УНИВЕРСИТЕТ ИТМО

И.С. Осетрова

РАЗРАБОТКА БАЗ ДАННЫХ в MS SQL Server 2014

Санкт-Петербург
2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

И.С. Осетрова

РАЗРАБОТКА БАЗ ДАННЫХ в MS SQL Server 2014

Учебное пособие

Санкт-Петербург

2016

Осетрова И.С., Разработка баз данных в MS SQL Server 2014. - СПб: Университет ИТМО, 2016. – 114 с.

В пособии представлены средства и технологии MS SQL Server 2014, которые предназначены для разработки, создания и администрирования баз данных в любой сфере деятельности.

Предназначено для подготовки бакалавров по направлению «11.03.02 Инфокоммуникационные технологии и системы связи» по бакалаврской программе «Инфокоммуникационные системы».

Рекомендовано к печати Ученым советом факультета инфокоммуникационных технологий, протокол № 09/16 от 24.11.2016г.

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ **5**

1 ОБЗОР MICROSOFT SQL SERVER 2014 ..... **6**

1.1 Выпуски SQL-сервера 6

1.2 Начало работы в SQL Server Management Studio 7

2 СОЗДАНИЕ БАЗ ДАННЫХ **10**

2.1 Принципы создания баз данных 10

2.2 Транзакция. Журнал транзакций 12

2.3 Параметры базы данных 14

2.4 Создание файловых групп 17

2.5 Создание схем 21

3 ТИПЫ ДАННЫХ. СОЗДАНИЕ ТАБЛИЦ **24**

3.1 Основные типы данных 24

3.2 Создание таблиц 29

4 СОЗДАНИЕ ИНДЕКСОВ **36**

4.1 Планирование индексов 36

4.2 Создание индексов 44

4.3 Оптимизация индексов 50

5 ВНЕДРЕНИЕ ПРЕДСТАВЛЕНИЙ **55**

5.1 Представления 55

5.2 Создание представлений и управление ими 58

6 РЕАЛИЗАЦИЯ ХРАНИМЫХ ПРОЦЕДУР **64**

6.1 Основные сведения о хранимых процедурах 64

6.2 Создание параметризованных хранимых процедур 67

6.3 Обработка ошибок 72

7 РЕАЛИЗАЦИЯ ФУНКЦИЙ **77**

7.1 Создание и использование функций 77

7.2 Работа с функциями 81

7.3 Контроль контекста выполнения 84

8 ОБЕСПЕЧЕНИЕ ЦЕЛОСТНОСТИ ДАННЫХ. ТРИГГЕРЫ **87**

8.1 Обзор целостности данных 87

8.2 Внедрение ограничений 90
8.3 Внедрение триггеров 100
ЛИТЕРАТУРА..... 111

Введение

Учебное пособие предназначено для студентов, обучающихся по профилю подготовки бакалавров направления 11.03.02 «Интеллектуальные инфокоммуникационные системы».

Основная цель учебного пособия состоит в ознакомлении со средствами и технологиями MS SQL Server 2014, которые предназначены для разработки, создания и администрирования баз данных в любой сфере деятельности.

В курсе «Создание клиент-серверных приложений» студенты должны овладеть навыками работы в среде SQL Server Management Studio.

Дополнительные сведения можно найти в списке рекомендуемой литературы, в первую очередь в электронной документации по SQL Server 2014 на сайте MSDN (Microsoft) [1-2].

1 ОБЗОР Microsoft SQL Server 2014

SQL-сервер является клиент-серверной системой. Это означает, что клиентское программное обеспечение, которое включает в себя SQL Server Management Studio и Visual Studio, является отдельным от SQL Server database engine. Это означает, что когда приложения-клиенты отправляют запросы на SQL Server database engine (такие, как операторы T-SQL), SQL-сервер выполняет все действия (файлы, загрузку памяти и использование процессора) от имени клиента, но приложения-клиенты никогда не получают напрямую доступ к файлам базы данных (в отличие от настольных приложений баз данных).

1.1 Выпуски SQL-сервера

SQL-сервер предлагает несколько выпусков, обеспечивающих различные наборы функций, предназначающиеся для различных деловых сценариев. Еще в выпуске SQL-сервера 2012 число выпусков было оптимизировано [1].

Основные выпуски:

- **Enterprise**, который является ведущим выпуском. Содержит все функции SQL-сервера, включая службы BI и поддержку виртуализации.
- **Standard**, включающий базовый механизм database engine, а также базовое создание отчетов и возможности аналитики. Однако поддерживает меньше ядер процессора и не предлагает всех возможностей, безопасности и функций организации хранилищ данных, представленных в версии Enterprise.
- **Business Intelligence**, который впервые появился в SQL Server 2012 [2]. Он обеспечивает базовый механизм database engine, полное создание отчетов и возможности аналитики, все возможности службы BI. Однако, как Standard, поддерживает меньше ядер процессора и не предлагает всех возможностей, безопасности и функций организации хранилищ данных.

SQL Server 2014 также предлагает и другие выпуски, такие как **Parallel Data Warehouse**, **Web**, **Developer** и **Express**, каждый из которых предназначен для определенных вариантов использования [1].

Есть «облачные» решения. SQL Server может работать как экземпляр SQL-сервера в сети на основанном в облаке сервере, который организация настроила и интегрировала со своей инфраструктурой. Другой альтернативой является Microsoft Azure™ SQL Database, предоставляющий базу данных, использующие технологии SQL-сервера в облаке, но без необходимости настраивать и конфигурировать целую виртуальную машину. Существуют некоторые ограничения к T-SQL при использовании Microsoft Azure SQL Database.

1.2 Начало работы в SQL Server Management Studio

SQL Server Management Studio (SSMS) является многофункциональным приложением, позволяющим администрировать SQL Server, создавать базы данных и делать к ним запросы. SSMS основывается на оболочке Visual Studio [5].

Запустить SSMS можно:

- из главного меню (кнопка **Пуск**),
- из командной строки (SSMS.EXE).

По умолчанию SSMS выведет на экран диалоговое окно **Connect to Server**, которое используют для определения сервера (или экземпляра сервера), а также для выбора учетных данных безопасности. При использовании кнопки Options для доступа к вкладке Connection Properties, можно также указать базу данных, с которой Вы хотите соединиться. Однако можно исследовать много функций SSMS, первоначально не соединяясь с экземпляром SQL-сервера, а выполнить соединение позже.

После того, как SSMS запущен, можно изменить некоторые его настройки по умолчанию (меню **Tools → Options**).

Соединение с SQL-сервером

Для соединения с экземпляром SQL-сервера необходимо определить несколько элементов, независимо от того какой инструмент Вы используете:

- **Имя экземпляра** (в форме: hostname\instancename). Например, MIA-SQL\Proseware соединился бы с экземпляром Proseware на Windows Server, названном MIA-SQL. Если Вы соединяетесь с экземпляром по умолчанию, можно опустить имя экземпляра. Для Microsoft Azure имя сервера находится в четырех частях в форме: < host >.database.windows.net.
- **Имя базы данных**. Если Вы не определите имя базы данных, то будете соединены с базой данных, определяемой по умолчанию для вашей учетной записи администратором базы данных, или к основной базе данных, если не было присвоено значение по умолчанию. В Microsoft Azure важно выбрать нужную базу данных, поскольку вы не можете изменить соединения между пользовательскими базами данных (необходимо будет разъединиться и повторно соединиться с требуемой базой данных).
- **Механизм аутентификации**. Это может быть ***Windows Authentication***, посредством которой учетные данные сети Windows передадут к SQL-серверу, или ***SQL Server Authentication***, в которой имя пользователя и пароль для учетной записи должны быть созданы администратором базы данных (во время соединения их вводят в окне диалога **Connect to Server**).

Механизм аутентификации ***SQL Server Authentication*** является единственным механизмом, поддерживаемым Microsoft Azure.

Работа с проводником (Object Explorer)

Проводник (Object Explorer) является графическим инструментом для того, чтобы управлять экземплярами SQL-сервера и базами данных. Это - одно из нескольких оконных панелей SSMS, доступных из меню **View**. Проводник предоставляет возможности прямого доступа к большинству объектов данных SQL-сервера, таких как базы данных, таблицы, представления, процедуры и т.д. Щелкая правой кнопкой по объекту, можно получать контекстно-зависимые команды, доступные для данного объекта, и выполнять их.

Любые действия, выполняемые в SSMS, требуют надлежащих полномочий, предоставленных администратором базы данных. Способность видеть объект или команду необязательно подразумевает разрешение на их использование или выполнение!

Можно использовать Проводник для изучения структуры объектов данных, которые, например, в дальнейшем будут использоваться в запросах.

Работа с файлами сценария и проектами

SSMS позволяет создавать и сохранять код T-SQL в текстовых файлах (обычно с типом ***.sql***). Как и в других приложениях Windows, SSMS обеспечивает доступ к управлению файлами через меню **File**, а также через кнопки на панели инструментов **Стандартная**.

В дополнение к прямому управлению отдельными файлами (сценариями) SSMS обеспечивает механизм хранения группы файлов (и для открытия, и для сохранения, и для закрытия всех вместе). Этот механизм использует с помощью области **Solution Explorer** несколько концептуальных уровней для работы с файлами T-SQL и связанными документами, чтобы вывести на экран и управлять ими.

Преимущества использования сценариев, организованных в проектах и решениях, включают возможность сразу открыть множество файлов в SSMS. Можно открыть решение или файл проекта из SSMS или Windows Explorer.

Для создания нового решения щелкните по меню **File** и нажмите **New Project**. Определите имя для начального проекта, его родительского решения, и хотите ли Вы, чтобы проект был сохранен в подпапке файла решения в месте, на которое Вы указываете. Нажмите "**OK**" для создания родительских объектов.

Для взаимодействия с Проводником Решений (Solution Explorer) надо при необходимости открыть эту область из меню **View**. Для создания нового сценария, который будет сохранен как часть проекта, можно щелкнуть правой кнопкой по папке ***Queries*** в проекте и нажать ***New Query***.

*Использование кнопки на панели инструментов **New Query** для нового запроса создаст новый сценарий, временно сохраненный решением в папке Miscellaneous Files. Если надо переместить существующий открытый запрос в решение, которое в настоящее время открыто в Проводнике Решений, то необходимо сначала сохранить файл, а потом перетащить его в дерево проекта для сохранения в папке Queries. Это сделает копию файла сценария и поместит его в решение.*

*Важно не забыть сохранять решение при выходе из SSMS или открытии другого решения! Сохранение сценария с помощью кнопки на панели инструментов **Save** сохранит только изменения в текущем файле сценария. Для сохранения всего решения и всех его файлов надо использовать команду **Save All** в меню **File** (или сохранить .ssmssln и .ssmssqlproj файлы на выходе).*

Выполнение запросов

Для выполнения кода T-SQL в SSMS сначала необходимо открыть .sql файл, содержащий запросы, или ввести запрос в новое окно запроса. Затем выделить код, который надо выполнить. Если ничто не будет выделено, то SSMS выполнит весь сценарий целиком.

Для выполнения кода:

- Надо нажать кнопку ***Execute*** на панели инструментов SSMS (или в меню **Query**).
- Или нажать клавишу F5, сочетание клавиш Alt+X или сочетание клавиш Ctrl+E.

По умолчанию SSMS выводит на экран результаты в новой области окна запроса [5]. Расположение и появление результатов может быть изменено в поле **Options**, доступном из меню **Tools**. Чтобы переключить дисплей результатов и возвратиться к полноэкранному редактору T-SQL, можно использовать сочетание клавиш Ctrl+R.

SSMS обеспечивает несколько форматов для представления результатов запроса:

- **Grid** (**Сетка**): подобие электронной таблицы результатов с номерами строк и столбцов, в которых можно изменить размеры. Для выбора этого представления прежде, чем выполнить запрос, надо нажать Ctrl+D.
- **Text** (**Текст**): является дисплеем Windows Notepad-like. Для выбора этого представления прежде, чем выполнить запрос, надо нажать Ctrl+T .
- **File** (**Файл**): позволяет сохранять запрос непосредственно в текстовом файле с .rpt расширением. Запрос запросит расположение файла результатов. Файл может быть открыт любым приложением для работы с текстовыми файлами, например, Windows Notepad и SSMS. Для выбора этого представления прежде, чем выполнить запрос, надо нажать Ctrl+Shift+F.

2 СОЗДАНИЕ БАЗ ДАННЫХ

Одной из первых задач, стоящих перед разработчиком баз данных, является создание самой базы данных и ее основных компонентов, таких как файловые группы и схемы. Эти вопросы рассматриваются в данной главе.

2.1 Принципы создания баз данных

Microsoft SQL Server (см. рисунок 1) имеет набор средств, позволяющий:

- хранить данные, которые можно обрабатывать с помощью транзакций в интерактивном режиме (OLTP - Online Transaction Processing);
- выполнять анализ данных с целью решения аналитических задач обработки данных (OLAP - Online Analytical Processing).

Рисунок 1 Microsoft SQL Server, OLTP, OLAP

Компонент СУРБД системы SQL Server отвечает за выполнение следующих функций [6]:

- Обеспечение связей между данными, содержащимися в базе данных.
- Обеспечение правильного хранения данных и соблюдения правил, определяющих связи между данными.
- Восстановление всех данных после отказа системы с возвратом к одному из последних согласованных состояний.

Базы данных OLTP

Реляционные таблицы, которые позволяют систематизировать данные для хранения в виде базы данных OLTP, что способствует сокращению объема избыточной информации и повышению скорости обновления. SQL Server дает возможность множеству пользователей выполнять транзакции и одновременно изменять содержимое баз данных OLTP в режиме реального времени.

Базы данных OLAP

OLAP используется для систематизации больших массивов данных и суммарной оценки их содержимого, позволяя аналитикам быстро исследовать данные в реальном времени. Службы Microsoft SQL Server Analysis Services

организуют эти данные, обеспечивая поддержку широкого спектра решений масштаба предприятия, начиная с систем корпоративной отчетности и анализа и заканчивая моделированием данных и процессами принятия решений.

