Server 2005, данный тип имеет еще один возможный формат — `nvarchar(`*max*`)`. Этот формат используется вместо типа `ntext` для хранения текстовых строк, длина которых 2^{31}–1 байтов

Таблица 5.2 *(окончание)*

Тип данных	Описание
`text`	Символьные данные переменной длины. Максимальная длина 2 147 483 647 символа. Начиная с SQL Server 2005 данный тип объявлен устаревшим, но он все еще часто используется в базах данных. Используйте `varchar(max)`
`ntext`	Символьные данные переменной длины в кодировке Unicode. Максимальная длина — 1 073 741 823 символа. Начиная с SQL Server 2005, данный тип объявлен устаревшим. Используйте `nvarchar(max)`
`image`	Двоичные данные. Максимальный размер — 2 Гбайт. Начиная с SQL Server 2005, данный тип объявлен устаревшим. Используйте `varbinary(max)`
`binary(n)`	Двоичные строковые данные фиксированной длины *n*. Максимальная длина — 8000 байтов
`varbinary(n)`	Двоичные строковые данные переменной длины *n*. Максимальная длина — 8000 байтов. Начиная с SQL Server 2005, данный тип имеет еще один возможный формат — `varbinary(max)`. Этот формат используется для двоичных данных, длина которых 2^{31}–1 байтов (2 Гбайт), и призван заменить тип `image`
`sql_variant`	Этот тип появился, только начиная с SQL Server 2005. Позволяет хранить в переменной или столбце различные другие типы данных
`xml`	Этот тип появился, только начиная с SQL Server 2005. Данный тип позволяет хранить документы в формате XML
`HierarchyId`	Этот тип данных появился в SQL Server 2008, и в нем хранятся данные иерархий, причем дерево иерархий будет довольно компактным
`Geometry`, `Geography`	Этот специальный тип данных появился в SQL Server 2008 и предназначен для хранения векторных объектов
`FileStream`	Этот тип данных появился в SQL Server 2008 и предназначен для хранения данных в файловой системе. Ограничение полей `varbinary` в 2 Гбайт не распространяется на этот тип данных

Параметры *NULL* и *NOT NULL*

Если столбец создается с параметром `NULL`, то при вводе данных столбец можно оставить пустым. Если столбец содержит обязательные данные, его следует создавать с параметром `NOT NULL`. По умолчанию столбцы создаются с параметром `NOT NULL`.

Уникальные идентификаторы

Базы данных требуют, чтобы в базе существовал способ однозначной идентификации записей. Иногда однозначная идентификация записей реализуется просто, но при отсутствии удобного способа приходится использовать ключ с последовательным приращением или другой неестественный механизм.

Столбцы счетчика

Столбец счетчика представляет собой столбец, автоматически генерирующий уникальные значения с некоторым приращением, например: 1, 2, 3, ... или 1, 4, 7, 10.

Они часто используются для определения уникальности записей таблицы. Столбцы счетчика не могут содержать неопределенные значения и должны относиться к числовому типу данных (например, `int`, `smallint`, `tinyint`, `numeric(`*p*`,0)` или `decimal(`*p*`,0)`).

Если при объявлении столбца не указаны начальное значение и приращение, столбец действует как счетчик с начальным значением 1 и приращением 1.

Нумерация столбцов счетчика может содержать пропуски, обусловленные незавершенными транзакциями или непредвиденными сбоями сервера.

При выборе типа данных для столбца счетчика следует быть внимательным. Недостаточный размер столбца счетчика может вызвать проблемы.

В следующем примере столбец счетчика `AuthorID` объявлен с типом `tinyint`. То есть невозможно добавить больше 255 авторов. При достижении максимума все попытки вставить новые записи будут вызывать ошибку.

```
CREATE TABLE Authors (
    AuthorID tinyint IDENTITY,
    Name nvarchar(50) NOT NULL
)
```

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

ROWGUIDCOL

Свойство столбца ROWGUIDCOL в сочетании с типом данных uniqueidentifier обеспечивает глобальную уникальность значения для всех компьютеров в сети. Для генерации таких идентификаторов используется время в миллисекундах, номера сетевых плат, а также другие параметры компьютера.

Такая возможность бывает особенно полезной при работе с несколькими базами данных, содержимое которых требуется объединить.

Пример таблицы со столбцов с глобально-уникальным идентификатором:

```
CREATE TABLE Authors (
    AuthorID uniqueidentifier NOT NULL ROWGUIDCOL
      DEFAULT (newid()),
    Name nvarchar(50) NOT NULL
)
```

В этом примере используется функция newid(), которая создает новый идентификатор при вставке новой записи.

Ограничения

Ограничения накладывают определенные условия на вводимые данные. Условием может быть уникальность значения, принадлежность к некоторому интервалу или списку значений, наличие первичного ключа, соответствующего внешнему ключу (ограничения ссылочной целостности).

SQL Server поддерживает четыре типа ограничений (PRIMARY KEY, FOREIGN, CHECK и UNIQUE).

PRIMARY KEY

Ограничение первичного ключа требует, чтобы содержимое столбца было уникальным. Объявление первичного ключа требует однозначную идентификацию записей таблицы. Ограничение PRIMARY KEY или UNIQUE является обязательным требованием для обеспечения ссылочной целостности посредством ограничения внешних ключей. Другими словами, когда вы определяете внешние ключи, ссылающиеся на таблицы, для внешнего ключа должен быть определен первичный ключ.

В следующем примере объявляется ограничение первичного ключа:

```
CREATE TABLE Authors (
    AuthorID int NOT NULL CONSTRAINT author_pk PRIMARY KEY,
    Name nvarchar(50) NOT NULL
)
```

Такой способ подходит для первичных ключей, состоящих из одного столбца, но для объявления первичного ключа, состоящего из нескольких столбцов, потребуется объявление на уровне таблицы. Пример такого объявления:

```
CREATE TABLE Authors (
    FirstName nvarchar(50) NOT NULL,
    LastName nvarchar(50)  NOT NULL,
    CONSTRAINT author_pk   PRIMARY KEY (FirstName, LastName)
)
```

В этом примере допускается, что не может быть авторов с одинаковым именем и фамилией. Имя и фамилия автора являются первичным ключом.

Указывать ключевое слово `CONSTRAINT` с именем ограничения необязательно. Тем не менее, если имя не указано, SQL Server присваивает его за вас. У ограничения `PRIMARY KEY` существует альтернатива — ограничение `UNIQUE`. Для ограничения `UNIQUE` допускает ввод неопределенных значений.

FOREIGN KEY

База данных представляет собой набор взаимосвязанных таблиц. Ограничение внешнего ключа позволяет обеспечить целостность данных в дочерней таблице. При установке ограничения внешнего ключа SQL Server обращается к столбцу первичного ключа связанной таблицы и проверяет совпадение данных в общих столбцах.

Пример ссылочного ограничения:

```
CREATE TABLE Books (
    BookName varchar(250) NOT NULL,
    AuthorID int NOT NULL
    CONSTRAINT author_fk REFERENCES Authors (AuthorID)
)
```

В этом примере создается таблица, содержащая имена книг и ID авторов. Ограничение `REFERENCES` требует, чтобы значение `AuthorID` уже присутствовало в таблице `Authors`.

Также это ограничение можно объявить на уровне таблицы:

```
CREATE TABLE Books (
    BookName varchar(250) NOT NULL,
    AuthorID int NOT NULL,
    CONSTRAINT author_fk FOREIGN KEY (AuthorID)
                REFERENCES Authors (AuthorID)
```

```
                ON DELETE { CASCADE }
                ON UPDATE { NO ACTION }
)
```

В этом примере устанавливается ограничение уровня таблицы, поэтому в нем использована форма с ключевым словом `FOREIGN KEY`, за которым указывается проверяемый столбец или столбцы таблицы. Также в команде определяется, что в случае удаления записи из родительской таблицы соответствующие записи из дочерней удаляются (это определяется в части `ON DELETE` команды). Но в случае обновления данных в родительской таблице SQL Server не будет предпринимать никаких действий. Это определяется в части `ON UPDATE` команды.

Ограничения внешнего ключа, состоящие из нескольких столбцов, должны объявляться на уровне таблицы. Они ссылаются на столбцы, объявленные в качестве первичных ключей или имеющие уникальные индексы.

CHECK

В некоторых ситуациях проверяется не существование данных (как ограничение внешнего ключа), а их правильность. Деловые правила могут предъявлять определенные требования к вводимым данным. Для их реализации можно воспользоваться ограничением проверки. Кроме того, ограничения проверки позволяют сравнивать данные из различных столбцов — при условии, что столбцы находятся в одной таблице, а ограничение объявлено на уровне таблицы.

В следующем примере создается таблица `Authors`, хранящая только записи авторов, имена которых начинаются с буквы A (специально для издательства, которое работает только с теми авторами, чье имя начинается с A).

```
CREATE TABLE Authors (
    AuthorID int NOT NULL CONSTRAINT author_pk PRIMARY KEY,
    FirstName nvarchar(50) NOT NULL,
    LastName nvarchar(50) NOT NULL
)
ALTER TABLE Authors ADD CONSTRAINT
lastname_ck CHECK (LastName LIKE 'A%')
```

Тот же пример, но с объявлением ограничения на уровне таблицы:

```
CREATE TABLE Authors (
    AuthorID int NOT NULL CONSTRAINT author_pk PRIMARY KEY,
    FirstName nvarchar(50) NOT NULL,
```

```
    LastName nvarchar(50) NOT NULL,
    CONSTRAINT lastname_ck CHECK (LastName LIKE 'A%')
)
```

При таком объявлении проверка позволяет сравнивать данные столбца с другим столбцом в той же таблице.

Значения по умолчанию

При вводе данных в базу столбцы либо заполняются, либо остаются неопределенными (имеют значение `NULL`). Иногда неопределенные значения запрещаются. В таких случаях требуется задать значение по умолчанию. Для этой цели используется параметр `DEFAULT`.

Пример определения значения по умолчанию:

```
CREATE TABLE Authors (
    AuthorID int NOT NULL CONSTRAINT author_pk PRIMARY KEY,
    [Имя] nvarchar(50) NOT NULL,
    [Фамилия] nvarchar(50) NOT NULL,
    [Гражданство] nvarchar(50) NOT NULL CONSTRAINT cit_def
    DEFAULT 'Россия'
)
```

То же объявление, но на уровне таблицы (с помощью ключевого слова `FOR` определяется конкретный столбец таблицы):

```
CREATE TABLE Authors (
    AuthorID int NOT NULL CONSTRAINT author_pk PRIMARY KEY,
    [Имя] nvarchar(50) NOT NULL,
    [Фамилия] nvarchar(50) NOT NULL,
    [Гражданство] nvarchar(50) NOT NULL,
    CONSTRAINT cit_def DEFAULT 'Россия' FOR [Гражданство]
)
```

Вычисляемые столбцы

Вы можете создавать виртуальные столбцы, значения которых вычисляются с использованием функций, констант или других столбцов таблицы. Вычисляемые столбцы определяются в виде SQL-выражений.

Пример таблицы с вычисляемым столбцом:

```
CREATE table bookPrice (
    BookID int NOT NULL CONSTRAINT author_pk PRIMARY KEY,
```

```
    [Оптовая цена] smallmoney NOT NULL,
    [Розничная цена] smallmoney NOT NULL,
    [Надбавка] AS ([Розничная цена] - [Оптовая цена])
)
```

В этой команде столбец `Надбавка` является вычисляемым. Результаты вычисления не хранятся в базе, а рассчитываются по мере необходимости. В итоге база занимает меньше места.

Модификация таблиц

После создания таблицы в базе данных в нее часто требуется вносить некоторые изменения. К числу возможных модификаций относится добавление столбцов, изменение типов данных в существующих столбцах, объявление первичных ключей и т. д.

Полный синтаксис команды `ALTER TABLE` показан в листинге 5.2.

Листинг 5.2. Синтаксис команды `ALTER TABLE`

```
ALTER TABLE имя_таблицы
{ [ ALTER COLUMN имя_колонки
     { DROP DEFAULT
     | SET DEFAULT выражение-константа
     | IDENTITY [ (начало, приращение)]
     }
  | ADD
     { <определение_колонки> | <ограничение_таблицы> } [, ...n]
  | DROP
     { [ CONSTRAINT ] имя_ограничения
       | COLUMN колонка}
] }

<определение_колонки> ::=
   { имя_колонки тип_данных }
   [ [DEFAULT выражение_константа]
      | IDENTITY [ (начало, приращение) ]
   ]
   [ROWGUIDCOL]
   [ <ограничение_колонки> ] [, ...n ] ]
```

```
<ограничение_колонки> ::=
   [ NULL | NOT NULL ]
   [ CONSTRAINT имя_ограничения ]
   {
      | { PRIMARY KEY | UNIQUE }
      | REFERENCES ссылка_таблица [ (ссылка_столбец) ]
      [ ON DELETE { CASCADE | NO ACTION } ]
      [ ON UPDATE { CASCADE | NO ACTION } ]
   }

<ограничение_таблицы> ::=
   [ CONSTRAINT имя_ограничения ]
   { [ { PRIMARY KEY | UNIQUE }
      { ( колонка [, ...n ] ) }
      | FOREIGN KEY
        ( колонка [, ...n ] )
        REFERENCES ссылка_таблица [ (ссылка_столбец [, ...n ] ) ]
      [ ON DELETE { CASCADE | NO ACTION } ]
      [ ON UPDATE { CASCADE | NO ACTION } ]
   }
```

Вы не сможете удалить ограничение первичного ключа при наличии ограничений внешнего ключа, ссылающегося на него.

Дополнение таблиц

С помощью команды `ALTER TABLE`, после создания таблицы в нее можно добавить новые столбцы, ограничения и значения по умолчанию. Например:

```
ALTER TABLE Author ADD
        [ИНН] char(16) NULL
           CONSTRAINT inn_ck CHECK([ИНН] LIKE '78%'),
        [Пол] bit NULL
```

В этой команде добавляются две новых колонки: для хранения информации об ИНН и поле автора. А для столбца `ИНН` устанавливается ограничение, что ИНН должен начинаться с цифр 78.

Новые столбцы могут представлять собой вычисляемые выражения и объявляться с ограничениями уровня столбцов.

Удаление из таблиц

Из созданной таблицы можно удалить столбцы или ограничения. При удалении ограничений следует помнить, что выполнению команд могут помешать некоторые зависимости. Например, если столбец является первичным ключом, вы должны удалить это ограничение до удаления столбца. Если в другой таблице существует ссылка на столбец, SQL Server также не позволит удалить его до снятия ограничения.

Примеры команды `ALTER TABLE DROP`:

```
ALTER TABLE Authors DROP CONSTRAINT inn_ck
ALTER TABLE Authors DROP COLUMN [ИНН]
```

Модификация столбцов

Иногда требуется изменить тип данных колонки. Неверный тип приводит к неэффективному хранению данных, или же данные могут оказаться слишком большими и не помещаться в столбцах.

Вы можете использовать команду `ALTER TABLE` для этого.

Например, выяснилось, что не хватает 50 символов для хранения имени автора. У издательства появился новый автор с экзотическим именем длиной 100 символов. Вам необходимо увеличить размер столбца, в котором хранится имя. Вот команда, которая делает это:

```
ALTER TABLE Authors
ALTER COLUMN FirstName varchar(100)
```

При модификации столбца должно существовать неявное преобразование старого типа данных в новый. И новый тип данных не может иметь типом `timestamp`. Если модифицированный столбец является столбцом счетчика, новый тип данных должен быть допустимым для столбцов счетчика.

Временные таблицы

Временные таблицы похожи на обычные, однако они не предназначены для постоянного хранения данных. Во временных таблицах часто хранятся данные, которые должны быть модифицированы или использованы позднее. Они создаются, удаляются и используются как обычные таблицы.

Перечислим главные отличия временных таблиц от обычных:

- имя временной таблицы должно начинаться с символов `#` или `##`;
- длина имени временной таблицы ограничивается 116 символами;

- временные таблицы удаляются при отключении пользователя от базы данных;
- временные таблицы используются так, как будто они входят в текущую базу данных, однако в действительности данные хранятся в базе TEMPDB.

В SQL Server существуют два типа временных таблиц: локальные и глобальные.

Локальные временные таблицы

Локальные временные таблицы доступны лишь для своего владельца. Имена локальных временных таблиц всегда начинаются с префикса `#`.

В следующем примере создается локальная временная таблица с именем `#templ`:

```
CREATE TABLE #templ (a int  NULL, b varchar(25) NULL)
```

Глобальные временные таблицы

Глобальные временные таблицы похожи на локальные, за исключением того, что они доступны для всех пользователей. Их имена должны начинаться с префикса `##`.

Следующий фрагмент создает глобальную временную таблицу:

```
CREATE TABLE ##globalTempTable1 (a int  NULL, b varchar(25) NULL)
```

Когда временная таблица становится ненужной, ее можно удалить так же, как и любую другую таблицу.

Удаление таблиц

Ненужные таблицы удаляются командой `DROP`. Удаленная таблица навсегда пропадает из базы данных вместе со всеми данными. Это действие отменить невозможно.

Команда `DROP TABLE` имеет следующий синтаксис:

```
DROP TABLE имя_таблицы
```

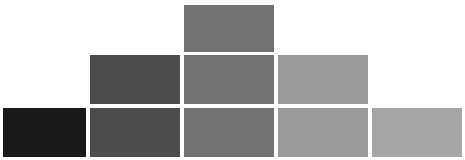
Например:

```
DROP TABLE Authors
```

Или вот пример удаления глобальной временной таблицы из предыдущего раздела:

```
DROP TABLE ##globalTempTable1
```

Таблицу, на которую ссылаются какие-либо ограничения, удалить нельзя. Перед удалением таблицы необходимо удалить эти ограничения. Таблица может быть удалена только владельцем.



ГЛАВА 6

Индексы

Индекс (index) — это объект базы данных, ускоряющий доступ к данным, по сравнению с просмотром каждой записи таблицы в поисках нужного результата.

На индексы в таблице возлагаются две задачи: ускорение поиска по индексируемым столбцам и гарантия уникальности значений, хранящихся в индексируемых столбцах.

С помощью индексов вы можете осуществлять поиск по таблице гораздо быстрее, чем с помощью простого перебора записей.

В SQL Server существуют индексы двух типов: кластерные и некластерные.

Определяя для таблицы *кластерный индекс*, вы тем самым приказываете серверу физически отсортировать данные в порядке индекса. Поскольку создание кластерного индекса приводит к физической сортировке данных, у каждой таблицы может быть лишь один кластерный индекс.

Однако одного индекса обычно оказывается недостаточно. Вы можете создать еще и *некластерный индекс*, в котором физическое расположение данных отличается от порядка следования индекса.

Создание индексов

Для повышения быстродействия кластерный индекс следует создавать до создания некластерных индексов. Создание кластерного индекса требует физической сортировки записей, а некластерные индексы содержат лишь "указатели на записи", которые приходится модифицировать при перемещении записей.

Упрощенный вариант создания индекса имеет следующий синтаксис:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX
    имя_индекса ON имя_таблицы (столбец [, ...])
```

Команда содержит следующие параметры:

- `UNIQUE` — запрещает наличие повторяющихся ключей в таблице. Обычно используется для столбцов первичного ключа. По умолчанию индекс не является уникальным;
- `CLUSTERED` — создает кластерный индекс;
- `NONCLUSTERED` — создает некластерный индекс (используется по умолчанию);
- *`имя_индекса`* — определяет имя индекса, уникальное в пределах таблицы;
- *`столбец`* — определяет столбец или столбцы, включаемые в индекс.

Следующая команда создает кластерный индекс:

```
CREATE UNIQUE CLUSTERED INDEX MyFirstIndex
    ON Author (FirstName, LastName)
```

Здесь создается уникальный кластерный индекс для таблицы `Author`, и индексация ведется сначала по столбцу `FirstName`, а затем по столбцу `LastName`.

А вот пример создания некластерного индекса:

```
CREATE NONCLUSTERED MySecondIndex ON Author ([ИНН])
```

Управление индексами

Чтобы изменить индекс таблицы, необходимо удалить, а затем создать его заново.

Удаление и повторное создание индексов часто выполняется в рабочих базах данных для пересчета коэффициента заполнения или изменения индексных структур в случае их смещения или нарушения сбалансированности. В таких случаях пересоздание индекса выполняется по соображениям быстродействия.

Команда удаления индекса из таблицы имеет следующий синтаксис:

```
DROP INDEX имя_таблицы.имя_индекса
```

Для управления индексами вы можете использовать редактор индексов в SQL Server Management Studio. Для этого при редактировании свойств таблицы выберите **Manage Indexes and Keys**. Откроется диалог, как на рис. 6.1, в котором вы можете создавать, редактировать и удалять любые индексы таблицы.

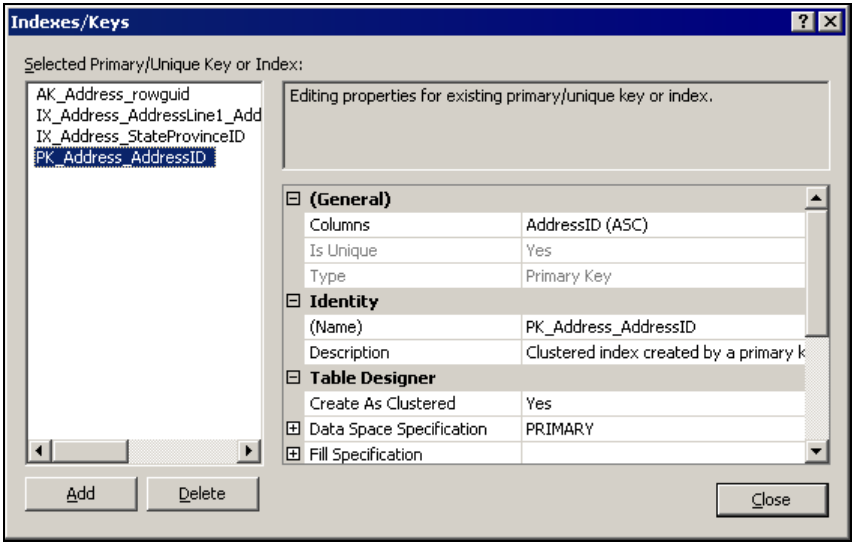
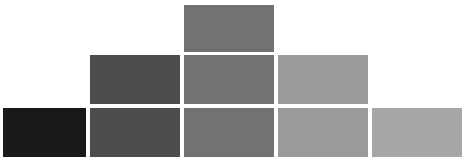


Рис. 6.1. Редактор индексов таблицы в SQL Server Management Studio



ГЛАВА 7

Представления

Представление (view) — это логическая таблица, созданная командой `SELECT` путем выборки данных из одной или нескольких таблиц.

Команда создания представления имеет следующий синтаксис:

```
CREATE VIEW [имя_схемы.]имя_представления [ (столбец [, ...n ]) ]
[ WITH <атрибуты_представления> [, ...n ] ]
AS команда_select [ ; ]
[ WITH CHECK OPTION ]

<атрибуты_представления> ::= {
   [ ENCRYPTION ]
   [ SCHEMABINDING ]
   [ VIEW_METADATA ]
}
```

Пример простого представления:

```
CREATE VIEW Customers AS
   SELECT * FROM CompanyData1..Customers
UNION ALL
SELECT * FROM CompanyData2..Customers
```

Это представление выбирает данные о покупателях из двух баз данных, объединяет их и возвращает как одну таблицу.

С практической точки зрения, представление является предварительно проанализированной командой `SELECT`, которая может интерпретироваться как таблица. Представление является не физической, а логической таблицей. Другими словами, представление позволяет изменять способ отображения данных, не меняя их основную структуру.

В большинстве ситуаций (в том числе и в командах SELECT) представления работают так же, как обычные таблицы.

С помощью представления можно ограничить доступ к записям и столбцам, скрыть объединения или упростить SQL-код посредством предварительного кодирования агрегатов, группировки и других сложных конструкций.

Если в команде новые имена столбцов не заданы явно, они наследуются из таблиц базы, как в примере ранее. Обратите внимание на то, что при использовании агрегатов имена столбцов задаются явно.

Параметры команды управляют доступом к представлению со стороны пользователей. Например, параметр ENCRYPT запрещает доступ к SQL-коду представления всем, кроме его создателя. То есть при создании представления с параметром ENCRYPT никто, кроме его создателя, не сможет просмотреть его SQL-код.

При использовании параметра WITH CHECK OPTION вставка и обновление записей через представление возможны лишь в том случае, если эти записи могут быть получены выборкой из представления.

Например, следующее представление возвращает список авторов, которые пишут на русском языке:

```
CREATE VIEW RussianAuthorsView AS
  SELECT [Имя] AS [rav_Имя], [Фамилия] AS [rav_Фамилия],
                [Язык написания книги]
  FROM Authors
  WHERE [Язык написания книги] = 'русский'
```

Из этого представления можно осуществлять выборку данных:

```
SELECT * FROM RussianAuthorsView
```

При попытке вставить данные в представление с установленным параметром WITH CHECK OPTION операция INSERT завершится неудачей, если значение поля Язык написания книги вставляемой записи отлично от 'русский'.

Другими словами, если команда INSERT нарушает представление, параметр WITH CHECK OPTION запрещает вставку.

При этом rav_Имя и rav_Фамилия становятся именами столбцов представления, и в команде INSERT используются они, а не реальные имена столбцов таблицы.

Имена столбцов представления должны соответствовать правилам выбора имен SQL Server.

Представления удаляются командой DROP VIEW, имеющей следующий синтаксис:

```
DROP VIEW имя_представления
```

При удалении представления его определение удаляется из базы. Команду DROP нельзя отменить. К удалению объекта следует относиться осторожно, особенно если у вас нет возможности создать его заново.

Представления могут использоваться для различных целей, связанных с защитой данных. Вы можете ограничить доступ к столбцам таблицы, предоставив его только к некоторым. Просто перечислите те столбцы, к которым вы хотите разрешить доступ пользователям, в команде SELECT.

Или вы можете ограничить доступ к записям, используя секцию WHERE запроса в представлении и возвращая пользователям только определенные данные.

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

Вы должны указать имя для каждого столбца в представлении, если выполняется одно из следующих условий:

- любой столбец представления является производным константы, встроенной функции или математического выражения;
- два или более столбцов таблицы имеют одинаковые имена;
- вы хотите переименовать столбец.

Представления не могут содержать следующие конструкции:

- SELECT INTO;
- COMPUTE;
- ORDER BY (однако секция ORDER BY может использоваться при выборке из представления);
- ссылки на временные таблицы.

Упрощение SQL-кода с помощью представлений

Представления используются для упрощения запросов, а также для маскировки сложных объединений и денормализованных таблиц.

Например, следующий фрагмент создает представление для работы с трехсторонним объединением:

```
CREATE VIEW AuthorBooks AS
SELECT b.BookID, b.[Название книги], b.AuthorID,
       a.[Имя автора], a.[Фамилия автора],
       s.[Стоимость книги]
FROM Books b
     LEFT OUTER JOIN Author a ON a.AuthorID = b. AuthorID
     LEFT OUTER JOIN Shop s ON b.BookID = s.BookID
```

Представления и команды *INSERT*, *UPDATE*, *DELETE*

Вы можете вставлять, удалять и обновлять данные в представлениях. Команда `INSERT` добавляет записи в базовую таблицу. Команды `UPDATE` и `DELETE` модифицируют записи базовой таблицы.

Например, следующий фрагмент удаляет записи из базовой таблицы через представление:

```
DELETE RussianAuthorsView
       WHERE [rav_Имя] = 'Александр' AND [rav_Фамилия] = 'Пушкин'
```

Если представление содержит столбцы из нескольких таблиц, вы не сможете удалить записи из представления или обновить столбцы нескольких таблиц в одной команде. Например, следующая команда недопустима:

```
UPDATE AuthorBooks
SET [Имя автора] = 'Николай', [Фамилия автора] = 'Гоголь',
    [Стоимость книги] = '100'
WHERE [Название книги] = "Мертвые души"
```

Из-за подобных ограничений могут возникнуть проблемы, поскольку одна команда может модифицировать лишь одну из базовых таблиц.

Проблема решается поочередным обновлением таблиц:

```
UPDATE AuthorBooks
SET [Имя автора] = 'Николай', [Фамилия автора] = 'Гоголь'
    WHERE [Название книги] = "Мертвые души"

UPDATE AuthorBooks
SET [Стоимость книги] = '100'
    WHERE [Название книги] = "Мертвые души"
```

Вставка в представлениях допускается 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.

Таким образом, вы не можете обновлять, удалять или вставить данные в представлении с секцией DISTINCT, поскольку DISTINCT дает временную рабочую таблицу. В такой ситуации вместо настоящей базовой таблицы будет модифицироваться временная таблица.

WITH CHECK OPTION

Пользователь может посредством вставки или обновления создать запись, которая не может быть выбрана из этого представления командой SELECT.

Вот пример создания "невидимой" записи:

```
UPDATE RussianAuthorsView
SET [Язык написания книги] = 'английский'
```

Флаг WITH CHECK OPTION запрещает пользователям вставку и обновление записей, не входящих в представление.

В следующем представлении параметр WITH CHECK OPTION разрешает пользователям выборку, модификацию и вставку лишь тех записей, которые входят в представление:

```
CREATE VIEW RussianAuthorsView AS
SELECT [Имя] AS [rav_Имя], [Фамилия] AS [rav_Фамилия],
       [Язык написания книги]
FROM Authors
WHERE [Язык написания книги] = 'русский'
WITH CHECK OPTION
```

ALTER VIEW

Для модификации представлений используется команда ALTER VIEW. Команда ALTER VIEW имеет следующий синтаксис:

```
ALTER VIEW [имя_схемы.]имя_представления [(колонка [, ...n ])]
[ WITH <атрибуты_представления> [, ...n ]]
AS выражение_select [ ; ]
[ WITH CHECK OPTION ]
```

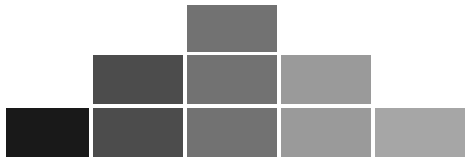
```
<атрибуты_представления> ::=
{
    [ ENCRYPTION ]
    [ SCHEMABINDING ]
    [ VIEW_METADATA ]
}
```

Аргументы `ALTER VIEW` означают то же, что и в команде `CREATE VIEW`.

Если представление было создано с флагами `WITH ENCRYPTION` или `WITH CHECK OPTION`, эти флаги остаются в модифицированном представлении лишь в том случае, если они присутствуют в команде `ALTER VIEW`. Другими словами, вы можете случайно отменить флаги `WITH ENCRYPTION` или `WITH CHECK OPTION`,
если забудете включить их в `ALTER VIEW`.

Пример команды `ALTER VIEW`:

```
ALTER VIEW RussianAuthorsView WITH ENCRYPTION AS
SELECT [Имя] AS [rav_Имя], [Фамилия] AS [rav_Фамилия]
FROM Authors
WHERE [Язык написания книги] = 'русский'
WITH CHECK OPTION
```



ГЛАВА 8

Настройка параметров базы данных

Вы можете создавать и удалять базы данных, используя команды `CREATE DATABASE` и `DROP DATABASE`. Например:

```
CREATE DATABASE [Моя самая важная база]
DROP DATABASE [Моя самая важная база]
```

Если вы захотите переименовать базу данных, используйте команду `sp_renamedb`. У нее следующий синтаксис:

```
sp_renamedb 'старое имя', 'новое имя'
```

Только системный администратор может переименовать базу данных. База данных до этого должна быть переведена в однопользовательский режим.

Конфигурация параметров базы данных может осуществляться хранимой процедурой `sp_dboption`. Эта процедура имеет следующий синтаксис:

```
sp_dboption ['база_данных'] [, 'имя_параметра'] [, 'значение']
```

Если параметры не указаны, процедура выводит список всех возможных параметров. Список всех параметров представлен в табл. 8.1.

Таблица 8.1. *Параметры конфигурации баз данных*

Параметр	Описание
`autoclose`	Если параметр равен `TRUE`, после выхода последнего пользователя база данных закрывается с освобождением всех ресурсов. По умолчанию `FALSE`
`autoshrink`	Если параметр равен `TRUE`, то файлы базы данных периодически автоматически сжимаются. По умолчанию `FALSE`

Таблица 8.1 *(продолжение)*

Параметр	Описание
`auto create statistics`	Установка опции разрешает серверу автоматическое создание статистики для таблиц базы данных. Статистика используется оптимизатором запросов для построения наиболее оптимальных планов исполнения запросов, пакетов, хранимых процедур и т. д. По умолчанию `ON`
`auto update statistics`	Установка опции разрешает серверу автоматическое обновление статистики для таблиц базы данных. Автоматическое обновление статистики позволяет оптимизатору запросов генерировать более эффективные планы исполнения, но требует определенных затрат процессорного времени на анализ информации. По умолчанию `ON`
`ANSI null default`	Если параметр равен `TRUE` и при создании или модификации таблицы для столбца не был задан параметр `NOT NULL`, то допустимость значений `NULL` в этом столбце определяется по правилам SQL-92. По умолчанию `FALSE`
`ANSI nulls`	Если параметр равен `TRUE`, результат всех сравнений с `NULL` считается равным `Unknown`. Если параметр равен `FALSE` и при этом не используются значения Unicode, результат `NULL = NULL` считается равным `TRUE`. По умолчанию `FALSE`
`ANSI warnings`	Если параметр равен `TRUE`, то при возникновении состояний типа "деление на ноль" выдаются ошибки или предупреждения. По умолчанию `FALSE`
`arithabort`	Управляет прерыванием запросов или транзакции при возникновении ошибок, таких как деление на ноль или переполнение. При установленной опции в случае возникновения ошибки производится откат транзакции. Когда же опция сброшена, возвращается значение `NULL`, и выполнение запроса или транзакции продолжается. Используя совместно с опцией `ANSI warnings`, можно контролировать выдачу состояний об ошибке
`concat null yields null`	Если параметр равен `TRUE`, то при объединении строки со значением `NULL` результат будет также равен `NULL`. При сброшенной опции значение `NULL` будет рассматриваться как пустая строка. По умолчанию `FALSE`

Таблица 8.1 *(продолжение)*

Параметр	Описание
`cursor close on commit`	Если параметр равен `TRUE`, любые курсоры, открытые на момент закрепления или отката изменений, закрываются. Если параметр равен `FALSE`, курсоры остаются открытыми при завершении транзакции. Откат транзакции при значении `FALSE` закрывает любые курсоры, кроме тех, которые определялись как `INSENSITIVE` или `STATIC`. По умолчанию `FALSE`
`dbo use only`	Если параметр равен `TRUE`, то базу данных может использовать лишь ее владелец
`default to local cursor`	Если параметр равен `TRUE`, при объявлениях курсоров по умолчанию принимается тип `LOCAL`. При сброшенной опции будет создаваться глобальный курсор (`GLOBAL`). По умолчанию `FALSE`
`merge publish`	Если параметр равен `TRUE`, база данных может быть опубликована для репликации сведением. По умолчанию `FALSE`
`numeric roundabort`	Определяет поведение сервера при попытке задать переменной или столбцу значение с более высокой точностью, чем допускает тип данных. Установка опции предписывает генерировать сообщение об ошибке, тогда как при сброшенной опции выполняется округление, и ошибка не выдается. Прерывание же запроса при установленной опции зависит от состояния опции `arithabort`
`offline`	Если параметр равен `TRUE`, база данных находится в отключенном состоянии и не может никем использоваться. По умолчанию `FALSE`
`published`	Если параметр равен `TRUE`, база данных может быть опубликована для репликации. По умолчанию `FALSE`
`quoted identifier`	Если параметр равен `TRUE`, идентификаторы могут заключаться в кавычки. По умолчанию `FALSE`
`read only`	Если параметр равен `TRUE`, пользователи могут только читать данные из базы, не модифицируя ее. По умолчанию `FALSE`
`recursive triggers`	Если параметр равен `TRUE`, то модификация данных в результате срабатывания триггера может приводить к срабатыванию того же самого триггера. По умолчанию `FALSE`
`single user`	Если параметр равен `TRUE`, то в произвольный момент времени с базой данных может работать лишь один пользователь. По умолчанию `FALSE`

Таблица 8.1 *(окончание)*

Параметр	Описание
`select into/bulkcopy`	В случае установки опции разрешается выполнение запроса `SELECT...INTO` и операция массированной закачки данных в базу данных. По умолчанию опция сброшена, т. к. указанные операции являются не протоколируемыми, т. е. информация о них не отображается в журнале транзакций
`subscribed`	Если параметр равен `TRUE`, база данных может быть подписана для публикации. По умолчанию `TRUE`
`torn page detection`	Если параметр равен `TRUE`, становится возможным обнаружение незавершенных страниц. По умолчанию `TRUE`
`trunc log on chkpt`	Путем установки данной опции можно разрешить автоматическое усечение (truncate) журнала транзакций при выполнении в базе данных контрольной точки (checkpoint). Это позволяет оперативно удалять неактивную часть журнала транзакций, но при этом становится невозможным создание резервной копии журнала транзакций

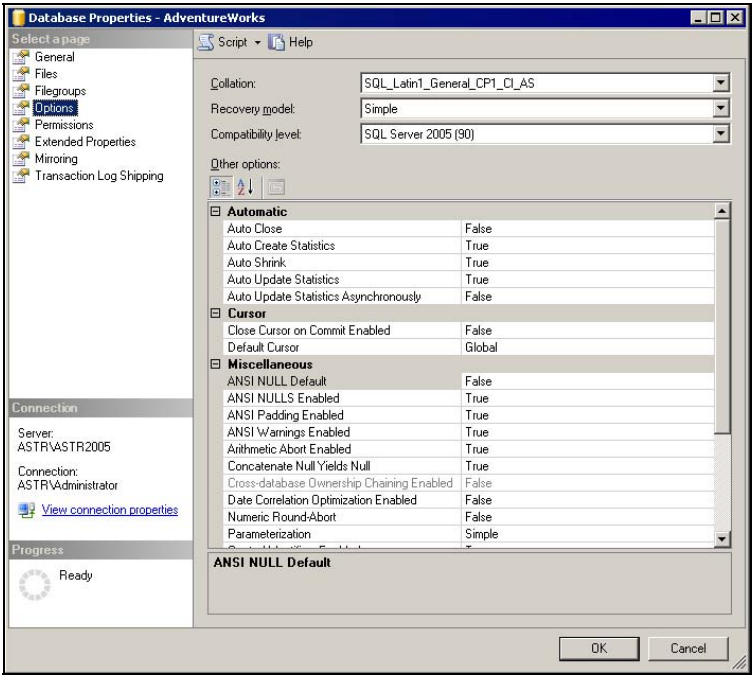


Рис. 8.1. Редактор свойств базы данных в SQL Server Management Studio

Пример использования команды `sp_dboption`:

```
EXEC sp_dboption 'AdventureWorks', 'autoshrink', 'true'
```

Так вы можете управлять всеми параметрами базы данных.

Кроме того, в SQL Server Management Studio, щелкнув правой кнопкой мыши на имени базы, вы можете вызвать редактор свойств базы данных (рис. 8.1) и настроить свойства так, как вам нужно.

Команда *USE*

При выполнении SQL-команды вы можете определить, для какой именно базы данных они предназначены, используя команду `USE`. Синтаксис этой команды:

```
USE [имя базы]
```

Например:

```
USE [Обычные покупатели]
SELECT * FROM [Покупатели]
USE [VIP покупатели]
SELECT * FROM [Покупатели]
```

Здесь сначала возвращается список покупателей из базы обычных покупателей, а потом из базы VIP-покупателей.

Выбор сопоставления

При работе с базой данных администратор должен выбрать *сопоставление*, которое будет использоваться при работе базы данных.

Сопоставление (collation) — это правила сравнения и хранения данных. Сопоставление определяет кодировку (кодовую страницу), в которой будут храниться данные. Также сопоставление отражается на правилах именования объектов базы данных и выборе имен пользователей. Например, при выборе сопоставления, нечувствительного к регистру (case insensitive), сервер не будет делать различия между символами, набранными в верхнем и нижнем регистрах. То есть имена `AUTHOR` и `Author` будут считаться именем одного и того же объекта. Если же для базы данных существует сопоставление, чувствительное к регистру (case sensitive), то сервер будет проверять уникальность имен объектов с учетом регистра.

При создании базы данных, если не указано ее сопоставление, по умолчанию используется сопоставление сервера.

Необходимо отметить, что при выполнении запросов к объектам различных баз данных чувствительность к регистру имен этих объектов определяется той базой данных, в которой расположены эти объекты, а не базой данных, в контексте которой обрабатывается запрос. Таким образом, в одном запросе можно использовать имена как чувствительные к регистру, так и не чувствительные.

Установленное на уровне базы данных сопоставление не является обязательным для таблиц, создаваемых в базе данных, а также столбцов этих таблиц. При создании таблицы или даже отдельного столбца можно явно определить сопоставление, которое должно использоваться для этой таблицы.

При определении локальных переменных, меток перехода (для команды `GOTO`), временных хранимых процедур и временных таблиц применяется сопоставление, определенное на уровне сервера. Это делается для того, чтобы запросы вели себя одинаково при переключении между различными базами данных. В противном случае нельзя гарантировать верного выполнения запросов. Например, если пользователь создает две переменные с одинаковыми именами, но набранными в разных регистрах, то необходимо гарантировать, что они будут различаться независимо от того, в контексте какой базы данных будут использоваться.

Данные, имеющие различные сопоставления, обрабатываются следующим образом. Например, в одном столбце таблицы хранятся данные, чувствительные к регистру, а в другом — нет.

Например:

```
CREATE TABLE TestTable(CaseInsensitiveColumn nvarchar(50)
                         COLLATE Latin1_General_CI_AS,
                       CaseSensitiveColumn nvarchar(50)
                         COLLATE Latin1_General_CS_AS,
)
```

Здесь создается таблица `TestTable` с двумя столбцами. Первый из них имеет сопоставление, чувствительное к регистру, а второй — нет. Теперь вставим в таблицу новую строку:

```
INSERT INTO TestTable VALUES ('Aaa', 'AAa')
```

Для столбцов были выбраны значения, отличающиеся только регистром второго символа. Однако даже такие небольшие различия приводят к конфликту сопоставлений. Например:

```
SELECT *
FROM TestTable
WHERE CaseInsensitiveColumn = CaseSensitiveColumn
```

При выполнении этого запроса будет выдано следующее сообщение об ошибке:

```
Msg 468, Level 16, State 9, Line 1
Cannot resolve the collation conflict between "Latin1_General_CS_AS" and
"Latin1_General_CI_AS" in the equal to operation.
```

Это сообщение о конфликте сопоставлений. Сервер не знает, по каким правилам должно выполняться сравнение значений. Для разрешения конфликта нужно явно указать, какое сопоставление (правила сравнения символов) должно использоваться при выполнении запроса. Для этого после выражения необходимо указать ключевое слово `COLLATE` с именем сопоставления.

Например:

```
SELECT *
  FROM TestTable
  WHERE CaseInsensitiveColumn=CaseSensitiveColumn
  COLLATE Latin1_General_CI_AS
```

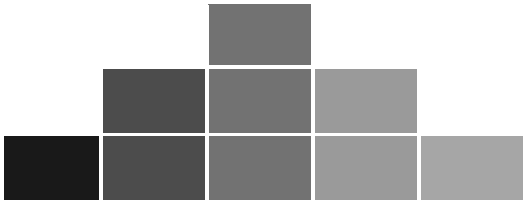
Параметр `COLLATE` позволяет разрешать конфликты сопоставлений. Данный параметр может применяться не только в запросах `SELECT`, но и в любых других выражениях, обрабатывающих символьные данные.

Имя сопоставления представляет собой описание предопределенных свойств. Список сопоставлений и их описание можно получить с помощью функции `fn_helpcollations`, выполнив следующий запрос:

```
SELECT * FROM fn_helpcollations()
```

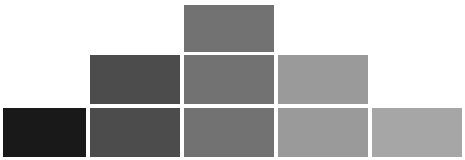
В ответ сервер возвратит 1011 строк с описанием отдельных сопоставлений. Например, следующий запрос возвращает все сопоставления для работы с кириллицей:

```
SELECT *
FROM fn_helpcollations()
WHERE name LIKE 'Cyrillic%'
```



ЧАСТЬ II

Программирование в SQL Server



ГЛАВА 9

Программирование на T-SQL

Язык SQL стандарта ANSI позволяет создавать и модифицировать базы данных и читать и изменять информацию в них. Для программирования это недостаточно. Программирование требует дополнительных возможностей. Поэтому SQL Server содержит расширенную версию SQL, которая называется Transact-SQL, или T-SQL.

Далее будут рассмотрены различные компоненты языка T-SQL, используемые в программировании.

Пакеты

Пакет (batch) — это одна или несколько команд SQL, передаваемых на SQL Server для выполнения. После подключения вы начинаете передавать пакеты с различными командами SQL Server.

Вы можете использовать команду `GO`, чтобы определить, когда передается пакет.

Например, следующий пакет состоит из двух команд `SELECT`:

```
SELECT * FROM Authors
SELECT * FROM Books
GO
SELECT * FROM Customers
```

В этом примере сначала две команды передаются серверу и выполняются им, а их результаты вместе возвращаются клиенту. Команды этого пакета анализируются, компилируются и выполняются как единая группа. Если сервер обнаруживает синтаксические ошибки, не выполняется весь пакет. А последняя команда, которая возвращает список покупателей, выполняется в отдельном пакете.

Комментарии

Комментарии представляют собой очень важную часть любой программной среды. Есть они и в T-SQL. Существует несколько основных вариантов записи комментариев.

Комментарии бывают однострочными и многострочными. Однострочный комментарий начинается с `--`. А многострочный начинается с `/*`, а заканчивается `*/`.

Например:

```
/* Многострочный
   комментарий
*/
-- Однострочный комментарий
SELECT * FROM Authors -- комментарий
```

Переменные

SQL Server поддерживает два типа переменных: локальные и глобальные. Локальные переменные существуют только в пределах сеанса, во время которого они были созданы. Глобальные используются для получения информации о сервере в целом.

Локальные переменные

Локальные переменные служат для хранения временной информации. Локальные переменные объявляются командой `DECLARE` и существуют лишь на время выполнения пакета. После завершения пакета вы уже не сможете обратиться к ним.

Значения переменных задаются командой `SELECT` или `SET`. Если значение присваивается одной переменной, эти команды эквивалентны.

Команда `SELECT` может присвоить значения сразу нескольким переменным. В следующем примере команда `DECLARE` объявляет две переменные, одна из которых задается командой `SELECT`, а другая — командой `SET`. Затем происходит чтение и вывод значения обеих переменных:

```
DECLARE @MyDate datetime, @Number int, @MyString nvarchar(50)
SELECT @Number = 1, @MyString='My string'
SET @MyDate = GETDATE()
SELECT @MyDate, @Number , @MyString
```

Вы можете присваивать значения нескольким переменным в одной команде SELECT и объявлять несколько переменных в одной команде DECLARE.

Начиная с SQL Server 2008, можно присваивать значение переменной прямо в команде DECLARE. Например:

```
DECLARE @Number int = 5
```

А также вы можете использовать математические операции типа +=, -= и подобные, которые прибавляют, вычитают некое значение от переменной, которой потом и присваивается вычисленный результат. Например:

```
SET @myVar += 100
```

Глобальные переменные

Глобальные переменные используются сервером для отслеживания информации уровня сервера или базы данных, относящейся к конкретному сеансу. Для глобальных переменных невозможно явное присваивание или объявление. Некоторые глобальные переменные представлены в табл. 9.1.

Таблица 9.1. *Некоторые глобальные переменные*

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

Пример вывода значения глобальной переменной @@VERSION:

```
SELECT @@VERSION
```

Результат:

```
Microsoft SQL Server 2008 (RTM) - 10.0.1600.22 (Intel X86) Jul 9 2008 14:43:34
Copyright (c) 1988-2008 Microsoft Corporation Enterprise Evaluation
Edition on Windows NT 5.1 <X86> (Build 2600: Service Pack 2)
```

Команды T-SQL

Transact-SQL (T-SQL) дополняет стандарт ANSI, не обладающий возможностями обработки сообщений или управления потоком команд. В сущности, все ваши программы будут написаны именно на T-SQL.

Команда *PRINT*

Команда `PRINT` передает сообщения длиной до 1024 байтов в качестве служебной информации. В SQL Server Management Studio служебные сообщения выводятся на вкладке **Messages** (рис. 9.1).

Рис. 9.1. Вкладка для вывода служебных сообщений в SQL Server Management Studio

Например, следующая команда `PRINT` просто передает сообщение:

```
PRINT 'Мое первое сообщение'
```

Таким образом вы можете выводить любую информацию.

Команды условного выполнения

Термин "условное выполнение" означает, что команды выполняются лишь в случае истинности некоторого критерия или условия.

IF...ELSE

Конструкция `IF...ELSE` определяет команды, выполнение которых зависит от некоторого критерия. Команда `IF...ELSE` имеет следующий синтаксис:

```
IF логическое_выражение
    {команда | блок_команд}
```

```
[ELSE
    {команда | блок_команд}]
```

В следующем примере команда `IF...ELSE` проверяет среднюю цену книг и возвращает соответствующий текст:

```
IF (SELECT AVG(price) FROM Books) > 19.95
    PRINT 'Средняя цена книг больше, чем 19,95'
ELSE
    PRINT 'Средняя цена книг меньше или равна 19,95'
```

IF EXISTS

Команда `IF EXISTS` является частным случаем команды `IF` и позволяет узнать, существуют ли какие-либо экземпляры, определяемые условием. Команда `IF EXISTS` заменяет конструкцию `COUNT(*) > 0` для проверки существования записей. При обнаружении первой совпадающей записи обработка команды `EXISTS` прекращается.

Далее представлен пример, где определяется, присутствуют ли в таблице данные об авторах по фамилии Азимов:

```
IF EXISTS(SELECT * FROM Authors WHERE LastName = 'Азимов')
    PRINT 'Автор с такой фамилией существует'
ELSE
    PRINT 'Такого автора не существует'
```

BEGIN...END

Конструкция `BEGIN...END` используется для создания блока команд. Все, что находится между `BEGIN` и `END`, является частью блока.

Например, проверка средней цены книги могла бы выглядеть так:

```
IF (SELECT AVG(price) FROM Books) > 19.95
  BEGIN
    PRINT 'Средняя цена книг больше, чем 19,95'
    PRINT 'Ура, мы увеличили среднюю стоимость книг!'
    INSERT INTO [Рекорды] ([Дата рекорда], [Значение])
           VALUES (GETDATE(), SELECT AVG(price) FROM Books)
  END
```

CASE

Для тех ситуаций, когда проверка нескольких условий требует многих команд `IF`, в SQL Server предусмотрена конструкция `CASE`. Конструкция `CASE`, в отличие от `IF`, может использоваться в команде `SELECT`.

Конструкция CASE имеет следующий синтаксис:

```
CASE выражение
     WHEN выражение1 THEN выражение2
     [...]
     [ELSE выражениеN]
END
```

Представим пример команды CASE:

```
SELECT [ID книги], [имя книги],
    CASE [жанр]
         WHEN 'Фант' THEN 'Фантастика'
         WHEN 'Дет' THEN 'Детектив'
         WHEN 'Ром' THEN 'Романтика'
         ELSE 'Неизвестно'
    END AS [Жанр книги],
           [Цена книги],
    CASE
         WHEN [Цена книги] < 10 THEN 'Дешевая'
         WHEN [Цена книги] BETWEEN 10 AND 20 THEN 'Средняя'
         WHEN [Цена книги] > 20 THEN 'Дорогая'
         ELSE 'Неизвестно'
    END AS [Категория книги],
FROM Books
```

WHILE

Конструкция WHILE используется для многократного выполнения команд. Команда WHILE вычисляет условие цикла и, если оно равно TRUE, выполняет команду или блок команд.

Команда WHILE имеет следующий синтаксис:

```
WHILE логическое_условие
  [{команда | блок_команд}]
[BREAK]
[CONTINUE]
```

Команда BREAK безусловно завершает выполнение цикла WHILE и продолжает выполнение с первой команды после END блока команд. Команда CONTINUE заново вычисляет логическое условие и, если оно равно TRUE, продолжает выполнение с начала цикла.

В следующем примере цикл выполняется до тех пор, пока средняя цена остается меньше 25:

```
WHILE (SELECT AVG(price) FROM Books) < 25
  BEGIN
    UPDATE Books SET price = price * 1.05
    IF EXISTS(SELECT * FROM Books WHERE price < 15)
      CONTINUE
    ELSE IF EXISTS(SELECT price FROM Books WHERE price > 50)
      BREAK
  END
```

В этом примере в каждом цикле сначала увеличивается цена книги на 5%, а потом выполняется проверка, есть ли еще книги с ценой меньше 15. Если есть, то цикл начинается сначала.

Циклы заканчивают выполняться, когда средняя цена книг становится больше или равна 25, или если не будет ни одной книги дешевле 15, но будет хоть одна книга дороже 50.

GOTO

Использование команды `GOTO` позволяет по-другому организовать многократно повторяющиеся вычисления.

Пример использования `GOTO`:

```
DECLARE @Counter INT
SET @Counter = 0
Top:
SET @Counter = @Counter + 1
PRINT 'Счетчик = ' + @Counter
IF @Counter < 10
    GOTO Top
```

В этом примере вначале вы устанавливаете метку `Top`. А позже, если значение счетчика меньше 10, с помощью команды `GOTO` управление снова передается метке `Top`.

WAITFOR

Команда `WAITFOR` переводит запрос в состояние ожидания на некоторое время или до наступления заданного времени.

Команда `WAITFOR` имеет следующий синтаксис:

```
WAITFOR {DELAY 'время' | TIME 'время'}
```

Параметр DELAY заставляет сделать паузу заданной длины (максимальное значение — 24 часа). Если в команде указан параметр TIME, процесс приостанавливается до наступления заданного времени. В обоих случаях время задается в формате hh:mi:ss (дату задать нельзя).

Примеры использования команды WAITFOR:

```
-- Пауза до 22:00
WAITFOR TIME '22:00:00'
-- Выводить текущий список пользователей каждые 30 секунд
WHILE 1 < 2
BEGIN
    WAITFOR DELAY '00:00:30'
    EXEC sp_who
END
```

RETURN

Командой RETURN осуществляется безусловный выход из обрабатываемого пакета. При желании, при выходе из хранимой процедуры в команде можно задать код возврата.

Команда RETURN имеет следующий синтаксис:

```
RETURN  {целое_число}
```

Пример использования команды RETURN:

```
IF EXISTS(SELECT * FROM Authors WHERE LastName = 'Азимов')
BEGIN
    PRINT 'Все в порядке'
    RETURN
END
PRINT 'У вас нет всех популярных книг'
PRINT 'Дополните вашу библиотеку!'
```

SET

Команда SET обычно используется для установки значения переменной. В этом случае она имеет следующий синтаксис:

```
SET имя_переменной = значение
```

Например:

```
SET @MyVar = 12345
```

Также командой SET задаются некоторые параметры, определяющие реакцию сервера на некоторые условия. Команда SET имеет следующий синтаксис:

```
SET условие {ON | OFF | значение}
```

Примеры некоторых команд SET:

```
SET rowcount 100
SET nocount on
SET showplan on
```

Параметры настраиваются на уровне сеанса или на уровне хранимой процедуры. Они не сохраняются между сеансами.

Обработка ошибок

При выполнении SQL-кода может возникнуть ошибка, и потребуется обработать ее. Для этого существует команда RAISERROR.

Команда RAISERROR передает состояния, коды и сообщения ошибок на программном уровне. RAISERROR позволяет использовать стандартное сообщение или определить новое сообщение об ошибке.

Если RAISERROR вызывается в блоке TRY...CATCH, то управление передается блоку CATCH. В противном случае выполнение цепочки команд продолжится (хотя вы можете использовать команду RETURN, чтобы прервать выполнение цепочки команд).

Синтаксис RAISERROR, показанный в следующем фрагменте, позволяет использовать стандартное сообщение или определить новое сообщение непосредственно в команде:

```
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.

Следующий пример вызывает незапланированное сообщение об ошибке:

```
IF NOT EXISTS(SELECT * FROM Authors)
   RAISEERROR ('Таблица с именами авторов не заполнена. Заполните ее.', 16, 1)
```

Вы можете использовать функцию @@ERROR, которая возвращает код последней ошибки, происшедшей в текущем соединении. Вот пример использования этой функции:

```
IF NOT EXISTS(SELECT * FROM Authors)
   RAISEERROR ('Таблица с именами авторов не заполнена. Заполните ее.', 16, 1)

IF @@ERROR <> 0
    PRINT 'Возникла ошибка при проверке таблицы Author.'
ELSE
    PRINT 'Проверка прошла успешно'
GO
```

Конструкция *TRY...CATCH*

Начиная с SQL Server 2005, существует гораздо более удобный способ обработки ошибок — с помощью конструкции TRY...CATCH. Эта конструкция имеет следующий синтаксис:

```
BEGIN TRY
    { sql_выражение | блок_выражений }
END TRY
BEGIN CATCH
    { sql_выражение | блок_выражений }
END CATCH
```

Каждый блок TRY...CATCH должен быть в одном пакете. Пример обработки ошибки:

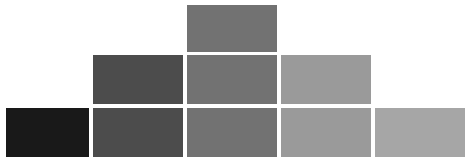
```
BEGIN TRY
    -- Обновление данных счета клиента
    UPDATE Account SET Balanse = @Sum WHERE ClientID = @ClientID
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() as ErrorNumber,
        ERROR_MESSAGE() as ErrorMessage;
END CATCH
```

В этом примере, если при обновлении таблицы Account произойдет ошибка, будет выведено описание этой ошибки.

В блоке TRY...CATCH вы можете использовать некоторые функции, чтобы получить более подробную информацию об ошибке. Их описание представлено в табл. 9.2.

Таблица 9.2. *Функции, возвращающие описание ошибки*

Имя функции	Описание
ERROR_NUMBER()	Возвращает номер ошибки
ERROR_MESSAGE()	Возвращает сообщение об ошибке
ERROR_LINE()	Возвращает номер строки, где возникла ошибка
ERROR_PROCEDURE()	Возвращает имя хранимой процедуры или триггера, где возникла ошибка
ERROR_SEVERITY()	Возвращает описание уровня серьезности ошибки
ERROR_STATE()	Возвращает номер состояния ошибки



ГЛАВА 10

Курсоры

Курсоры — это механизм, позволяющий приложениям выполнять действия с отдельными записями, входящими в результат запроса, вместо обработки итогового набора в целом.

SQL проектировался как язык обработки данных, ориентированный на работу с наборами данных. Курсоры работают медленнее, чем выполняется обычная обработка наборов данных с помощью SQL.

Вот пример использования курсора. Предположим, если книга стоит дешевле некоторой суммы, вы хотите увеличить ее стоимость на 15%, а если дороже — уменьшить цену на 10%.

С помощью курсоров вы можете определить цену книги для каждой записи и обновить ее соответствующим образом. Далее описано, как это сделать.

Последовательность действий с курсорами

При использовании курсоров приняты следующие основные действия:

1. Свяжите курсор с итоговым набором и определите характеристики курсора.
2. Заполните курсор командами T-SQL.
3. Произведите выборку обрабатываемых записей.
4. Выполните действия с записями.
5. Закройте курсор.

Типы курсоров T-SQL

Курсоры T-SQL делятся на три основных типа:

- *динамические курсоры* — курсоры, при использовании которых изменения данных отображаются при перемещении курсора. Динамические курсоры используют минимальное количество ресурсов сервера за пределами базы данных;
- *статические курсоры* — курсоры, изолированные от изменений данных. Сервер сохраняет весь итоговый набор курсора при первой выборке;
- *ключевые курсоры* — курсоры, в которых отображается бóльшая часть изменяемых данных, хотя новые записи не появляются. При открытии курсора сервер запоминает ключи возвращаемых записей. Новые записи не включаются в курсор, а при обращении к записи, удаленной другим пользователем, происходит ошибка.

Работа с курсорами в T-SQL

Приведем последовательность действий при использовании курсора. Следите за тем, чтобы все операции выполнялись в нужном порядке:

1. Объявите курсор.
2. Откройте курсор.
3. Произведите выборку записей.
4. Модифицируйте или удалите записи из курсора.
5. Продолжайте выборку записей до завершения всей обработки.
6. Закройте курсор.
7. Освободите ресурсы курсора.

Опишем каждый из этих шагов более подробно.

Объявление курсоров

Синтаксис объявления курсора:

```
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] ]
```

Например:

```
DECLARE authors_curs  CURSOR FOR
    SELECT * FROM Authors WHERE City='Москва'
    FOR READ_ONLY
```

Этот курсор обращается к таблице `Authors` и отбирает все записи, для которых указано, что автор живет в Москве. Здесь запрещено обновление данных через курсор.

Имя курсора подчиняется стандартным правилам выбора имен для объектов баз данных.

Аргументы в команде `DECLARE` имеют следующий смысл:

- `LOCAL` — курсор является локальным по отношению к пакету, хранимой процедуре или триггеру, где объявляется курсор. В конце процедуры или триггера локальные курсоры автоматически освобождаются. Если курсор передается в качестве выходного параметра, он освобождается лишь после освобождения последней переменной, ссылающейся на него;
- `GLOBAL` — курсор доступен в любой момент в течение подключения. Глобальный курсор освобождается явно или уничтожается в конце сеанса;
- `FORWARD_ONLY` — курсор допускает перебор записей только в прямом направлении (от первой записи к последней). Если ключевое слово `STATIC | KEYSET | DYNAMIC` не указано, курсор является динамическим. При отсутствии этих ключевых слов параметр `FORWARD_ONLY` используется по умолчанию; в противном случае по умолчанию используется `SCROLL`;
- `SCROLL` — создается прокручиваемый курсор. К записям в нем можно обращаться в любом порядке и направлении;
- `STATIC` — статический курсор, доступный только для чтения. Содержимое курсора формируется один раз и больше не обновляется. Следовательно, изменения в базе данных на нем не сказываются;
- `KEYSET` — ключевой курсор. Ключевой курсор представляет собой набор ключей, идентифицирующих строки набора данных;
- `DYNAMIC` — динамический курсор. Каждый раз, когда вы обращаетесь к динамическому курсору, происходит выборка из соответствующих таблиц. Следовательно, курсор всегда отображает актуальные данные;
- `FAST_FORWARD` — указывается для курсора, имеющего свойство "только для чтения". Курсор будет оптимизирован для быстрого доступа к данным. Эту опцию нельзя использовать вместе с опциями `FORWARD_ONLY` и `OPTIMISTIC`;
- `READ_ONLY` — курсор доступен только для чтения;

- SCROLL_LOCKS — параметр гарантирует, что операции обновления и удаления будут успешными, а блокировки сохраняются до выборки следующей записи;
- OPTIMISTIC — успешность операций обновления и удаления не гарантируется, а блокировки не сохраняются до выборки следующей записи;
- FOR UPDATE — режим используется по умолчанию, если при объявлении курсора не указано обратное;
- COLUMN_LIST — список всех обновляемых столбцов.