Планирование базы данных

При планировании новой базы данных следует учитывать ряд важных аспектов.

К ним относится, в частности, следующее:

- **Предназначение хранилища данных**. Базы данных OLTP и OLAP предназначены для разных целей, и потому требования к их проектированию различаются.
- **Производительность обработки транзакций**. К базам данных OLTP, как правило, предъявляются высокие требования в отношении числа транзакций, обрабатываемых за минуту, час или день. Эффективный проект базы данных с надлежащим уровнем нормализации и адекватной конфигурацией индексов и секций данных позволит достичь высокой скорости обработки транзакций.
- **Потенциальное расширение физического хранилища данных**. Для больших объемов данных требуется соответствующее оборудование с подходящими параметрами памяти, места на жестком диске и мощности центральных процессоров. Необходимо оценить объем данных, который будет храниться в базе данных в течение последующих месяцев и лет; это гарантирует эффективную работу базы данных в будущем. Базы данных можно настраивать так, чтобы файлы автоматически расширялись до установленного максимального размера. Однако автоматический рост файлов может неблагоприятно отразиться на быстродействии. В решениях баз данных, использующих сервер, рекомендуется создавать базу данных с приемлемым размером файлов и следить за использованием пространства базы данных, выделяя дополнительное пространство только по мере необходимости.
- **Размещение файлов**. Место расположения файлов также может влиять на быстродействие. Если имеется возможность использовать несколько дисководов, можно распределить файлы базы данных по нескольким дискам. Это позволит использовать в SQL Server несколько соединений и несколько головок дисков для повышения эффективности считывания и записи данных.

Для создания базы данных можно использовать визуальные средства SQL Server Management Studio или инструкцию Transact-SQL CREATE DATABASE [4].

В следующем примере показано, как создать базу данных с помощью Transact-SQL:

```
CREATE DATABASE TestDB
ON (NAME = 'TestDB_Data',
    FILENAME = 'D:\DATA\TransactTestDB.mdf',
    SIZE = 20 MB,
    FILEGROWTH = 0)
LOG ON (NAME = 'TestDB_Log',
    FILENAME = 'D:\DATA\TestDB_Log.ldf',
    SIZE = 5 MB,
FILEGROWTH = 0)
```

В результате выполнения кода будет создана база данных TestDB.

2.2 Транзакция. Журнал транзакций

Транзакция

Транзакция — это элементарная единица работы, выполняемая в базе данных. Характеристики транзакций часто описывают аббревиатурой ACID: Atomicity, Consistency, Isolation и Durability, которая применяется для описания основных требований по организации транзакций, предъявляемых к надежным серверам баз данных [3].

- **Атомарность** (**Atomicity**) — транзакция должна быть атомарной для исполняемого блока команд. Или все её изменения данных будут исполнены, или не будет исполнено ни одно из них.
- **Последовательность** (**Consistency**) — после завершения транзакции данные должны оставаться в непротиворечивом состоянии. В реляционной базе данных, должны быть предприняты все меры к тому, чтобы поддерживать целостность данных во время исполнения модифицирующих данные транзакций. Все внутренние структуры данных, например: бинарные деревья индексов или дважды связанные списки, не должны нарушаться после завершения транзакции.
- **Изолированность** (**Isolation**) — изменения, сделанные в параллельных транзакциях, должны быть изолированы от изменений, сделанных другими, исполняемыми параллельно транзакциями. Транзакция должна видеть данные в том состоянии, какими они были до изменения их другой параллельной транзакцией, или в том состоянии, в каком данные окажутся после завершения второй транзакции, но не должна видеть их промежуточное состояние. Всё это называется сериализацией, потому что предоставляет системе возможность перезагружать изначальные данные и повторять серию транзакций, чтобы в результате данные оказались в том состоянии, в каком они должны быть после исполнения первичной транзакции.
- **Неизменность** или **живучесть** (**Durability**) — после того, как транзакция завершена, результаты её работы в системе не должны изменяться. Изменения должны сохраняться даже в случае отказа системы.

SQL Server поддерживает неявные транзакции для отдельных инструкций, изменяющих данные, и явные транзакции для групп инструкций, которые должны быть выполнены как единое целое.

Журнал транзакций

Все транзакции SQL Server заносятся в журнал с упреждающей записью, что позволяет поддерживать базу данных в согласованном состоянии и помогает восстанавливать ее после сбоев. Журнал представляет собой область хранилища, в которой автоматически отслеживаются вносимые в базу данных изменения [6].

SQL Server записывает в журнал все изменения, когда они выполняются на диске, но до того, как они будут отражены в базе данных.

Ведение журнала транзакций

Процесс ведения журнала представлен на рисунке 2.

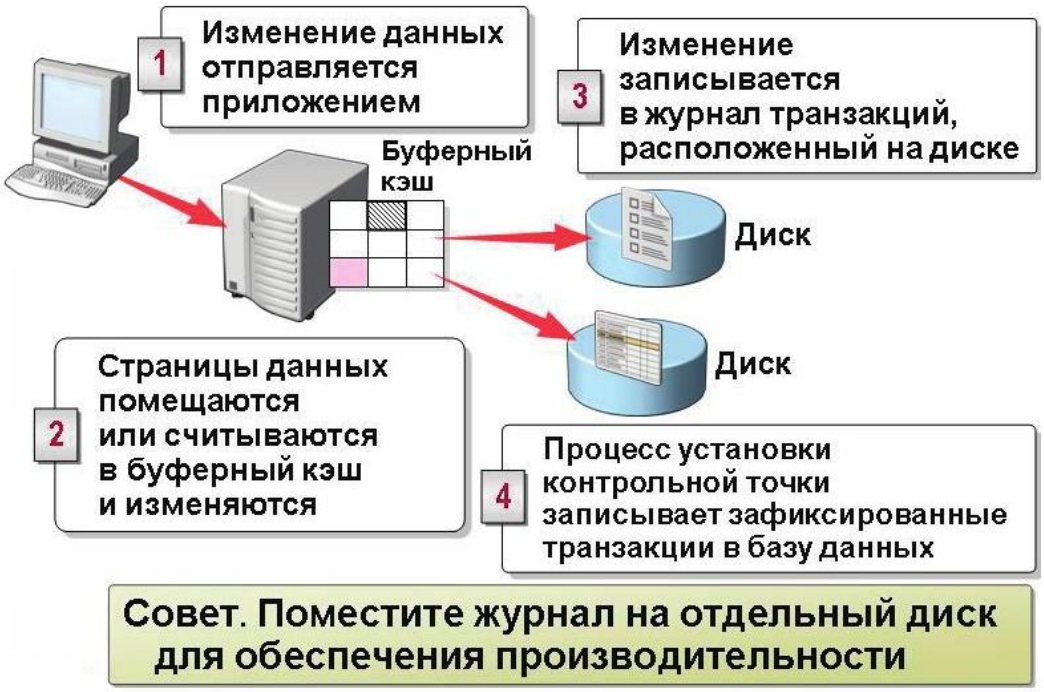

Рисунок 2 Процесс ведения журнала транзакции

Процесс ведения журнала транзакций состоит из нижеперечисленных таких этапов:

1. Приложение отправляет запрос на изменение данных.
2. При выполнении изменения затрагиваемые страницы данных загружаются с диска в буферный кэш, если они не были уже помещены туда при обработке предыдущего запроса.
3. В журнал записывается каждая инструкция изменения данных, как только это изменение производится. Изменение всегда заносится в журнал и записывается на диск до того, как оно будет отражено в базе данных. Журнал такого типа называется журналом с упреждающей записью.

4. Периодически инициируемый процесс установки контрольной точки записывает все завершенные транзакции в базу данных на диск.

В случае отказа системы автоматический процесс восстановления, используя журнал транзакций, выполнит накат всех фиксированных транзакций и откат всех незавершенных транзакций.

В процессе автоматического восстановления используются маркеры транзакций, определяющие начальную и конечную точки каждой транзакции.

Транзакция считается завершенной, если для маркера BEGIN TRANSACTION существует соответствующий ему маркер COMMIT TRANSACTION. При достижении контрольной точки страницы данных записываются на диск.

2.3 Параметры базы данных

Создав базу данных, можно настроить ее параметры (см. Таблицу 1), используя либо визуальные средства SQL Server Management Studio, либо инструкцию Transact-SQL ALTER DATABASE [6].

Можно одновременно настроить целый ряд параметров, но только для одной базы данных. Чтобы изменить параметры сразу для всех новых баз данных, следует внести исправления в базу данных ***model***.

Таблица 1. Виды категорий параметров баз данных

Категория параметров базы данных	**Описание**
Автоматические	Управляет автоматическим поведением, таким как ведение статистики, закрытие базы данных и сжатие.
Доступность	Контролирует, находится ли база данных в оперативном состоянии, кто может подключиться к этой базе данных и предназначена ли база данных только для чтения
Курсор	Управляет поведением курсора и областью
Восстановление	Управляет моделью восстановления для базы данных
SQL	Управляет параметрами соответствия ANSI, такими как значения NULL в формате ANSI и рекурсивные триггеры

Размещение файла журнала

Для повышения быстродействия рекомендуется размещать файл журнала транзакций отдельно от файлов данных, на другом физическом диске. Это позволит избежать конфликтов при доступе к ресурсам: одну группу головок дисков можно будет использовать для записи транзакций в журнал, а другую, в то же самое время — для считывания данных из файлов данных. Данные обновляются быстро, поскольку транзакции немедленно записываются на диск без ожидания завершения операций чтения данных. Запись в файлы журналов ведется в режиме последовательного доступа, поэтому если журнал хранится на выделенном диске, его головки всегда будут оставаться в позиции, требуемой для очередной операции записи.

Категории параметров баз данных

В базах данных используется ряд параметров, которые для удобства управления группируются в различные категории. В Таблице 2 перечислены наиболее часто используемые параметры.

Таблица 2. Параметры баз данных

Категория	Параметр базы данных	Описание
Автоматические	AUTO_CREATE_ STATISTICS	Автоматическое создание любой отсутствующей статистики, необходимой для оптимизации обработки запроса. По умолчанию задано значение ON.
	AUTO_UPDATE_ STATISTICS	Автоматическое обновление устаревшей статистики, необходимой для оптимизации запроса. По умолчанию задано значение ON.
	AUTO_CLOSE	Если задано значение ON, база данных автоматически закрывается при выходе последнего пользователя. По умолчанию имеет значение OFF для всех версий SQL Server 2012, кроме SQL Server 2012 Express.
	AUTO_SHRINK	Если задано значение ON, файлы базы данных становятся кандидатами на периодическое сжатие. По умолчанию OFF.
Доступность	OFFLINE | ONLINE | EMERGENCY	Устанавливает режим работы базы данных: интерактивный или автономный. EMERGENCY означает, что всем, кроме системных администраторов, запрещается подключение, и база данных становится доступной только для чтения. По умолчанию ONLINE.
	READ_ONLY | READ_WRITE	Определяет, могут ли пользователи изменять данные. По умолчанию READ_WRITE.
	SINGLE_USER | RESTRICTED_USER | MULTI_USER	Определяет, кому из пользователей разрешается подключаться к базе данных. SINGLE_USER означает, что может подключиться только один пользователь. Настройка RESTRICTED_USER разрешает подключаться членам роли базы данных db_owner и серверных ролей dbcreator и sysadmin. MULTI_USER разрешает подключение всем пользователям, обладающим необходимыми правами доступа. По умолчанию MULTI_USER.
Курсор	CURSOR_CLOSE_ON_COMMIT	Автоматическое закрытие открытых курсоров при фиксации транзакции. По умолчанию OFF — курсоры остаются открытыми.
	CURSOR_DEFAULT	CURSOR_DEFAULT_LOCAL ограничивает область действия курсора. Этот параметр является локальным для пакета, хранимой процедуры или триггера, в котором курсор был создан. По умолчанию область действия курсора – CURSOR_DEFAULT_GLOBA – является глобальной для подключения.

Категория	Параметр базы данных	Описание
Восстановление	RECOVERY	Значение FULL (используется по умолчанию) обеспечивает полное восстановление после отказа носителя. В режиме BULK_LOGGED журнал занимает меньше места, поскольку ведется на минимальном уровне, однако при этом повышается уязвимость данных. SIMPLE задает восстановление только последней полной или разностной резервной копии базы данных.
	PAGE_VERIFY	Позволяет автоматически выявлять незавершенные операции ввода-вывода, прерванные из-за сбоя питания или других отказов системы. CHECKSUM задает сохранение в заголовке страницы значения, вычисленного на основе содержимого страницы. Это значение пересчитывается и сравнивается с сохраненной версией при считывании страниц данных с диска. Данная настройка используется по умолчанию. TORN_PAGE_DETECTION задает сохранение в заголовке страницы определенного бита для каждого 512-байтного сектора страницы размером 8 килобайт (КБ). Биты, сохраненные в заголовке страницы, сравниваются с фактическим содержимым секторов страницы при считывании страниц данных с диска.
SQL	ANSI_NULL_DEFAULT	Разрешает пользователю контролировать стандартный режим использования значений NULL. В SQL Server 2012 по умолчанию устанавливается OFF, т. е. используется NOT NULL.
	ANSI_NULLS	ON означает, что любое сравнение со значением NULL дает результат NULL (неизвестно). OFF означает, что сравнение значений не из Юникода со значением NULL дает результат TRUE, если оба значения суть NULL. По умолчанию OFF.
	RECURSIVE_TRIGGERS	Разрешает рекурсивный запуск триггеров AFTER. По умолчанию OFF, т. е. прямая рекурсия запрещается.