SQL Server разрешает всем пользователям объявлять и использовать курсоры.

В курсорах может использоваться любая команда SELECT, удовлетворяющая следующим условиям:

- команда SELECT не может содержать секции SELECT INTO, COMPUTE, COMPUTE BY и FOR BROWSE;
- при использовании секций GROUP BY, HAVING, DISTINCT или UNION курсор становится статическим;
- если все таблицы, используемые в курсоре, не имеют уникального индекса, курсор становится статическим.

Если все столбцы, используемые в ORDER BY, не входят в уникальный индекс, динамический курсор преобразуется в ключевой или статический.

Объявленный курсор готов к открытию.

Открытие курсоров

Открытие курсора начинает обработку команды SELECT и готовит курсор к выборке записей. Команда открытия курсора имеет следующий синтаксис:

```
OPEN {{ [GLOBAL] имя_курсора } | @переменная}
```

Параметр GLOBAL означает, что курсор является глобальным. Если существуют локальный и глобальный курсоры с одинаковыми именами, для открытия глобального курсора необходимо задать параметр GLOBAL. Переменная *@переменная* содержит имя курсора.

Пример открытия курсора:

```
DECLARE Authors_curs CURSOR
    FOR SELECT * FROM Authors
    FOR READ_ONLY
OPEN Authors_curs
```

Приведенный фрагмент объявляет и открывает курсор. Открыть можно только объявленный курсор. Глобальная переменная @@CURSOR_ROWS содержит количество записей, полученных последней командой открытия курсора. После открытия курсора следующим шагом является выборка записей.

Выборка записей

Выборка записей курсора осуществляется командой FETCH. Для выборки записей необходимо предварительно открыть курсор.

Синтаксис выборки записей:

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

Параметры GLOBAL, *имя_курсора* и *@переменная* имеют тот же смысл, что и при открытии курсора. 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 данных);
- PRIOR — курсор перемещается на одну запись назад в текущем наборе;
- ABSOLUTE *n*/*-n* — если n является положительным числом, курсор перемещается к *n*-ой записи от начала. Если *n* является отрицательным числом, курсор перемещается к последней записи набора и отсчитывает *n* записей в обратном направлении. Если *n* = 0, записи не возвращаются;
- RELATIVE *n*/*-n* — если *n* является положительным числом, курсор перемещается на *n* записей вперед от текущей позиции. Если *n* является отрицательным числом, курсор перемещается на *n* записей назад. Если *n* = 0, возвращается текущая запись.

В качестве аргументов при использовании параметров ABSOLUTE и RELATIVE курсор может получать целые переменные или параметры хранимых процедур.

Следующий пример объявляет и открывает курсор, после чего выбирает из него данные:

```
DECLARE Authors_curs CURSOR
    FOR SELECT * FROM Authors
    FOR READ_ONLY
OPEN Authors_curs
```

```
FETCH NEXT FROM Authors_curs
DECLARE @LastName NVARCHAR(50), @FirstName NVARCHAR(50)

-- Чтение следующей записи из набора
FETCH NEXT FROM Authors_curs INTO @LastName, @FirstName
-- Прочитать 25-ю запись от текущей
FETCH RELATIVE 25 FROM Authors_curs
CLOSE Authors_curs
DEALLOCATE Authors_curs
```

Если при объявлении курсора не указано ключевое слово `SCROLL`, допускается выборка только следующей записи (`FETCH NEXT`). При наличии списка переменных их типы должны совпадать с типами в списке выборки. Если список переменных не задан, результат возвращается непосредственно пользователю.

В табл. 10.1 перечислены глобальные переменные, значения которых можно использовать при работе с курсорами.

Таблица 10.1. *Глобальные переменные, используемые при работе с курсорами*

Глобальная переменная	Описание
`@@CURSOR_ROWS`	Количество записей в последнем открытом курсоре: • `-#` — количество записей, находящихся в данный момент в ключевом наборе, если выборка происходит асинхронно; • `#` — количество записей после завершения выборки; • `0` — открытые курсоры отсутствуют
`@@FETCH_STATUS`	Результат команды `FETCH`: • `0` — выборка успешна; • `-1` — выборка завершилась неудачей или запрашиваемая запись выходит за границы диапазона; • `-2` — возвращенная запись отсутствует в таблице (она была удалена после открытия курсора)

В следующем примере переменная `@@FETCH_STATUS` использована в условии цикла `WHILE` при обработке курсора:

```
DECLARE Author_curs CURSOR FOR
    SELECT FirstName, LastName, NumberBooks FROM Authors
    FOR READ_ONLY
DECLARE @LastName NVARCHAR(50), @FirstName NVARCHAR(50),
```

```
    @NumberBooks int

OPEN Author_curs
FETCH Author_curs INTO @FirstName, @LastName, @NumberBooks
/* Цикл работает до тех пор, пока есть необработанные записи */
WHILE @@FETCH_STATUS = 0
BEGIN
    IF @NumberBooks > 10
        SELECT FirstName = @FirstName, LastName = @LastName,
        Discount = '10%';
    ELSE
        SELECT FirstName = @FirstName, LastName = @LastName,
        Discount = '3%';

    FETCH Author_curs INTO @FirstName, @LastName, @NumberBooks
END

IF @@FETCH_STATUS = -2
    RAISERROR ("Выход из цикла по ошибке выборки", 16, 1)

CLOSE Author_curs
DEALLOCATE Author_curs
```

В этом примере выбран очень сложный путь для получения информации о скидке для каждого конкретного автора.

Модификация записей с помощью курсоров

Вы можете модифицировать записи с помощью курсора. Для этого в команды `UPDATE` и `DELETE` включается секция `WHERE CURRENT OF`.

Синтаксис команд:

```
UPDATE имя_таблицы
    SET имя_столбца = выражение [, ...]
   [FROM имя_таблицы [, имя_таблицы]]
    WHERE CURRENT OF имя_курсора
      [{AND | OR} ...]

DELETE [FROM] имя_таблицы [FROM имя_таблицы [, имя_таблицы]
    WHERE CURRENT OF имя_курсора
    [{AND | OR} ...]
```

Примеры использования этих команд:

```
UPDATE Authors
       SET AuthorLevel = AuthorLevel + 1
       WHERE CURRENT OF Authors_curs

DELETE Authors WHERE CURRENT OF Authors_curs
```

Не разрешается удаление из многотабличных курсоров (с секцией JOIN или представлением, содержащим JOIN). Обновление многотабличного курсора разрешается лишь в том случае, если оно затрагивает только одну таблицу.

Операция удаления удаляет запись из таблицы. Обновление не изменяет текущей позиции курсора, что делает возможным многократное обновление записи.

При попытке выполнить команду UPDATE или DELETE для курсора READ_ONLY выдается сообщение об ошибке, в котором говорится, что курсор доступен только для чтения.

Закрытие курсоров

Команда CLOSE переводит курсор в неактивное состояние и фактически удаляет его итоговый набор. Курсор можно открыть повторно, при этом все его указатели сбрасываются.

Синтаксис команды:

```
CLOSE {{[GLOBAL]  имя_курсора} | @переменная}
```

Пример использования:

```
CLOSE Authors_curs
```

Освобождение курсоров

В процессе освобождения курсора удаляется ссылка на него. Допускается освобождение курсора даже в том случае, если в нем остались незавершенные транзакции.

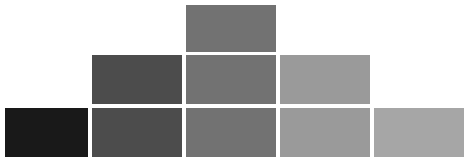
Синтаксис команды:

```
DEALLOCATE {{GLOBAL} имя_курсора} | @переменная}
```

После выполнения команды имя курсора может быть использовано и другой командой — DECLARE CURSOR. Если эта ссылка на курсор была последней, команда освобождает всю память, занятую структурой курсора. Чтобы заново открыть курсор, необходимо повторно объявить его.

Пример использования:

```
DEALLOCATE Authors_curs
```



ГЛАВА 11

Хранимые процедуры

Хранимые процедуры — это предварительно откомпилированные процедуры, программы, написанные на T-SQL и находящиеся в базе. Хранимые процедуры выполняются быстрее обычного T-SQL, т. к. хранятся в откомпилированном виде. Хранимые процедуры могут помочь изолировать пользователей от базовых табличных структур, скрыть особенности реализации какой-либо возможности.

Хранимые процедуры являются объектами, находящимися в базе данных. Создание хранимой процедуры очень похоже на создание таблицы. Например:

```
CREATE PROCEDURE MyFirstProcedure
    @Name nvarchar(50),
    @SexIsMale bit = 1
AS
BEGIN
    IF @SexIsMale = 1
        PRINT 'Уважаемый ' + @Name
    ELSE
        PRINT 'Уважаемая ' + @Name
END
GO
```

Эта хранимая процедура выводит различные сообщения, в зависимости от пола человека. У хранимой процедуры два параметра: имя и пол человека. При этом пол человека — необязательный параметр, вы можете не задавать его. В этом случае считается, что у человека мужской пол.

Вот примеры вызова этой процедуры:

```
EXEC MyFirstProcedure 'Иванов Андрей'
EXEC MyFirstProcedure 'Иванов Андрей', 1
EXEC MyFirstProcedure @Name = 'Иванова Мария', @SexIsMale = 0
```

Вы можете не писать, имя какого параметра вы инициализируете.

В этом случае параметры должны передаваться в том же порядке, как они следуют в хранимой процедуре.

Создание процедуры возможно только в текущей базе данных. Синтаксис создания хранимой процедуры выглядит следующим образом:

```
CREATE PROCEDURE имя_процедуры [ ; номер ]
    [ { @параметр тип_данных }
        [ VARYING ] [ = значение ] [ [OUT[PUT]] [ ,...n ]
[ WITH {RECOMPILE|ENCRYPTION}
[ FOR REPLICATION ]
AS
[BEGIN]
    выражения
[END]
```

Аргументы, используемые при создании хранимых процедур:

- *имя_процедуры* — подчиняется стандартным правилам выбора имен объектов. В рамках одной базы данных для процедур с одинаковыми именем и владельцем номер должен быть уникальным;
- *номер* — используется для группировки процедур как единого целого (часто используется для создания нескольких версий процедуры, для поддержки разных версий приложения);
- *@параметр* — представляет параметр, передаваемый или получаемый из процедуры. Процедура может иметь до 1024 параметров. Параметры могут принимать значения NULL. Параметр может относиться к любому типу данных SQL Server. Для параметров типа CURSOR ключевые слова VARYING и OUTPUT являются обязательными. VARYING означает, что параметр возвращает итоговый набор и используется только для типа данных CURSOR;
- *значение* — определяет значение параметра по умолчанию. Может быть любой константой или NULL;
- OUTPUT или OUT — означает, что параметр является выходным, т. е. может возвращаться стороне, вызвавшей процедуру;
- RECOMPILE — эта опция отменяет кэширование плана выполнения процедуры;
- ENCRYPTION — данная опция предписывает серверу шифровать текст процедуры, скрывая его от постороннего взгляда;
- FOR REPLICATION — означает, что процедура участвует в процессе репликации. Параметры FOR REPLICATION и RECOMPILE являются взаимоисключающими;

- *выражения* — произвольное количество команд SQL, составляющих тело процедуры. При этом ключевые слова BEGIN и END, создающие блок с командами SQL, не являются обязательными.

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

Процедуры вызываются с помощью ключевого слова EXECUTE (EXEC). Использование EXECUTE обязательно, если вызов хранимой процедуры не является первой командой пакета. При вызове процедуры должны передаваться все параметры, для которых не определены значения по умолчанию.

Пример хранимой процедуры, которая возвращает сообщение через один из параметров:

```
CREATE PROCEDURE MyProcedure
    @Name nvarchar(50),
    @SexIsMale bit = 1,
    @Message nvarchar(250) OUTPUT
AS
    IF @SexIsMale = 1
        SET @Message = 'Уважаемый ' + @Name
    ELSE
        SET @Message = 'Уважаемая ' + @Name
GO
```

Пример вызова этой процедуры:

```
DECLARE @Mess nvarchar(250)
EXECUTE MyProcedure 'Иванов Андрей ', DEFAULT, @Mess OUTPUT
PRINT @Mess
```

Для создания, удаления и модификации хранимых процедур можно использовать SQL Server Management Studio. Выберите нужную вам процедуру, щелкните на ней правой кнопкой мыши и выберите пункт **Modify**. После этого Management Studio создаст код ALTER PROCEDURE, который может обновить выбранную процедуру. Пример окна редактирования хранимой процедуры вы можете видеть на рис. 11.1.

Синтаксис вызова хранимой процедуры:

```
[ { EXEC | EXECUTE } ]
   {
     [ @код_возврата = ]
     { имя_процедуры [ ; номер] | @список_параметров }
       [ [ @параметр = ] { значение | @переменная [ OUTPUT ] | [ DEFAULT ] } ]
```

```
    [ ,...n ]
    [ WITH RECOMPILE ]
}
```

Параметры команды:

- *код_возврата* — переменная для сохранения кода возврата процедуры;
- *номер* — числовой идентификатор процедуры. При отсутствии этого аргумента выполняется процедура с максимальным номером версии;
- *список_параметров* — список параметров процедуры. Параметры передаются как по имени (*имя = значение*), так и по позиции в списке. Но если некоторый параметр передается по имени, все последующие параметры в списке тоже должны передаваться по имени. Чтобы пропустить параметр в середине списка, следует передавать остальные параметры по имени;
- `WITH RECOMPILE` — процедура при каждом выполнении заново компилируется.

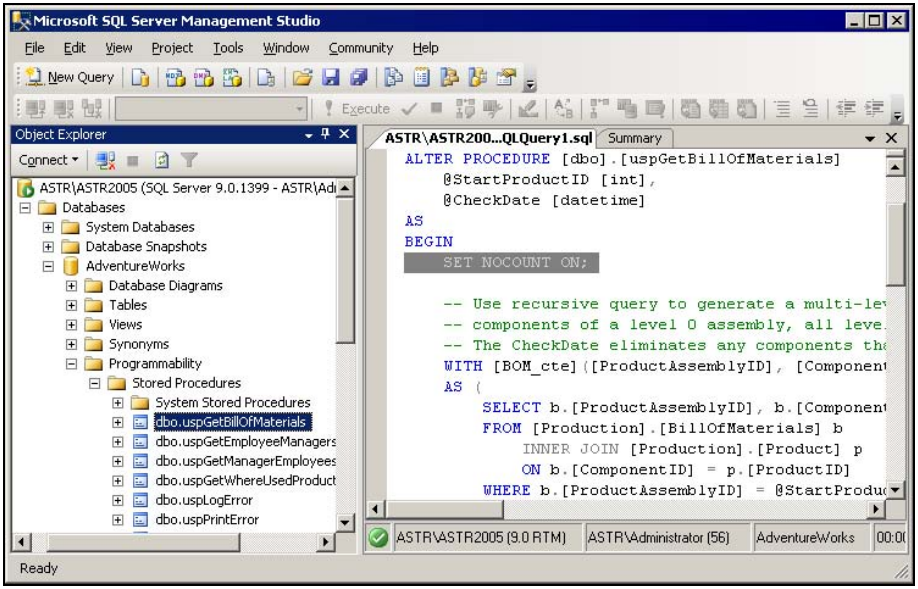


Рис. 11.1. Редактирование хранимой процедуры в SQL Server Management Studio

Вот примеры вызова этой процедуры:

```
EXECUTE MyFirstProcedure 'Иванов Андрей'
EXECUTE MyFirstProcedure 'Иванов Андрей', DEFAULT
EXEC MyFirstProcedure @Name = 'Иванова Мария', @SexIsMale = DEFAULT
```

В хранимой процедуре можно вызывать другие хранимые процедуры. SQL Server допускает до 32 уровней вложенности вызовов хранимых процедур.

Удаление хранимой процедуры

Команда удаления хранимых процедур похожа на команду удаления таблицы или базы данных и имеет следующий синтаксис:

```
DROP PROC имя_процедуры [, имя_процедуры[, ...]]
```

Команда `DROP PROC` может удалять несколько процедур сразу. Для заданного имени процедуры удаляются все процедуры со всеми номерами (вся группа процедур). Нельзя удалить отдельную процедуру из группы.

Изменение хранимой процедуры

Для изменения хранимой процедуры вы можете использовать команду `ALTER PROCEDURE`. У нее следующий синтаксис:

```
ALTER PROCEDURE имя_процедуры [ ; номер ]
    [ { @ параметр тип_данных }
        [ VARYING ] [ = значение ] [ [OUT[PUT]] [ ,...n ]
[ WITH {RECOMPILE|ENCRYPTION}
[ FOR REPLICATION ]
AS
[BEGIN]
    выражения
[END]
```

Команда `ALTER` ведет себя так же, как и `CREATE PROCEDURE`, только вместо слова `CREATE` пишется `ALTER`. Она не изменяет зависимостей или разрешений.

Переименование хранимой процедуры

Хранимые процедуры можно не только модифицировать, но и переименовывать. Переименование осуществляется системной процедурой `sp_rename`. Например:

```
sp_rename старое_имя_процедуры, новое_имя
```

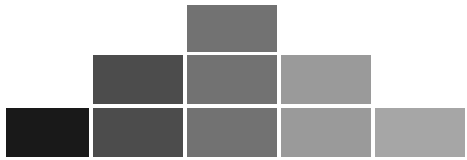
Таблицы как параметры хранимой процедуры

В SQL Server 2008 появилась новая возможность — передавать таблицы в хранимые процедуры через параметры. Передача таблицы через параметры может упростить обработку больших объемов табличных данных в хранимых процедурах.

Работая с параметром-таблицей, вы знаете четко структуру табличных данных (т. к. таблица строго типизирована). Вы можете сортировать данные в таблице, используя `ORDER BY`.

Вот пример использования параметра-таблицы:

```
-- Создание таблицы-типа
CREATE TYPE TBook AS TABLE (id int, Title nvarchar(250), Annotation nvarchar(500));
GO
-- Создание таблиц Books и BooksAnnotation в текущей базе данных
CREATE TABLE Books (id int, Title nvarchar(250));
GO
CREATE TABLE BooksAnnotation(id int, Annotation nvarchar(500));
GO
-- Заполнение таблиц данными
INSERT INTO Books VALUES (1, 'Все обо всем')
INSERT INTO BooksAnnotation VALUES (1, 'Эта книга обо всем на свете')
INSERT INTO Books VALUES (2, 'Удаленная работа')
INSERT INTO BooksAnnotation VALUES (2, 'Работа с сервисом DoWork.ru')
INSERT INTO Books VALUES (3, 'Новинки от Microsoft')
INSERT INTO BooksAnnotation VALUES (3, 'Подробности новых разработок')
GO
-- Создание хранимой процедуры
CREATE Procedure AddBooks (@book TBook READONLY)
AS
INSERT INTO Books
      SELECT id, Title FROM @book
INSERT INTO BooksAnnotation
      SELECT id, Annotation FROM @book
GO

-- Использование
DECLARE @myNewBook TBook
INSERT INTO @myNewBook VALUES (4, 'Жизнеописание великих людей',
                                  'Кащей бессмертный. Монография')
EXEC AddBooks @myNewBook
GO
```



ГЛАВА 12

Функции

Встроенные функции

В SQL Server есть множество встроенных функций, которые помогут вам в самых различных ситуациях. Вы можете использовать их в SQL-выражениях, хранимых процедурах — практически везде.

Пример использования функций:

```
SELECT PI() AS [Число пи]
```

Математические функции

В табл. 12.1 приведен полный перечень математических функций, которые можно использовать в T-SQL.

Таблица 12.1. *Математические функции*

Формат функции	Назначение
abs(*numeric_expression*)	Возвращает абсолютное значение числового выражения
acos(*float_expression*)	Возвращает угол в радианах, косинус которого равен выражению *float_expression*
asin(*float_expression*)	Возвращает угол в радианах, синус которого равен выражению *float_expression*
atan(*float_expression*)	Возвращает угол в радианах, тангенс которого равен выражению *float_expression*
atn2(*float_expression1*, *float_expression2*)	Возвращает угол в радианах, тангенс которого равен частному от деления первого аргумента на второй
ceiling(*numeric_expression*)	Возвращает ближайшее целое число, большее или равное аргументу (округление вверх)

Таблица 12.1 *(окончание)*

Формат функции	Назначение
cos(*float_expression*)	Возвращает значение косинуса для угла в радианах
cot(*float_expression*)	Возвращает значение котангенса для угла в радианах
degrees(*numeric_expression*)	Осуществляет преобразования угла из радиан в градусы
exp(*float_expression*)	Возвращает значение экспоненты
floor(*numeric_expression*)	Возвращает ближайшее целое число, меньшее или равное аргументу (округление вниз)
log(*float_expression*)	Возвращает значение натурального логарифма
log10(*float_expression*)	Возвращает значение десятичного логарифма
pi()	Возвращает значение числа π
power(*numeric_expression*, *y*)	Возводит число в степень — *numeric_expression* в степени *y*
radians(*numeric_expression*)	Возвращает угол в радианах по значению угла в градусах
rand([*seed*])	Возвращает случайное число типа float в промежутке от 0 до 1. *seed* — некоторое целое число. При одном и том же значении *seed* возвращается одно и то же значение функции. Если аргумент отсутствует, то сервер выбирает его случайно
round(*numeric_expression*, *length*[, *function*])	Функция округления, *numeric_expression* — округляемое число, *length* — точность округления. Если *length* > 0, то округление происходит после десятичной точки; *length* < 0, округление происходит до десятичной точки. Если аргумент *function* отсутствует или равен 0, то происходит обычное округление. Если этот аргумент отличен от нуля, то происходит отбрасывание соответствующих разрядов (truncation)
sign(*numeric_expression*)	Возвращает знак выражения
sin(*float_expression*)	Возвращает значение синуса для угла в радианах
sqrt(*float_expression*)	Возвращает квадратный корень аргумента
square(*float_expression*)	Возвращает квадрат выражения
tan(*float_expression*)	Возвращает значение тангенса для угла в радианах

Строковые функции

В табл. 12.2 перечислены строковые функции.

Таблица 12.2. *Строковые функции*

Формат функции	Назначение
`ascii(`*`character_expression`*`)`	Возвращает код ASCII первого символа строки
`char(`*`integer_expression`*`)`	Символ по ASCII-коду
`charindex(`*`expression1`*`, `*`expression2`*`[, `*`start_location`*`])`	Осуществляет поиск подстроки *`expression1`* в строке *`expression2`* и возвращает порядковый номер символа, с которого начинается первое вхождение, *`start_location`* — порядковый номер символа, с которого следует начать поиск. По умолчанию поиск начинается с первого символа. Нумерация символов в строке начинается с единицы
`difference(`*`character_expression1`*`, `*`character_expression2`*`)`	Возвращает целое число в диапазоне от 0 до 4, по которому можно судить о совпадении звучания двух строк. Чем выше число, тем ближе по звучанию слова
`left(`*`character_expression`*`, `*`integer_expression`*`)`	Возвращает *`integer_expression`* первых символов строки *`character_expression`*
`len(`*`string_expression`*`)`	Возвращает длину строки
`lower(`*`character_expression`*`)`	Переводит все символы строки в нижний регистр
`ltrim(`*`character_expression`*`)`	Удаляет все пробелы в начале строки
`nchar(`*`integer_expression`*`)`	Возвращает символ по Unicode-коду
`patindex('%`*`attern`*`%', `*`expression`*`)`	Осуществляет поиск подстроки в строке по шаблону. В шаблоне допускаются такие же символы-заменители, что используются в операторе `LIKE`
`quotename('`*`character_string`*`' [, '`*`quote_character`*`'])`	Заключает строку (первый аргумент) в ограничители. По умолчанию ограничителями являются квадратные скобки. С помощью второго аргумента можно указать другие ограничители. Например: `SELECT quotename('string', '()')` Результат: `(string)`

Таблица 12.2 *(окончание)*

Формат функции	Назначение
replace('*string_expression1*', '*string_expression2*', '*string_expression3*')	Заменяет все вхождения подстроки *string_expression2* в строке *string_expression1* на значение *string_expression3*
replicate(*character_expression*, *integer_expression*)	Осуществляет тиражирование строки *integer_expression* раз
reverse(*character_expression*)	Возвращает строку символов, записанную в обратном порядке
right(*character_expression*, *integer_expression*)	Возвращает *integer_expression* последних символов строки *character_expression*
rtrim(*character_expression*)	Удаляет все пробелы в конце строки
soundex(*character_expression*)	Возвращает четырехзначный код строки для оценки ее звучания
space(*integer_expression*)	Возвращает строку из указанного количества пробелов
str(*float_expression* [, *length*[, *decimal*]])	Конвертирует число в строку. Первый аргумент определяет само конвертируемое число, второй — длину строки, третий — количество знаков после десятичной точки
stuff(*character_expression*, *start*, *length*, *new_character_expression*)	Удаляет определенное количество символов (*length*), начиная со *start*, и заменяет их новой подстрокой *new_character_expression*
substring(*expression*, *start*, *length*)	Для заданной строки возвращает подстроку, начиная с указанного символа (*start*), указанной длины (*length*)
unicode('*ncharacter_expression*')	Возвращает Unicode-код самого левого символа строки
upper(*character_expression*)	Переводит все символы указанной строки в верхний регистр

Функции управления датой и временем

В табл. 12.3 представлены функции управления датой и временем.

T-SQL может легко преобразовать строку с датой к типу `datetime`. Вы можете спокойно использовать строку типа `'2.1.2006'`, приравнивая ее к переменным типа `datetime`. Следует только помнить, что формат даты может

быть разным. Например, порядок следования компонентов даты может быть другим.

Таблица 12.3. *Функции управления датой и временем*

Формат функции	Назначение
dateadd(*datepart*, *number*, *date*)	Добавляет к дате указанное число дней, месяцев, часов и т. д. Первый аргумент определяет тип добавляемого значения (*number*) и может быть равен: • yy или yyyy — год; • qq или q — квартал; • mm или m — месяц; • dd или d — день; • wk или ww — неделя; • hh — час; • mi или n — минута; • ss или s — секунда; • ms — миллисекунда
datediff(*datepart*, *startdate*, *enddate*)	Возвращает разницу между двумя датами (*startdate* и *enddate*). Первый аргумент имеет такое же назначение, что и для функции *dateadd*
datename(*datepart*, *date*)	Возвращает указанную первым аргументом часть даты в символьном формате
datepart(*datepart*, *date*)	Возвращает указанную первым аргументом часть даты в числовом формате
day(*date*)	Возвращает количество дней для указанной даты
getdate()	Возвращает текущую дату
getutcdate()	Возвращает текущую дату по Гринвичу
month(*date*)	Возвращает количество месяцев для указанной даты
year(*date*)	Возвращает количество лет для указанной даты

Чтобы обезопасить себя от возможных ошибок, нужно использовать команду SET DATEFORMAT, которая явно задает формат даты. Например:

```
SET DATEFORMAT mdy
```

Здесь явно указывается формат даты *'месяц, день, год'*.

Системные функции

В табл. 12.4 представлены основные системные функции.

Таблица 12.4. *Системные функции*

Формат функции	Назначение
app_name()	Возвращает имя приложения, которое установило текущую сессию
cast(*expression* as data *type* [(*length*)])	Осуществляет явное преобразование типов
convert(*data_type* [(*length*)], *expression*[, *style*])	Еще одна функция, которая осуществляет явное преобразование типов
coalesce(*expression*[, ...n])	Возвращает первое в списке значение, отличное от NULL
current_timestamp()	Возвращает текущее значение даты и времени. Эквивалентна функции getdate()
current_user()	Возвращает имя текущего пользователя и аналогична функции user_name()
datalength(*expression*)	Возвращает количество байтов, необходимых для хранения данного выражения
host_id()	Возвращает идентификационный номер компьютера, на котором выполняется команда
host_name()	Возвращает имя компьютера, на котором выполняется команда
ident_current(*table_name*)	Возвращает последнее значение, которое было присвоено столбцу-идентификатору в текущем соединении
ident_inc(*table_name*)	Возвращает текущее значение инкремента для указанной таблицы
Ident_seed(*table_name*)	Указывает начальное значение столбца identity для заданной таблицы
@@identity	Возвращает последнее значение, которое было присвоено столбцу-идентификатору
isdate(*expression*)	Проверяет правильность формата даты
isnull(*check_expression*, *replacement_value*)	Проверяет выражение, и если выражение принимает значение NULL, то она возвращает replacement_value

Таблица 12.4 *(окончание)*

Формат функции	Назначение
`isnumeric(`*`expression`*`)`	Возвращает значение 1, если аргумент функции имеет числовой тип
`newid()`	Возвращает глобально уникальный идентификатор (GUID)
`nullif(`*`expression1`*`, `*`expression2`*`)`	Возвращает значение `NULL`, если оба аргумента равны. В противном случае возвращает первый аргумент
`parsename('`*`object_name`*`', `*`object_piece`*`)`	Осуществляет разбор полного имени объекта. Первый аргумент — полное имя объекта. Второй аргумент — номер части имени (1 — имя объекта, 2 — имя схемы и т. д.)
`@@rowcount`	Возвращает количество строк, выданных последним запросом
`row_count_big()`	Функция аналогична `@@rowcount`, но возвращает значение типа `bigint`
`scope_identity()`	Возвращает последнее значение идентификатора, которое было присвоено в пределах данного программного модуля
`serverproperty(`*`propertyname`*`)`	Возвращает свойство сервера. Например, `serverproperty('servername')` возвращает имя сервера
`session_user()`	Возвращает имя пользователя, установившего текущее соединение
`update(`*`column`*`)`	Возвращает значение `TRUE`, если данный столбец обновлен. Функция используется в триггерах
`user_name([`*`id`*`])`	Возвращает текущее имя пользователя, аналогична `current_user()`

Функции конфигурирования

В табл. 12.5 перечислены функции конфигурирования.

Таблица 12.5. *Функции конфигурирования*

Формат функции	Назначение
`@@datefirst`	Возвращает текущее значение первого дня недели
`@@dbts`	Выводит текущее значение счетчика `timestamp` для текущей базы данных
`@@langid`	Возвращает идентификатор локального языка, являющегося текущим
`@@language`	Показывает, какой язык является текущим
`@@lock_timeout`	Возвращает количество миллисекунд, которое сервер будет ожидать до выполнения запроса
`@@max_connection`	Выводит максимальное количество подключений, которое одновременно могут установить пользователи (по умолчанию 32 767)
`@@max_precision`	Возвращает максимальное количество знаков, поддерживаемое сервером для типов данных `numeric` и `decimal`
`@@nestlevel`	Возвращает текущий уровень вложенности хранимых процедур
`@@remserver`	Возвращает имя удаленного сервера
`@@servername`	Возвращает имя локального сервера
`@@spid`	Возвращает идентификационный номер текущего процесса
`@@textsize`	Возвращает максимальный размер блока в байтах, который может быть возвращен командой `SELECT` для столбцов типа `text`, `image`, `varchar(`*`max`*`)`, `varbinary(`*`max`*`)`, `nvarchar(`*`max`*`)`
`@@version`	Возвращает информацию о версии сервера, типе процессора, операционной системе

Пользовательские функции

В SQL Server пользователи могут создавать свои собственные *пользовательские функции.* Пользовательская функция (user-defined function) — это функция на T-SQL, определенная пользователем. Функция включает в себя логику, определенную пользователем, и может вызываться в выражениях T-SQL.

В SQL Server имеется несколько типов определяемых пользователем функций:

- `Scalar` — возвращает скалярное значение любого типа данных, поддерживаемого SQL Server, за исключением `timestamp`, `text`, `ntext`, `image`, `table` и `cursor`;
- `Inline` — всегда возвращает значения типа `table` и состоит из одной команды `SELECT`. Особенностью этого типа функции является то, что код функции при выполнении вставляется непосредственно в исполняемый набор команд, т. е. происходит не вызов функции, а ее встраивание;
- `Multi-Statement` — тоже возвращает значение типа `table`. Однако функции этого типа могут состоять из нескольких команд. Вы можете использовать в функции транзакции, курсоры, хранимые процедуры и т. д.

Функции, возвращающие значения типа `table` (`Inline` и `Multi-Statement`), могут вызываться непосредственно в разделе `FROM` запросов `SELECT`, `INSERT`, `DELETE` и `UPDATE`.

Простой пример пользовательской функции:

```
CREATE FUNCTION MyFirstFunction (
    @FirstName varchar(50), @LastName varchar(50)
)
RETURNS varchar(101)
BEGIN
    RETURN @FirstName + ' ' + @LastName;
END
```

А вызвать ее можно так:

```
SELECT MyFirstFunction(AuthorFirstName, AuthorLastName) AS Name
FROM Authors
```

В этом примере в команде `SELECT` вызывается функция `MyFirstFunction`, в которую передаются имя и фамилия автора, а функция возвращает имя и фамилию как одну строку.

Пользовательские функции напоминают хранимые процедуры. И у тех, и у других есть входные параметры. При этом входные параметры могут иметь любой из типов данных, поддерживаемых SQL Server, за исключением `image`, `text`, `ntext`, `cursor` и `table`. Входные параметры в пользовательской функции нельзя применять для возврата значений, т. е. нельзя использовать ключевое слово `OUTPUT`. Но вы можете определять значения по умолчанию.

Функция должна в обязательном порядке явно вернуть какое-то значение. В функции не разрешается использование команд, которые могут вернуть значения непосредственно в соединение, а не как результат вычисления

функции. Следовательно, в теле функции не допускается вызов команды PRINT, а также вариант команды SELECT, предназначенный для вывода данных.

В функциях не разрешаются изменения данных в таблицах базы данных, создание, модификация и удаление объектов базы данных. Но можно использовать команды UPDATE, INSERT и DELETE, изменяющие данные в переменной типа table, которая является возвращаемым функцией значением.

Создание определяемых пользователем функций выполняется с помощью команды CREATE FUNCTION. Но для создания функций разных типов команды пишутся по-разному.

Функции *Scalar*

Синтаксис команды для создания функций типа Scalar:

```
CREATE FUNCTION имя_функции
( [{@имя_параметра [ AS ] тип_параметра [=значение_по_умолчанию] }
  [ ,...n ] ])
RETURNS тип_данных
    [ WITH [ ENCRYPTION ] | [SCHEMABINDING] |
           [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]]
    [ AS ]
BEGIN
    SQL-код
    RETURN скалярное_выражение
END
```

Назначение и использование параметров команды:

- *имя_функции* — данный параметр предназначен для указания имени функции, а при необходимости дополнительно и владельца (owner_name) создаваемой функции. Если имя функции или владельца содержит неразрешенные символы, то следует использовать ограничители (двойные кавычки или квадратные скобки). Указание имени базы данных или сервера не разрешается, т. е. функция может быть создана только в текущей базе данных;
- @*имя_параметра* [AS] *тип_параметра* [=*значение_по_умолчанию*] — в этой конструкции определяются входные параметры создаваемой функции. Это делается так же, как и в хранимой процедуре. Параметры должны начинаться с символа @ и быть уникальными в пределах функции. Задается имя параметра, его тип и, если необходимо, значение по умолчанию. Не допускается применение типов данных timestamp, text, ntext, image, table и cursor;

- RETURNS *тип_данных* — тип данных, который возвращает функция;
- WITH ENCRYPTION — при наличии данного параметра SQL-код функции хранится в базе в зашифрованном виде;
- WITH SCHEMABINDING — при наличии данного параметра сервер выполняет связывание создаваемой функции со всеми объектами, на которые ссылаются команды функции. Это позволяет избежать модификации объектов базы данных, которая может привести к нарушению работы функции (например, удаление столбца таблицы, из которого осуществлялась выборка данных). При этом имена объектов в функции должны состоять из собственного имени объекта и имени схемы;
- WITH RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT — этот параметр задает реакцию на посылку в качестве параметра значения NULL. По умолчанию установлен параметр CALLED ON NULL INPUT, который говорит о том, что параметр со значением NULL будет принят и функция будет выполняться. Параметр RETURNS NULL ON NULL INPUT означает, что если хоть один параметр функции будет NULL, то функция выполняться не станет, и будет возвращено значение NULL;
- BEGIN...END — используются для объединения команд, которые будут являться телом функции. Все команды, являющиеся телом функции, должны находиться между ключевыми словами BEGIN...END;
- RETURN *скалярное_выражение* — когда в теле функции встречается команда RETURN, то завершается выполнение функции и возвращается значение, полученное в результате вычисления выражения *скалярное_выражение*. Этот результат должен иметь тип данных, указанный после ключевого слова RETURNS. В отличие от хранимых процедур, не требующих обязательного использования команды RETURN, при работе с функцией выход из нее происходит только при выполнении команды RETURN.

Последняя команда тела функции всегда должна быть RETURN, даже если гарантированно до достижения конца функции будет выполнена другая команда RETURN.

Вот пример Scalar-функции:

```
CREATE FUNCTION MyFunction (
    @Name nvarchar(100), @IsSexMale bit
)
RETURNS nvarchar(200)
BEGIN
    IF @IsSexMale = 1
```

```
        RETURN 'Уважаемый ' + @Name;
    RETURN 'Уважаемая ' + @Name;
END
```

Эта функция возвращает вежливое обращение к человеку в зависимости от его пола.

Функции *Inline*

Синтаксис команды для создания `Inline`-функций:

```
CREATE FUNCTION имя_функции
  ( [ {@имя_параметра [AS] тип_параметра [=значение_по_умолчанию] }
    [, ...n ] ] )
RETURNS table
    [ WITH [ ENCRYPTION ] | [SCHEMABINDING] |
           [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]]
    [ AS ]
    RETURN SELECT_выражение
```

Определение функций `Inline` практически полностью аналогично определению функций `Scalar`, рассмотренных ранее.

Отличия: явно указан тип значения, возвращаемого функцией — `table`, и после ключевого слова `AS` не допускается указание конструкции `BEGIN...END`, а разрешается лишь использование запроса `SELECT`, с помощью которого будет сформирован набор данных, возвращаемый функцией.

В этом запросе можно использовать параметры функции.

Пример `Inline`-функции:

```
CREATE FUNCTION MyFunction (@Char char)
RETURNS table
RETURN
    SELECT FirstName, LastName, Address
      FROM Author
      WHERE LEFT(LastName, 1) = @Char;
```

Эта функция возвращает список авторов, чье имя начинается с определенной буквы.

Функции *Multi-Statement*

Синтаксис команды для создания `Multi-Statement`-функции:

```
CREATE FUNCTION имя_функции
  ( [{@имя_параметра [ AS ] тип_параметра [=значение_по_умолчанию] }
```

```
    [, ...n ] ])
RETURNS @возвращаемая_переменная TABLE <определение_типа_таблицы>
    [ WITH [ ENCRYPTION ] | [SCHEMABINDING] |
           [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]]
    [ AS ]
BEGIN
    SQL-код
    RETURN
END
```

Определение функций `Multi-Statement` также практически полностью аналогично определению функций `Scalar` и `Inline`, рассмотренных ранее. Но есть и отличия. Самое заметное отличие — в определении типа возвращаемого значения. В отличие от функций `Inline` и `Scalar`, здесь необходимо указать имя локальной переменной, содержимое которой будет возвращено как результат выполнения функции. Эта переменная имеет тип `table`.

При этом необходимо явно определить набор столбцов, которые будут применяться для хранения данных. Дополнительно можно определить индексы, ограничения целостности и т. д.

Еще одно отличие связано с завершением работы функции. Завершение работы функции происходит при выполнении команды `RETURN`, однако не требуется указание значения, которое будет возвращено как результат выполнения функции. Всегда возвращается содержимое переменной типа `table`, указанной после ключевого слова `RETURNS`.

Изменение функций

Может возникнуть необходимость внести в уже созданную функцию некоторые изменения. Для этого вы можете использовать команду `ALTER FUNCTION`.

Параметры и синтаксис команды `ALTER FUNCTION` аналогичны команде `CREATE FUNCTION`, требуется лишь заменить слово `CREATE` на `ALTER`.

Пример команды для обновления функции:

```
ALTER FUNCTION MyFunction (@Char char)
RETURNS table
RETURN
    SELECT FirstName, LastName, Address, Phone, NumberBooks
      FROM Author
      WHERE LEFT(LastName, 1) = @Char;
```

Удаление функций

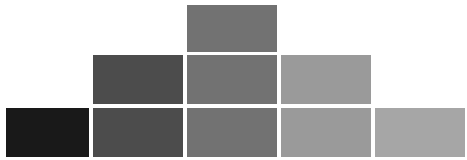
Для удаления функции используется команда DROP FUNCTION, имеющая синтаксис:

```
DROP FUNCTION { [имя_схемы.]имя_функции} [, ...n]
```

С помощью одной команды можно удалить множество функций, просто перечислив их имена через запятую.

Пример использования команды:

```
DROP FUNCTION MyFunction, MyFirstFunction
```



ГЛАВА 13

Транзакции

Транзакция представляет собой одну или несколько команд SQL, которые либо успешно завершаются, либо отменяются как единое целое.

Транзакции являются одной из самых важных концепций, необходимой для построения надежных баз данных. Именно транзакции обеспечивают целостность данных в любых ситуациях.

Транзакция является логической единицей работы и обладающей следующими свойствами:

- *атомарность* — все изменения данных, выполненные в транзакции, рассматриваются как единый блок. То есть все изменения, сделанные в блоке команд, либо принимаются, либо отклоняются целиком;
- *согласованность* — после успешного завершения транзакции все данные должны удовлетворять всем ограничениям целостности, определенным в базе;
- *изолированность* — две транзакции могут работать с одними и теми же данными независимо от изменений, вносимых другой транзакцией;
- *устойчивость* — после фиксации изменений, сделанных транзакцией, данные не могут быть возвращены в состояние, в котором они были до начала транзакции. Они могут быть лишь изменены другой транзакцией.

Примером транзакции является пересылка денег с одного счета на другой. Предположим, вам нужно перевести 5000 руб. с вашего счета на счет вашего друга Васи. Команда SQL, которая делает это, должна зарегистрировать операцию, вычесть 5000 руб. с вашего счета и зачислить их на счет вашего друга. То есть здесь 3 отдельные операции.

Но что будет, если по ходу выполнения этих операций произойдет сбой питания? Допустим, это произойдет после второй операции. Деньги с вашего счета уже сняты. Но на счет вашего друга они не поступили. И не поступят. Они пропали в дебрях банковской системы.

Решение для этой проблемы — объединить все эти 3 команды в одну транзакцию. Это будет гарантировать, что все три команды либо успешно завершатся, либо не завершится ни одна из них.

Вот другой пример полезности транзакций. Предположим, вы выполняете команду UPDATE, обновляющую несколько записей. В процессе обновления происходит сбой (например, нарушение питания или разрыв соединения с сервером). Что произойдет в тот момент, когда данные снова станут доступными? Завершится ли операция обновления? Нет. Произойдет откат.

По умолчанию любая команда SQL представляет собой транзакцию. Кроме того, несколько команд SQL также могут быть объединены в одну транзакцию.

Команды управления транзакциями

Команды управления транзакциями определяют логические единицы выполняемой работы. В табл. 13.1 перечислены команды управлении транзакциями SQL Server и приведены их краткие описания.

Таблица 13.1. *Команды управления транзакциями и их применение*

Команда	Описание
BEGIN TRAN[SACTION] [*имя_транзакции* | @*переменная*]	Определяет начало транзакции
COMMIT TRAN[SACTION] [*имя_транзакции* | @*переменная*]	Информирует сервер о завершении текущей транзакции и делает изменения данных 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_транзакции* | @*переменная*	Регистрирует промежуточную точку, к которой может выполняться откат (обеспечивает возможность частичного отката). Транзакция остается активной

Приведем пример транзакции, демонстрирующий применение BEGIN TRAN... COMMIT TRAN для объединения нескольких команд SQL в одну транзакцию:

```
BEGIN TRAN
    IF (SELECT Sum FROM Account WHERE AccountNumber=123456) >= 5000
    BEGIN
        ROLLBACK TRAN
        RETURN
    END

    UPDATE Account SET Sum = Sum - 5000 WHERE AccountNumber = 123456
    UPDATE Account SET Sum = Sum + 5000 WHERE AccountNumber = 547891

    IF @@error != 0
    BEGIN
        ROLLBACK TRAN
        RETURN
    END
COMMIT TRAN
```

В этом примере команда BEGIN TRAN сообщает серверу о начале транзакции. Это означает, что до получения сервером команды завершения транзакции (COMMIT TRAN) все изменения являются временными. Следовательно, если на сервере произойдет сбой после первого обновления, выполнится откат транзакции.

Термин *"откат"* означает, что все последствия транзакции будут отменены, и данные на сервере будут выглядеть так, словно никаких изменений не было.

Чтобы обеспечить согласованность транзакций, в этом примере блокируется модификация информации до завершения транзакции. Термин *"блокировка"* (lock) означает, что никакой процесс не сможет обратиться к данным до их освобождения (посредством закрепления или отката транзакции). Только процесс, являющийся владельцем блокировки, сможет просмотреть заблокированные данные.

Если во время транзакции произойдут ошибки (или если этого требует логика программы), выполняется команда ROLLBACK TRAN.

Команды управления транзакциями не влияют на последовательность выполнения программы.

> ***Примечание***
>
> Команда ROLLBACK TRAN не влияет на последовательность выполнения команд программы. Следовательно, после отката необходимо выполнить команду RETURN.

Последовательность выполнения транзакций

Рассмотрим пример из листинга 13.1, которым иллюстрируется последовательность выполнения транзакций.

Листинг 13.1. Последовательность выполнения транзакций

```
BEGIN TRAN first_tran
    INSERT MyTable VALUES (1)
    PRINT @@TRANCOUNT     -- значение 1
SAVE TRAN Point_one

INSERT MyTable VALUES (2) -- любой SQL-код

IF @@error != 0
    -- Откат к точке Point_one, если была хоть одна ошибка
    ROLLBACK TRAN Point_one
ELSE IF (SELECT COUNT(*) FROM MyTable) > 3
    -- если меньше 3 циклов, то откат к началу транзакции
    ROLLBACK TRAN first_tran

INSERT MyTable VALUES (3) -- любой SQL-код

IF @@error != 0
    ROLLBACK TRAN      -- если была ошибка, то откат к началу транзакции
COMMIT TRAN first_tran
```

Сервер запоминает факт начала транзакции, обновляя глобальную переменную `@@TRANCOUNT`. Вначале переменная `@@TRANCOUNT` равна 0. Она увеличивается на 1 для каждой новой вложенной транзакции.

После открытия транзакции добавляется запись в таблицу `MyTable`. Добавленная запись находится в таблице и может участвовать в запросах со стороны процесса, открывшего транзакцию, но будет недоступна со стороны всех остальных процессов до закрепления транзакции.

Команда `SAVE TRAN` используется в качестве "закладки". Она не влияет на значение `@@TRANCOUNT` и не изменяет уровня вложенности транзакции, а лишь позволяет позднее вернуть транзакцию к этой определенной точке.

Обратите внимание: откаты, закрепления и команды BEGIN TRAN не обязаны образовывать пары, аналогичные парам BEGIN...END. Каждая транзакция завершается откатом или закреплением, однако программа может содержать несколько команд отклонения или закрепления.

Независимо от количества вложенных команд BEGIN TRAN вложенность является лишь синтаксической. Другими словами, действительно активной является лишь одна транзакция. Эта транзакция закрепляется в базе данных только командой COMMIT самого верхнего уровня. Это похоже на матрешку, но для закрепления всех результатов матрешки вы должны вызвать COMMIT для той матрешки-транзакции, которая включает в себя все остальные.

Каждое закрепление транзакции уменьшает значение @@TRANCOUNT, а неуточненная команда ROLLBACK вызывает откат к самой внешней транзакции.

Если глобальная переменная @@TRANCOUNT так и не возвращается к нулевому значению, транзакция считается незавершенной и откатывается.

> ***Примечание***
>
> Настоятельно рекомендуется, чтобы транзакция содержала как можно меньше команд и заканчивалась как можно быстрее. Это очень важно с точки зрения производительности.

Ограничения транзакций

Тем не менее, существуют некоторые действия, которые не могут выполняться в транзакциях. Например, транзакция относится к конкретному серверу (и не может распространяться на несколько серверов).

Некоторые команды не могут использоваться в транзакциях. Их список:

- ALTER/CREATE/DROP DATABASE;
- DUMP/LOAD DATABASE/TRANSACTION;
- UPDATE STATISTICS.

Распределенные транзакции (транзакции, распространяющиеся на две и более базы данных) обладают собственным набором правил и ограничений.

Распределенные транзакции

Распределенной транзакцией называется транзакция, распространяющаяся на две и более базы данных.

Существуют дополнительные команды управления распределенными транзакциями:

```
BEGIN DISTRIBUTED TRAN[SACTION] [имя_транзакции | переменная]
```

Распределенные транзакции закрепляются командой `COMMIT TRAN`, завершающей любую другую транзакцию.

Команда закрепления или отката распределенной транзакции выдается тем же сервером, на котором была инициирована транзакция.

На удаленных серверах распределенная транзакция может вызывать лишь хранимые процедуры, в то время как на сервере, инициирующем транзакцию, можно выполнять как хранимые процедуры, так и SQL-код.

Блокировки в транзакциях

Блокировка объектов обеспечивает согласованность данных. Пока один процесс изменяет данные, другой процесс не должен работать с ними. Блокировка данных сохраняется до завершения транзакции. Для разных ситуаций предназначены разные уровни блокировки.

Уровни изоляции

Уровень изоляции — это параметр, определяющий степень блокировки данных в SQL Server. Стандарт языка SQL ANSI-92 определяет четыре уровня изоляции транзакций SQL. Каждый уровень изоляции описывает типы действий, выполнение которых для данного уровня запрещено. Далее описаны все уровни изоляции ANSI. Высокие уровни включают в себя более низкие.

Уровень изоляции 0

Уровень 0 допускает *"грязное" чтение*. "Грязным" чтением называется чтение данных, которые были изменены, но не закреплены в базе данных.

На этом уровне обеспечивается изолированность изменений данных транзакциями. Одни и те же данные в каждый момент времени может изменять только одна транзакция. Если какая-то другая транзакция пытается изменить те же данные, то она должна ожидать завершения работы первой транзакции. Только после этого разрешается изменять данные. Но другие транзакции могут читать измененные, но не закрепленные данные в базе данных.

Уровень 0 реализуется командой `NOLOCK` или параметром `READUNCOMMITED` команд `SELECT` и `SET`.

Уровень изоляции 1

Уровень 1 является уровнем блокировки по умолчанию. Он предотвращает "грязное чтение", т. е. чтение данных, которыми пользуются другие тразакции.

На этом уровне изоляции, до тех пор, пока транзакция не будет зафиксирована или отменена, данные нельзя будет прочитать. Транзакции, подавшие запрос на чтение данных, должны будут ожидать разблокирования ресурсов.

Уровень 1 реализуется командой `SET LOCKIMPLEMENTS` или параметром `READ COMMITED` команд `SELECT` и `SET`.

Уровень изоляции 2

Уровень 2 запрещает неповторяемые операции чтения. Таким образом, другая транзакция не сможет изменить запись, которая в настоящее время читается текущей транзакцией.

Уровень 2 реализуется параметром `REPEATABLE` команд `SELECT` и `SET`.

Уровень изоляции 3

Уровень 3, последний уровень изоляции, запрещает чтение *фантомных записей*. Фантомная запись — это запись, которой на самом деле уже нет в базе данных, или наоборот, запись уже есть, но в запросе выбора данных ее еще нет.

Этот уровень изоляции предотвращает ситуацию, при которой набор выбранных записей отличается от набора, прочитанного ранее в этой транзакции с тем же критерием выборки.

Уровень 3 реализуется параметрами `HOLDLOCK` и `SERIALIZABLE` команд `SELECT` и `SET`.

Установка уровня изоляции

По умолчанию в SQL Server используется уровень изоляции 1. Но вы можете изменить уровень изоляции, используемый по умолчанию для сеанса, с помощью команды `SET`:

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED |
        READ UNCOMMITED | REPEATABLE READ | SERIALIZABLE}
```

При установке значения `SERIALIZABLE` ко всем командам `SELECT` транзакции автоматически применяется параметр `HOLDLOCK`.

Уровень изоляции транзакции продолжает действовать на протяжении всего сеанса или до установки нового уровня.

Пример установки уровня изоляции:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE
```

Мертвые блокировки

Когда с данными одновременно работают несколько пользователей, возможна ситуация, когда две или более транзакции обращаются к одному и тому же набору данных.

Может случиться следующее: есть две транзакции (A и B), которые обращаются к двум ресурсам (1 и 2). Сначала транзакция A обращается к ресурсу 1, а транзакция B — к ресурсу 2. При этом каждая из транзакций устанавливает блокировку на используемый ресурс. На следующем этапе транзакция A обращается к ресурсу 2, который уже блокирован транзакцией B. Транзакция A приостанавливается и ожидает снятия блокировки. В это время транзакция B обращается к ресурсу 1, который блокирован транзакцией A. Это приводит к тому, что транзакция B станет ожидать снятия блокировки с ресурса 1, а транзакция A — с ресурса 2.

Таким образом, каждая из транзакций ожидает разблокирования ресурсов, используемых противоположной транзакцией. Но ни одна из транзакций не может продолжаться, т. к. необходимо разблокирование ресурсов, используемых другой транзакцией.

Подобная ситуация называется *мертвой блокировкой*. Возникновение мертвой блокировки невозможно при выполнении операций чтения данных. Обычно мертвые блокировки возникают при попытке изменения одних и тех же данных несколькими транзакциями.

Мертвая блокировка возникает, когда две или более транзакций пытаются обратиться к ресурсам, блокированным ранее другими транзакциями, участницами мертвой блокировки.

Могут возникнуть мертвые блокировки, в которые втянуто множество транзакций, блокирующих множество ресурсов.

Мертвая блокировка не может разрешиться сама собой, и если не предпринимать каких-либо действий, она способна существовать бесконечно долго.

Одним из механизмов, позволяющих разрешать конфликты мертвых блокировок, является ограничение времени ожидания транзакцией разблокирования ресурса.

Например, если транзакция пытается обратиться к блокированным ресурсам, то можно установить максимальный период ожидания, в течение которого транзакция будет ожидать разблокирования требуемых ресурсов. По истечении этого времени транзакция получит соответствующее сообщение об ошибке. Однако при этом не произойдет установки мертвой блокировки.

Для установки времени ожидания разблокирования ресурсов используется следующая команда:

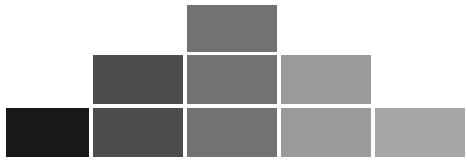
```
SET LOCKTIMEOUT время_ожидания
```

Время ожидания задается в миллисекундах. Например:

```
SET LOCKTIMEOUT 1800
```

Период ожидания разблокирования ресурсов задается на уровне соединения. По умолчанию устанавливается период ожидания, равный −1, что соответствует бесконечному ожиданию.

Максимальное время ожидания установки блокировки — лишь механизм, позволяющий на уровне соединения избежать возникновения мертвых блокировок. Но на уровне сервера остается высокая вероятность возникновения мертвых блокировок, т. к. не для всех соединений может быть назначен период ожидания разблокирования ресурса.



ГЛАВА 14

Триггеры

Триггеры являются особой разновидностью хранимых процедур, выполняемых автоматически и срабатывающих при модификации данных таблицы. Триггеры можно применять в разных целях, от проверки данных до обеспечения сложных деловых правил.

Триггеры имеют доступ к версии записи до и после модификации и позволяют откатить сделанные изменения. Таким образом, вы можете сравнивать две записи.

Триггеры используются для различных целей, в том числе:

- обеспечение ссылочной целостности, особенно когда ее критерии слишком сложны для определения на уровне таблиц;
- обеспечение проверки правильности значений атрибутов. При необходимости триггеры могут использоваться для проверки значений в других таблицах и базах данных, а также для сравнения образов данных до и после обновления;
- обеспечение сложных деловых правил. Например, автоматическое увеличение количества текущих заказов в таблице, где хранятся агрегированные данные;
- выбор сложных значений по умолчанию. Триггеры позволяют принимать решения при выборе значения по умолчанию для столбцов. Полезная возможность, учитывая, что обычно допускается лишь одно значение по умолчанию для каждого столбца.

Срабатывание триггеров

Триггеры закрепляются за таблицей и срабатывают в ответ на выполнение команды, модифицирующей запись. Поскольку триггер является частью исходной транзакции, он может вызвать откат отдельной команды или всей

транзакции. Благодаря этому у вас появляется возможность контролировать изменение данных в таблице.

При модификации данных в таблице с триггером происходит следующее:

1. Сначала происходит проверка всех ограничений. Если ограничение не нарушено (или не существует), запись изменяется или удаляется.
2. Затем, перед завершением транзакции, выполняется триггер, который принимает решение об откате или закреплении транзакции.

Триггеры выполняются только один раз для одной команды модификации данных, независимо от количества обрабатываемых записей. То есть триггер срабатывает, даже если команда не влияет ни на одну запись или обрабатывает несколько записей таблицы.

Создание триггеров

Таблица может содержать триггеры вставки, обновления и удаления или любую комбинацию из этих трех типов. Количество триггеров в таблице не ограничено.

Синтаксис команды создания триггера:

```
CREATE TRIGGER имя_триггера ON { имя_таблицы | имя_представления }
[ WITH ENCRYPTION ]
{ FOR | AFTER | INSTEAD OF }
{ [INSERT] [,] [UPDATE] [,] [DELETE] }
AS { SQL_выражение }
```

Параметры в команде означают следующее:

- `INSERT`, `UPDATE`, `DELETE` — определяют, когда срабатывает триггер. При создании триггера можно определить, срабатывает ли триггер при вставке, обновлении или удалении записи. При выборе нескольких операций они разделяются запятыми (`INSERT, UPDATE, DELETE`);
- `WITH ENCRYPTION` — при наличии данного параметра SQL-код функции хранится в базе в зашифрованном виде;
- `FOR | AFTER` — вы можете использовать любое из этих ключевых слов, они означают одно и то же;
- `INSTEAD OF` — триггеры этого типа выполняются вместо пользовательских действий, т. е. запрос пользователя не выполняется, а вносятся лишь изменения, осуществляемые в теле триггера.

Пример триггера, который выводит сообщение при каждой вставке записи в таблицу `Books`:

```
CREATE TRIGGER myFirstTrigger ON Books FOR INSERT
AS
    PRINT "Добавлена новая книга"
RETURN
```

Команда `CREATE TRIGGER` должна быть первой командой в пакете. Для одной таблицы может определяться несколько триггеров.

Хотя триггеры во многом похожи на хранимые процедуры, они не получают параметров при вызове и не могут явно вызываться или выполняться из программы. Триггеры не могут создаваться для представлений, только для таблиц.

В триггерах запрещены следующие команды SQL:

- любые команды `CREATE`;
- любые команды `DROP`;
- `ALTER TABLE/DATABASE`;
- `DENY`, `GRANT`;
- `REVOKE`;
- `SELECT INTO`;
- `TRUNCATE TABLE`;
- `UPDATE STATISTICS`;
- `RECONFIGURE`;
- `LOAD/RESTORE DATABASE/TRANSACTION`.

Удаление триггеров

Синтаксис команды для удаления `DROP TRIGGER`:

```
DROP TRIGGER имя_триггера [, имя_триггера [, ...]]
```

При удалении таблицы удаляются также все связанные с ней триггеры.

Модификация триггеров

Изменение триггеров осуществляется командой `ALTER TRIGGER`. Команда похожа на команду `CREATE TRIGGER`, просто вместо слова `CREATE` используется `ALTER`.

Таблицы *deleted* и *inserted*

В таблицах `deleted` и `inserted` отражаются изменения, внесенные в таблицу командой, которая привела к срабатыванию триггера. Эти таблицы доступны лишь во время срабатывания триггера и только в пределах триггера.

Структура таблиц `inserted` и `deleted` в точности совпадает со структурой таблицы, к которой привязан триггер. Таблица `inserted` содержит новые записи, созданные командой `INSERT` или `UPDATE`, а таблица `deleted` содержит записи, удаленные при выполнении команд `DELETE` или `UPDATE`. SQL Server интерпретирует команду `UPDATE` как команду `DELETE` с последующей командой `INSERT`.

Все вставленные и удаленные записи включаются в таблицы `inserted` и `deleted`. Эти таблицы видны только внутри триггера, для которого они создавались. Триггер может запретить удаление или вставку записи.

Вот пример, демонстрирующий применение таблицы `deleted`. В этом примере из таблицы `Books` удаляется определенная запись. В таблице `Books` есть три колонки: `BookID`, `Name`, `Price`. Триггер обращается к таблице `deleted` и просматривает удаляемую информацию. Триггер может запретить удаление записи.

Таблица `Books` до выполнения команды:

```
BookID Name                                      Price
------ ----------------------------------------- -----------
1      Война и мир                               350
2      Разработка на C++                         250
3      Веб-дизайн для чайников                   300
```

Выполняется команда:

```
DELETE FROM Books WHERE BookID = 1
```

Далее следует содержимое таблиц в триггере.

Таблица `Books`:

```
BookID Name                                      Price
------ ----------------------------------------- -----------
2      Разработка на C++                         250
3      Веб-дизайн для чайников                   300
```

Таблица `inserted`:

```
BookID Name                                      Price
------ ----------------------------------------- -----------
1      Война и мир                               350
```

Таблица `deleted`:

```
BookID Name                                      Price
------ ----------------------------------------- -----------
```

А вот что получается, когда выполняется команда `UPDATE`.

Выполняется команда:

```
UPDATE Books SET Price = 450 WHERE BookID = 2
```

Далее следует содержимое таблиц в триггере.