Источники сведений о базе данных

Если требуется получить информацию об объекте базы данных, проще всего использовать SQL Server Management Studio. При создании приложений, извлекающих метаданные об объектах базы данных, следует использовать Transact-SQL для запроса представлений каталога, формируемых системой, вызова системных функций и выполнения системных хранимых процедур (см. Таблицу 3).

Таблица 3. Способы получения сведений о БД

Источник сведений	Описание
SQL Server Management Studio	Наглядное средство, отображающее метаданные базы данных в среде управления.
Transact-SQL	**Представления каталога.** Представление метаданных об объектах базы данных, возвращающих строки данных
Transact-SQL	**Функции метаданных.** Каждая функция возвращает одно значение сведений о метаданных
Transact-SQL	**Системные хранимые процедуры.** Извлекают метаданные при помощи хранимых процедур.

SQL Server также поддерживает представления информационной схемы, позволяющие получать внутреннее, независимое от системных таблиц представление метаданных SQL Server. Такие представления соответствуют определению информационной схемы в стандарте языка структурированных запросов (SQL) Американского национального института стандартов (ANSI). Эти представления поддерживаются для совместимости с предыдущими версиями, однако для запроса метаданных базы данных рекомендуется использовать представления каталога.

2.4 Создание файловых групп

SQL Server для хранения данных используются файлы данных, которые можно объединить в одну или несколько файловых групп. База данных может успешно работать при наличии только одной файловой группы, однако в некоторых случаях удобнее иметь несколько файловых групп.

Ниже объясняется, что представляют собой файловые группы и как с их помощью можно улучшить структуру базы данных, увеличить ее быстродействие и повысить эффективность ее сопровождения, как создавать файловые группы.

Файловая группа

Файловая группа — это логическое объединение файлов данных, позволяющее администраторам управлять всеми файлами группы как единым целым. Возможность контроля физического размещения отдельных объектов в базе данных дает целый ряд преимуществ в отношении управляемости и производительности. Например, создав несколько файловых групп, можно контролировать физическое сохранение содержимого базы данных на запоминающих устройствах и отделять перезаписываемые данные от доступных только для чтения. Пример размещения файловых групп см. на рисунке 3.

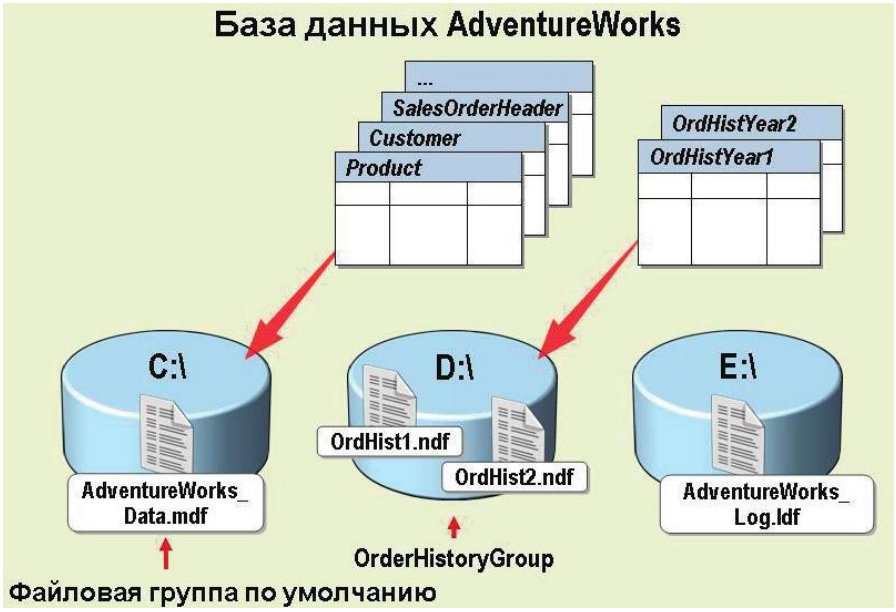

Рисунок 3 Пример размещения файловых групп

Типы файловых групп

В SQL Server имеются два типа файловых групп:

- **Первичная файловая группа** содержит первичный файл данных вместе с системными таблицами. Первичный файл данных обычно имеет расширение имени .mdf.
- **Пользовательская файловая группа** содержит файлы данных, объединенные вместе для удобства распределения и администрирования. Эти файлы данных называются вторичными и обычно имеют расширение имени .ndf.

На рисунке 3 представлен пример возможного размещения файлов базы данных на отдельных дисках, как описано ниже [6]:

- С помощью пользовательских файловых групп можно отделять файлы, на которые часто выдаются запросы, от интенсивно обновляемых файлов. На данном рисунке файлы OrdHist1.ndf и OrdHist2.ndf размещаются на другом диске, отдельно от таблиц Product, Customer и SalesOrderHeader, поскольку они запрашиваются для процессов принятия решений, а не обновляются сведениями о текущих заказах.
- Если оба файла OrdHist1.ndf и OrdHist2.ndf к тому же, часто запрашиваются, их также можно разместить на разных дисках.
- В файловые группы не могут включаться файлы журналов транзакций. Управление областью журнала транзакций осуществляется отдельно от области данных. Файлы журналов транзакций обычно имеют расширение имени .ldf.

В следующем примере кода Transact-SQL реализован сценарий размещения файлов базы данных с помощью инструкции CREATE DATABASE.

```
CREATE DATABASE [AdventureWorks] ON PRIMARY
(NAME=N'AdventureWorks_Data',
    FILENAME=N'C:\AdventureWorks_Data.mdf' ),
FILEGROUP [OrderHistoryGroup]
(NAME = N'OrdHist1', FILENAME = N'D:\OrdHist1.ndf'),
(NAME = N'OrdHist2', FILENAME = N'D:\OrdHist2.ndf')
LOG ON
(NAME = N'AdventureWorks_log', FILENAME =
     N'E:\AdventureWorks_log.ldf')
```

Можно также воспользоваться инструкцией ALTER DATABASE, позволяющей добавлять и удалять файлы и файловые группы в существующих базах данных.

Создание файловых групп

Пользователь может создать несколько файлов данных на разных дисках, а также определить пользовательскую файловую группу для объединения этих файлов. Использования файловых групп позволяет повысить производительность и контролировать физическое расположение данных.

Использование одной файловой группы для повышения производительности

Лучший способ повысить производительность базы данных — использовать массив независимых дисков с избыточностью (RAID — Redundant Array of Independent Disks), однако можно вместо этого объединить несколько файлов, расположенных на разных дисках, в одну файловую группу. Рост производительности достигается благодаря применению методов чередования данных в SQL Server. Поскольку в SQL Server при записи данных в файловую группу используется стратегия пропорционального заполнения, данные фактически расслаиваются по файлам, а, следовательно, — и по разделам физических дисков. Такой подход позволяет детализировать контроль чередования данных, что становится возможным при создании чередующегося набора томов в операционной системе Microsoft Windows® или при использовании контроллера массива RAID.

В большинстве случаев использование возможностей чередования, предлагаемых технологией RAID, дает практически такой же выигрыш в производительности, как и пользовательские файловые группы, но при этом не приводит к повышению административной нагрузки, которого требует определение файловых групп и управление ими.

Использование нескольких файловых групп для контроля физического размещения данных

Для облегчения обслуживания и реализации целей конструирования с помощью файловых групп можно использовать следующие методы:

- Хранить данные, допускающие чтение и запись, отдельно от данных, доступных только для чтения, чтобы разграничить различные виды операций дискового ввода-вывода.
- Хранить индексы и таблицы на разных дисках, что может способствовать повышению быстродействия.
- Выполнять резервное копирование и восстановление отдельных файлов или файловых групп, а не всей базы данных. Резервное копирование файлов или файловых групп может оказаться необходимым для реализации эффективной стратегии резервирования больших баз данных.
- Объединять таблицы и индексы со сходными требованиями к обслуживанию в одну файловую группу. Для некоторых объектов может требоваться чаще выполнять процедуры обслуживания, чем для других. Например, можно создать две файловые группы и назначить им таблицы так, чтобы для таблиц одной группы задачи обслуживания выполнялись ежедневно, а для другой — еженедельно. Это позволит уменьшить число конфликтов между файловыми группами при доступе к дискам.
- Отделить пользовательские таблицы и прочие объекты базы данных от системных таблиц, сведенных в первичную файловую группу. Можно также изменить файловую группу, используемую по умолчанию, чтобы непредвиденный рост размеров таблиц не мешал работать с системными таблицами в первичной группе.
- Хранить таблицу, разбитую на секции, в нескольких файловых группах. Это позволит физически разделить данные одной таблицы в зависимости от требований к доступу, а также улучшить управляемость и производительность.

В следующем примере кода Transact-SQL реализован сценарий создания файловой группы

```
ALTER DATABASE [TransactTestDB]
ADD FILEGROUP [SECONDARY]
GO

ALTER DATABASE [TransactTestDB]
ADD FILE (
    NAME = N'Test2', FILENAME = N'C:\Program Files\Microsoft SQL
           Server\MSSQL.1\MSSQL\Data\Test2.ndf')
TO FILEGROUP [SECONDARY]
GO

ALTER DATABASE [TransactTestDB] MODIFY FILEGROUP [SECONDARY]
     DEFAULT
GO
```

После выполнения кода можно будет увидеть две файловые группы и список файлов, содержащихся в каждой из них.

2.5 Создание схем

Пространство имен позволяет сгруппировать взаимосвязанные объекты, чтобы сделать громоздкие списки объектов более управляемыми. В SQL Server для объектов базы данных предлагается средство, реализованное в виде схем.

Схемы как пространства имен

Объекты базы данных (таблицы, представления, хранимые процедуры) создаются в рамках схемы. Прежде чем планировать и внедрять базу данных SQL Server, важно понять, что представляет собой схема.

Схема — это пространство имен для объектов базы данных. Другими словами, схема определяет рамки, в пределах которых все имена уникальны.

Поскольку сами имена схем должны быть уникальны в пределах базы данных, каждому объекту базы данных присваивается полное имя в формате **имя_сервера.имя_базаданных.имя_схемы. объект**. В пределах базы данных это имя можно сократить до **имя_схемы.объект**.

На рисунке 4 показаны три схемы в базе данных **AdventureWorks** для экземпляра SQL Server с именем **Server1**. Имена этих схем — **Person**, **Sales** и **dbo**. Каждая из этих схем содержит таблицу; полное имя таблицы состоит из имени сервера, имени базы данных и имени схемы. Например, полное имя таблицы **ErrorLog** в схеме **dbo** выглядит так: **Server1.AdventureWorks.dbo.ErrorLog**.

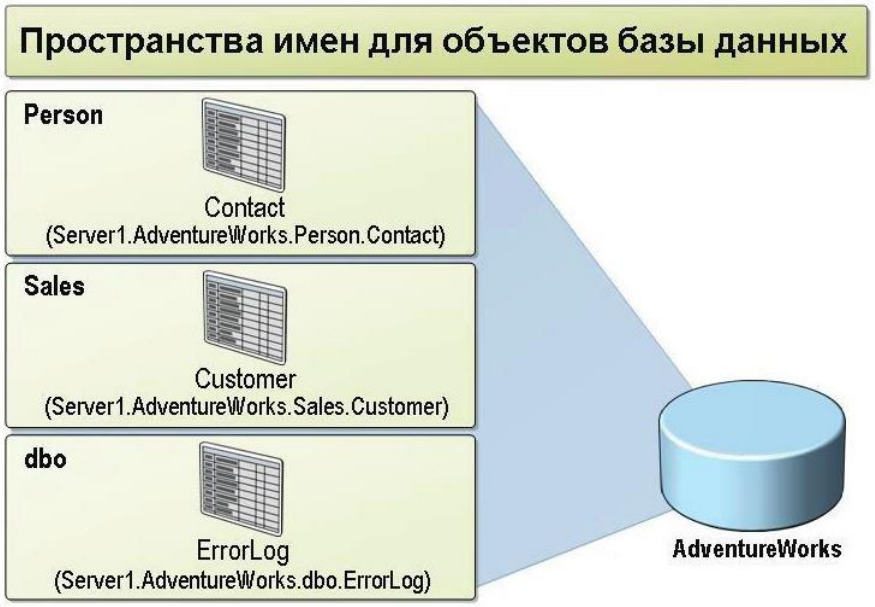

Рисунок 4 Схема базы данных AdventureWorks

В предыдущих выпусках SQL Server до 2005 пространство имен объекта определялось именем пользователя его владельца [3]. Начиная с SQL Server 2005 схемы не связаны с владением объектов, что дает ряд преимуществ [6]:

- Становятся более гибкими возможности распределения объектов базы данных по пространствам имен, поскольку объекты группируются в схемы независимо от того, кто ими владеет.
- Упрощается управление разрешениями на доступ — они могут предоставляться как в масштабе схемы, так и в отношении отдельных объектов.
- Улучшается управляемость, поскольку удаление пользователя не означает необходимость переименования всех объектов, которыми он владел.

Схема dbo

В каждой базе данных имеется схема с именем **dbo**. Схема **dbo** используется по умолчанию для всех пользователей, которые не назначены явным образом никакой другой стандартной схеме.

В следующем примере кода Transact-SQL реализован сценарий создания схемы.

Для создания схемы используется инструкция CREATE SCHEMA, как показано в следующем примере.

```
Use AdventureWorks
GO

CREATE SCHEMA Sales
GO
```

После выполнения кода в AdventureWorks кроме схемы dbo можно будет использовать схему Sales, посмотреть наличие которой можно развернув Безопасность, Схемы.

В базе данных, содержащей несколько схем, разрешение имен объектов может оказаться непростой задачей. Предположим, например, что в базе данных есть две таблицы **Order** в двух разных схемах, **Sales** и **dbo**. Уточненные имена объектов базы данных являются однозначными: **Sales.Order** и **dbo.Order** соответственно. Однако использование неуточненного имени **Order** может привести к неожиданным результатам. Чтобы контролировать разрешение неуточненных имен объектов, можно назначить пользователям схему по умолчанию.