Таблица `Books`:

```
BookID Name                                      Price
------ ----------------------------------------- -----------
1      Война и мир                               350
2      Разработка на C++                         450
3      Веб-дизайн для чайников                   300
```

Таблица `inserted`:

```
BookID Name                                      Price
------ ----------------------------------------- -----------
2      Разработка на C++                         450
```

Таблица `deleted`:

```
BookID Name                                      Price
------ ----------------------------------------- -----------
2      Разработка на C++                         250
```

Триггеру доступна как сама основная таблица с уже внесенными изменениями, так и таблицы `inserted` и `deleted`, с помощью которых можно узнать, какие именно изменения были осуществлены.

Как видите, в примере, при удалении записи из таблицы, таблица `inserted` остается пустой, а в таблице `deleted` находятся удаленные записи. При обновлении записи в таблице `inserted` будет уже обновленная версия записи, а в таблице `deleted` ее старая версия. А при вставке записи таблица `deleted` будет пустой, а в таблице `inserted` будут вставленные записи.

Если команда затрагивает несколько записей, то в таблицах `inserted` и `deleted` будут все записи, обработка которых затрагивалась данной командой.

Проверка столбцов при модификации

При срабатывании триггера часто бывает необходимо узнать, обновлялась ли та или иная колонка. Конструкция `IF UPDATE` позволяет определить, выполнялась ли вставка или обновление для конкретного столбца.

Вот пример использования этой конструкции:

```
CREATE TRIGGER myFirstTrigger ON Books FOR INSERT, UPDATE, DELETE
AS
IF UPDATE(Price)
    PRINT "Обновлена цена книги " + SELECT Name FROM inserted
IF UPDATE(Name)
    PRINT "Обновлено название книги " + SELECT Name FROM inserted
RETURN
```

Результат проверки `IF UPDATE` равен `TRUE`, если в столбец было вставлено значение, отличное от `NULL`, или если столбец был включен в секцию `SET` команды `UPDATE`.

Триггер может отменить действие команды, вызвав откат транзакции с помощью команды `ROLLBACK TRAN`.

Вот пример триггера, который не допускает, чтобы цена книги была меньше 100 рублей:

```
CREATE TRIGGER myTrigger ON Books FOR INSERT, UPDATE, DELETE
AS
IF UPDATE(Price) AND
   EXISTS(SELECT * FROM inserted WHERE price < 100)
BEGIN
   PRINT "Недопустимое значение цены книги"
   ROLLBACK TRAN;
   RETURN
END
PRINT "Обновление цены книги прошло успешно"
RETURN
```

Триггер является неотъемлемой частью команды, вызвавшей его срабатывание, и транзакции, в которую входит эта команда. Выполнение `ROLLBACK TRAN` в триггере приводит к откату всех операций, начиная с самой внешней команды `BEGIN TRAN`, завершает вычисления в триггере и прерывает текущий пакет.

Очень важно, что в отличие от `ROLLBACK TRAN` в хранимой процедуре, `ROLLBACK TRAN` в триггере отменяет выполнение оставшейся части пакета.

При использовании триггеров в транзакциях необходимо помнить о некоторых нюансах.

- Команды, следующие в триггере за `ROLLBACK TRAN`, будут выполнены. Откат относится к обработке транзакций, а не к потоку команд.

- Не забывайте ставить RETURN после ROLLBACK TRAN в триггере, чтобы предотвратить нежелательные последствия.
- Не включайте в триггер команду BEGIN TRAN — вы уже находитесь в транзакции, поэтому лишняя команда BEGIN TRAN не нужна. В триггере можно установить точку сохранения и вернуться к ней. Откат будет выполнен только для команд триггера, находящихся за точкой сохранения.
- Транзакция остается активной до последующего закрепления или отката, а выполнение пакета продолжится.
- Не выполняйте в триггере откат для именованной транзакции. Это приведет к ошибке времени выполнения, откату всей работы и отмене пакета.

Использование точек сохранения в триггерах

Команда SAVEPOINT позволяет избежать отката всей транзакции в триггере. Вы определяете точку сохранения по своему желанию и ограничиваете откат транзакции точкой сохранения с заданным именем. В этом случае следует 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.

Вот пример:

```
CREATE TRIGGER myTrigger ON Books FOR INSERT
AS
SAVE TRAN checkPoint
IF UPDATE(Price) AND
   EXISTS(SELECT * FROM inserted WHERE price < 100)
BEGIN
   ROLLBACK TRAN checkPoint;
   RAISERROR ('Недопустимое значение цены книги', 16, 1)
   RETURN
END
UPDATE Statistic
SET NumberBooks = NumberBooks + 1
   WHERE BookID = (SELECT BookID FROM inserted)
RETURN
```

А вот пример использования этого триггера:

```
BEGIN TRAN myTran
INSERT Books (BookID, Name, Price) VALUES (4, 'Java-2', 50)
IF @@ERROR <> 0
BEGIN
    PRINT 'Ошибка при вставке записи'
    ROLLBACK TRAN myTran
    RETURN
END
INSERT Books (BookID, Name, Price) VALUES (4, 'C#', 250)
IF @@ERROR <> 0
BEGIN
    PRINT 'Ошибка при вставке записи'
    ROLLBACK TRAN myTran
    RETURN
END
COMMIT TRAN
```

В этом примере вставляются записи в таблицу, для которой определен триггер вставки. После каждой вставки проверяется значение `@@ERROR`, чтобы узнать, произошел ли в триггере откат, и если произошел — откатывается вся транзакция.

Вложенные триггеры

Предположим, триггер A выполняет команду `INSERT`, `UPDATE` или `DELETE` для другой таблицы, в которой для выполняемой операции определен триггер B. В результате сработает и триггер B. Это пример вложенных триггеров. Эта модель поведения используется сервером по умолчанию.

Триггеры ограничиваются 32 уровнями вложенности. Если вы полагаете, что ограничение на глубину вложенности триггеров может оказаться существенным для вашего приложения (хотя в большинстве ситуаций такое ограничение не вызывает проблем), проверьте глобальную переменную `@@NESTLEVEL` или вызовите функцию `NEST_LEVEL()`, чтобы избежать превышения.

Для вложенных триггеров недоступно содержимое таблиц `inserted` или `deleted` других триггеров — в них видны только таблицы `inserted` или `deleted` триггера, вызвавшего срабатывание вложенного триггера.

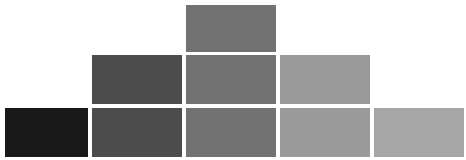
Обычно триггеры не являются рекурсивными. Иначе говоря, при модификации собственной таблицы они не срабатывают, хотя при желании это поведение можно изменить.

Команды модификации данных внутри триггера не влияют на содержимое таблиц `inserted` или `deleted` для данного триггера, т. е. значения в базовой таблице можно заменить без входа в циклическую рекурсию.

Если для некоторого действия в таблице определено сразу несколько триггеров, эти триггеры срабатывают в порядке их определения. Если важен порядок их вызова, то будет проще и разумнее использовать один триггер.

При модификации данных через представление будут вызваны все триггеры для базовых таблиц. Триггеры могут выполнять хранимые процедуры. В триггерах можно использовать курсоры, хотя лучше этого не делать — триггеры выполняются часто, а курсоры работают медленно.

Если вам потребуется вернуть информацию из триггера, используйте команды `PRINT` и `RAISERROR`.



ГЛАВА 15

Полнотекстовый поиск

Если вам необходимо найти записи в таблице со словом "компьютер", вы можете сделать следующий запрос:

```
SELECT * FROM Books WHERE Name LIKE '%компьютер%'
```

Этот запрос работает крайне медленно, поскольку серверу приходится просматривать каждую запись таблицы. Очень медленно.

Стандартный индекс сортирует столбец по возрастанию или убыванию символов. Полнотекстовый поиск создает специальный *полнотекстовый индекс* для упрощения нахождения слова или фразы в текстовых столбцах. Фактически создается индекс слов столбца.

Однако перед тем как использовать полнотекстовый поиск, необходимо выполнить ряд подготовительных действий.

Вам потребуется создать полнотекстовый индекс — определить колонки, которые будут входить в полнотекстовый индекс. В полнотекстовый индекс могут входить колонки типа `char`, `varchar`, `nvarchar`, `text`, `ntext`.

Подготовка к полнотекстовому поиску

В первую очередь, вам необходимо создать полнотекстовый каталог. В SQL Server 2005 и более поздних версиях SQL Server для создания полнотекстовых каталогов используется команда `CREATE FULLTEXT CATALOG`. В предыдущих версиях SQL Server это делалось с помощью других специальных команд.

Синтаксис команды `CREATE FULLTEXT CATALOG`:

```
CREATE FULLTEXT CATALOG имя_каталога
    [WITH ACCENT_SENSITIVITY = {ON|OFF}]
    [AS DEFAULT]
```

Параметр `AS DEFAULT` определяет, является ли создаваемый каталог каталогом по умолчанию. А параметр `ACCENT_SENSITIVITY` определяет, будет ли индекс создан как регистрозависимый или нет.

Пример создания полнотекстового каталога в базе данных:

```
CREATE FULLTEXT CATALOG DefaultCatalog AS DEFAULT
```

Изменение и удаления каталога делаются так же, как других объектов базы данных. То есть используются команды `ALTER FULLTEXT CATALOG` и `DROP FULLTEXT CATALOG`.

Также вы можете создать полнотекстовый каталог с помощью SQL Server Management Studio. Откройте в окне **Object Explorer**, в папке нужной вам базы данных, папку **Storage**, а там **Full Text Catalogs**. Выберите в контекстном меню пункт **New Full-Text Catalog**. В появившемся диалоге введите имя каталога и нажмите кнопку **OK**. Каталог будет создан.

Создание индекса для полнотекстового поиска

Теперь необходимо создать индексы, с помощью которых будет осуществляться поиск. Индексы состоят из определенных столбцов какой-либо таблицы базы данных.

Для настройки индекса вам необходимо вызвать диалог редактирования свойств полнотекстового каталога. Для этого щелкните правой кнопкой мыши на имени каталога и выберите пункт меню **Properties**. Выберите страницу **Tables/Views**. Сам диалог вы можете увидеть на рис. 15.1.

В этом диалоге есть список таблиц. Вам необходимо выбрать таблицы, участвующие в индексе. После этого вам потребуется выбрать конкретные колонки таблиц, которые участвуют в индексе.

> ***Внимание!***
>
> После изменения свойств полнотекстового индекса или параметров каталога необходимо перестроить каталог, чтобы он содержал актуальные данные. Для этого щелкните на имени каталога правой кнопкой мыши и выберите пункт **Rebuild**.

После этого вы можете выбрать страницу **Population Schedule** и настроить расписание, по которому полнотекстовый каталог будет обновляться. Сам диалог вы можете увидеть на рис. 15.2. В нем вы можете настроить, будет ли обновление ежедневным, еженедельным или ежемесячным, когда именно будет запускаться обновление. Или же вы можете выбрать автоматический перенос в индекс всех сделанных в таблицах изменений.

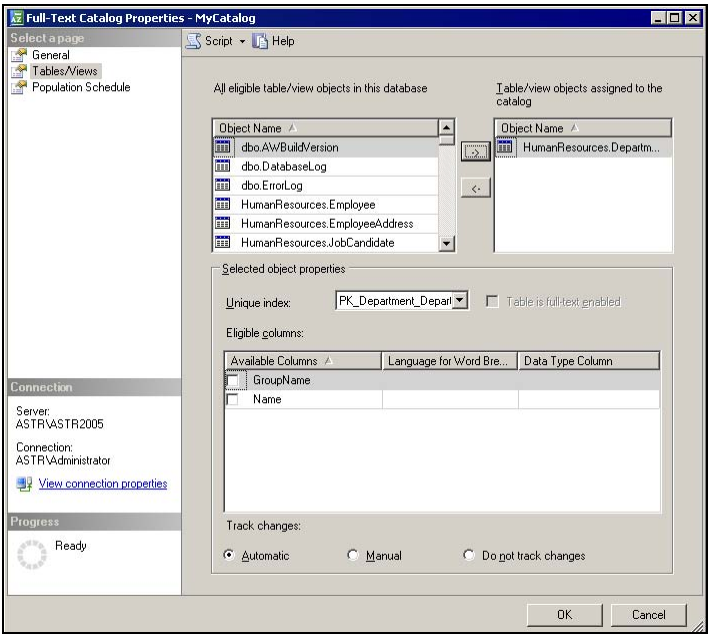


Рис. 15.1. Диалог настройки полнотекстового индекса

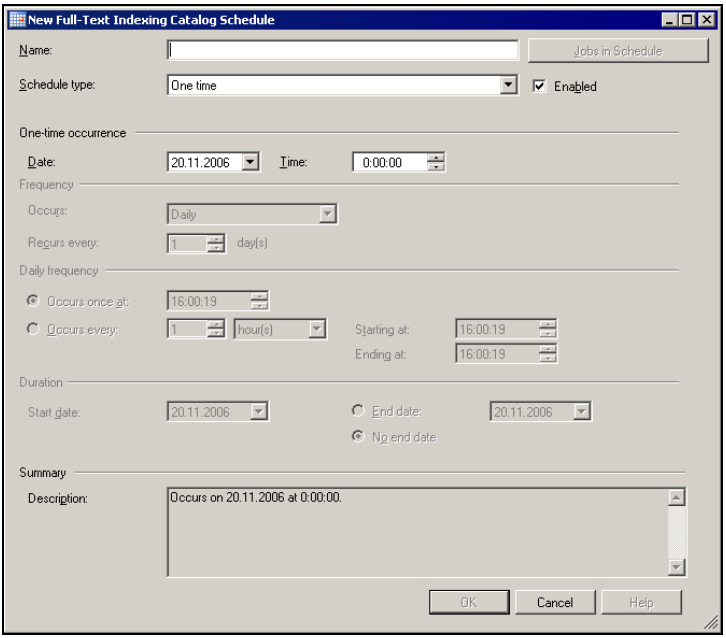


Рис. 15.2. Диалог настройки расписания обновления полнотекстового индекса

Как обычно в SQL Server Management Studio, сделав изменения, вы всегда можете увидеть SQL-код, который их обеспечит. Для этого просто нажмите на кнопку **Script** на панели инструментов.

Полнотекстовые запросы

Полнотекстовые запросы находят записи баз данных с помощью индексов, созданных для полнотекстовых запросов. Для выполнения полнотекстовых запросов используется команда `SELECT`. SQL Server поддерживает два ключевых слова для реализации полнотекстовых запросов: `CONTAINS` и `FREETEXT`.

CONTAINS

Ключевое слово `CONTAINS` организует поиск слова или фразы в столбце, зарегистрированном в полнотекстовом индексе. Условие поиска может состоять из начальных символов слова, за которыми следует звездочка (универсальный символ). При этом искомую фразу или слово надо заключать в двойные кавычки.

Синтаксис ключевого слова `CONTAINS`:

```
CONTAINS ({имя_столбца | *}, 'критерий_поиска')
```

Вот пример использования `CONTAINS`:

```
SELECT BookID, Name FROM Books WHERE CONTAINS (Name, 'Computer')
```

А вот пример поиска фразы:

```
SELECT BookID, Name
FROM Books
WHERE CONTAINS (Name, '"для чайников"')
```

В этом примере `CONTAINS` используется для поиска слов, начинающихся на "Ста" с произвольным продолжением:

```
SELECT BookID, Name FROM Books WHERE CONTAINS (Name, '"Ста*"')
```

Здесь `*` (звездочка) используется в качестве универсального символа.

Также в запросе вы можете использовать ключевые слова `AND`, `OR`, `NOT`, чтобы более точно определить, какие именно записи возвращать. Например:

```
SELECT BookID, Name
FROM Books
WHERE CONTAINS (Name, '"для чайников" AND "C#" AND NOT "JAVA"')
```

Этот запрос возвращает список книг, в названии которых есть фраза "для чайников" и "C#", но нет слова "JAVA".

Поиск слов, расположенных рядом

SQL Server также позволяет искать слова, находящиеся поблизости друг от друга. Для выполнения такого поиска в критерий включается функция NEAR(). Функция вызывается без параметров, однако по обе стороны от нее располагаются искомые слова или фразы. Например:

```
SELECT BookID, Name
FROM Books
WHERE CONTAINS (Name, '"C#" NEAR() "начинаю*"')
```

Этот запрос возвращает записи, в которых слово "C#" и слова, начинающиеся с "начинаю", находятся в столбце Name недалеко друг от друга.

Поиск единственного и множественного числа

SQL Server существует функция FORMSOF(), предназначенная для поиска слов в различных грамматических формах. Функция FORMSOF() ищет как единственное, так и множественное число слов (существительных и глаголов). Одна функция FORMSOF() может работать либо с существительными, либо с глаголами, но не с обеими категориями одновременно.

У функции FORMSOF() следующий синтаксис:

```
FORMSOF(INFLECTIONAL, 'фраза')
```

Вот пример использования функции FORMSOF():

```
SELECT BookID, Name
FROM Books
WHERE CONTAINS(Name, 'FORMSOF(INFLECTIONAL, computer)')
```

Этот запрос возвращает записи, в которых столбец Name содержит слово "computer" или любую из его грамматических форм.

Весовые коэффициенты в критериях поиска

Функция ISABOUT() в сочетании с функцией WEIGHT() позволяет задать весовые коэффициенты, определяющие относительную значимость критериев поиска.

Функция ISABOUT() определяет список весовых коэффициентов. В качестве параметра функции WEIGHT() передается весовой коэффициент — десятичное число в интервале от 0,0 до 1,0.

Синтаксис этой функции:

```
ISABOUT('фраза', WEIGHT(весовой_коэффициент, ...))
```

Приведем пример использования:

```
SELECT BookID, Name
FROM Books
WHERE CONTAINS (Name,
                'ISABOUT("разработка" WEIGHT(.8), "язык" WEIGHT(.4))')
```

Запрос возвращает записи из таблицы `Books`, содержащие слово "разработка" или "язык" в столбце `Name`. Однако в данном примере слову "разработка" назначен более высокий весовой коэффициент, поэтому сервер придает записям, содержащим слово "разработка", более высокий ранг.

FREETEXT

Команда `FREETEXT` позволяет обращаться к базе данных с запросами, основанными на значащих словах из произвольной текстовой строки. Механизм полнотекстовых запросов SQL Server анализирует текст, выделяет все значащие слова и обороты, после чего генерирует запрос на основании полученных результатов.

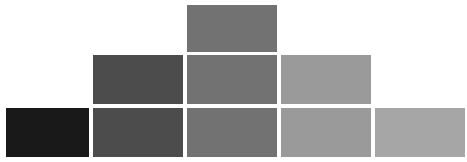
Синтаксис команды `FREETEXT`:

```
FREETEXT('имя_столбца', 'искомый_текст')
```

Приведем пример использования предиката `FREETEXT`:

```
SELECT BookID, Name
FROM Books
WHERE FREETEXT(Name,
               'Разработка приложений в экстремальных условиях')
```

Запрос возвращает записи из таблицы `Books` со словами, похожими на слова фразы "Разработка приложений в экстремальных условиях".



ГЛАВА 16

SQL Server и XML

XML — прекрасная технология, которая упрощает решение самых разных задач: от транспортировки данных до их структурированного хранения. Этот язык необходим для новейшей технологии Web-сервисов и используется в конфигурационных файлах приложений .NET. В общем, XML — это универсальная технология представления данных.

В этой главе нет описания, что же такое XML. Для этого вам потребуется обратиться к другим книгам. В данной главе кратко рассматриваются лишь возможности работы с XML в SQL Server 2005 (и более поздних версиях SQL Server), т. к. только там впервые появился хранимый тип данных XML.

XML и SQL Server

Впервые возможность экспортировать данные как XML появилась в SQL Server 2000. Но единственной естественной формой хранения XML в SQL Server 2000 является длинная строка. Этот подход неэффективен и не поддерживает поиск с учетом структуры данных. Поскольку такие XML-данные на самом деле являются лишь символьными, для их изменения нужно использовать функции работы со строками.

Начиная с SQL Server 2005, содержит новый тип данных — `xml`, с помощью которого решены многие подобные проблемы. Его можно с легкостью индексировать, запрашивать и обрабатывать. Новая технология XQuery позволяет непосредственно обрабатывать и запрашивать данные, извлекать из данных отдельные значения и проверять наличие данных.

Если вы хотите преобразовывать данные в XML и обратно по мере надобности, то можете использовать две возможности: конструкцию `FOR XML` и функцию `OPENXML`.

Первая позволяет преобразовать реляционные данные в XML-код. Вторая принимает XML-код и создает "представление", которое вы можете запрашивать, что позволяет с легкостью использовать данные для внесения изменений в базу данных.

Хранение XML

Если вы решили хранить XML-данные, то можете использовать тип данных XML и определить столбцы, которые будут содержать XML-данные. Это как дата, которая на самом деле представляет собой особый тип данных, а не символьные данные. XML является особым типом данных, поддерживающим дополнительные возможности, такие как применение схем, проверка и типизация.

Так как данные будут храниться в форме XML, а не в символьной форме, вы можете запрашивать данные и управлять ими. XML также можно индексировать, и из XML-данных можно создавать представления.

Тип данных `xml` имеет пять методов, использующих технологию XQuery, что позволяет запрашивать и даже изменять данные в XML-коде без сложного анализа строк, как если бы XML-данные хранились как строка.

Конструкция *FOR XML*

Конструкцию `FOR XML` выражения `FROM` можно использовать для получения результата запроса в виде XML. Для ее использования нужно только указать ее в конце оператора `SELECT`, как в запросе:

```
SELECT ProductID, SUM(OrderQty) AS Quantity
FROM Sales.SalesOrderHeader AS OrderHeader
    INNER JOIN Sales.SalesOrderDetail AS OrderDetail
        ON OrderHeader.SalesOrderID = OrderDetail.SalesOrderID
GROUP BY ProductID
FOR XML AUTO
```

Вы можете выполнить этот код в SQL Server Management Studio для базы AdventureWorks и убедиться, что он вернет XML с информацией о заказах.

В SQL Server 2005 и более поздних версиях конструкция `FOR XML` улучшена. Теперь она может возвращать XML как специфический тип данных, а не как символьные данные. Используя опцию `AUTO` конструкции `FOR XML` вместе с подзапросами, вы можете сгенерировать XML, содержащий и элементы, и атрибуты, что раньше можно было выполнить только с помощью более сложной опции `EXPLICIT` конструкции `FOR XML`.

Пример далее извлекает заголовок заказа как элемент, а каждый компонент заказа отображает как элемент, а его данные как атрибуты:

```
SELECT TOP 2 SalesOrderID, OrderDate, CustomerID,
    (SELECT ProductID, OrderQty, UnitPrice
       FROM Sales.SalesOrderDetail AS SalesOrderDetail
       WHERE SalesOrderDetail.SalesOrderID=SalesOrderHeader.SalesOrderID
       FOR XML AUTO, TYPE)
FROM Sales.SalesOrderHeader AS SalesOrderHeader
ORDER BY SalesOrderID
FOR XML AUTO, ELEMENTS
```

Конструкция `TOP` использована только для ограничения объема возвращаемых результатов. Этот запрос вернет следующий результат:

```
<SalesOrderHeader>
    <SalesOrderID>43659</SalesOrderID>
    <OrderDate>2001-07-01T00:00:00</OrderDate>
    <CustomerID>676</CustomerID>
    <SalesOrderDetail ProductID="776" OrderQty="1" UnitPrice="2024.9940"/>
    <SalesOrderDetail ProductID="777" OrderQty="3" UnitPrice="2024.9940"/>
    <SalesOrderDetail ProductID="778" OrderQty="1" UnitPrice="2024.9940"/>
</SalesOrderHeader>
<SalesOrderHeader>
    <SalesOrderID>43660</SalesOrderID>
    <OrderDate>2001-07-01T00:00:00</OrderDate>
    <CustomerID>117</CustomerID>
    <SalesOrderDetail ProductID="762" OrderQty="1" UnitPrice="419.4589"/>
    <SalesOrderDetail ProductID="758" OrderQty="1" UnitPrice="874.7940"/>
</SalesOrderHeader>
```

Новая опция `TYPE`, которая использована в этом коде, приказывает конструкции `FOR XML` возвратить настоящий XML-код вместо символьных данных. В полученных результатах записи `SalesOrderDetail` являются обособленными фрагментами XML с атрибутами. Из-за опции `TYPE` эти результаты затем рассматриваются в главном запросе как один столбец с XML-данными.

Кроме того, в SQL Server 2005 и более поздних версиях `FOR XML` поддерживает другую опцию, позволяющую указать путь к данным для XML-вывода в операторе `SELECT`. Пример, представленный далее, выводит те же результаты, что и предыдущий пример, но с использованием конструкции `FOR XML PATH`.

```
SELECT TOP 2 SalesOrderID AS '@SalesOrderID',
             OrderDate AS 'SalesOrderHeader/OrderDate',
```

```
            CustomerID AS 'SalesOrderHeader/CustomerID',
   (SELECT ProductID AS '@ProductID',
           OrderQty AS '@OrderQty', UnitPrice AS '@UnitPrice'
      FROM Sales.SalesOrderDetail AS SalesOrderDetail
      WHERE SalesOrderDetail.SalesOrderID=SalesOrderHeader.SalesOrderID
      FOR XML PATH ('SalesOrderDetail'), TYPE)
FROM Sales.SalesOrderHeader AS SalesOrderHeader
ORDER BY SalesOrderID
FOR XML PATH ('SalesOrder')
```

В этом примере имена столбцов определяют путь к данным при выводе XML. Аргумент конструкции `PATH` определяет имя главного элемента данных. Добавление префикса `@` к выходному имени столбца говорит о том, что данные столбца должны быть выведены как атрибут, а не как элемент.

В результате этого запроса вы получите следующее:

```
<SalesOrder SalesOrderID="43659">
   <SalesOrderHeader>
      <OrderDate>2001-07-01T00:00:00</OrderDate>
      <CustomerID>676</CustomerID>
   </SalesOrderHeader>
   <SalesOrderDetail ProductID="776" OrderQty="1" UnitPrice="2024.9940"/>
   <SalesOrderDetail ProductID="777" OrderQty="3" UnitPrice="2024.9940"/>
   <SalesOrderDetail ProductID="778" OrderQty="1" UnitPrice="2024.9940"/>
   <SalesOrderDetail ProductID="771" OrderQty="1" UnitPrice="2039.9940"/>
</SalesOrder>
<SalesOrder SalesOrderID="43660">
   <SalesOrderHeader>
      <OrderDate>2001-07-01T00:00:00</OrderDate>
      <CustomerID>117</CustomerID>
   </SalesOrderHeader>
   <SalesOrderDetail ProductID="762" OrderQty="1" UnitPrice="419.4589"/>
   <SalesOrderDetail ProductID="758" OrderQty="1" UnitPrice="874.7940"/>
</SalesOrder>
```

Как видите, это совсем другой XML-код, но для его получения нам понадобилось внести лишь несколько небольших изменений.

Функция *OPENXML*

Функция `OPENXML` появилась еще в SQL Server 2000. Она позволила выполнять разбор XML-данных, чтобы их можно было сохранить в базе данных. Начиная с SQL Server 2005, функция `OPENXML` поддерживает новый тип данных `xml`.

Предположим, что у вас есть таблица, созданная с помощью кода:

```
CREATE TABLE Contacts (
    ContactID int IDENTITY(1, 1) NOT NULL PRIMARY KEY NONCLUSTERED,
    LastName varchar(50)  NOT NULL,
    FirstName varchar(50) NOT NULL,
    Address varchar(150)  NOT NULL,
    City varchar(30)      NOT NULL
)
```

Допустим, что вам нужно вставить в эту таблицу XML-данные:

```
<Root>
    <Contact id="1">
        <LastName>Иванов</LastName>
        <FirstName>Андрей</FirstName>
        <Address>улица Компьютерщиков, д. 3, кв. 11</Address>
        <City>Санкт-Петербург</City>
    </Contact>
    <Contact id="2">
        <LastName>Сидоров</LastName>
        <FirstName>Сергей</FirstName>
        <Address>пр. Машиностроителей, д. 17, кв. 156</Address>
        <City>Саратов</City>
    </Contact>
</Root>
```

Вы можете создать следующую хранимую процедуру для загрузки этого XML в таблицу:

```
CREATE PROCEDURE ParseMyXml
    @Xml varchar(2000)
AS
DECLARE @hXml int
EXEC sp_xml_preparedocument @hXml output, @Xml
INSERT INTO Contacts(LastName, FirstName, Address, City)
    SELECT * FROM OPENXML (@hXml, '/Root/Contact', 10)
        WITH (LastName varchar(50),
              FirstName varchar(50),
              Street varchar(150) 'Address',
              City varchar(30) 'City')
EXEC sp_xml_removedocument @hXml
GO
```

В этой процедуре сначала с помощью процедуры `sp_xml_preparedocument` выполняется подготовка XML-документа, а затем его разбор. После этого вам останется только вызвать эту процедуру, передав через параметр XML-данные для разбора.

В SQL Server 2000 текстовый тип данных можно было передавать в функцию `OPENXML`, но размер XML-информации, которую можно было обработать за один раз, был ограничен, потому что SQL Server 2000 не поддерживал переменные текстового типа. Начиная с SQL Server 2005, вы можете использовать любой из символьных типов данных с опцией размера *max* (например, `varchar(`*max*`)`) или новый тип данных `xml`.

Тип данных *xml*

Начиная с SQL Server 2005, поддерживается встроенный тип данных `xml`, который в предыдущей версии отсутствовал. Этот тип данных не только повышает эффективность работы с XML в контексте языка T-SQL, но и поддерживает возможности, характерные для других типов данных, такие как типизация переменных и столбцов. Кроме того, он имеет собственные методы, позволяющие использовать возможности XQuery для проверки существования данных или индивидуальных значений, запроса данных и даже изменения XML-данных.

Есть две разновидности типов данных `xml`: типизированные и нетипизированные. Нетипизированный XML легко реализовать, и почти каждый программист может довольно быстро научится с ним работать. Однако типизированный XML часто работает лучше и эффективнее, чем нетипизированный.

Типизация XML

Начиная с SQL Server 2005, можно зарегистрировать схему XML в базе данных, чтобы SQL Server мог ограничивать и типизировать XML. Типизированный XML предоставляет механизм проверки, чтобы любые вставляемые или изменяемые XML-данные проверялись на соответствие определенной схеме.

Для создания типизированного XML нужно сначала создать набор схем внутри базы данных с помощью оператора `CREATE XML SCHEMA COLLECTION`. Он создает набор, состоящий из одной или нескольких схем, каждая из которых описывает некоторое пространство имен.

Синтаксис команды для создания схемы:

```
CREATE XML SCHEMA COLLECTION имя_коллекции AS XML-схема
```

После создания схемы вы можете создавать переменные типа `xml` или типизированные колонки типа `xml` в таблицах, как показано в примерах:

```
DECLARE @x xml (MySchemaCollection)
GO
CREATE TABLE MyTable(
        i int primary key,
        x xml (MySchemaCollection))
```

При вставке в колонку таблицы или в переменную XML-данных, не соответствующих схеме, будет выдано сообщение об ошибке.

Схемы расширяют XML-возможности SQL Server, благодаря чему вы можете хранить строго типизированные XML-данные.

Методы типа данных *xml*

У типа данных `xml` есть пять методов для выполнения различных действий над XML-данными. В четырех методах используется синтаксис XQuery, а пятый выполняет простые обновления с помощью XQuery. Далее описаны все эти методы.