Разрешение имен объектов

В SQL Server разрешение неуточненного имени объекта производится в следующем порядке:

1. Если для пользователя определена схема по умолчанию, SQL Server пытается найти объект в этой схеме.

2. Если в схеме по умолчанию, назначенной пользователю, объект не найден или если для пользователя схема по умолчанию не определена, SQL Server пытается найти объект в схеме **dbo**.

Например, пользователь, которому по умолчанию назначена схема **Person** (см. рисунок 5) [6], выполняет следующую инструкцию Transact-SQL.

```
SELECT * FROM Contact
```

SQL Server сначала пытается разрешить имя объекта в имя **Person.Contact**. Если в схеме **Person** нет объекта с именем **Contact**, SQL Server попытается разрешить имя объекта в имя **dbo.Contact**.

Если пользователь, для которого не определена схема по умолчанию, выполнит ту же инструкцию, SQL Server немедленно разрешит имя объекта в имя **dbo.Contact**.

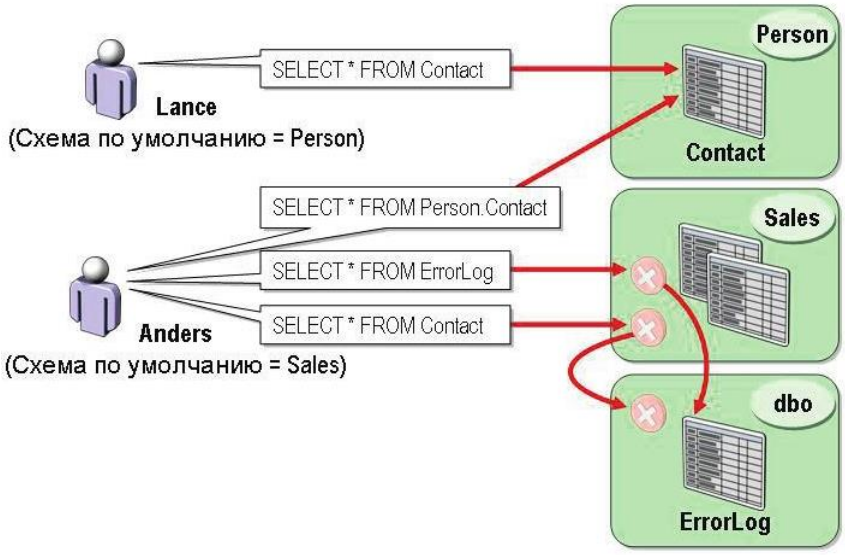

Рисунок 5 Разрешение имен объектов

Назначение схемы по умолчанию

Назначить пользователю схему по умолчанию можно в диалоговом окне «Свойства пользователя базы данных» или с помощью инструкции CREATE USER или ALTER USER, указав имя схемы в предложении DEFAULT_SCHEMA. Например, следующий код Transact-SQL назначает схему Sales пользователю Anders в качестве схемы по умолчанию.

```
ALTER USER Anders WITH DEFAULT
```

3 ТИПЫ ДАННЫХ. СОЗДАНИЕ ТАБЛИЦ

Данные в базе данных хранятся в таблицах, и каждый элемент данных определяется как конкретный тип данных. В этом разделе описываются поддерживаемые системой типы данных в Microsoft SQL Server, способы определения пользовательских типов данных Transact-SQL, создание таблиц и как следует использовать секционированную таблицу для организации данных в несколько секций.

Прежде чем создавать таблицу, необходимо определить типы данных для тех данных, которые будут в ней храниться. Типы данных определяют тип информации (символы, числа или даты), которые могут содержаться в столбце, а также способ хранения этих данных. SQL Server 2014 предоставляет определенные системные типы данных и поддерживает «псевдонимные» типы, которые представляют собой пользовательские типы данных, основанные на системных типах данных.

3.1 Основные типы данных

Типы данных определяют значения данных, допустимые для каждого столбца в таблице базы данных. Некоторые категории типов данных, которые обычно используются в языках программирования, имеют несколько соотнесенных с ними типов данных SQL Server. Примеры типов данных см. в Таблице 4. Следует всегда использовать наименьший тип данных, отвечающий вашим потребностям, чтобы сэкономить место на жестком диске и обеспечить наибольшее количество строк на странице данных, что, в свою очередь, обеспечит наилучшую производительность системы [1].

Таблица 4. Системные типы данных

Категория		**Типы данных**
Числовые	Целое число	***int, bigint, smallint, tinyint***
	Точное	***decimal, numeric***
	Приблизительное	***float, real***
	Денежный	***money, smallmoney***
Дата и время		***datetime, smalldatetime***
Символ	Не поддерживающий Юникод	***char, varchar, varchar(max), text***
	Поддерживающий Юникод	***nchar, nvarchar, nvarchar(max), ntext***
Двоичный		***binary, varbinary, varbinary(max)***
Изображение		***image***
Глобальный идентификатор		***uniqueidentifier***
XML		***xml***
Специальный		***bit, cursor, timestamp, sysname, table, sql_variant***

В Таблице 5 описываются типы поддерживаемые системой SQL Server 2005 и указываются ANSI-синонимы (American National Standards Institute) для стандартных типов данных, используемых в ANSI-совместимых системах баз данных. При ссылке на типы данных в коде Transact-SQL можно использовать специфичное для SQL Server имя типа данных или соответствующий ANSI-синоним [6].

Таблица 5. Соответствие типов данных Transact-SQL и ANSI

Категория типа данных	**Системные типы данных**	**ANSI-синоним**	**Число байтов**
Целое число	int	integer	4
	bigint	—	8
	smallint	—	2
	tinyint	—	1
Точный числовой	decimal[(p[, s])]	dec	5–17
	numeric[(p[, s])]	—	5–17
Приблизительный числовой	float[(n)]	double precision, float[(n)] для n=8–15	8
	real	float[(n)] для n=1–7	4
Денежный	money	—	8
	smallmoney	—	4
Дата и время	datetime	—	8
	smalldatetime	—	4
Символ не Юникода	char[(n)]	character[(n)]	0–8000
	varchar[(n)]	char VARYING[(n)] character VARYING[(n)]	0–8000
	varchar(max)	char VARYING(max) character VARYING(max)	0–2 гигабайта (ГБ) (в строке или 16-разрядный указатель)
	text	—	0–2 ГБ (16-разрядный указатель)
Символ Юникода	nchar[(n)]	national char[(n)] national character[(n)] national char	0–8000 (4000 символов)
	nvarchar[(n)]	VARYING([n]) national character VARYING([n])	0–8000 (4000 символов)
	nvarchar(max)	national char VARYING(max) national character VARYING(max)	0–2 ГБ (в строке или 16-разрядный указатель)
	ntext	—	0–2 ГБ (16-разрядный указатель)
Двоичный	binary[(n)]	—	0–8000
	varbinary[(n)]	binary VARYING[(n)]	0–8000
	varbinary(max)	binary VARYING(max)	0–2 ГБ (строке или 16-разрядный указатель)
Изображение	image	—	0–2 ГБ (16-разрядный указатель)
Глобальный идентификатор	uniqueidentifier	—	16

Категория типа данных	Системные типы данных	ANSI-синоним	Число байтов
XML	xml	—	0–2 ГБ
Специальный	bit	—	1
	cursor	—	0–8
	timestamp	rowversion	8
	sysname	—	256
	table	—	
	sql_variant	—	0–8016

Для получения дополнительных сведений о системных типах данных см. раздел «Типы данных (ядро СУБД)» в электронной документации по SQL Server.

Использование системных типов данных

SQL Server предоставляет несколько типов данных, каждый из которых является более подходящим для использования в тех или иных условиях. Прежде чем решить, какой тип данных следует использовать, необходимо принять во внимание требования к этим данным, а также каким образом их планируется использовать.

Точный числовой и приблизительный числовой типы данных

Точный числовой тип данных позволяет точно определить, какое масштабирование и точность следует использовать. Например, можно определить, что будет использоваться три знака справа от десятичного разделителя и четыре — слева от десятичного разделителя. Запрос всегда будет возвращать точно то, что было вами введено. SQL Server поддерживает два точных числовых типа данных для ANSI-совместимости: **decimal** и **numeric**.

Обычно точные числовые типы данных используются в финансовых приложениях, в которых требуется согласованно отображать данные (например, всегда использовать два десятичных разряда) и осуществлять запрос по этому столбцу (например, найти все займы со ставкой процента, равной 8,75 %).

При использовании приблизительных числовых типов данных соответствующие данные хранятся с максимально возможной точностью.

Например, дробь «одна треть» представляется в десятичной системе как 0,33333 (в периоде). Это число не может храниться в виде точного числа, поэтому оно сохраняется как приблизительное число. SQL Server поддерживает два приблизительных числовых типа данных: **float** и **real**. Если осуществляется округление чисел или выполняется контроль качества для числовых данных, следует избегать использования приблизительных числовых типов данных.

Рекомендуется избегать ссылок на столбцы с типами данных float или real в предложениях WHERE.

Чтобы обеспечить целостность данных при перемещении баз данных между компьютерами, столбцы (включая вычисляемые столбцы), в которых используется приблизительный тип данных, могут использоваться только в неключевых столбцах индексированных представлений. Это объясняется тем, что точный результат может зависеть от архитектуры процессора или версии микрокода.

Символьные типы данных

В случае точного размера элемента данных, можно воспользоваться типом данных фиксированного размера, таким как **nchar**, и указать количество байтов, которое должно быть зарезервировано для каждого значения. Когда длина значения может меняться, более эффективным может оказаться использование типа данных с переменной длинной, такого как **nvarchar**, и задание максимальной длины значения. Когда задается тип данных с переменной длинной, SQL Server резервирует два байта для каждого значения переменной длины в качестве маркера и использует только дополнительный пробел, необходимый для каждого значения.

Данные даты и времени

SQL Server предоставляет типы данных **datetime** и **smalldatetime** для хранения данных, основанных на дате и времени. Предоставляется также тип данных timestamp для управления версиями на основе строк.

- Тип данных **datetime** используется для представления данных о дате и времени, начиная с 1 января 1753 г. по 31 декабря 9999 г., с точностью, равной трем сотым секунды. Значения типа данных **datetime** округляются с приращениями, равными **0,000**; **0,003** или **0,007** секунды.
- Тип данных **smalldatetime** используется, чтобы представлять данные о дате и времени, начиная с 1 января 1900 г. по 6 июня 2079 г., с точностью, равной одной минуте. Значения типа данных smalldatetime округляются до ближайшей минуты. Тип данных **smalldatetime** обычно используется, когда не требуется сохранять время или не требуется сохранять точное время, например в случае даты истечения срока. Тип данных datetime обычно используется, когда требуется сохранять время дня с повышенной точностью.
- Тип данных **timestamp** представляет автоматически генерируемые уникальные двоичные числа. Тип данных timestamp обычно используется для пометки версий строк таблицы. Каждая база данных поддерживает счетчик, который SQL Server увеличивает для каждой операции обновления или вставки. Значение поля временной метки является относительным временем внутри базы данных, а не действительным временем, связанным с часами. Можно иметь только один столбец **timestamp** на таблицу, однако не следует задавать имя столбца для столбцов, использующих тип данных **timestamp**, как показывается в следующем примере Transact-SQL.

```
CREATE TABLE ExampleTable (PriKey int PRIMARY KEY, timestamp);
```

Тип данных timestamp языка Transact-SQL отличен от типа данных timestamp, определенных в стандарте SQL-2003, который является эквивалентом типа данных datetime языка Transact-SQL.

Большие значения данных

Столбцы, которые будут использоваться для хранения средних и больших значений данных (обычно более 8 000 байтов) могут храниться с использованием столбца **varchar**, **nvarchar** или **varbinary**, объявляемого с помощью спецификатора max. Использование спецификатора max заменяет типы данных больших объектов (LOB — **text**, **ntext** и **image**) из более ранних версий SQL Server, что обеспечивает обратную совместимость.

В следующем примере Transact-SQL показывается, как осуществляется вставка текста в столбец **LongText varchar(max)** таблицы **BigTable**.

```
DECLARE @longText varchar(max)
SET @longText = replicate(cast('*' AS varchar(max)), 10000)
INSERT BigTable (LongText)
VALUES (@longText)
GO
```

В следующем примере Transact-SQL показывается, как осуществляется вставка текста в столбец **LongText varchar(max)** таблицы **BigTable**.

```
INSERT INTO Photos (FileName, FileType, Picture)
SELECT 'Photo1.jpg' AS FileName, '.jpg' AS FileType,
* FROM OPENROWSET(BULK N'C:\Photo1.jpg', SINGLE_BLOB) AS Picture
GO
```

Псевдонимный тип данных

Псевдонимный тип данных представляет собой пользовательский тип данных, основанный на системном типе данных.

Псевдонимный тип данных:

- позволяет дополнительно уточнить тип данных, чтобы обеспечить совместимость при работе с общими элементами данных в различных таблицах или базах данных;
- определен в конкретной базе данных;
- должен иметь уникальное имя в базе данных. (Однако псевдонимные типы данных с различными именами могут иметь одинаковое определение.)

Псевдонимные типы данных:

- Основаны на типах, поддерживаемых системой
- Используются для элементов общих данных с определенным форматом

- Создаются с помощью инструкции CREATE TYPE

```
CREATE TYPE dbo.StateCode
FROM char (2)
NULL
```

Псевдонимные типы данных следует создавать, когда требуется определить часто используемый элемент данных определенного формата. Например, столбец, в котором будет храниться код страны, основанный на стандарте alpha-2 Международной организации по стандартизации (ISO — International Organization for Standardizationa) для аббревиатур названий стран (например, JP для Японии и CH для Швейцарии), может быть определен как **char(2).** Однако если код страны будет использоваться регулярно по всей базе данных, вместо этого можно определить и в дальнейшем использовать тип данных **CountryCode**. Это позволит облегчить понимание определений объектов и кода в базе данных.