Метод *exist*

Метод `exist` следует использовать для проверки наличия определенных значений в XML-данных, но не для простой проверки существования XML-данных. Если вы хотите узнать, действительно ли столбец содержит XML-данные, проверьте, не содержит ли он `NULL`.

Пример использования метода `exist`:

```
SELECT ProductModelID, Name
FROM Production.ProductModel
WHERE  Instructions.exist('declare namespace
    AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions";
    /AWMI:root/AWMI:Location[@LocationID=50]')=1
```

В этом запросе выбираются все записи, в котором `LocationID=50`. Путь представляется как */родительский_элемент/дочерний_элемент [атрибут]*. Такое представление пути используется во всех методах типа данных `xml` и входит в спецификации XPath и XQuery.

Если столбец, определенный как типизированный XML, содержит данные (т. е. его значение не `NULL`), это корректные XML-данные, потому что схема этого столбца не позволила бы поместить в него что-либо, кроме синтаксически правильных XML-данных, соответствующих этой схеме.

Метод *value*

Метод `value` позволяет извлечь из XML-кода одно значение как встроенный тип данных SQL Server, такой как `int` или `varchar`. Он принимает два аргумента: выражение XQuery, используемое для извлечения значения, и тип данных T-SQL для возвращаемого значения (кроме `xml`, пользовательских типов данных, `image`, `text`, `ntext` и `timestamp`).

Пример использования метода `value`:

```
SELECT Name, CatalogDescription.value('declare namespace
    PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
    (/PD:ProductDescription/@ProductModelID)[1]', 'int') as Result
FROM Production.ProductModel
WHERE CatalogDescription.value('declare namespace
    PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
    (/PD:ProductDescription/@ProductModelID)[1]', 'int')
    BETWEEN 10 AND 50
```

Этот запрос возвращает только те продукты, у которых `ProductModelID` между 10 и 50.

При этом если функция `value` не сможет найти значение, соответствующее запросу, то она вернет `NULL`.

Метод *query*

Метод `query` позволяет выполнить XQuery-запрос XML-данных и возвращает нетипизированные XML-данные. В отличие от метода `value`, этот метод возвращает не единственное значение, а множество узлов запрашиваемых XML-данных.

Пример использования метода `query`:

```
SELECT CatalogDescription.query('declare namespace
    PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelDescription";
    <Product ProductModelID="{
    /PD:ProductDescription[1]/@ProductModelID }" />') as Result
FROM Production.ProductModel
WHERE CatalogDescription.exist('declare namespace
    PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works
/ProductModelDescription";
    declare namespace
```

```
    wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works
/ProductModelWarrAndMain";
    /PD:ProductDescription/PD:Features/wm:Warranty') = 1
```

Этот запрос возвращает записи, в которых в колонке `CatalogDescription` есть элемент `Product`, который содержит атрибут `ProductModelID`.

Даже если данные хранятся в SQL Server как XML, вы можете извлекать, преобразовывать и переформатировать XML в то, что вам нужно, не прибегая к написанию кода на .NET.

Метод *modify*

Метод `modify` позволяет вносить изменения в данные типа `xml` без переписывания всей структуры данных. Эта возможность может существенно упростить обновление XML-данных.

Пример использования этого метода:

```
UPDATE Production.ProductModel
SET Instructions.modify('declare namespace
    PMMI="http://schemes.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions"
    replace value of (/PMMI:root/PMMI:Location[@LocationID=20]/
PMMI:step[1]/PMMI:blueprint)[1]
    with "1234"')
FROM Production.ProductModel AS ProductModel
WHERE ProductModelID = 47
```

Этот запрос ищет первый узел `<blueprint>` в первом узле `<step>` узла `<Location>` с идентификатором 20 и изменяет номер чертежа на 1234. Как и в других методах, в этом методе используются синтаксис XQuery и ссылки на путь.

При изменении столбца XML всегда используется оператор `UPDATE` языка T-SQL, показывающий, что вы изменяете запись таблицы, но в методе `modify` надо также использовать выражение `insert`, `replace value` или `delete`. Например, чтобы удалить какую-либо запись, вы можете использовать следующий запрос:

```
UPDATE Production.ProductModel
SET Instructions.modify('declare namespace
    PMMI="http://schemas.microsoft.com/sqlserver/2004/07/
adventureworks/Product ModelManuInstructions"
    delete(/PMMI:root/PMMI:Location[@LocationID=20] /PMMI:step[1]
/PMMI:blueprint)[1]')
```

```
FROM Production. ProductModel. AS ProductModel
WHERE ProductModellD = 47
```

Код T-SQL в данном запросе остался тем же, и только код в методе `modify` чуть изменился. Вы все так же обновляете запись в БД, пусть даже для XML-данных это означает вставку или удаление.

Метод *nodes*

Метод `nodes` позволяет получить из XML-данных реляционные данные. Простой пример этого, который возвращает три строки:

```
DECLARE @x xml
SET @x='<Root>
<row id="1"><name>Mike</name><oflw>any text</oflw></row>
<row id="2"><name>Ann</name></row>
<row id="3" />
</Root>'
SELECT Tbl.col.value('@id','int') as ID,
       Tbl.col.query('name') as Name
FROM   @x.nodes('/Root/row') Tbl(col)
```

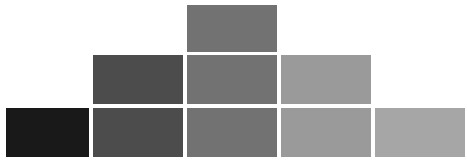
Метод `nodes` определяет итоговые таблицу и столбец — `Tbl` и `col` — для пути `Root/row` в XML-данных. В этом примере запрос возвращает каждый элемент `row` как отдельную запись, а итоговая таблица содержит две колонки: `ID` и `Name`.

Конструкция *WITH XMLNAMESPACES*

Конструкция `WITH XMLNAMESPACES` позволяет сильно упростить запросы. В ней просто перечисляются пространства имен с их псевдонимами. Например:

```
WITH XMLNAMESPACES (
   'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelDescription' AS PD,
   'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelWarrAndMain' AS wm)

SELECT CatalogDescription.query('
    <Product ProductModelID="{
    /PD:ProductDescription[1]/@ProductModelID }" />
    ') as Result
FROM Production.ProductModel
where CatalogDescription.exist('
     /PD:ProductDescription/PD:Features/wm:Warranty ') = 1
```

ГЛАВА 17



Интеграция с .NET Framework

Одной из новых возможностей SQL Server (еще версии 2005) являлась его интеграция с Microsoft .NET Framework. В этой главе дано краткое описание этих возможностей.

Вы можете создавать хранимые процедуры, триггеры, пользовательские функции на любом .NET-совместимом языке (например, на C# или VB.NET), используя все богатство возможностей классов .NET Framework, а не только T-SQL. Также вы можете решать административные задачи, используя новый API, состоящий из объектов .NET.

У вас есть выбор. Вы можете писать код на T-SQL или на .NET. Однако вам все равно потребуется определять процедуры, триггеры и т. д. на T-SQL, который, по сути, регистрирует код сборки, чтобы ее можно было использовать в SQL Server.

Например, решив написать хранимую процедуру на C#, вы сначала пишете код C#, компилируете сборку, а затем регистрируете ее в SQL Server, используя оператор `CREATE ASSEMBLY` языка T-SQL для загрузки кода сборки в БД. Наконец, вы используете оператор `CREATE PROCEDURE` языка T-SQL для подключения хранимой процедуры к функции, находящейся в зарегистрированной сборке.

Как вы можете видеть, в этом процессе T-SQL по-прежнему играет важную роль.

Кроме того, есть определенные возможности, для которых T-SQL просто необходим. Возможности нескольких операторов языка манипулирования данными, таких как `SELECT` и `INSERT`, реализованы только в T-SQL. То есть хранимая процедура, написанная на C#, все же должна вызывать для выборки данных код T-SQL.

На самом деле, .NET расширяет, а не заменяет T-SQL. Что касается фактического кода процедуры, функции и т. д., то тут вы свободны в выборе между .NET и T-SQL.

Какой вариант выбрать?

Нельзя сказать однозначно, когда писать код на T-SQL, а когда на языке .NET. Далее приведены простые правила, которые помогут вам в выборе, но помните, что окончательное решение зависит от многих факторов.

- Если код преимущественно обращается к данным или изменяет их, его надо писать на T-SQL. Для обращения к данным .NET-код должен "подключаться" к SQL Server, и поэтому по скорости доступа к данным он значительно уступает коду T-SQL. К тому же, код T-SQL гораздо проще изменять и поддерживать.
- Если код большей частью выполняет операции, требовательные к вычислительной мощности (математические вычисления, шифрование и т. д.), базу кода следует писать на языке .NET.
- Если вы хотите использовать возможности .NET Framework, вам определенно придется писать на языке .NET.

Однако комбинирование этих условий может затруднять принятие решения.

Далее будет описано создание различных объектов базы данных, использующих код .NET. Все примеры кода написаны на C#, и вам необходимо некоторое знание этого языка, чтобы понимать их.

Создание объектов БД с помощью .NET

В первую очередь вам потребуется создать новый проект SqlServerProject. Выберите в меню **File | New | Project**, а после этого в появившемся диалоге выберите язык, который вам нравится, например Visual C#, а потом тип проекта **SqlServerProject** и введите его имя (рис. 17.1). Результатом компилирования такого проекта будет динамическая библиотека (файл с расширением dll), а точнее, *сборка*. Сборка, или assembly, представляет собой исполняемый модуль в .NET.

После создания проекта появится диалог настройки соединения с базой данных (рис. 17.2). Введите все необходимые параметры и проверьте соединение с помощью кнопки **Test Connection**. Если проверка соединения пройдет успешно, нажмите кнопку **OK**, и теперь вы сможете работать с проектом.

Теперь, с помощью окна **Server Explorer** в Visual Studio, вы можете просматривать и изменять объекты базы данных (рис. 17.3).

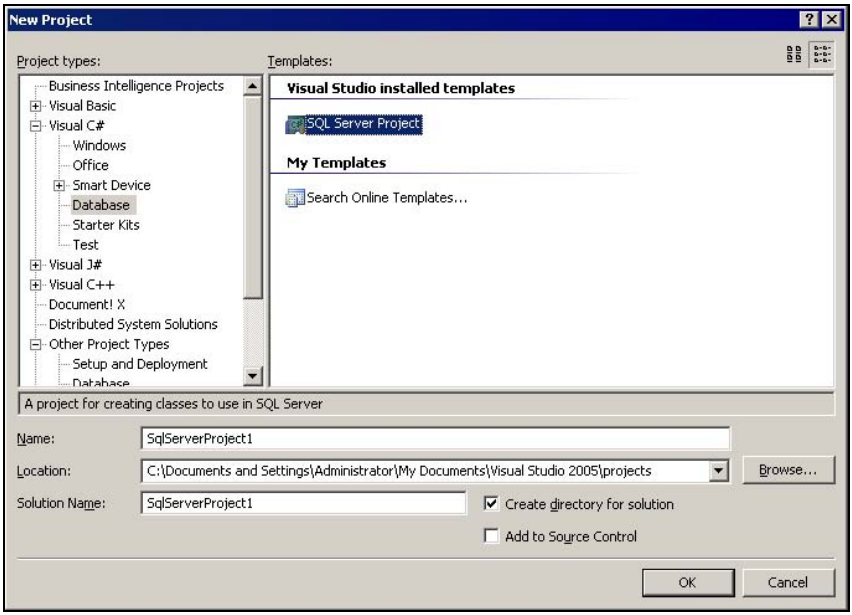


Рис. 17.1. Диалог создания нового проекта в Visual Studio 2008

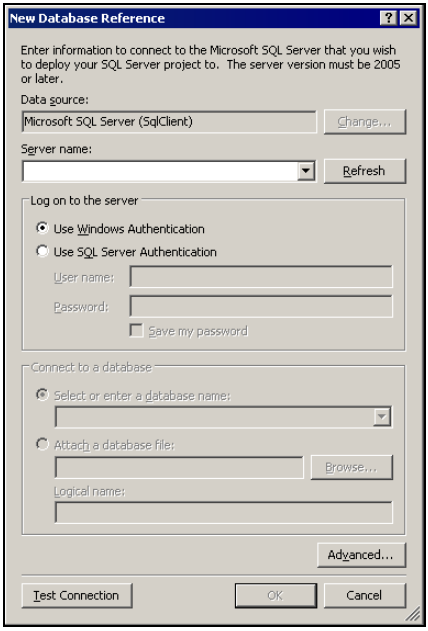


Рис. 17.2. Диалог настройки соединения с SQL Server в Visual Studio 2008

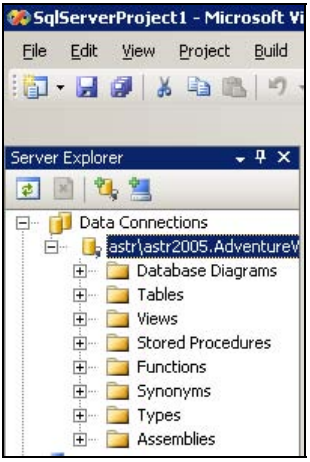


Рис. 17.3. Окно **Server Explorer** в Visual Studio 2008

ВНИМАНИЕ!

Следует иметь в виду, что поддержка .NET в SQL Server по умолчанию отключена. Для того чтобы подключить эту возможность, следует выполнить команды:

```
execute sp_configure 'clr_enabled', 1
reconfugure
```

Интеграция сборки в SQL Server

Сборку, созданную в Visual Studio, вначале требуется поместить в SQL Server. В окне **Object Explorer** SQL Management Studio в списке объектов каждой базы данных находится раздел **Programmability**, а в нем подраздел **Assemblies**. Здесь должны размещаться все зарегистрированные сборки.

Для того чтобы зарегистрировать сборку, нужно щелкнуть правой кнопкой мыши по строке **Assemblies** и выбрать в контекстном меню пункт **New Assembly** (рис. 17.4).

В появившемся окне следует указать полный путь к сборке. Имя для сборки присваивается по имени файла, где она содержится.

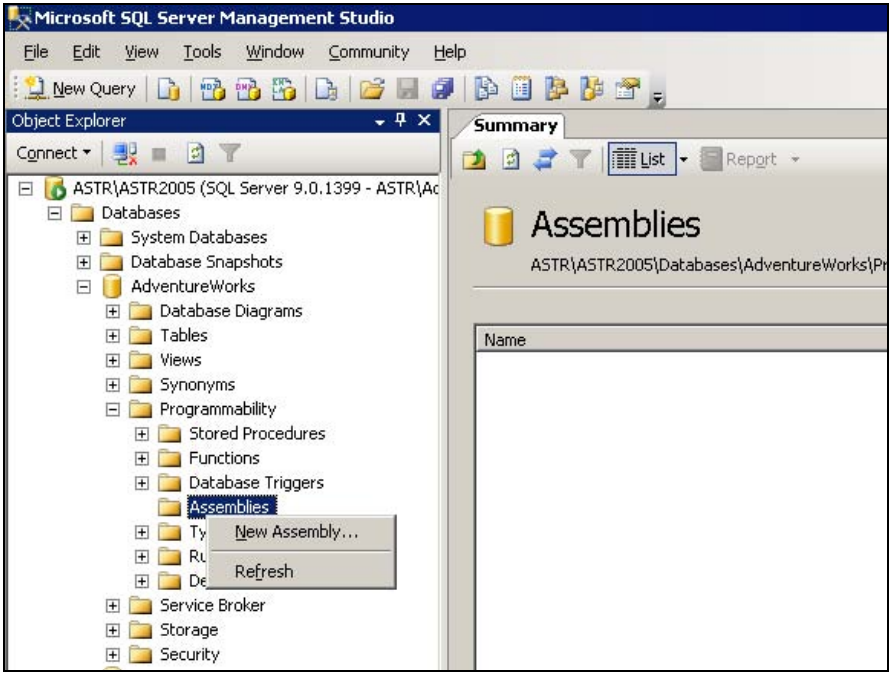


Рис. 17.4. Регистрация сборки в SQL Server Management Studio

Другой программный путь добавления сборки основывается на использовании команды `CREATE ASSEMBLY`. В SQL Server, используя команду `CREATE ASSEMBLY`, можно явно задать имя сборки.

Вот пример создания сборки с помощью команды `CREATE ASSEMBLY`:

```
CREATE ASSEMBLY MyAssembly
    authorization dbo
    from 'C:\MyAssembly.dll' with permission_set = safe
```

Параметры этой команды:

- `MyAssembly` — имя создаваемой сборки;
- `authorization dbo` — здесь определяется имя владельца сборки. Если данный раздел опустить, то собственником сборки будет текущий пользователь;
- `from 'C:\MyAssembly.dll'` — полный путь к сборке;
- `with permission_set = safe` — здесь настраиваются параметры безопасности сборки.

  Существуют три уровня безопасности:

  - `safe` — этот самый высокий уровень безопасности означает, что сборка не может иметь доступ к внешним ресурсам (файлам, реестру, сети и т. п.);
  - `external_access` — данный уровень безопасности означает, что сборка может получить доступ к некоторым внешним ресурсам;
  - `unsafe` — данный уровень самый небезопасный. Сборка получает полный доступ к внешним ресурсам. В том числе она может запускать любой код.

Для удаления сборки из SQL Server используется команда:

```
DROP ASSEMBLY <имя_сборки>
```

Хранимые процедуры

Вернемся к нашему вновь созданному проекту в Visual Studio и создадим на базе него хранимую процедуру на основе .NET. Если вы посмотрите меню **Project** Visual Studio, то увидите, что можно выполнить следующие действия:

- **Add User-Defines Function** — добавить к проекту класс для создания пользовательской функции;
- **Add Stored Procedure** — добавить к проекту класс для создания хранимой процедуры;

- **Add Aggregate** — добавить к проекту класс для создания агрегирующей функции;
- **Add Trigger** — добавить к проекту класс для создания триггера;
- **Add User-defined Type** — добавить к проекту класс для создания пользовательского типа данных.

Нам нужно создать хранимую процедуру, так что используем команду **Project | Add Stored Procedure**. Будет создана хранимая процедура с именем StoredProcedure1, код которой представлен далее:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void StoredProcedure1()
    {
        // Код хранимой процедуры
    }
};
```

Все, что от вас теперь требуется — это написать код процедуры. Рассмотрим этот код подробнее.

Основным пространством имен для работы с SQL Server является Microsoft.SqlServer.Server, поэтому его объявление добавлено в начало. Атрибут Microsoft.SqlServer.Server.SqlProcedure определяет, что функция, которая идет после него, является хранимой процедурой.

Далее представлен пример реализации хранимой процедуры, которая выполняет простой запрос и возвращает результат запроса клиенту:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
```

```
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void MyFirstProcedure()
    {
        SqlCommand comm = new SqlComnand();
        comm.CommandType = CommandType.Text;
        comm.CommandText =
          "SELECT * FROM Books";
Microsoft.SqlServer.Server.SqlContext.Pipe.ExecuteAndSend(comm);
    }
};
```

Здесь сначала создается обычный объект `SqlCommand` для запроса из некоторой таблицы. Но после этого для выполнения команды T-SQL используется метод `ExecuteAndSend`. Этот метод и передает результат выполнения клиенту, т. е. в SQL Server. Так можно возвращать наборы данных.

После того как процедура готова, мы можем откомпилировать проект и получить сборку. Далее сборка должна быть помещена в SQL Server, например, с помощью следующей команды:

```
CREATE ASSEMBLY MyProcs
authorization dbo
from 'C:\MyProcs.dll' with permission_set = safe
```

Теперь на процедуру `MyFirstProcedure` можно ссылаться при создании хранимых процедур. Далее представлена команда создания хранимой процедуры, которая ссылается на процедуру `MyFirstProcedure` из сборки `MyProcs`.

```
CREATE procedure MyProc
AS
external name MyProcs.StoredProcedures.MyFirstProcedure
```

В команде для ссылки на хранимую процедуру используется последовательность:

имя_сборки.имя_класса.имя_процедуры

Теперь вы можете вызывать хранимую процедуру `MyProc` из любого запроса и объекта.

Мы создали хранимую процедуру без параметров, но она может иметь их. Для этого вам необходимо понять соответствие между типами данных SQL Server и типами данных языков .NET. Хотя пространство имен `System.Data.SqlType` содержит типы данных, созданные специально для работы с SQL Server. Это соответствие показано в табл. 17.1.

Таблица 17.1. *Соответствие между типами SQL Server и типами .NET*

Типы SQL Server 2008	Типы пространства имен `System.Data.SqlType`	Стандартные типы .NET
varbinary	SqlBytes, SqlBinary	Byte[ ]
binary	SqlBytes, SqlBinary	Byte[ ]
varbinary(1), binary(1)	SqlBytes, SqlBinary	byte, Byte[ ]
image	—	—
varchar	—	—
char	—	—
nvarchar(1), nchar(1)	SqlChars, SqlString	Char, String, Char[ ]
nvarchar	SqlChars, SqlString	String, Char[ ]
nchar	SqlChars, SqlString	String, Char[ ]
text	—	—
ntext	—	—
uniqueidentifier	SqlGuid	Guid
rowversion	—	Byte[ ]
bit	SqlBoolean	Boolean
tinyint	SqlByte	Byte
smallint	SqlIntl6	Int16
int	SqlInt32	Int32
bigint	SqlInt64	Int64
smallmoney	SqlMoney	Decimal
money	SqlMoney	Decimal
numeric	SqlDecimal	Decimal
decimal	SqlDecimal	Decimal
real	SqlSingle	Single
float	SqlDouble	Double
smalldatetime	SqlDateTime	DateTime
datetime	SqlDateTime	DateTime
sql_variant	—	Object

Таблица 17.1 *(окончание)*

Типы SQL Server 2008	Типы пространства имен `System.Data.SqlType`	Стандартные типы .NET
table	—	—
cursor	—	—
timestamp	—	—
xml	SqlXml	—

Приведем пример простой хранимой процедуры, которая возвращает список покупателей с определенной фамилией:

```
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void GetCustomers(SqlString lastName)
    {
      SqlCommand comm = new SqlComnand();
      comm.CommandType = CommandType.Text;
      comm.CommandText = "SELECT * FROM Customer " +
                         " WHERE LastName='" + lastName + "'";
     Microsoft.SqlServer.Server.SqlContext.Pipe
    .ExecuteAndSend(comm);
    }
};
```

Для регистрации этой хранимой процедуры нужно выполнить следующую команду:

```
-- регистрируем сборку
CREATE ASSEMBLY MyProcs
authorization dbo
from 'C:\MyProcs.dll' with permission_set = safe
GO

-- создаем хранимую процедуру
CREATE procedure GetCustomers (@lastName nvarchar(60))
AS
external name MyProcs.StoredProcedures.GetCustomers
GO
```

Скалярные пользовательские функции

Создание пользовательской функции очень похоже на создание хранимой процедуры.

Выберите **Add User-Defines Function** из меню **Project** Visual Studio. Назовите вновь созданный класс `MyFunction`.

Пример пользовательской функции:

```
public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString MyUserFunc (SqlString lastName)
    {
        return new SqlString("Hello " + lastName);
    }
};
```

Для регистрации этой пользовательской функции нужно выполнить следующую команду:

```
-- регистрируем сборку
CREATE ASSEMBLY MyProcs
authorization dbo
from 'C:\MyProcs.dll' with permission_set = safe
GO

-- создаем пользовательскую функцию
CREATE function MyUserFunc (@lastName nvarchar(60))
return nvarchar(50)
AS
external name MyProcs.UserDefinedFunctions.MyUserFunc
GO
```

Созданная функция является скалярной — она возвращает строковое значение. Теперь функция может быть вызвана как обычная функция в T-SQL, как показано далее:

```
SELECT MyUserFunc('Иванов')
DECLARE @ln nvarchar(250)
SET @ln = MyUserFunc('Иванов') + '!'
```

В примере далее показано, как создавать функцию, которая возвращает результат, получаемый в результате выполнения некоторого запроса к базе данных:

```
public partial class UserDefinedFunctions
{
    [SqlFunction(DataAccess = DataAccessKind.Read)]
    public static SqlString MyUserFunc(SqlInt32 userId)
    {
        SqlConnection conn =
            new SqlConnection("context connection=true");
        conn.Open();
        SqlComand cmd = new SqlCommand();
        cmd.commandText = "SELECT UserName FROM Users " +
            "WHERE userId=" + userId.ToString();
        cmd.Connection = conn;
        return (string)cmd.ExecuteScalar();
    }
};
```

Свойство `DataAccess` определяет возможность доступа к данным. А метод `ExecuteScalar` выполняет запрос и возвращает значение первого столбца первой колонки полученной таблицы.

Для того чтобы получить доступ к данным, необходимо также создать объект `SqlConnection` — соединение, посредством которого будет осуществлен запрос.

Табличные функции

Теперь рассмотрим, как создаются табличные пользовательские функции. Рассмотрите пример, представленный далее.

Выберите **Add User-Defines Function** из меню **Project** Visual Studio. Назовите вновь созданный класс `MyTableFunction`.

После этого определите структуру, которая будет представлять строку результирующего набора:

```
public struct User
{
  public Int32 Id;
  public String Name;
}
```

После этого вы можете определить саму функцию:

```
public partial class UserDefinedFunctions
{
    [SqlFunction(DataAccess = DataAccessKind.Read,
                 FillRowMethodName = "FillRow")]
    public static  IEnumerable MyTableFunc(SqlInt32 id)
    {
        User[] users = new User[1];

        SqlDataReader rd = null;
        SqlConnection conn =
            new SqlConnection("context connection=true");
        conn.Open();

        SqlCommand cmd = new SqlCommand();
        cmd.Connection = conn;
        cmd.CommandText = "SELECT Id, Name FROM Users " +
                          "WHERE Id>" + id.ToString();
        rd = cmd.ExecuteReader();

        int i = 1;
        while (rd.Read())
        {
            users [i-l].Id = (Int32)rd[0];
            users [i-1].Name = (String)rd[l];
            i ++;
            Array.Resize(ref users, i);
    }
    Array.Resize(ref users, i-1);
    conn.Close();
    return users;
}

public static void FillRow(Object obj, out Sqllnt32 id,
                           out SqlString name)
{
    User u = (User)obj;
    id = new SqlInt32(u.Id);
    name = new SqlString(u.Name);
}
```

Здесь в классе имеются две функции. Однако лишь один метод `MyTableFunc` объявляется непосредственно как табличная функция. Функция `FillRow` играет вспомогательную роль — она формирует строку, которая будет отправлена приложению, вызвавшему функцию `MyTableFunc`.

В атрибуте функции `MyTableFunc`, кроме свойства `DataAccess`, необходимо также определить свойство `FillRowMethodName`. Оно указывает на второй метод, необходимый для заполнения строк результирующего набора. Учтите, что функция `FillRow` будет вызываться столько раз, сколько строк в результирующем наборе.

```
-- регистрируем сборку
CREATE ASSEMBLY MyProcs
authorization dbo
from 'C:\MyProcs.dll' with permission_set = safe
GO

-- создаем пользовательскую функцию
CREATE function MyTableFunc (@id int)
returns table (id int, name nvarchar(50))
AS
external name MyProcs.UserDefinedFunctions.MyTableFunc
GO
```

После создания функции к ней можно обращаться из любого запроса. Например:

```
SELECT * FROM MyTableFunc(14)
```

Триггеры

Также вы можете создать триггер на языке .NET. Сам процесс практически такой же, как и для хранимой процедуры.

Вот пример создания простого триггера:

```
public partial class Triggers
{
    [Microsoft.SqlServer.Server.SqlTrigger (Name="Trigger1",
        Target="students", Event="FOR INSERT")]
    public static void Trigger1()
    {
    }
}
```

Обратите внимание на атрибут метода `Triggerl` и его свойства. Свойство `Name` указывает имя триггера, свойство `Target` — таблицу, к которой будет относиться данный триггер, а свойство `Event` — тип триггера и событие, на которое должен реагировать триггер.

Регистрация триггера осуществляется так же, как и хранимой процедуры:

```
CREATE function MyTrigger on dbo.Users AFTER INSERT
AS
external name MyProcs.Triggers.Trigger1
GO
```

Агрегирующие функции

Вы также можете создать собственную агрегирующую функцию. Далее представлен пример агрегирующей функции для вычисления среднего значения. Эта функция аналогична встроенной функции `avg`.

```
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct NewAvg
(
    public void Init()
    {
        sum = 0;
        count = 0;
    }

    public void Accumulate(SqlDouble s)
    {
        sum += s;
        count ++;
    }
    public void Merge(NewAvg Group)
    {
        sum += Group.sum;
        count ++;
    }

    public SqlDouble Terminate()
    {
```

```
        return sum/count;
    }

    private SqlDouble sum;
    private Int32 count;
}
```

Структура, описывающая агрегирующую функцию, должна содержать четыре обязательных функции:

- `Init` — эта функция вызывается перед началом вычисления агрегирующей функции и используется для инициализации значений переменных с целью вычисления;
- `Accumulate` — эта функция предназначена для накапливания агрегирующих значений (сумм, счетчиков и т. п.);
- `Merge` — эта функция предназначена для объединения текущего экземпляра агрегата с передаваемым в качестве параметра;
- `Terminate` — эта функция вызывается в конце вычисления и используется для формирования окончательного результата и возвращения его как результата.

Регистрация агрегирующей функции очень похожа на регистрацию хранимых процедур или пользовательских функций:

```
CREATE aggregate NewAvg (@val float)
returns float
AS
external name MyProcs.NewAvg
GO
```

После создания агрегирующей функции к ней можно обращаться из любого запроса. Например:

```
SELECT NewAvg('Возраст') FROM Users
```

Пользовательские типы данных

Вы можете создать собственный новый тип данных — *пользовательский тип данных*, который затем можно применять как при определении столбцов таблиц, так и при декларировании переменных при использовании T-SQL. Новый тип может иметь свои свойства и методы, что значительно расширяет функциональность программирования на стороне сервера.

Добавим к нашему проекту класс пользовательского типа данных, выбрав **Add User-defined Type** из меню **Project** в Visual Studio.

Пример пользовательского типа данных:

```
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]
public struct MyPoint : INullable
{
    private int _X;

    // Координата X
    public int X
    {
        get { return _X; }
        set { _X = value; }
    }

    private int _Y;

    // Координата Y
    public int Y
    {
        get { return _Y; }
        set { _Y = value; }
    }

    private bool _Null;

    public bool IsNull
    {
        get { return _Null; }
        set { _Null = value; }

    }
    public static MyPoint Null
    {
        get
        {
            MyPoint o = new MyPoint();
            o._Null = true;
            return o;
```

```
        }
    }

    public override string ToString()
    {
        if (this.IsNull)
            return "NULL";
        return X.ToString() + "-" + Y.ToString();
    }

    public static MyPoint Parse(SqlString s)
    {
        if (s.IsNull) return Null;

        MyPoint o = new MyPoint();
        String ss = s.ToString();
        String[] arr = ss.Split(new Char[]{"-"}, 2);
        o.X = int.Parse(arr[0]);
        o.Y = int.Parse(arr[1]);
        o.IsNull = false;
        return o;
    }

    public int GetXDistance(MyPoint o)
    {
        if (this.IsNull || o.IsNull)
            return 0;

        return this.X - o.X;
    }

    public int GetYDistance(MyPoint o)
    {
        if (this.IsNull || o.IsNull)
            return 0;
        return this.Y - o.Y;
    }
}
```

Этот пользовательский тип данных представляет собой точку с координатами по осям *x* и *y*. Также этот тип содержит функции `GetXDistance` и `GetYDistance` для расчета расстояния между двумя точками.