Псевдонимные типы данных, созданные в базе данных model автоматически включаются во все базы данных, которые будут созданы в дальнейшем.

Псевдонимные типы данных могут быть созданы с помощью обозревателя объектов в среде SQL Server Management Studio или с помощью инструкции CREATE TYPE языка Transact-SQL. В следующем примере кода показывается, как осуществляется создание псевдонимного типа данных с именем **CountryCode.**

```
CREATE TYPE dbo.CountryCode
FROM char(2)
NULL
```

Обратите внимание, что при создании псевдонимного типа данных можно определить его возможность содержать значения NULL. Псевдонимный тип данных, созданный с признаком NOT NULL, никогда не может быть использован для сохранения значения NULL. При создании типа данных важно определить возможность содержания в нем значений NULL, поскольку процедура изменения типа данных может занять в дальнейшем много времени. Необходимо будет воспользоваться инструкцией DROP TYPE, чтобы удалить тип данных, а затем еще раз создать новый тип данных для замены удаленного типа. Поскольку удалять тип данных, используемый таблицами в базе данных, нельзя, необходимо будет также применить команду ALTER к каждой таблице, в которой используются данные первого типа.

3.2 Создание таблиц

После определения всех требуемых типов данных для базы данных, вы можете создать таблицы, необходимые для хранения данных. При создании таблицы необходимо понимать, каким образом SQL Server физически организует данные и как следует определить столбцы, чтобы обеспечить оптимальное хранение информации и быстродействие системы.

Организация данных в строках

Строка данных состоит из заголовка строки и фрагмента данных. Важно понимать, какие элементы фрагмента данных содержатся в каждой строке, чтобы точно оценить раздел таблицы (см. рисунок 6) [6].

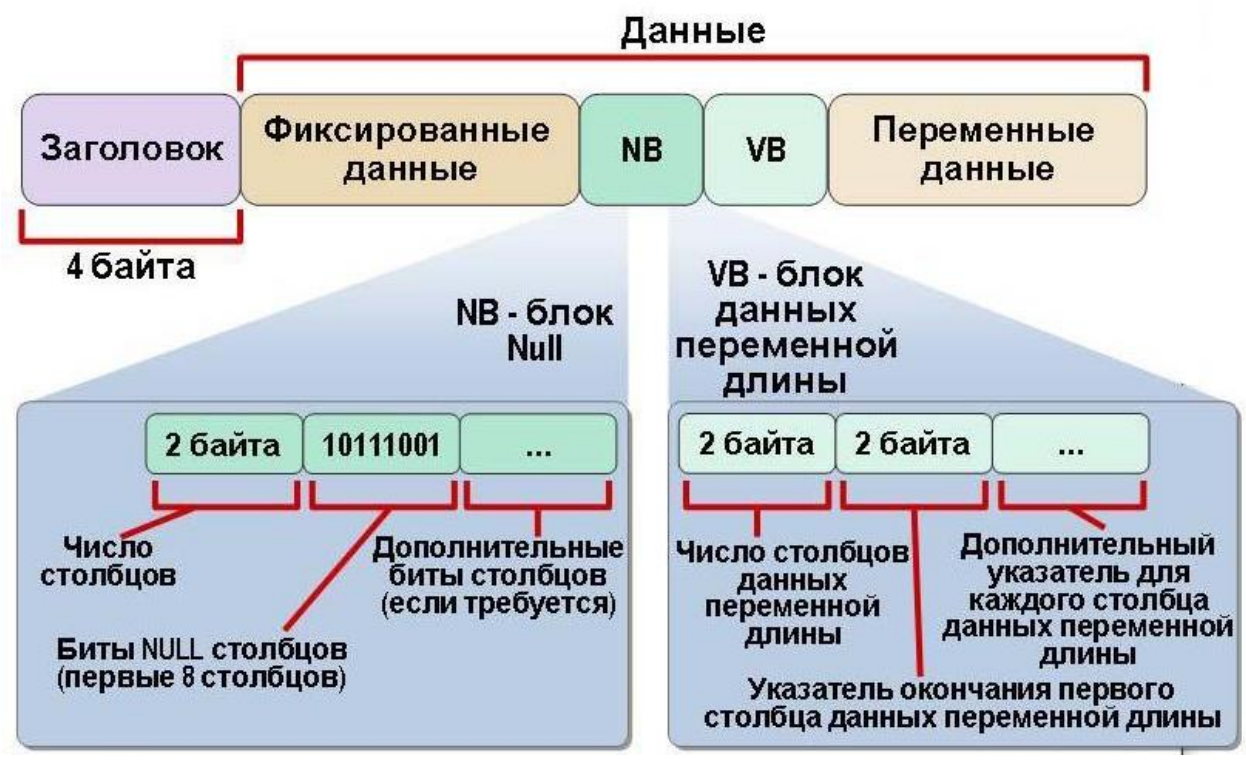

Рисунок 6 Организация данных в строках

Заголовок строки

4-байтовый заголовок строки содержит информацию о столбцах в строке данных, такую как указатель местоположения конца фрагмента данных фиксированной длины, а также имеются ли в этой строке столбцы переменной длины.

Данные

Данные строки могут содержать следующие элементы:

- **Фиксированные данные**. Данные фиксированной длины вводятся на страницу перед данными переменной длины. Пустая строка данных фиксированной длины занимает столько же места, как и заполненная данных фиксированной длины. В таблице, содержащей только столбцы фиксированной длины, всегда хранится одно и то же количество строк на странице.
- **Блок нулевых столбцов (NB – Null Block)**. Блок нулевых столбцов представляет собой набор байтов переменной длины. Он состоит из двух байтов, в которых хранится информация о количестве столбцов. Далее следует битовая карта блока нулевых столбцов, указывающая, является ли

нулевым каждый отдельный столбец. Размер битовой карты блока нулевых столбцов составляет один бит на столбец; полученное значение округляется до ближайшего байта. Если количество столбцов составляет от 1 до 8, для них требуется однобайтовая битовая карта. Если количество столбцов составляет от 9 до 18, для них требуется двухбайтовая битовая карта.

- **Блок столбцов переменной длины (VB – Variable Block)**. Блок столбцов переменной длины состоит из двух байтов, описывающих, сколько имеется столбцов переменной длины. Дополнительные два байта на столбец указывают конец каждого столбца переменной длины. Блок столбцов переменной длины опускается, если такие столбцы отсутствуют.
- **Данные переменной длины**. Данные переменной длины вводятся на страницу после блока столбцов переменной длины. Пустая строка данных переменной длины места на странице не занимает. Таблица со столбцами переменной длины может иметь несколько длинных строк или большое количество коротких строк.

Организация больших значений данных

Строка не может быть больше страницы. Поэтому когда столбец может содержать значение, превышающее 8 000 байтов, SQL Server должен хранить эти данные отдельно, сохраняя в исходной строке только указатель на эти данные.

Спецификатор max

Спецификатор **max** может использоваться с типами данных **varchar**, **nvarchar** и **varbinary**. Он позволяет использовать эти типы данных для хранения значений, превышающих 8 000 (231) байтов.

Пользователь может управлять тем, как хранятся столбцы, определенные с использованием спецификатора **max**. Параметр **large value types out of row** (типы больших значений за пределами строки) может быть задан с помощью системной хранимой процедуры **sp_tableoption**. Когда этот параметр включен, указатель на корневой узел сбалансированного дерева для данных хранится в строке; когда этот параметр выключен, значения, который являются достаточно малыми, хранятся непосредственно в строке, а указатель используется только для значений, являющихся слишком большими, чтобы помещаться в этой строке.

Создание таблиц

При создании таблицы необходимо задать имя таблицы, имена столбцов и типы данных столбцов.

Имена столбцов должны быть уникальными для данной конкретной таблицы. Однако одно и то же имя столбца может использоваться в различных таблицах одной и той же базы данных. Для каждого столбца следует задать тип данных.

Количество элементов, которое может иметься при работе с базой данных:

- более 2 млрд объектов на базу данных, включая таблицы;
- до 1024 столбцов на таблицу;
- до 8 060 байтов на строку. (Эта примерная максимальная длина не применяется к столбцам, определенным с использованием спецификатора max.)

Параметры сортировки столбцов

Параметры сортировки определяют порядок сортировки данных, в котором осуществляется перечисление значений при выполнении последовательной сортировки данных. Различные параметры сортировки определяют порядок сортировки в зависимости от того, учитывается ли в них регистр букв, какие правила сортировки применяются для специальных символов и букв с надстрочными знаками, а также других соображений. SQL Server поддерживает хранение элементов данных с различными параметрами сортировки в одной и той же базе данных. Отдельные параметры сортировки SQL Server могут быть заданы на уровне столбца, в результате чего каждому столбцу могут быть назначены различные параметры сортировки.

Возможность столбца содержать значения NULL

В определении таблицы можно задать, могут ли в каждом столбце содержаться значения NULL. Если признаки NULL или NOT NULL не будут заданы, SQL Server определит их на основе параметров по умолчанию, заданных на уровне сеанса или базы данных. Однако эти параметры по умолчанию могут быть изменены, поэтому на них не следует полагаться.

Когда возможность столбца содержать значения NULL не задана, она определяется параметром сеанса или базы данных ANSI_NULL_DEFAULT. Когда значением этого параметра является ON, столбцы по умолчанию могут содержать значения NULL. Когда значением этого параметра является OFF, для столбцов по умолчанию используется признак NOT NULL. В SQL Server исходным значением параметра ANSI_NULL_DEFAULT является OFF.

Специальные типы столбцов

К специальным типам столбцов относятся:

- **Вычисляемые столбцы**. Вычисляемый столбец представляет собой виртуальный столбец, который не хранится в таблице в физическом виде. В SQL Server для расчета значения такого столбца, когда в нем возникнет потребность, используется формула, определенная пользователем при создании этого столбца. Эта формула определяется с использованием других столбцов этой же таблицы. Использование имени вычисляемого столбца в запросе может упростить синтаксис этого запроса.
- **Столбцы идентификаторов**. Свойство **Identity** может использоваться для создания столбцов (которые называют столбцами идентификаторов),

содержащих генерируемые системой последовательные значения, идентифицирующие каждый столбец, вставленный в таблицу. Столбец идентификаторов часто используется для значений первичного ключа. Возможность SQL Server автоматически предоставлять значения позволяет снизить затраты и повысить производительность при работе с системой. Эта возможность упрощает программирование, позволяет сохранять кроткие значения первичного ключа и снижает вероятность возникновения узких мест для пользовательских транзакций.

- **Столбцы временных меток**. Столбцы, определенные с использованием типа данных **timestamp**, имеют значение по умолчанию, состоящее из автоматически генерируемой временной метки, гарантирующей уникальность внутри этой базы данных.
- **Столбцы uniqueidentifier**. Столбцы, определенные с использованием типа данных **uniqueidentifier**, могут применяться для хранения идентификаторов GUID (Globally Unique Identifier), гарантирующих глобальную уникальность. Значение для столбца **uniqueidentifier** может генерироваться с помощью функции NEWID языка Transact-SQL.

Таблицы можно создавать с помощью обозревателя объектов в среде SQL Server Management Studio или с помощью инструкции CREATE TABLE языка Transact-SQL. В следующем примере программного кода Transact-SQL осуществляется создание таблицы с именем CustomerOrders в схеме с именем Sales. Данная таблица включает столбец идентификаторов.

```
CREATE TABLE Sales.CustomerOrders
(OrderID int identity NOT NULL,
OrderDate datetime NOT NULL,
CustomerID int NOT NULL,
Notes nvarchar(200) NULL)
```

Изменение и удаление таблиц

Определение таблицы можно изменить с помощью обозревателя объектов или с помощью инструкции ALTER TABLE языка Transact-SQL. Например, следующие инструкции Transact-SQL показывают, каким образом можно добавить столбец и изменить возможность столбца содержать значения NULL в таблице Sales.CustomerOrders, определенной ранее.

```
ALTER TABLE Sales.CustomerOrders
ADD SalesPersonID int NOT NULL
GO
ALTER TABLE Sales.CustomerOrders
ALTER COLUMN Notes nvarchar(200) NOT NULL
GO
```

Чтобы удалить таблицу из базы данных, можно удалить ее в обозревателе объектов или с помощью инструкции DROP TABLE языка Transact-SQL. В следующем примере кода Transact-SQL показывается, каким образом можно удалить таблицу с помощью инструкции DROP TABLE.

```
DROP TABLE Sales.CustomerOrders
```

Формирование сценариев Transact-SQL

Сценарии Transact-SQL могут использоваться для поддержки сценария резервного копирования, создания или обновления кода разработки базы данных, создания среды тестирования или разработки, а также для обучения новых сотрудников.

Сценарии могут формироваться для существующих объектов базы данных с помощью обозревателя объектов в среде SQL Server Management Studio. Может также использоваться мастер формирования сценариев для создания сценариев сразу же для нескольких объектов базы данных.

Обозреватель объектов

Используйте обозреватель объектов для формирования сценариев для базы данных или отдельного объекта базы данных с помощью параметров по умолчанию. Сценарий можно формировать в окне редактора запросов, файле или буфере обмена.

Сценарии можно формировать для создания или удаления объекта. У некоторых объектов базы данных имеются дополнительные параметры формирования сценариев, такие как ALTER, SELECT, INSERT, UPDATE, DELETE и EXECUTE.

Используйте следующую процедуру для формирования сценария для отдельного объекта базы данных:

1. В обозревателе объектов подключитесь к экземпляру ядра СУБД SQL Server, а затем разверните этот экземпляр.
2. Разверните обозреватель объектов и найдите в нем нужный объект.
3. Щелкните правой кнопкой мыши этот объект, а затем щелкните Создать сценарий для <тип объекта>.

Мастер формирования сценариев

Мастер формирования сценариев позволяет создавать сценарии одновременно для нескольких объектов, а также задавать различные параметры, определяющие, например, следует ли включать разрешения или ограничения, задавать параметры сортировки и т. п.

Используйте следующую процедуру, чтобы открыть мастер формирования сценариев:

1. В обозревателе объектов **Базы данных**, щелкните правой кнопкой мыши нужную базу данных, укажите элемент **Задачи**, а затем щелкните **Сформировать сценарий**.
2. На странице **Выбор базы** данных, щелкните базу данных, для которой формируется сценарий. Чтобы сформировать сценарии для всех объектов в этой базе данных, установите флажок **Внести в сценарий все объекты в выделенной базе данных**.
3. На странице **Выбор параметров сценария** выберите необходимые параметры сценариев, определяющие, например, внесение в сценарий индексов, триггеров и ключей.
4. На странице Выбор типов объектов, выберите типы объектов базы данных, для которых будет формироваться сценарий.
5. На странице (страницах) **Выбор <тип объекта>**, выберите отдельные объекты базы данных, которые требуется внести в сценарий.
6. На странице **Параметр вывода**, выберите назначение для вывода результатов мастера формирования сценариев. Сценарии можно формировать в файле, буфере обмена или новом окне редактора запросов.
7. На странице **Сводка мастера сценариев** просмотрите определенные вами параметры и нажмите кнопку **Готово.**

4 СОЗДАНИЕ ИНДЕКСОВ

Индекс — это набор страниц, связанный с таблицей (или представлением) и используемый для ускорения получения строк из таблицы или обеспечения уникальности. Например, без индекса-указателя пришлось бы листать всю книгу страница за страницей, чтобы найти сведения по нужной теме. В Microsoft SQL Server индексы используются для указания местонахождения строки и позволяют избежать просматривания всех страниц данных таблицы.

Индекс содержит ключи, построенные по одному или нескольким столбцам таблицы. Способ хранения этих ключей позволяет SQL Server быстро и эффективно находить строки, связанные со значениями ключа.

В этом разделе содержится обзор планирования, создания и оптимизации индексов, объясняются различия между кучами, кластеризованными индексами и некластеризованными индексами, а также области применения каждого из этих понятий, описывается, как создавать различные типы индексов и поддерживать их для получения оптимальной производительности.

Сведения, связанные с индексами для таблиц, представленные в данном разделе, относятся и к индексам для представлений, поэтому для простоты и во избежание повторения «таблица или представление» в разделе описываются таблицы, но материал применим и к индексам.

4.1 Планирование индексов

Планирование полезных индексов – это один из важных аспектов повышения производительности запроса. Он требует понимания и структуры индексов, и использования данных. Понимание основ хранения данных и доступа к ним является первым шагом в понимании того, как работают индексы, почему их использование может быть полезным, и по каким причинам может оказаться предпочтительным каждый из различных вариантов индексации, предлагаемых SQL Server.

Доступ к данным в SQL Server

В SQL Server доступ к данным осуществляется одним из двух способов (см. рисунок 7) [2].

1 способ: С помощью просмотра всех страниц данных в таблице, называемого просмотром строк таблицы. Выполняя просмотр строк таблицы, SQL Server:

- начинает просмотр с начала таблицы;
- просматривает все строки таблицы от страницы к странице;
- извлекает строки, соответствующие условиям запроса.

2 способ : С помощью индексов. При использовании индексов SQL Server:

- просматривает структуру дерева индексов, чтобы найти строки, запрашиваемые запросом;

- извлекает только нужные строки, соответствующие условиям запроса.

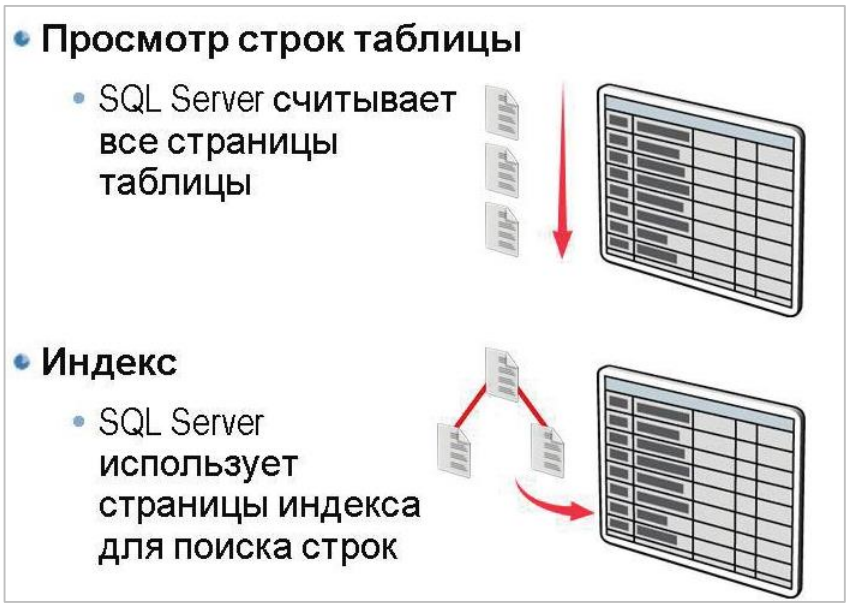

Рисунок 7 Доступ к данным

SQL Server сначала определяет, существует ли индекс. Затем оптимизатор запросов — компонент, отвечающий за оптимальное планирование выполнения запроса — определяет, какой из двух способов доступа к данным, просмотр строк таблицы или использование индексов, более эффективен.

Кластеризованный индекс

Кластеризованный индекс сортирует и сохраняет строки данных таблицы в зависимости от ключа кластеризованного индекса. Кластеризованный индекс реализован в виде сбалансированного дерева (B-дерева). В сбалансированном дереве каждая страница называется узлом индекса. Верхний узел сбалансированного дерева называется корневым узлом. Нижний уровень узлов индекса называется конечным уровнем. Все уровни индекса между корневым узлом и конечными узлами вместе называются промежуточными узлами. Каждая страница промежуточного или конечного уровня содержит указатели на предыдущую и последующую страницу, образуя дважды связанный список. Эта структура обеспечивает высокоэффективный механизм ускорения процесса нахождения данных.

В кластеризованном индексе корневой и промежуточные узлы содержат страницы индекса, хранящие строки индекса. Каждая строка индекса содержит значение ключа и указатель либо на страницу промежуточного уровня сбалансированного дерева, либо на строку данных на конечном уровне индекса. На каждом уровне страницы индекса связаны в дважды связанный список (см. рисунок 8).

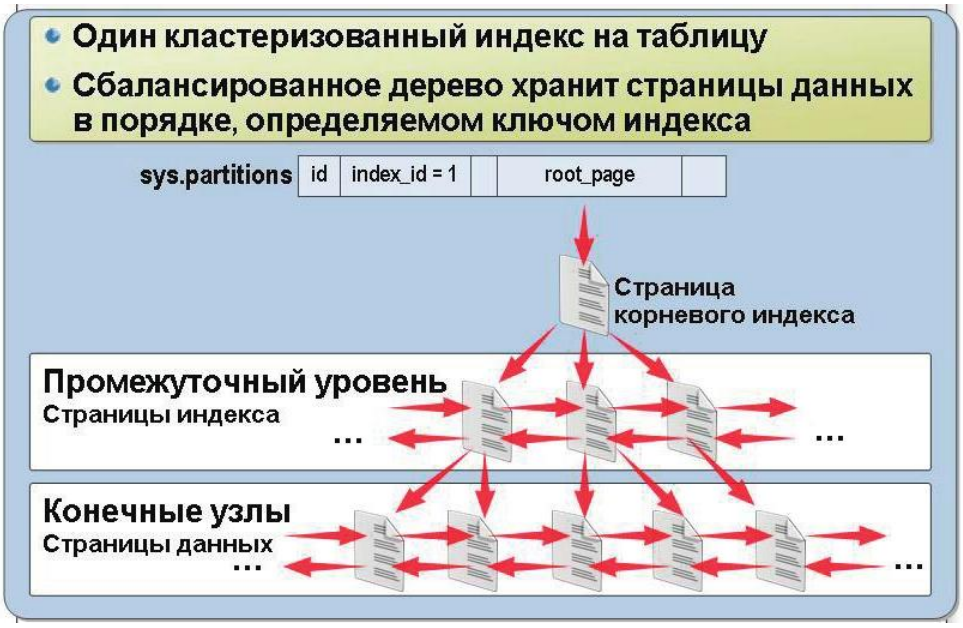

Рисунок 8 Кластеризованный индекс

Так как кластеризованный индекс определяет порядок, в котором фактически хранятся строки таблицы, у каждой таблицы может быть только один кластеризованный индекс — строки таблицы могут храниться только в одном порядке [6].

В качестве ключа кластеризованного индекса нельзя использовать столбцы со следующими типами данных: **varchar(max), nvarchar(max)**, **varbinary(max)** или **xml**.

Использование кластеризованного индекса

Так как можно использовать только один кластеризованный индекс для таблицы, необходимо гарантировать, что он будет использоваться для получения максимально возможных преимуществ. Перед созданием кластеризованного индекса необходимо понять, как будет осуществляться доступ к данным. Так как кластеризованный индекс определяет порядок, в котором строки данных таблицы хранятся в SQL Server, кластеризованные индексы больше подходят для конкретных типов данных и шаблонов использования.

Кластеризованные индексы наиболее эффективны для поддержки запросов, выполняющих следующие действия:

- Возвращение диапазона значений с помощью таких операторов, как BETWEEN, >, >=, < и <=. Поскольку данные таблицы физически хранятся в порядке индекса, то после нахождения с помощью кластеризованного индекса строки с первым значением гарантируется, что последующие индексированные значения физически находятся по соседству.
- Возвращение данных, отсортированных с помощью предложения ORDER BY или GROUP BY. Индекс для столбцов, указанных в предложении ORDER BY или GROUP BY, может избавить ядро базы данных от

необходимости сортировать данные, так как строки уже отсортированы. При этом производительность запроса повышается.

- Возвращение данных, объединенных с помощью предложений JOIN, обычно это столбцы внешних ключей.
- Возвращение больших наборов данных.

При определении кластеризованного индекса следует определять ключ индекса так, чтобы он содержал как можно меньше столбцов. Поддержание небольшого значения кластеризованного ключа увеличивает количество строк индекса, которые SQL Server может разместить на странице индекса, и уменьшает число просматриваемых уровней. Это позволяет минимизировать ввод-вывод. Кроме того, следует проанализировать столбцы, обладающие одним или несколькими из следующих свойств:

- Уникальны или содержат много различных значений. Среди таковых столбцы, определяемые как IDENTITY, так как гарантируется, что такой столбец будет уникальным в таблице. Уникальность значения ключа поддерживается в явном виде с помощью ключевого слова UNIQUE или неявно с помощью внутреннего уникального идентификатора. Если кластеризованный индекс содержит дублированные значения, SQL Server должен различать строки, содержащие идентичные значения в столбце или столбцах ключа. Это выполняется с помощью 4-байтового целого (значение uniqeidentifier) в дополнительном системном столбце uniqeidentifier. Эти уникальные идентификаторы являются внутренними для SQL Server и недоступны пользователю.
- Часто используются для сортировки данных, возвращаемых из таблицы, так как это позволяет сэкономить на операциях сортировки всякий раз, когда запрос сортирует результаты по этому столбцу.
- Доступ к которым часто осуществляется последовательно.

Кластеризованные индексы не стоит использовать в следующих случаях:

- Данные в индексированных столбцах будут часто меняться. Изменения кластеризованного индекса приводят к перемещению всей строки данных, так как ядро базы данных должно поддерживать физический порядок значений данных в строке. Это важный аргумент в системах обработки транзакций большого объема, в которых данные обычно быстро меняются.
- Ключи индексов являются широкими. Широкие ключи представляют собой составные ключи из нескольких столбцов или нескольких столбцов большого размера. Все некластеризованные индексы используют значения ключей кластеризованного индекса в качестве ключей поиска. Все некластеризованные индексы, определенные для той же таблицы, оказываются заметно больше, так как записи некластеризованного

индекса содержат ключ кластеризации, а также столбцы ключа, определенного для этого некластеризованного индекса.

Понятие кучи

Куча – это таблица без кластеризованного индекса. Для хранения строк данных никакой конкретный порядок не используется, отсутствует какой-либо конкретный порядок и в последовательности страниц данных. Эти страницы данных не объединены в связанный список. SQL Server всегда хранит страницы данных в куче, если для таблицы не определен кластеризованный индекс (см. рисунок 9).

Рисунок 9 Куча

Для обслуживания кучи SQL Server использует страницы карты распределения индекса (Index Allocation Map, IAM). Страницы IAM:

- Содержат сведения о том, где SQL Server хранит экстенты кучи. Указатель на первую страницу IAM, связанную с кучей, хранится в системной таблице **sys.partitions**. Этот указатель будет содержаться в записи с **index_id** = 0.
- Обеспечивают перемещение по куче для поиска доступного места при вставке в таблицу новых строк.
- Связывают страницы данных с таблицей. Страницы данных и строки внутри них не упорядочены и не связаны друг с другом. Единственной логической связью между страницами данных является запись в страницах IAM.

Использование кучи

Куча используется по умолчанию всегда, когда для таблицы не определен кластеризованный индекс. Кучу следует использовать, когда таблица содержит данные, структура или способ использования которых не подходят для реализации кластеризованного индекса. Для таблицы, использующей кучу, тем не менее, можно реализовать некластеризованные индексы [6].

Кучи следует использовать для таблиц, которые:

- Содержат часто меняющиеся данные, строки которых часто добавляются, удаляются и обновляются. Затраты на ведение индекса могут оказаться более дорогостоящими, чем получаемые преимущества.
- Содержат небольшое количество данных. Использование просмотра строк таблицы для поиска данных может оказаться быстрее, чем ведение и использование индекса.
- Содержат преимущественно дублированные строки данных. Просмотр строк таблицы может оказаться быстрее поиска по индексу.
- Содержат данные, которые редко записываются и читаются, например журнал аудита. Ведение индекса может привести к ненужному расходованию дискового пространства и дополнительным затратам на обслуживание.