Структура, на основе которой создается пользовательский тип данных, всегда является наследником интерфейса `INullable`. Обратите внимание на обязательные свойства `Null` и `IsNull`, которые определяют, присвоено или нет переменной данного типа значение `NULL`. Без этих свойств невозможно зарегистрировать новый тип данных.

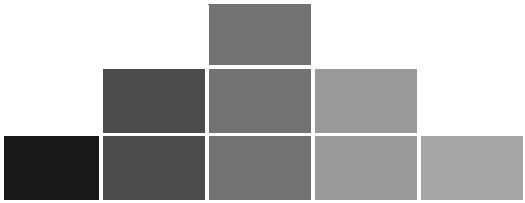
Также в пользовательском типе должны присутствовать метод `ToString`, преобразующий данный тип к строковому представлению, и метод `Parse` обратного преобразования из строки к данному типу.

Для регистрации пользовательского типа в SQL Server используйте следующую команду:

```
CREATE type MyPoint
external name MyProcs.MyPoint
GO
```

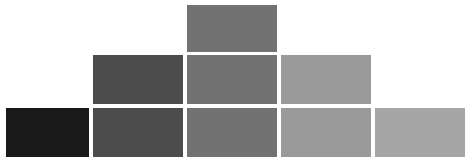
Теперь созданный тип может быть использован как тип переменной в коде T-SQL, а также как тип столбца таблицы. Например:

```
DECLARE @point MyPoint
DECLARE @point2 MyPoint
SET @point = '15-54'

-- Или другой вариант инициализации
SET @point2.X = 15
SET @point2.Y = 86

-- Вы также можете вызывать функции данного типа
PRINT @point.GetXDistance(@point2)
```



ЧАСТЬ III

Администрирование SQL Server



ГЛАВА 18

Безопасность

В этой главе описаны реализованные в SQL Server функции защиты, о которых нужно знать разработчикам.

Пользователи

Для того чтобы получить доступ к какой-либо из баз данных, следует вначале создать учетную запись. В случае Windows-аутентификации учетная запись берется из списка локальных пользователей компьютера или из списка пользователей домена. Причем учетной записью может быть и группа пользователей. Это бывает очень удобно, т. к. одним действием предоставляется доступ к серверу целой группе пользователей.

Если предполагается смешанная аутентификация (когда наряду с Windows-аутентификацией возможно задание логина и пароля определенного пользователя прямо в SQL Server), то следует указать имя учетной записи и ее пароль. Для учетной записи можно указать одну из девяти встроенных ролей, которые определяют возможность выполнять определенный набор действий на уровне сервера, но об этом далее.

После создания учетной записи следует создать пользователя базы данных, доступ к которой будет разрешен данной учетной записи, и увязать пользователя с этой учетной записью. После создания пользователя можно указать, что пользователь будет являться членом одной из ролей базы данных.

Также вы можете создать пользователя с помощью SQL Server Management Studio. Для этого откройте список пользователей базы данных, выбрав **Security | Users**, а потом щелкните правой кнопкой мыши на этом списке и выберите пункт **New User**. Появится окно, как на рис. 18.1. Введите имя пользователя, его роль и выберите Windows-пользователя для него. Нажмите кнопку **OK** — и пользователь базы данных будет создан.

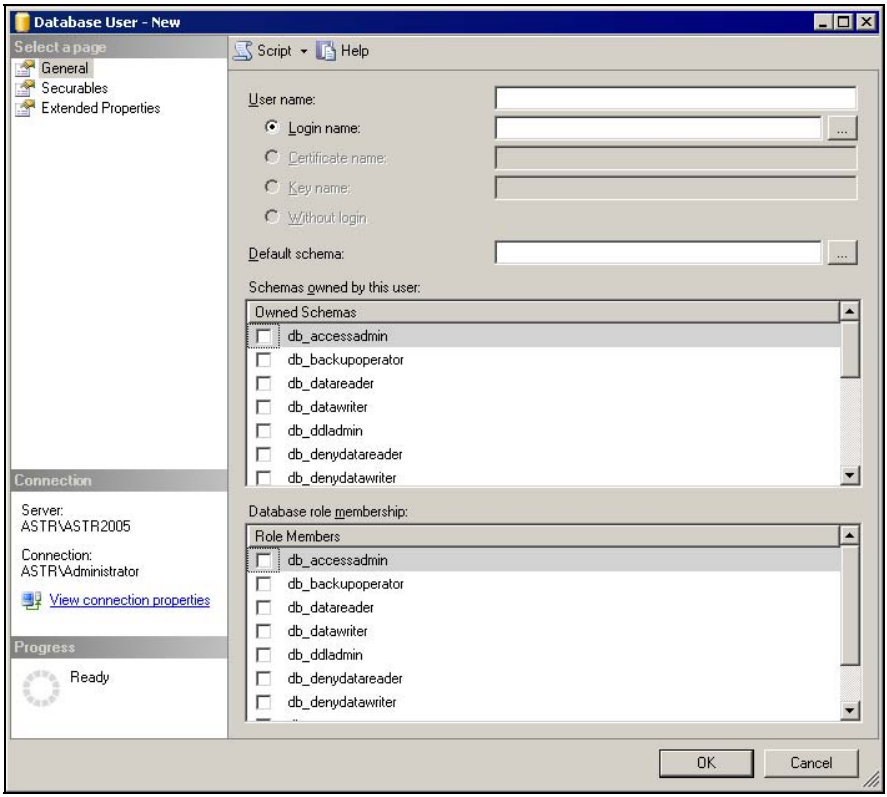


Рис. 18.1. Диалог создания нового пользователя базы данных

Роли

Роль в SQL Server похожа на группу пользователей. Роль позволяет объединять пользователей, выполняющих одинаковые функции (т. е. играющие одинаковые роли), для упрощения администрирования SQL Server. Существуют роли уровня сервера и уровня базы данных. При установке на уровне сервера создается 9 предопределенных фиксированных ролей. При создании базы данных также создается 10 предопределенных фиксированных ролей.

Кратко рассмотрим фиксированные роли на уровне сервера:

- `sysadmin` — разрешено выполнять любые действия в SQL Server;
- `setupadmin` — управление связанными серверами;
- `serveradmin` — доступно конфигурирование и выключение сервера;
- `securityadmin` — управляет учетными записями и правами на создание базы данных;

- `processadmin` — может управлять процессами, запущенными в SQL Server;
- `diskadmin` — управляет файлами SQL Server;
- `dbcreator` — разрешено создавать и модифицировать базы данных;
- `bulkadmin` — может управлять процессом массовой вставки данных.

Фиксированные роли базы данных:

- `db_accessadmin` — может удалять или добавлять пользователей базы данных;
- `db_backupoperator` — может выполнять операцию резервного копирования;
- `db datareader` — может просматривать содержимое таблиц базы данных;
- `db_datawriter` — может изменять данные таблиц;
- `db_ddladmin` — может выполнять команды DDL (команды SQL для создания таблиц и структур баз данных);
- `db_denydatareader` — запрещается просматривать данные в любой таблице;
- `db_denydatawriter` — запрещается модифицировать данные в любой таблице;
- `db_owner` — обладает всеми правами в базе данных (собственник базы данных);
- `db_securityadmin` — может управлять ролями и разрешениями на уровне базы данных;
- `public` — любой пользователь базы данных автоматически становится членом этой роли. Данную роль используют для предоставления минимальных прав пользователю.

Кроме использования встроенных ролей, можно создавать свои роли, причем как для пользователей, так и для приложений. Вы можете создать роль с помощью SQL Server Management Studio.

Для каждого пользователя базы данных и для каждой роли можно явно прописать права и запреты для конкретных объектов базы данных. Более того, можно прописывать права и ограничения даже на уровне отдельных столбцов таблицы. Но гораздо проще и лучше использовать схемы для этих целей.

Схемы

Вы, должно быть, знакомы с концепцией групп и ролей, служащих для управления пользователями в Windows. Вкратце, на пользователя, относящегося к определенной группе, распространяются все права и ограничения этой группы.

Схема — это средство группировки объектов (не пользователей), позволяющее обращаться с набором объектов как с единым целым (в смысле владения и прав).

Например, вы можете при помощи одного оператора T-SQL предоставить роли право на выполнение всех хранимых процедур, относящихся к определенной схеме. Это очень удобно, если вы имеете дело с сотнями хранимых процедур.

Вот некоторая информация о схемах:

- для каждого пользователя базы данных определяется своя схема по умолчанию. При создании пользователем объекта базы данных этот объект по умолчанию помещается в данную схему;
- каждый объект базы данных, за исключением объектов безопасности, относится к какой-либо схеме;
- при создании объекта можно явно указать схему, в которую его нужно поместить. В базе данных могут существовать объекты с одинаковыми именами, но принадлежащие разным схемам. Но одинаковые имена объектов в одной схеме не допускаются;
- для пользователей и ролей, для облегчения администрирования указываются определенные права на объекты той или иной схемы.

Отделение пользователей от схем, начиная с SQL Server 2005, дает вам некоторые преимущества:

- более легкое удаление пользователей. Пользователь не владеет объектами базы данных. Если в SQL Server 2000 пользователь владел каким-либо объектом базы, и вы хотели его удалить, вы должны были сначала передать эти объекты во владение другому пользователю, что сильно усложняло процесс удаления. Начиная с SQL Server 2005, объекты назначаются схемам, а не пользователям. Пользователь может иметь схемы по умолчанию, не владея на самом деле этими схемами. А поскольку пользователи не владеют никакими схемами и объектами, удалить их проще;
- схемой (и ее объектами) могут управлять несколько пользователей.

Использование схемы

Перед использованием схемы ее нужно создать, что легко сделать с помощью оператора `CREATE SCHEMA`:

```
CREATE SCHEMA MySchema
```

Этот простой оператор создает схему `MySchema`, которой владеет пользователь, выполняющий данный оператор. После создания схемы вы можете

использовать ее двумя способами. Первый: он просто требует, чтобы при создании объектов в БД вы указывали имя схемы. Например, следующий код создает в схеме `MySchema` таблицу `MyNewTable`. При создании этой таблицы используется явная нотация *`схема.объект`*.

```
CREATE TABLE MySchema.MyNewTable (ID int, Name nvarchar(100))
```

Второй путь основан на использовании схемы по умолчанию и потому не требует указания схемы в операторе создания объекта. Пример создания объекта:

```
CREATE TABLEMyNewTable (ID int, Name nvarchar(100))
```

В этом примере таблица создается в схеме по умолчанию, соответствующей пользователю, который выполняет данный оператор. Например, если этот код выполняет пользователь `Alex` и его схемой по умолчанию является схема `Staff`, то таблица создается в схеме `Staff` и владельцем объекта будет схема `Staff`.

> ***ВНИМАНИЕ!***
>
> В SQL Server 2000 не было различия между пользователями и схемами, поэтому "схемой по умолчанию" был пользователь. Начиная с SQL Server 2005, пользователи отделены от схем, поэтому вы можете связать нескольких пользователей с одной схемой по умолчанию. Лучше всегда создавать объекты явно — это исключит многие потенциальные затруднения.

Доступ к объектам

Если пользователю назначена схема по умолчанию, при ссылке на объект из этой схемы ее имя указывать не нужно. Если есть разные схемы, то при обращении к объекту без указания его схемы порядок действий таков:

1. SQL Server проверяет, существует ли объект в схеме `sys`. Если да, то он использует этот объект.
2. SQL Server проверяет, существует ли объект в назначенной пользователю схеме по умолчанию. Если да, то он использует этот объект.
3. SQL Server проверяет, существует ли объект в схеме `dbo`. Если да, то он использует этот объект.

То есть, если пользователь `Alex` имеет схему по умолчанию `Staff`, а таблица `Contact` существует в схемах `Staff` и `dbo`, получается следующее:

1. Проверяется, есть ли таблица `Contact` в схеме `sys`. Если нет, SQL Server продолжает проверку.
2. Проверяется, есть ли таблица `Contact` в схеме `Staff`. Если да, используется эта найденная таблица.

Вот другой пример. Если таблица `Product` существует в схемах `Production` и `dbo`:

1. Проверяется, есть ли таблица `Product` в схеме `sys`. Если нет, проверка продолжается.
2. Проверяется, есть ли таблица `Product` в схеме `Staff`. Если нет, проверка продолжается.
3. Проверяется, есть ли таблица `Product` в схеме `dbo`. Если да, используется найденная таблица.

Если происходит запрос к таблице `Employee`, которая существует только в схеме `Company`, происходит следующее:

1. Проверяется, есть ли таблица `Employee` в схеме `sys`. Если нет, проверка продолжается.
2. Есть ли таблица `Employee` в схеме `Staff`. Если нет, проверка продолжается.
3. Есть ли таблица `Employee` в схеме `dbo`. Если нет, происходит ошибка.

В этом примере схемой по умолчанию для пользователя `Alex` является схема `Staff`, а таблица `Employee` существует только в схеме `Company`. Поэтому в запросе необходимо было явно указать, к какой схеме относится таблица.

Чтобы у вас не возникали вопросы о том, к какому объекту вы обращаетесь на самом деле, при наличии нескольких схем придерживайтесь явной методики обращения к объектам базы данных.

Права

Если в SQL Server 2000 вы хотели предоставить владельцу хранимых процедур право на их выполнение, а также позволить ему выполнять выборку данных из принадлежащих ему таблиц, то вы должны были назначить роли права в отношении каждого объекта. Покажем, как это было бы:

```
GRANT EXECUTE ON MyProcedure TO MyDatabaseRole
GRANT SELECT ON MyTable1 TO MyDatabaseRole
```

При большом количестве таблиц и процедур этот способ становится очень громоздким, а риск ошибиться — очень высоким. Но начиная с SQL Server 2005, вы можете сгруппировать объекты в схему и назначить права самой схеме. Например, оператор `GRANT` предоставляет пользователю `Alex` право на выборку данных из всех объектов:

```
GRANT SELECT, EXECUTE ON Schema::Sales TO Alex
```

Эта новая возможность появилась, начиная с SQL Server 2005. Кроме того, как и в предыдущих версиях, SQL Server позволяет назначать права и ролям, только теперь вы можете сделать это на уровне схемы:

```
GRANT SELECT, EXECUTE ON Schema::Sales TO MyDatabaseRole
```

С помощью одного простого оператора вы можете назначить группе пользователей все необходимые права в отношении набора объектов БД.

Синонимы

Начиная с SQL Server 2005, появились *синонимы*, позволяющие создавать постоянные псевдонимы для объектов базы данных.

Например, у нас есть представление:

```
CREATE VIEW MyView
    SELECT ProductID, ProductName
    FROM Production.Product
```

В этом примере создается представление, использующее таблицу `Product` из схемы `Production`.

Но вы можете создать синоним для этой таблицы, чтобы было проще к ней обращаться:

```
CREATE SYNONYM synProduct FOR Production.Product
```

Теперь вы можете заменить обращение к таблице обращением к синониму, как показано:

```
ALTER VIEW MyView
    SELECT ProductID, ProductName
    FROM synProduct
```

Конструкция *EXECUTE AS*

Эта конструкция появилась еще в SQL Server 2005. Она позволяет изменять контекст безопасности, в котором выполняется команда. Конечно, пользователь, желающий выполнить метод, должен иметь право на его выполнение. В начале выполнения метода контекст безопасности изменяется на новый контекст, определяемый конструкцией `EXECUTE AS`.

Конструкцию `EXECUTE AS` можно использовать с хранимыми процедурами и пользовательскими функциями (за исключением встраиваемых табличных функций).

Допустим, что Alex имеет право на выполнение хранимой процедуры Sales.GetOrderList и выполняет ее. Схемой Sales владеет Ann. Эта процедура обращается к ряду таблиц, относящихся к нескольким разным схемам. Всеми этими схемами владеет Ann. Хотя Alex не имеет права на непосредственную выборку данных из этих таблиц, благодаря цепочке владения и тому факту, что все нужные схемы принадлежат одному владельцу, процедура выполняется без проблем.

Однако, если одна из используемых в процедуре таблиц будет относиться к схеме, не принадлежащей Ann, цепочка владения оборвется, и для выборки данных из этой таблицы Alex должен будет получить дополнительные права.

Конструкция *EXECUTE AS CALLER*

При использовании этого варианта метод выполняется в контексте вызвавшего его пользователя. EXECUTE AS CALLER соответствует также контексту выполнения по умолчанию, поэтому при создании хранимой процедуры или пользовательской функции эту конструкцию можно не указывать. Приведем пример хранимой процедуры без заданного контекста выполнения:

```
CREATE PROCEDURE procProductDetails (@ProductID int)
AS
SELECT ProductID, ProductName
FROM Production.Product
WHERE ProductID=@ProductID
```

Отсутствие конструкции EXECUTE AS CALLER идентично ее наличию:

```
CREATE PROCEDURE procProductDetails (@ProductID int)
WITH EXECUTE AS CALLER
AS
SELECT ProductID, ProductName
FROM Production.Product
WHERE ProductID=@ProductID
```

Иначе говоря, если вы хотите, чтобы метод выполнялся в контексте вызвавшего пользователя, вы можете включить в код конструкцию EXECUTE AS CALLER или просто опустить ее.

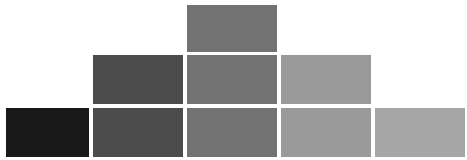
Конструкция *EXECUTE AS* в контексте пользователя

Этот вариант конструкции `EXECUTE AS` позволяет выполнить метод в контексте заданного пользователя. Приведем пример использования этой конструкции:

```
CREATE PROCEDURE procProductDetails (@ProductID int)
WITH EXECUTE AS 'Ann'
AS
SELECT ProductID, ProductName
FROM Production.Product
WHERE ProductID=@ProductID
```

Когда кто-то вызывает эту процедуру, она выполняется так, как если бы ее вызвал пользователь `Ann`. Если в коде процедуры осуществляется доступ к каким-либо другим объектам базы данных, то все будет работать так, как если бы к ним пыталась обратиться `Ann`, при этом используются права `Ann`. Например, если `Ann` имеет право на выборку данных из таблицы `Product`, то при вызове процедуры пользователем `Alex` доступ к таблице `Product` будет выполнен так, как если бы к ней обратилась `Ann`.

Без конструкции `EXECUTE AS` пользователю, выполняющему эту процедуру, потребовалось бы дополнительное право на выборку данных из таблицы `Product`, но т. к. обращение к этой таблице выполняется в контексте пользователя `Ann`, обрыва цепочки владения не происходит. Пока `Ann` принадлежит право на выборку данных из таблицы `Product`, любой пользователь, имеющий право на выполнение этой хранимой процедуры, может использовать ее без каких-либо нарушений защиты.



ГЛАВА 19

SQL Server Agent

С помощью службы SQL Server Agent вы можете автоматизировать администрирование SQL Server. Управлять работой службы SQL Server Agent вы можете с помощью SQL Server Management Studio, открыв группу свойств сервера **SQL Server Agent**. Далее будут рассмотрены компоненты службы SQL Server Agent.

Задачи

Задачи (jobs) — это административные операции, которые можно определить один раз и затем выполнять многократно по определенному расписанию. Можно осуществлять контроль, успешно ли была завершена задача или ее выполнение было прервано вследствие возникшей ошибки.

Операторы

Оператор — это сотрудник, ответственный за поддержку SQL Server. В некоторых случаях все задачи выполняет один оператор. В крупных системах с несколькими серверами задачи распределены между несколькими операторами.

Операторы могут получать оповещения несколькими способами: через электронную почту, на пейджер или средствами сети (в этом случае будет создано диалоговое окно или окно сообщения). Оповещение может быть отправлено группе сотрудников.

Оповещения

Оповещение (alert) — это определенный системным администратором ответ на некоторое событие в SQL Server.

Обычно администратор не может контролировать возникновение события, но при его наступлении он может принять соответствующие меры. Оповещения могут быть использованы для того, чтобы проинформировать одного или нескольких операторов или выполнить задачу.

Этапы настройки автоматизированного администрирования

Настройка автоматизированного администрирования включает в себя несколько главных этапов:

1. Сначала вы должны определить себя как оператора.
2. Создайте задачу, которая будет выполнять некоторые команды T-SQL, и установите график выполнения ее. Например, вы можете создать задачу.
3. Настройте задачу таким образом, чтобы она извещала вас (например, по электронной почте), а также записывала сообщение о событии в журнал событий Windows всякий раз при возникновении сбоя при выполнении задачи.
4. После завершения всех предыдущих шагов запустите службу SQL Server Agent (эта служба должна работать для выполнения задач автоматизированного администрирования).

Для настройки задач, операторов и оповещений можно использовать SQL Server Management Studio.

> ***ВНИМАНИЕ!***
>
> Задачи автоматизированного администрирования не будут выполнены, если не запущен SQL Server Agent.

Настройка задач

Чтобы создать задачу, необходимо сделать следующее:

1. Вызовите контекстное меню пункта **SQL Server Agent | Jobs** и выберите пункт **New Job**. Появится диалог, как на рис. 19.1.
2. В появившемся диалоге введите имя задачи.
3. Перейдите на страницу **Steps** и добавьте шаги задачи. Диалог добавления шага задачи показан на рис. 19.2. Введите имя шага, выберите базу, с которой он будет работать, а также введите код T-SQL, который будет выполнен. Во вкладке **Advanced** диалога добавления шага задачи вы можете

выбрать, с правами какого пользователя будет выполнен этот код. А также вы можете выбрать, что будет делать SQL Server в случае успешного или неуспешного выполнения шага задачи.

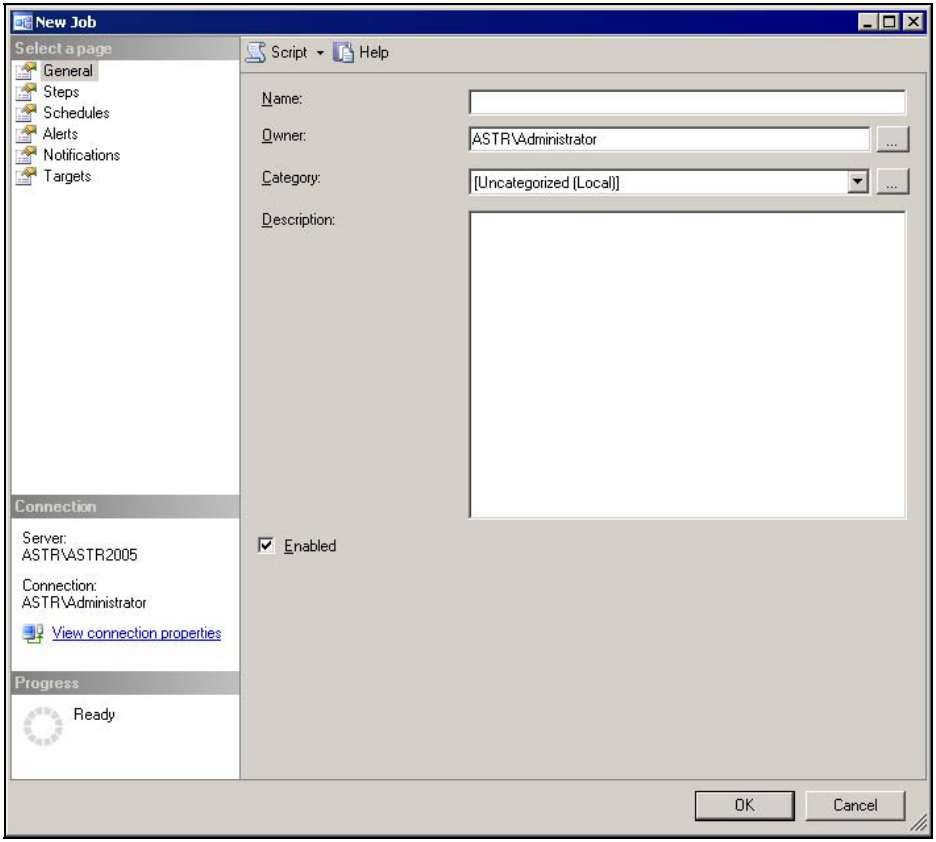


Рис. 19.1. Диалог свойств задачи в SQL Server Agent

4. После определения шагов задачи вам необходимо определить расписание, когда эта задача будет выполняться. Вы можете увидеть диалог определения расписания задачи на рис. 19.3.
5. После этого вы можете настроить оповещения, которые будут посланы при успешном или неудачном выполнении задачи.
6. Сохраните вновь созданную задачу, нажав кнопку **OK**.

После того как вы создадите задачу, можете выбрать пункт **Start Job** из контекстного меню для выполнения задачи.

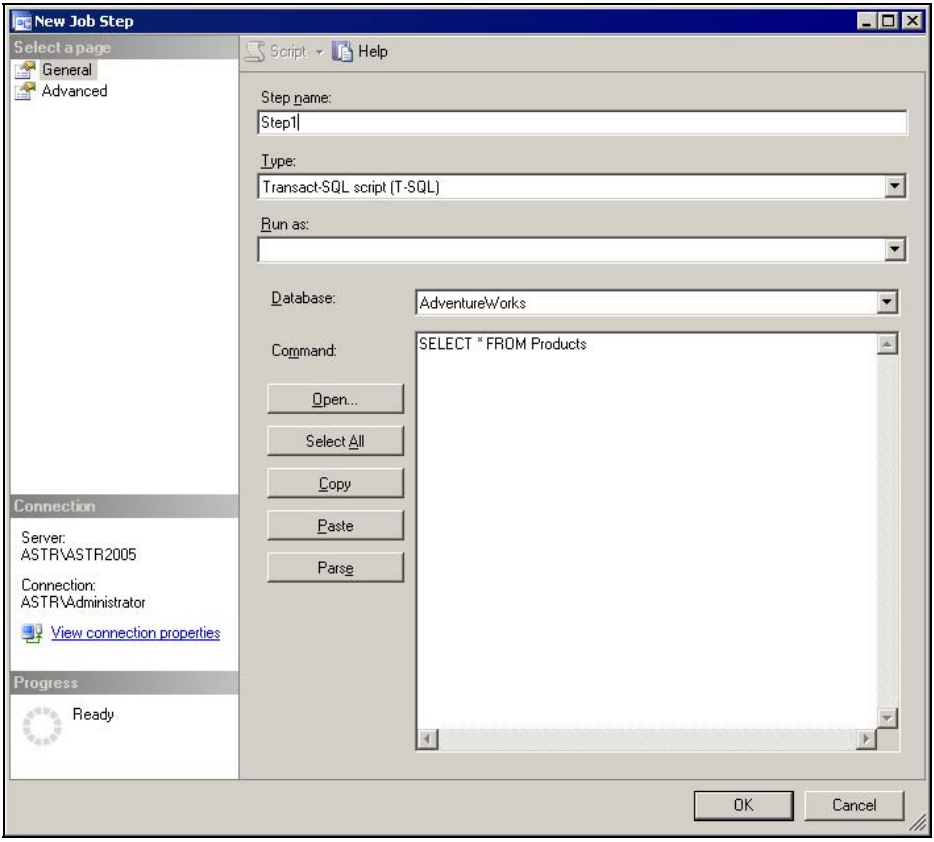


Рис. 19.2. Диалог свойств шага задачи в SQL Server Agent

Настройка операторов

Все, что нужно, чтобы создать оператора, — это указать его имя и информацию о том, каким образом ему можно передавать сообщения. Каждый оператор должен иметь уникальное имя.

Настраивайте операторы перед тем, как настроите оповещения. В противном случае вам придется вернуться к настройке оповещений.

Операторы могут получать оповещения посредством электронной почты, пейджера или сетевых сообщений. Посылка сообщений на пейджер производится через электронную почту. Диалог настройки оператора вы можете видеть на рис. 19.4.

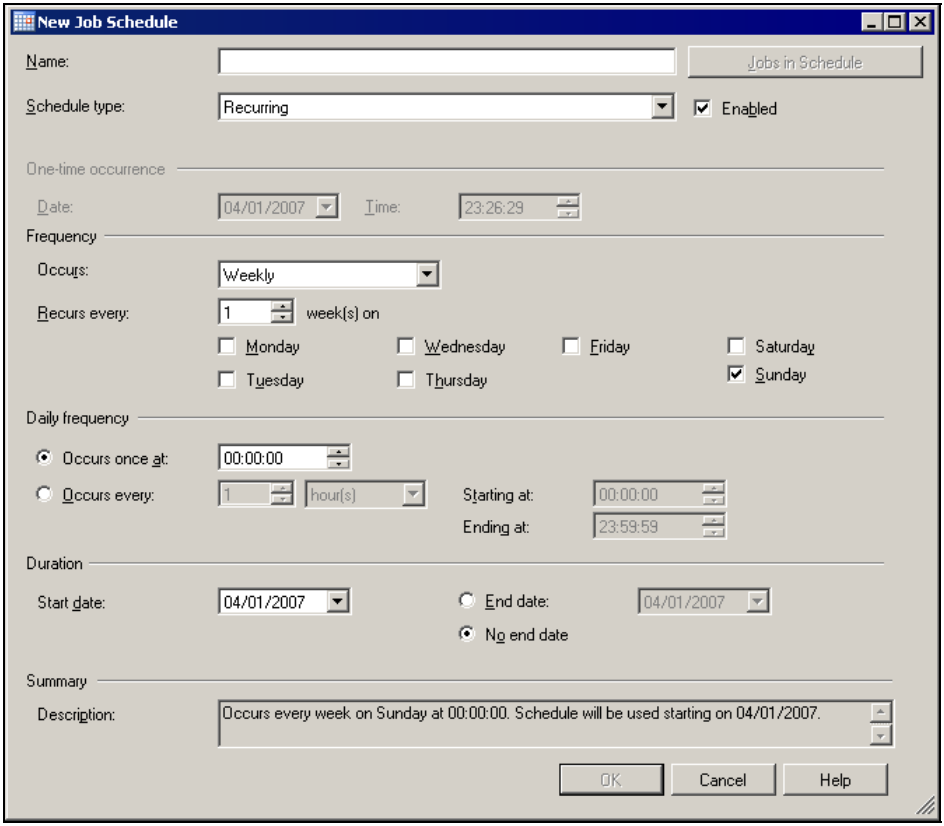


Рис. 19.3. Диалог определения расписания выполнения задачи в SQL Server Agent

Настройка оповещений

Ошибки, сообщения или события генерируются SQL Server и записываются во вкладку **Applications** журнала событий Windows. SQL Server Agent может читать журнал событий и определять, соответствует ли какому-либо из событий заранее указанное оповещение. Если такое соответствие найдено, посылается оповещение.

Перед отправкой оповещения должны быть настроены. Главное, что нужно указать для оповещения, — это его название и событие, в ответ на которое оно будет послано.

Каждому оповещению должно соответствовать уникальное название. Сам диалог настройки оповещения представлен на рис. 19.5.