При создании для таблицы ограничения PRIMARY KEY автоматически создается уникальный индекс для столбца (или столбцов). По умолчанию этот индекс является кластеризованными, то есть, таблица больше не будет храниться как куча. При создании этого ограничения можно определить некластеризованный индекс, задав аргумент NONCLUSTERED.

Некластеризованный индекс

Некластеризованные индексы используют ту же структуру сбалансированного дерева, что и кластеризованные индексы, за исключением того, что строки данных базовой таблицы не сортируются и хранятся в порядке, основанном на их некластеризованных ключах. В некластеризованном индексе данные и индекс хранятся отдельно, и конечный уровень индекса состоит из страниц индекса, а не из страниц данных (см. рисунок 10).

Строки в некластеризованном индексе хранятся в порядке значений ключей индекса, но упорядоченность соответствующих строк данных не гарантируется, если для таблицы не создается кластеризованный индекс. Каждая строка индекса в некластеризованном индексе содержит значение некластеризованного ключа и указатель на строку. Этот указатель указывает на строку данных, если таблица является кучей, и содержит ключ кластеризованного индекса для строки, если для таблицы создан кластеризованный индекс.

Если для индексированной таблицы создан кластеризованный индекс, столбец или столбцы, определенные в кластеризованном индексе, автоматически добавляются в конец каждого некластеризованного индекса таблицы. Это позволяет создать покрывающий запрос, не задавая столбцов кластеризованного индекса в определении некластеризованного индекса. Например, если для таблицы создан кластеризованный индекс по столбцу C, некластеризованный индекс по столбцам **B** и **A** будет использовать в качестве значений ключа столбцы **B**, **A** и **C**.

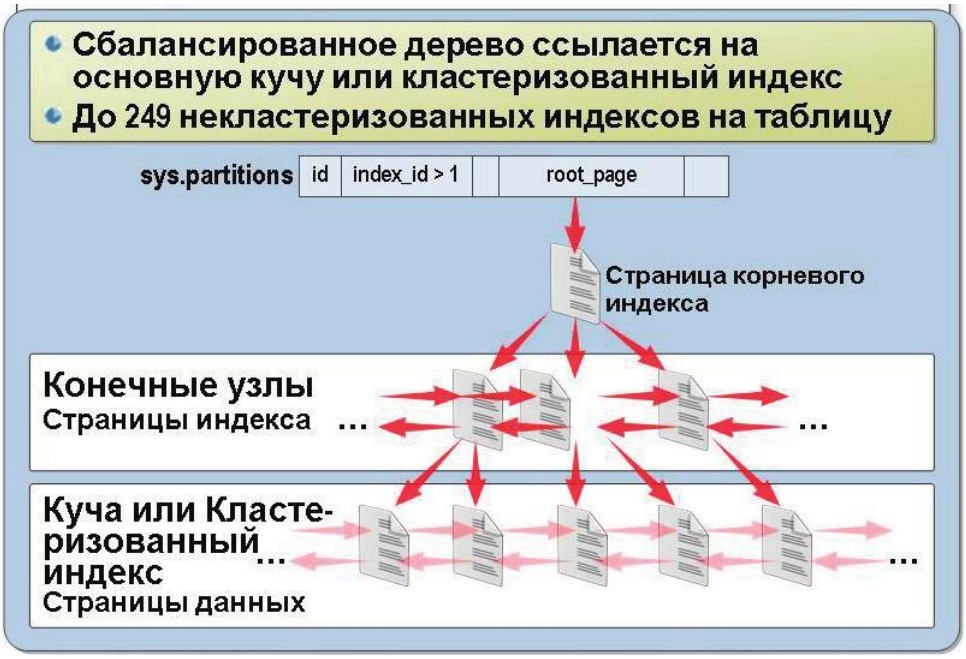

Рисунок 10 Некластеризованный индекс

Для таблицы может быть создано до 249 некластеризованных индексов. Некластеризованные индексы можно определить для таблицы независимо от того, является ли таблица кучей или для нее используется кластеризованный индекс.

В качестве ключа некластеризованного индекса нельзя использовать столбцы со следующими типами данных: **varchar(max)**, **nvarchar(max)**, **varbinary(max)** или **xml**.

Использование некластеризованного индекса

Некластеризованные индексы полезны, когда нужны различные способы поиска данных. Например, пользователь может часто выполнять поиск в базе данных по садоводству, разыскивая общеупотребительные и научные названия растений. Можно создать некластеризованный индекс для возвращения научных названий и кластеризованный индекс для возвращения общеупотребительных названий.

Некластеризованные индексы создаются для повышения производительности часто используемых запросов, не охватываемых кластеризованным индексом. Если для таблицы уже создан кластеризованный индекс, и нужно создать индекс для другого столбца, единственный вариант – использование некластеризованного индекса.

Перед созданием некластеризованных индексов нужно понять то, как будет осуществляться доступ к данным.

Максимальное повышение производительности запросов достигается, если индекс содержит все столбцы запроса. Термин покрытие относится к количеству столбцов, поддерживаемых индексом и задействованных в запросе. При полном покрытии оптимизатор запроса может найти все

значения столбца внутри индекса, то есть, данные кучи или кластеризованного индекса не используются, что приводит к уменьшению числа дисковых операций ввода-вывода. Но широкие ключи и большое количество индексов также могут отрицательно повлиять на общую производительность базы данных.

Некластеризованный индекс следует использовать, когда:

- Нужно повысить производительность запросов, использующих предложения JOIN или GROUP BY. Следует создать несколько некластеризованных индексов по столбцам, участвующим в операциях объединения и группирования, и кластеризованный индекс по любому из столбцов внешних ключей.
- Таблица обновляется редко, но содержит большие объемы данных, например, используется приложением поддержки решений. Такие приложения содержат большие объемы данных, предназначенных в основном только для чтения, и могут заметно выиграть от большого количества некластеризованных индексов. При наличии многих индексов оптимизатор запроса может выбрать из большего количества индексов, чтобы определить самый быстрый способ доступа, а низкая частота обновления базы данных означает, что обслуживание индекса вряд ли ухудшит производительность.
- Известно, что запросы не возвращают больших наборов данных.
- Нужно индексировать столбцы, часто участвующие в условиях поиска запроса — например в предложении WHERE — которые возвращают точные соответствия.
- Нужно индексировать столбцы, содержащие много различных значений, например объединение фамилии и имени. Некластеризованные индексы оптимально работают со столбцами, в которых избирательность данных лежит в диапазоне от высоко избирательных до уникальных.

Создавая некластеризованный индекс, следует учитывать:

- Создавайте кластеризованные индексы до создания некластеризованных индексов. SQL Server автоматически перестраивает существующие некластеризованные индексы в следующих случаях:
  a удаление существующего кластеризованного индекса;
  b создание кластеризованного индекса;
  c изменение столбцов, составляющих кластеризованный индекс.
- Избегайте большого количества некластеризованных индексов, если таблица используется в приложении оперативной обработки транзакций (online transaction processing, OLTP) или часто обновляется. Большое количество индексов для таблицы влияет на производительность инструкций INSERT, UPDATE и DELETE, так как все индексы должны обновляться при изменении данных в таблице.

При использовании некластеризованных индексов так же следует учитывать:

- Ядро базы данных создаст некластеризованный индекс, если не указать, что индекс должен быть кластеризованным.
- Порядок страниц конечного уровня некластеризованного индекса отличается от физического порядка таблицы.

4.2 Создание индексов

Решив, что нужно индексировать и какой индекс использовать, кластеризованный или некластеризованный, необходимо создать индекс. При создании индексов доступно множество вариантов, выбор которых может заметно повлиять на производительность индекса и удобство его обслуживания.

Индекс можно создать, используя обозреватель объектов SQL Server Management Studio или инструкцию CREATE INDEX Transact-SQL [4].

Для создания индекса с помощью Transact-SQL используйте инструкцию CREATE INDEX. Для инструкции CREATE INDEX применяется следующий синтаксис:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]
INDEX index_name ON {table | view} (column [ASC | DESC] [,...n])
INCLUDE (column [,...n ])
[WITH
    [PAD_INDEX = {ON | OFF}]
    [[,] FILLFACTOR = fillfactor]
    [[,] IGNORE_DUP_KEY = {ON | OFF}]
    [[,] ONLINE = {ON | OFF}]
    [[,] ALLOW_ROW_LOCKS = { ON | OFF}]
    [[,] ALLOW_PAGE_LOCKS = { ON | OFF}]]
[ON {partition_scheme (column) | filegroup | „default”}]
```

Не используйте в индексах столбцы, определенные с помощью типов **varchar(max)**, **nvarchar(max)**, **varbinary(max)** или **xml**. Столбцы с этими типами данных индексировать нельзя.

Инструкция CREATE INDEX предлагает и дополнительные параметры, не рассматриваемые в данном разделе.

Код следующего примера создает некластеризованный индекс по возрастанию с именем **AK_Employee_LoginID** по столбцу **LoginID** таблицы **HumanResources.Employee** в базе данных **AdventureWorks**.

```
CREATE NONCLUSTERED INDEX [AK_Employee_LoginID]
ON [HumanResources].[Employee] ( [LoginID] ASC)
```

Уникальные индексы

Уникальный индекс — это индекс, гарантирующий, что все данные в индексированном столбце являются уникальными и не содержат дублированных значений (см. рисунок 11).

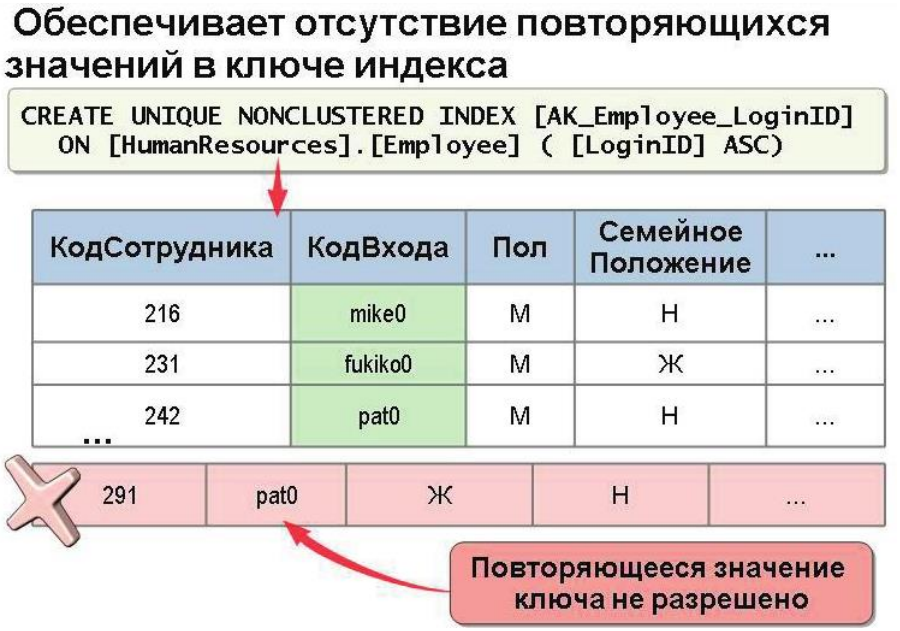

Рисунок 11 Уникальные индексы

Если существует уникальный индекс, ядро базы данных при каждом добавлении данных с помощью операции вставки проверяет, не возникает ли дублирование значений. Для операций вставки, выполнение которых привело бы к появлению дублированных значений ключа, выполняется откат, и ядро базы данных выводит сообщение об ошибке. Это справедливо, даже если операция вставки, изменяя множество строк, создает лишь одно дублированное значение. Но если выполняется попытка ввода данных, для которых существует уникальный индекс и предложение IGNORE_DUP_KEY в инструкции CREATE INDEX установлено равным ON, заканчивается неудачей только вставка строк, нарушающих уникальность индекса.

При создании индекса для таблицы, уже содержащей данные, ядро базы данных гарантирует отсутствие существующих дублированных значений.

Уникальный индекс создается для кластеризованных или некластеризованных индексов, если данные являются уникальными по своей сути. Следует создавайть уникальные индексы только для столбцов, в которых может быть обеспечена целостность сущности. Например, не следует создавать уникальный индекс для столбца **LastName** (фамилия) таблицы **Person.Contact** в базе данных **AdventureWorks**, так как фамилии некоторых сотрудников могут совпадать. Если необходимо обеспечить уникальность, вместо уникального индекса следует создать для столбца ограничение PRIMARY KEY или UNIQUE.

Если для таблицы задано ограничение PRIMARY KEY или UNIQUE, при выполнении инструкции CREATE TABLE или ALTER TABLE уникальный индекс в SQL Server создается автоматически. По умолчанию этот индекс будет кластеризованным, но можно создать этот индекс как некластеризованный, используя параметр NONCLUSTERED инструкции CREATE TABLE.

В среде SQL Server Management Studio уникальный индекс можно создать, устанавливая при создании индекса флажок «Уникальный» в диалоговом окне «Создание индекса».

В примере показана инструкция Transact-SQL, необходимая для создания некластеризованного индекса с именем **AK_Employee_LoginID** для столбца **LoginID** таблицы **HumanResources.Employee** в базе данных **AdventureWorks**.