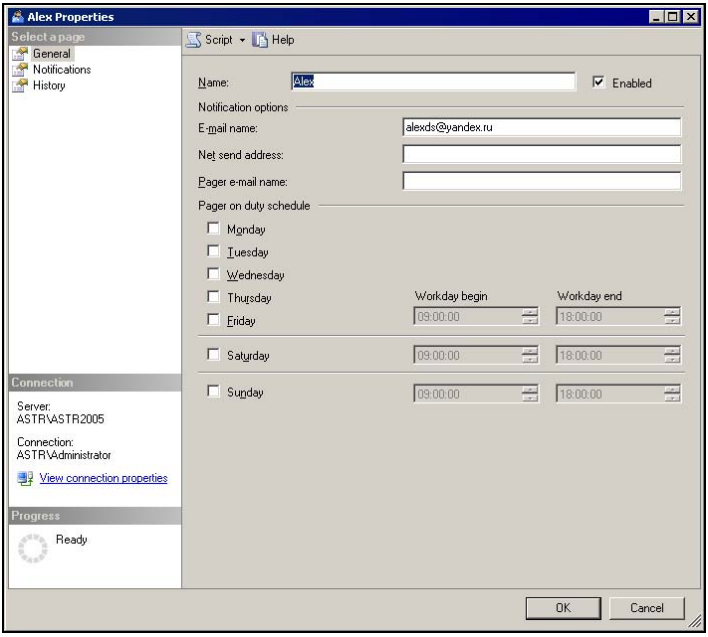


Рис. 19.4. Диалог настройки свойств оператора в SQL Server Agent

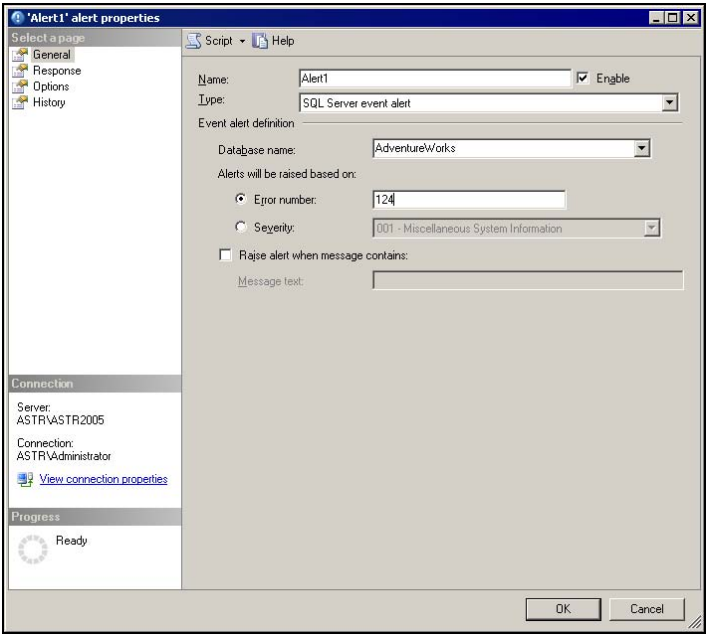
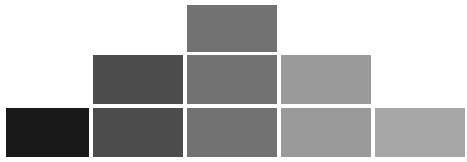


Рис. 19.5. Диалог настройки оповещения в SQL Server Agent



ГЛАВА 20

Репликации

Репликация — это механизм поддержания данных SQL Server в синхронизированном состоянии с данными на другом SQL Server (посредством дублирования). Эти базы данных могут находиться на том же сервере, что и база данных-источник, или же на других серверах. И SQL Server обеспечивает, чтобы данные на всех серверах были одинаковые. Вне зависимости от того, на каком сервере происходило изменение данных, об этом станет известно всем серверам, которых затрагивает репликация.

В этой главе кратко описывается модель репликации с издателем и подписчиком, а также три типа репликации — репликация моментальных снимков, репликация транзакций и репликация сведением.

Модель репликации с издателем и подписчиком

В SQL Server используется модель репликации с издателем и подписчиком. Эта модель состоит из трех следующих элементов:

- *издатель* (Publisher) — сервер, который хранит данные, предназначенные для репликации на другие базы данных;
- *распределитель* (Distributor) — сервер, хранящий специальную базу данных распределения (distribution database), которая содержит в себе распределяемые данные и действует в качестве промежуточного звена между издателем и подписчиком;
- *подписчик* (Subscriber) — сервер, на котором хранится копия публикуемой информации. Подписчик получает все изменения для опубликованных данных.

Подписчики также могут участвовать в изменении данных. При этом информация о таких изменениях посылается на сервер-издатель (через сервер-распределитель).

SQL Server использует модель издателя и подписчика для поддержки следующих трех типов репликации:

- *репликация моментальных снимков* (snapshot replication) — это тип репликации, при котором производится периодическое формирование моментальных снимков (полной копии данных, выбранных для репликации) публикуемых данных и применение их к базе данных подписчиков;
- *репликация транзакций* (transactional replication) — это тип репликации, при котором передаются не наборы данных целиком, а только лишь информация о сделанных изменениях;
- *репликация сведением* (merge replication) — это тип репликации, при котором данные могут изменяться в нескольких базах и информация об этом сводится вместе в центральной базе данных.

В своей работе репликации интенсивно используют *журнал транзакций* — специальный файл базы данных, в который заносятся все изменения, перед тем как они будут внесены непосредственно в базу данных.

Репликация моментальных снимков

В этом типе репликации используются так называемые моментальные снимки (snapshot). Моментальный снимок — это копия структуры публикуемых объектов и хранящихся в них данных. В моментальном снимке записывается структура и данные. Затем моментальный снимок посылается подписчикам. Подписчики получают не изменения к уже имеющимся данным, а полностью новый набор данных, который заменяет прежний. При этом старые данные уничтожаются и на их место записываются новые. Для наглядности можно представить, что при этом типе репликации старая фотография убирается из фотоальбома и заменяется новой.

Репликация транзакций

При репликации транзакций передаются не наборы данных целиком, а только лишь информация о сделанных изменениях.

Подписчикам предоставляется начальный набор данных. После этого специальный агент просматривает журнал транзакций на предмет записей типа `INSERT`, `UPDATE` или `DELETE`, относящихся к опубликованным объектам, и осу-

ществляет их пакетное применение к базе данных распределения. Этот агент может работать постоянно или по графику, определенному администратором.

Изменения хранятся в базе данных распределения до того момента, пока их не перешлют подписчикам. Подписчики могут после этого вносить изменения в своей базе данных.

Преимущество репликации транзакций заключается в том, что подписчикам передаются только изменения, но не целые таблицы.

Репликация сведением

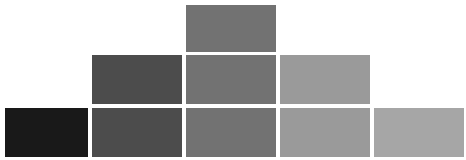
Репликация сведением — это особый тип репликации, при котором данные могут изменяться в нескольких базах и информация об этом сводится вместе в центральной базе данных.

В репликации моментальных снимков и репликации транзакций база данных издателя является источником всех изменений. При репликации сведением изменения в базе-источнике отслеживаются и затем синхронизируются с базами данных издателя и подписчика. Так же, как и при репликации моментальных снимков и репликации транзакций, в процессе репликации записывается структура и данные базы данных подписчика, которые передаются распределителю.

Как и при репликации моментальных снимков, сначала получаются начальные данные для передачи. Однако при репликации сведением изменения согласовываются между издателем и подписчиками.

Здесь лишь основные данные о репликациях в SQL Server: можно написать о них целую книгу. Используя механизм репликации, вы можете очень просто и с минимальными усилиями синхронизировать данные между центральной базой и базами подписчиками, которые могут находиться на другом конце земного шара.

Вы можете легко настроить все, что связано с репликацией, посредством SQL Server Management Studio и с помощью группы свойств **Replication**.



ГЛАВА 21

SQL Server Reports services

В SQL Server 2008 (как и в версии 2005) встроены службы отчетов. С помощью этих служб можно создать интерактивные, табличные, графические отчеты и отчеты свободной формы из реляционных, многомерных и XML-источников данных. Можно публиковать отчеты, планировать их обработку или осуществлять доступ к отчетам по требованию.

В этой главе изложены лишь основы работы со службами отчетов. Для описания некоторых из них может потребоваться отдельная книга. Далее представлен список этих служб с их кратким описанием.

Создание отчетов

Для создания отчета необходимо создать файл определения отчета с помощью конструктора или построителя отчетов. С SQL Server поставляется Business Intelligence Development Studio, с помощью которой вы можете графически конструировать отчеты (рис. 21.1).

Отчеты представляют собой файлы, хранимые в проекте отчета. Проект отчета выступает в качестве контейнера для самих отчетов и дополнительных ресурсов. Когда проект развернут, каждый файл проекта отчета публикуется на сервере отчетов.

Сами отчеты создаются на компьютере-клиенте, независимом от сервера отчетов. Когда отчет создан, его можно опубликовать на сервере отчетов или узле SharePoint, где им можно пользоваться. Отчеты также можно экспортировать на локальный компьютер в различных форматах файлов, например TIFF, PDF, Microsoft Office Excel или HTML.

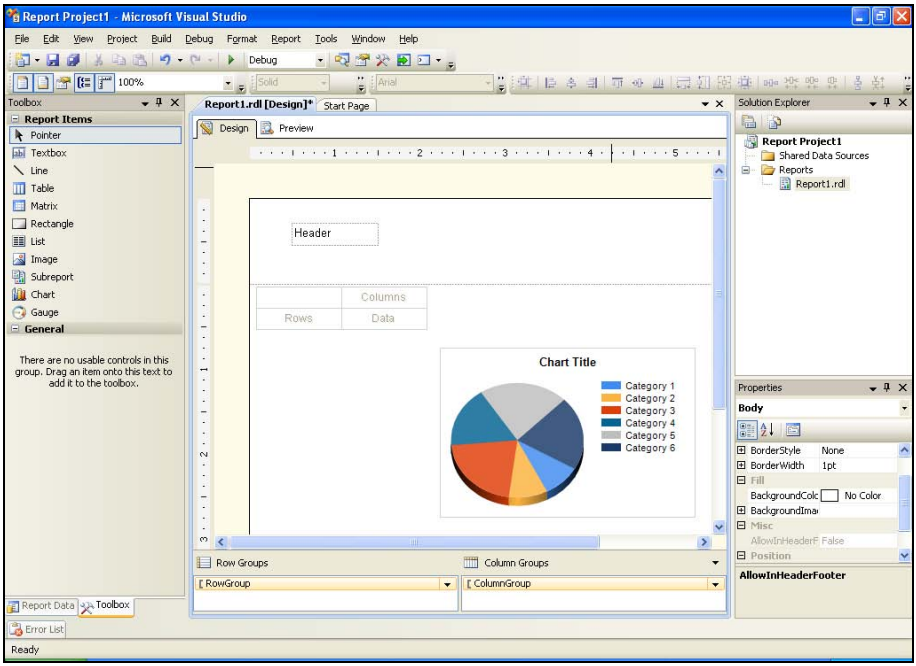


Рис. 21.1. Конструирование отчета с помощью Business Intelligence Development Studio

Основы конструирования отчетов

Чтобы создать отчет, необходимо указать данные для отчета, способ их организации на странице и способ просмотра отчета пользователем.

Отчет может содержать верхний колонтитул, текст и нижний колонтитул. Обычно по умолчанию он имеет размер страницы, включающий верхний и нижний колонтитулы. В колонтитулах можно размещать такие элементы отчета, как изображения, текстовые поля и линии. При этом основная часть отчета, текст, содержит данные.

В тексте отчета можно разместить элементы отчета любого типа, в том числе таблицы, матрицы, списки, диаграммы и датчики. При этом при конструировании отчета данные связываются с элементами отчета.

При непосредственном создании отчета для его просмотра данные и элементы макета объединяются. У отчета могут быть параметры, применяемые при формировании отчета для определения данных, которые будут использоваться в отчете, или других его параметров.

Для организации данных на странице отчета используются разнообразные элементы отчета. Эти элементы размещены в области элементов конструктора

отчетов. Их можно перетащить непосредственно в область отчета, а затем перетащить данные отчета из области "Данные отчета" на элементы. Вы даже можете перенести всю таблицу данных целиком.

Область конструктора отчетов может не совпадать с тем, как отчет выглядит при просмотре. Элементы отчета при конструировании имеют определенное положение на макете, которое может измениться при построении отчета.

Далее представлены основные элементы отчета.

- *Текстовое поле*. Используется для заголовков, меток даты и имен отчетов. Пространство текстового поля заполняется текстом из связанных с отчетом данных. Также текстовое поле может заполняться вычисляемыми значениями.
- *Таблица*, *матрица*. Предназначены для отображения табличных или матричных данных из набора данных отчета. При этом данные могут группироваться и могут рассчитываться суммарные значения.
- *Диаграмма*. Используется для графического отображения данных из набора данных отчета.
- *Датчик*. Используется для визуального изображения одного значения в диапазоне значений. При этом это отображается как прибор со шкалой.
- *Список*. Служит для создания списка свободной формы.
- *Изображение*. Используется для добавления в отчет существующих изображений.
- *Линия*, *прямоугольник*. Используются как графические элементы для оформления отчета.

Вы можете настроить параметры отображения всех элементов: их цвет, цвет фона, тип шрифта и т. д.

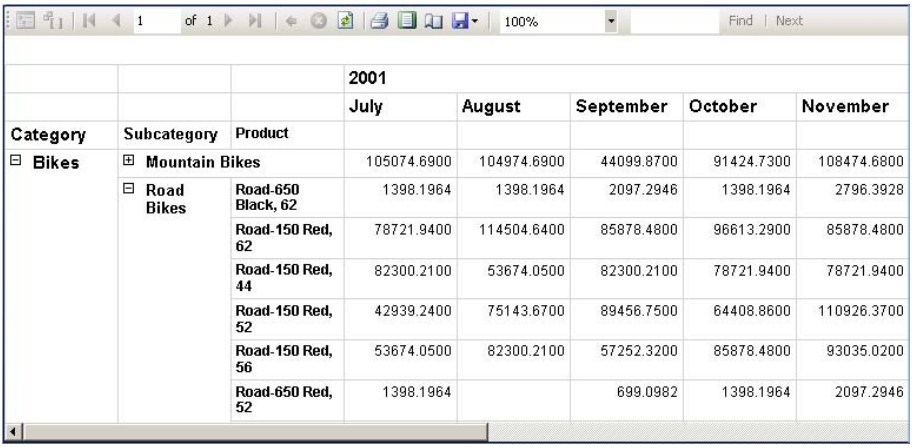
В любом отчете может содержаться вложенный отчет. Например, это можно использовать так: в основном отчете выводится список магазинов, а во вложенном отчете — список наиболее продаваемых товаров.

Обычно отчеты состоят из этих нескольких элементов. В типичном отчете в основном используются *табликсы* (так называются элементы, которые выводят данные в табличном виде: таблица и матрица).

Можно перетащить поля из области "Данные отчета" и поместить в ячейку табликса. При желании можно настроить группировку данных (рис. 21.2), используя которую вы можете скрыть всю сложность отчета и позволить пользователю детализировать только интересующие его области.

Вы можете также добавить параметры для отчета, чтобы позволить пользователю изменять его данные, например, определить период, за который выво-

дятся данные. Параметры отчета, связанные с параметрами запроса к данным, могут сократить количество данных, получаемых из источника данных. Можно также предоставить список допустимых значений для параметра, чтобы пользователь мог выбрать только те значения, которые наверняка есть в источнике данных.



			2001				
			July	August	September	October	November
Category	Subcategory	Product					
⊟ Bikes	⊞ Mountain Bikes		105074.6900	104974.6900	44099.8700	91424.7300	108474.6800
	⊟ Road Bikes	Road-650 Black, 62	1398.1964	1398.1964	2097.2946	1398.1964	2796.3928
		Road-150 Red, 62	78721.9400	114504.6400	85878.4800	96613.2900	85878.4800
		Road-150 Red, 44	82300.2100	53674.0500	82300.2100	78721.9400	78721.9400
		Road-150 Red, 52	42939.2400	75143.6700	89456.7500	64408.8600	110926.3700
		Road-150 Red, 56	53674.0500	82300.2100	57252.3200	85878.4800	93035.0200
		Road-650 Red, 52	1398.1964		699.0982	1398.1964	2097.2946

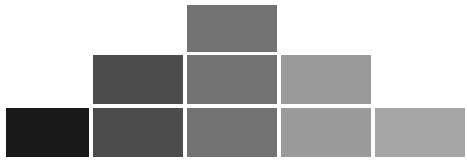
Рис. 21.2. Пример отчета с группировкой данных

Просмотр или экспорт отчета

В Business Intelligence Development Studio формируется отчет с помощью кнопки **Preview** в конструкторе отчета. Формируется, а потом отображается отчет. Чтобы получить отчет в других форматах, используйте кнопку **Export** на панели инструментов.

Разбиение на страницы

Порядок разбиения на страницы определяется размером страницы и разрывами страниц, размещенными в элементах отчета. При подготовке отчета в формате, поддерживающем размеры страницы (например, в формате изображений или PDF), данные отчета форматируют так, чтобы они соответствовали размеру страницы. Некоторые форматы вывода, как например HTML, не поддерживают размер страниц, могут использовать "мягкие" разрывы страниц.



ГЛАВА 22

Другие возможности SQL Server

В SQL Server существуют и другие службы, которые не рассмотрены в данной книге. Для описания некоторых из них может потребоваться отдельная книга. Далее представлен список этих служб с их кратким описанием.

Сервисы интеграции

Сервисы интеграции раньше назывались *сервисами преобразования данных* (Data Transformation Services, DTS). Это мощный инструмент, который позволяет создавать решения, служащие для перемещения, преобразования и объединения данных, извлекаемых из разных источников. С помощью сервисов интеграции вы можете решать такие задачи, как:

- слияние данных, извлекаемых из различных источников данных;
- пакетная загрузка данных в разные базы данных;
- обновление данных с использованием внешних источников;
- интеллектуальная обработка данных при их переносе из одного источника в другой;
- автоматизация административных функций.

Конечно, для этих функций можно было бы написать специальное приложение, извлекающее данные из файла и обновляющее базу данных, но реализовать аналогичное решение с помощью сервисов интеграции гораздо проще.

Сервисы уведомлений

Сервисы уведомлений обеспечивают функциональность, связанную с доставкой обновлений информации (сообщений) на различные стационарные и мобильные устройства.

Интегрированная поддержка HTTP

SQL Server позволяет выполнять доступ к данным по протоколу HTTP и управлять этим доступом. Благодаря этой технологии, появившейся еще в SQL Server 2000, вы просто получаете результаты, предварительно отформатированные как XML (например, с использованием конструкции `FOR XML`). Полученные XML-данные можно легко отобразить в Web-браузере или использовать в других клиентах.

Начиная с SQL Server 2005, поддерживаются настоящие Web-сервисы, официально называемые *конечными точками HTTP* (HTTP Endpoints). Кроме данных, преобразованных в XML при помощи конструкции `FOR XML`, эта новая возможность Web-сервисов позволяет возвращать наборы строк, скалярные значения, сообщения и даже ошибки, причем все эти данные автоматически переводятся в XML.

Аналитические сервисы

Аналитические сервисы обеспечивают возможности онлайновой обработки и анализа больших и комплексных наборов данных, находящихся в многомерных хранилищах данных. Эти сервисы появились еще в SQL Server 2000.

Service Broker

Service Broker — это встроенный в SQL Server механизм асинхронного обмена сообщениями между удаленными приложениями. Service Broker отличается от других технологий обработки очередей, поддерживая возможность группировки сообщений в потоки. Таким образом, если задача требует выполнения множества разных операций, Service Broker позволяет отправить много кратких связанных сообщений от одного сервиса другому. Service Broker обрабатывает детали маршрутизации, обеспечивает надежную доставку сообщений, упорядочивает и координирует их.

Service Broker поддерживает блокировку группы взаимодействий, гарантирующую, что в любой конкретный момент времени только один объект, читающий очередь, может изменять заказ. Но важнее всего то, что даже при полном отключении всей компьютерной сети сообщения будут доставлены после восстановления работоспособности компьютеров.

Сжатие данных в SQL Server

Эта возможность появилась в SQL Server 2005 SP2 и SQL Server 2008. Объемы данных в хранилищах баз данных продолжают расти, занимая все больше и больше места. Теперь вы можете сжимать данные в колонках таблицы, уменьшая размер занимаемого места, а также повышая производительность благодаря сокращению объемов операций ввода/вывода.

Также вы можете сжимать резервные копии. И благодаря этому будет требоваться меньше пространства для их хранения. И работа с ними будет идти быстрее, поскольку сокращается объем операций дискового ввода/вывода.

Параметры сжатия настраиваются через SQL Server Management Studio.

Глоссарий

ANSI (American National Standards Institute, Американский институт национальных стандартов) — организация, занимающаяся разработкой и утверждением всевозможных стандартов.

DCL (Data Control Language) — команды SQL, управляющие разрешениями для объектов баз данных.

DDL (Data Definition Language) — команды SQL для создания таблиц и структур баз данных.

DML (Data Manipulation Language) — команды SQL для вставки, удаления, обновления и выборки данных.

NULL — неопределенное значение столбца при отсутствии присвоенного значения.

SQL (Structured Query Language, язык структурированных запросов) — язык, используемый для запросов, обновления и управления реляционной базы данных. Диалекты языка SQL, используемые в различных реляционных базах данных, имеют некоторые различия, но в целом схожи. Диалект SQL, который используется в Microsoft SQL Server, называется Transact-SQL (T-SQL).

Transact-SQL (T-SQL) — диалект SQL, который используется в Microsoft SQL Server.

UNIQUEIDENTIFIER — тип данных в SQL Server для хранения глобально-уникальных идентификаторов (GUID).

Агрегатная функция (aggregate functions) — функция для вычисления сводных показателей.

Альтернативный ключ — любой возможный ключ, не являющийся первичным.

Атрибут (attribute) — свойство, описывающее сущность. В физической модели атрибут называется колонкой или столбцом.

Внешний ключ (foreign key) — один или несколько столбцов, совпадающих с первичным ключом другой таблицы. В SQL Server внешний ключ может совпадать с любым ограничением уникальности любой другой таблицы.

Возможный ключ (candidate key) — любой набор атрибутов, однозначно идентифицирующих запись в таблице. Возможный ключ называется также суррогатным.

"Грязное" чтение (dirty read) — чтение данных, которые были модифицированы, но не закреплены в базе данных.

Деловое правило (business rule) — алгоритм, определяющий последовательность действий, соответствующих какому-либо деловому процессу. Ограничения, накладываемые на данные в соответствии с требованиями бизнеса.

Денормализация (denormalization) — это умышленное изменение структуры базы, нарушающее правила нормальных форм.

Динамическая блокировка (dynamic locking) — процесс SQL Server, определяющий наиболее эффективную схему блокировки для каждого запроса.

Динамический курсор (dynamic cursor) — курсор с изменяющимся итоговым набором. Другими словами, при перемещении курсора все изменения, сделанные кем-либо в базе данных, отображаются и в итоговом наборе.

Журнал транзакций (transaction log) — специальный файл базы данных, в который заносятся все изменения, перед тем как они будут внесены непосредственно в базу данных.

Задачи (jobs) — это административная операция, которую можно определить один раз и затем выполнять многократно по определенному расписанию.

Идентификатор (identifier) — имя объекта базы данных. В границах своей области видимости идентификатор должен быть уникальным.

Издатель (Publisher) — сервер, который хранит специальную базу данных, называемую базой данных распределения (distribution database), в этой базе хранятся данные, предназначенные для репликации на другие базы данных.

Индекс (index) — объект базы данных, ускоряющий доступ к данным, по сравнению с просмотром каждой записи таблицы в поисках нужного результата.

Итоговый набор — таблица с результатами запроса. Итоговый набор состоит из трех частей: столбцов, заголовков столбцов и записей. Итоговый набор может содержать один или несколько столбцов. Кроме того,

он может содержать пустые заголовки столбцов, а также ноль или более строк с данными.

Каскадное обновление (cascading update) — обновление всех зависимых записей (или столбцов).

Каскадное удаление (cascading delete) — удаление зависимых записей (или столбцов) из зависимых таблиц.

Кластерный индекс (clustered index) — индекс, при котором данные располагаются в порядке значений индекса.

Ключ (key) — набор атрибутов, однозначно определяющий запись в таблице.

Ключевой курсор (keyset-driven cursor) — курсор, в котором не учитываются результаты вставок и удалений, но отображаются все модификации данных.

Конкатенация (concatenation) — объединение двух и более символьных строк, выражений или двоичных цепочек.

Константа (constant) — любая статическая величина, используемая в запросе (не содержащая вызовов функций, столбцов или объектов баз данных).

Курсор (cursor) — механизм последовательной обработки записей, входящих в итоговый набор. Курсоры делятся на клиентские и серверные.

Левое внешнее объединение (left outer join) — внешнее объединение, в результат которого включаются все записи первой таблицы, даже при отсутствии связанных записей во второй таблице.

Логические операторы (logical operators) — операторы `AND`, `NOT` и `OR`.

Логическое выражение (Boolean expression) — любое выражение, возвращающее `TRUE`, `FALSE` или `NULL`.

Логическое моделирование — процесс сбора требований и разработки модели базы данных, не зависящей от конкретной СУБД.

Мертвая блокировка (deadlock) — ситуация, когда две или более транзакций пытаются обратиться к ресурсам, блокированным ранее другими транзакциями.

Метаданные (metadata) — данные, описывающие другие объекты баз данных.

Многие-ко-многим, отношение (many-to-many, relationship) — отношение, при котором одной записи первой сущности соответствуют несколько записей второй сущности, а каждой записи второй сущности — несколько записей первой сущности.

Монопольная блокировка (exclusive lock) — режим блокировки, при котором другие процессы не могут установить блокировку для заблокированных записей.

Некластерный индекс (non-clustered index) — индекс, в котором физическое расположение данных отличается от порядка следования индекса.

Нормализация (normalization) — процесс удаления избыточных данных из сущности.

Объединение (join) — процесс объединения данных из двух и более таблиц, представлений или процедур.

Объект базы данных (database object) — один из следующих элементов базы данных: таблица, индекс, триггер, представление, ключ, ограничение, значение по умолчанию, правило, пользовательский тип данных или хранимая процедура.

Ограничения (constrains) — это правила, за соблюдением которых следит система управления базы данных. Ограничения определяют множество значений, которые можно вводить в столбец или столбцы.

Один-к-одному, отношение (one-to-one, relationship) — отношение, при котором одной записи первой сущности соответствует одна запись второй сущности, а каждой записи второй сущности соответствует ровно одна запись первой сущности.

Один-ко-многим, отношение (one-to-many, relationship) — отношение, при котором одной записи первой сущности соответствует несколько записей второй сущности, по каждой записи второй сущности соответствует лишь одна запись первой сущности.

Одновременность (concurrency) — возможность одновременного обращения к SQL Server со стороны нескольких пользователей.

Оповещение (alert) — определенный системным администратором ответ на некоторое событие в SQL Server.

Ортогональное объединение — тип объединения, при котором возвращаются все записи из всех объединяемых таблиц.

Откат (rollback) — отмена транзакции и возврат модифицированных ею данных к исходному состоянию.

Отношение (relationship) — это ситуация, при которой одна сущность ссылается на первичный ключ второй сущности.

Пакет (batch) — это одна или несколько команд SQL, выполняемых SQL Server.

Первичный ключ (primary key) — возможный ключ, выбранный для идентификации записей в таблице.

Подзапросы (subquery) — это команда `SELECT`, создающая новый запрос, находясь в другом запросе.

Подписчик (Subscriber) — сервер, на котором хранится копия публикуемой информации. Подписчик получает все изменения для опубликованных данных.

Полнотекстовый поиск (full-text search) — механизм поиска слов или фраз в текстовых столбцах.

Пользовательская функция (user-defined function) — функция на T-SQL, определенная пользователем. Функция включает в себя логику, определенную пользователем, и может вызываться в выражениях T-SQL.

Пользовательский тип данных (user-defined data type) — тип данных, определенный пользователем.

Предикат — это выражение, которое возвращает `TRUE`, `FALSE` или `Unknown`.

Представление (view) — логическая таблица, созданная командой `SELECT` путем выборки данных из одной или нескольких таблиц.

Псевдоним (alias) — альтернативное имя таблицы или столбца в выражении.

Распределитель (Distributor) — сервер, который хранит специальную базу данных, называемую базой данных распределения (distribution database), которая содержит в себе распределяемые данные и действует в качестве промежуточного звена между издателем и подписчиком.

Репликация (replication) — процесс поддержания данных на SQL Server в синхронизированном состоянии с данными на другом SQL Server (посредством дублирования).

Репликация моментальных снимков (Snapshot Replication) — тип репликации, при котором производится периодическое формирование моментальных снимков (полной копии данных, выбранных для репликации) публикуемых данных и применение их к базе данных подписчиков.

Репликация транзакций (Transactional Replication) — тип репликации, при котором передаются не наборы данных целиком, а только лишь информация о сделанных изменениях.

Репликация сведением (Merge Replication) — тип репликации, при котором данные могут изменяться в нескольких базах, и информация об этом сводится вместе в центральной базе данных.

Сборка (assembly) — исполняемый модуль в .NET.

Система с принятием решений (decision support system) — приложение баз данных, используемое для анализа данных.

Системные базы данных (system databases) — четыре базы данных, создаваемые при установке SQL Server: master, tempdb, model и msdb.

Сопоставление (collation) — правила сравнения и хранения данных.

Составной индекс (composite index) — индекс, состоящий из нескольких столбцов.

Составной ключ (composite key) — ключ, состоящий из нескольких атрибутов.

Союз (union) — слияние результатов двух и более запросов.

Список выборки (select list) — элементы, возвращаемые командой `SELECT`.

Ссылочная целостность (referential integrity) — правило, которое гарантирует, что для внешних ключей существуют ассоциированные первичные ключи.

Статический курсор (static cursor) — курсор, в котором отображается состояние данных на момент открытия.

Столбец (column) — смотри атрибут.

Столбец счетчика (identity column) — столбец, значение которого сгенерировано базой данных, обладающий атрибутом `IDENTITY`. В таблице может быть лишь один столбец счетчика.

Схема — это средство группировки объектов (не пользователей), позволяющее обращаться с набором объектов как с единым целым (в смысле владения и прав).

СУБД — системы управления реляционными базами данных.

Суррогатный ключ (surrogate key) — то же, что и возможный ключ.

Сущность (entry) — это объект в базе данных, в котором хранятся данные. Сущность может представлять нечто вещественное (дом, человек, предмет, место) или абстрактное (банковская операция, отдел компании, маршрут автобуса). В физической модели сущность называется таблицей. Сущности состоят из атрибутов (столбцов таблицы) и записей (строк в таблице).

Тип данных (data type) — атрибут столбца, определяющий, какая информация может храниться в данном столбце.

Транзакция (transaction) — единица работы, рассматриваемая как атомарная операция. То есть все операции в рамках транзакции успешны или неуспешны как единое целое.

Триггер (trigger) — это аналог хранимой процедуры, который вызывается автоматически при изменении данных в таблице.

Уровень изоляции (isolation level) — это параметр, определяющий степень блокировки данных в SQL Server.

Фантомная запись — это запись, которой на самом деле уже нет в базе данных, или наоборот, запись уже есть, но в запросе выбора данных ее еще нет.

Физическое моделирование — процесс создания модели базы данных, оптимизированной для конкретного приложения и СУБД.

Хранимая процедура (stored procedure) — это предварительно откомпилированные процедуры, хранящиеся в базе данных.

Целостность данных (data integrity) — правильность и надежность данных в таблицах.

Предметный указатель

(окончание рубрики см. на стр. 214)

П

Р

С

Т

Ф

Х