```
CREATE UNIQUE NONCLUSTERED INDEX [AK_Employee_LoginID]
ON [HumanResources].[Employee] ( [LoginID] ASC)
```

Создание индексов для нескольких столбцов

Составной индекс определяет в качестве значения ключа несколько столбцов. С помощью составных индексов можно повысить производительность запросов, особенно если пользователи регулярно выполняют поиск данных разными способами. Но широкие ключи увеличивают требования, предъявляемые индексом к дисковому пространству.

Для составных индексов справедливы следующие требования и ограничения:

- Для создания одного составного индекса можно объединить до 16 столбцов.
- Сумма длин столбцов, образующих составной индекс, не может превосходить 900 байтов.
- Чтобы оптимизатор запроса воспользовался составным индексом, предложение WHERE запроса должно ссылаться на первый столбец составного индекса. Это ограничение не распространяется на охватывающие индексы.
- Все столбцы составного индекса должны принадлежать одной таблице, за исключением создания индекса для представления — в этом случае все столбцы должны браться из одного представления.
- Составной индекс (столбец1, столбец2) не совпадает с индексом (столбец2, столбец1) — в этих индексах используется различный порядок столбцов.

Если индекс содержит все столбцы, на которые ссылается запрос, он обычно называется покрывающим запрос. Покрытые запросы уменьшают ввод-вывод с диска и могут заметно повысить производительность запроса, так как оптимизатор запроса может получить все результаты поиска из страниц индекса, извлечение дополнительных данных со страниц данных не требуется.

Составные индексы рекомендуется создавать в следующих случаях:

- Оптимальным для поиска является использование в качестве ключа двух или более столбцов.
- Запросы ссылаются только на входящие в индекс столбцы.

При создании составного индекса следует:

- Сначала определить наиболее уникальный столбец. Первый столбец, определенный в инструкции CREATE INDEX, является столбцом наивысшего порядка.
- Использовать составные индексы для таблиц с ключами из нескольких столбцов.
- Использовать составные индексы для увеличения производительности запросов и уменьшения числа индексов, создаваемых для таблицы.

Включенные столбцы

SQL Server позволяет определить дополнительные столбцы таблицы, содержимое которых хранится вместе с некластеризованным индексом. Включение столбцов, не являющихся ключом, позволяет создавать некластеризованные индексы, покрывающие большее количество запросов.

Неключевые столбцы обеспечивают следующие дополнительные преимущества:

- Они могут относиться к типам данных, которые нельзя использовать в качестве столбца ключа индекса.
- Ядро базы данных не учитывает их при подсчете числа столбцов ключа индекса или размера ключа индекса.

Индекс с включенными столбцами можно создать с помощью среды SQL Server Management Studio. Для этого выберите дополнительно включенные столбцы в разделе **Включить столбцы** диалогового окна **Создание индекса**.

Следующий пример показывает использование инструкции Transact-SQL, необходимой для включения столбцов **ContactID** и **NationalIDNumber** в индекс с именем **AK_Employee_LoginID** для столбца **LoginID** таблицы **HumanResources.Employee** в базе данных **AdventureWorks**.

```
CREATE UNIQUE NONCLUSTERED INDEX [AK_Employee_LoginID]
ON [HumanResources].[Employee] ( [LoginID] ASC)
INCLUDE ( [ContactID], [NationalIDNumber])
```

Индексы для вычисляемых столбцов

Индексы для вычисляемых столбцов можно создавать в следующих случаях:

- Выражение вычисляемого столбца является точным и детерминированным. Детерминированные выражения всегда возвращают один и тот же результат для заданного набора входов. Точные выражения не содержат данных типов с плавающей запятой, и такие типы данных не входят в определение выражений.

- При выполнении инструкции CREATE TABLE параметр уровня соединения ANSI_NULLS включен (ON). Функция OBJECTPROPERTY с помощью свойства IsAnsiNullsOn сообщает, включен ли этот параметр.
- У соединения, для которого создается индекс — и у всех соединений, пытающихся выполнить инструкции INSERT, UPDATE или DELETE, которые изменят значения индекса — есть шесть параметров SET со значением ON и один параметр со значением OFF (выключен). Значение ON должно быть установлено для следующих параметров:

  a ANSI_NULLS;
  b ANSI_PADDING;
  c ANSI_WARNINGS;
  d CONCAT_NULL_YIELDS_NULL;
  e QUOTED_IDENTIFIER;
  f ARITHABORT.

В SQL Server установка для ANSI_WARNINGS значения ON неявно устанавливает значение ARITHABORT равным ON. В предыдущих версиях SQL Server параметр ARITHABORT необходимо было явно устанавливать равным ON.

Для параметра NUMERIC_ROUNDABORT должно быть установлено значение OFF.

Оптимизатор запросов игнорирует индекс вычисляемого столбца для любой инструкции SELECT, выполняемой соединением с отличающимися настройками этих параметров.

Параметры для включения свободного пространства в индексах

Доступность свободного пространства в странице индекса может заметно повлиять на производительность операций обновления индекса. Если должна быть вставлена запись индекса, а свободное пространство отсутствует, должна быть создана новая страница индекса, а содержание старой страницы разделено между двумя этими страницами. Такие действия, если происходят слишком часто, могут повлиять на производительность. SQL Server предлагает два важных параметра, позволяющих управлять объемом свободного пространства, поддерживаемого внутри индекса: FILLFACTOR и PAD_INDEX.

Параметр FILLFACTOR

Чтобы уменьшить разбиение страниц, параметр FILLFACTOR позволяет задать для страниц индекса конечного уровня процент свободного пространства (от 0 до 100). Этот процент определяет, насколько должны быть заполнены страницы конечного уровня. Например, при коэффициенте заполнения 65 процентов заполняется 65 процентов страницы конечного уровня, а 35 процентов пространства страницы свободно для новых строк.

Таблица 4.2 содержит значения параметра FILLFACTOR и типичные среды, в которых используются эти значения коэффициента заполнения.

Таблица 6. Значение параметра FILLFACTOR

Процент FILLFACTOR	**Страницы конечного уровня**	**Страницы неконечного уровня**	**Действия со значениями ключа**	**Типичная бизнес-среда**
1–99	Заполнение до заданного процента	Оставляет место для одной записи индекса	Изменение от умеренного до сильного	Смешанная или OLTP
100	Полное заполнение	Оставляет место для одной записи индекса	Отсутствие изменения или незначительное изменение	OLAP

Значение коэффициента заполнения, которое применялось к индексу последним, хранится в системной таблице **sysindexes** вместе с другими сведениями об индексе. Значение коэффициента заполнения по умолчанию можно изменить на уровне сервера с помощью системной хранимой процедуры **sp_configure**.

Основное назначение параметра FILLFACTOR — отсрочить разбиение страницы. Следовательно, он не имеет смысла для таблиц, кластеризованных по столбцу идентификаторов, так как для них не используется разбиение страниц.

Значение коэффициента заполнения, заданное для таблицы, зависит от того, как часто изменяются данные (инструкциями INSERT и UPDATE), и от среды организации. Обычно следует:

- Использовать низкое значение коэффициента заполнения для сред оперативной обработки транзакций (online transaction processing, OLTP). Это обеспечит максимальное пространство для роста в таблицах, для которых часто выполняются операции вставки строк или часто изменяются значения индекса.
- Использовать высокое значение коэффициента заполнения для сред оперативного анализа данных (online analytical processing, OLAP).

Параметр PAD_INDEX

Параметр PAD_INDEX позволяет определить, применяется или нет коэффициент заполнения, применяемый к страницам конечного уровня, и к страницам неконечного уровня. Параметр PAD_INDEX можно использовать, только если определен параметр FILLFACTOR, так как значение процента PAD_INDEX определяется на основе процентного значения, определенного для FILLFACTOR.

В Таблице 4.3. показано влияние значений параметра FILLFACTOR на использование параметра PAD_INDEX и типичную среду, в которой используются значения PAD_INDEX [1].

Таблица 7. FILLFACTOR для PAD_INDEX

Процент FILLFACTOR	Страницы конечного уровня	Страницы неконечного уровня	Действия со значениями ключа	Типичная бизнес-среда
1–99	Заполнение до заданного процента	Заполнение до заданного процента	Изменение от умеренного до сильного	OLTP

По умолчанию SQL Server всегда оставляет достаточно места для размещения хотя бы одной строки индекса максимального размера для каждой страницы неконечного уровня, независимо от значения коэффициента заполнения.

Количество элементов страницы неконечного уровня индекса никогда не может быть меньше двух, независимо от значения коэффициента заполнения.

Значения параметров FILLFACTOR и PAD_INDEX можно настроить при создании индекса с помощью обозревателя объектов в среде SQL Server Management Studio. На вкладке «Параметры» диалогового окна «Создание индекса» установите флажок «Коэффициент заполнения». Это позволяет ввести значение коэффициента заполнения и установить флажок «Дополнить индекс», который позволяет включать и выключать параметр PAD_INDEX.

В примере показана инструкция Transact-SQL, необходимая для установки значения FILLFACTOR равным 65 процентам и включения параметра PAD_INDEX при создании индекса.

```
CREATE UNIQUE NONCLUSTERED INDEX [AK_Employee_LoginID]
ON [HumanResources].[Employee] ( [LoginID] ASC)
WITH ( FILLFACTOR = 65, PAD_INDEX = ON)
```

Способы получения сведений об индексах

Перед созданием, изменением или удалением индекса может понадобиться просмотреть сведения о существующих индексах. SQL Server предлагает множество способов получения сведений об индексах. Эти способы можно разделить на следующие категории:

- среда SQL Server Management Studio;
- системные хранимые процедуры;
- представления каталогов:
- системные функции.

4.3 Оптимизация индексов

Эффективность индексов играет важную роль в общей производительности базы данных. Следовательно, важно убедиться, что индексы с самого начала

разрабатываются и реализовываются так, чтобы наилучшим образом поддерживать работу приложений. После реализации индексов необходимо обслуживать индексы, чтобы гарантировать их постоянную оптимальную производительность.

При выполнении с данными в базе данных операций добавления, изменения или удаления, индексы становятся фрагментированными. В зависимости от бизнес-среды и назначения приложения базы данных фрагментация может улучшить или ухудшить производительность, но она требует обслуживания в соответствии с потребностями бизнеса.

Фрагментация индекса

Фрагментация индекса – это неэффективное использование страниц внутри индекса. Фрагментация возникает спустя некоторое время по мере изменения данных. Например, когда строки данных добавляются в таблицу или удаляются из нее, или когда изменяются значения в индексированных столбцах, SQL Server настраивает страницы индекса в соответствии с выполненными изменениями и с целью обслуживания хранилища индексированных данных. Настройка страниц индекса называется разбиением страниц. Процесс разбиения увеличивает размер таблицы и время обработки запросов.

Тип фрагментации

Существует два типа фрагментации индекса, внешняя и внутренняя:

Внутренняя Неэффективное использование страниц внутри индекса из-за того, что количество данных, хранимое внутри каждой страницы, меньше, чем может вместить страница данных.

Внутренняя фрагментация приводит к увеличению логического и физического ввода-вывода, и для хранения строк в кэш-памяти требуется больше памяти. Соответствующие дополнительные чтения могут привести к падению производительности запросов. Но для таблиц с большим количеством операций вставки внутренняя фрагментация может обеспечить определенные преимущества, так как вероятность разбиения страницы при вставке уменьшается.

Внешняя Неэффективное использование страниц внутри индекса из-за неправильного логического порядка страниц.

Внешняя фрагментация замедляет дисковый доступ к строкам вследствие позиционирования головки диска и не может создавать никаких преимуществ.

Обнаружение фрагментации

Для определения экстента, приводящего к фрагментации индексов, можно использовать среду SQL Server Management Studio или динамическую функцию управления sys.dm_db_index_physical_stats.

Для просмотра сведений о фрагментации индексов в среде SQL Server Management Studio, следует открыть окно Свойства для нужного индекса, а затем выберать страницу Фрагментация. Кроме ряда основных свойств страниц индекса, окно Свойства показывает среднее заполнение страниц и общую фрагментацию по индексу в виде процента. Чем больше это значение, тем больше фрагментирован индекс.

Для просмотра фрагментации в конкретном индексе, всех индексов в таблице или индексированном представлении, всех индексов в базе данных или всех индексов во всех базах данных можно использовать sys.dm_db_index_physical_stats. Аргументы, передаваемые функции sys.dm_db_index_physical_stats, включают идентификаторы базы данных, таблицы, индекса и секции, которые нужно оценить. Набор результатов функции включает столбец avg_fragmentation_in_percent, показывающий среднюю фрагментацию индекса в виде процента (то же значение, что и в окне свойств индекса в среде SQL Server Management Studio).

В следующем примере кода показано, как получать среднюю фрагментацию всех индексов для таблицы Production.Product с помощью sys.dm_db_index_physical_stats.

```
SELECT a.index_id, name, avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats (DB_ID(N'AdventureWorks'),
    OBJECT_ID(N'Production.Product'), NULL, NULL, NULL) AS a
    JOIN sys.indexes AS b ON a.object_id = b.object_id AND
          a.index_id =
b.index_id;
```

Результаты запроса:

index_id	имя	avg_fragmentation_in_percent
1	PK_Product_ProductID	23,0769230769231
2	AK_Product_ProductNumber	50
3	AK_Product_Name	66,6666666666667
4	AK_Product_rowguid	50

Параметры дефрагментации индексов

Существует два способа дефрагментации индекса: реорганизация и перестроение.

Реорганизация индекса дефрагментирует конечный уровень кластеризованных и некластеризованных индексов таблиц, физически изменяя порядок страниц конечного уровня для соответствия логическому порядку (слева направо) узлов конечного уровня. Упорядочивание страниц улучшает производительность просмотра индексов. Индекс реорганизуется внутри существующих страниц, выделенных для индекса, новые страницы не выделяются. Если индекс занимает несколько файлов, файлы реорганизовываются по одному. Страницы не перемещаются между файлами.

Реорганизация индекса также сжимает страницы индекса. Все пустые страницы, созданные этим сжатием, удаляются, высвобождая дисковое пространство. Сжатие основано на значении коэффициента заполнения в представлении каталога sys.indexes.

Перестроение индекса удаляет индекс и создает новый. При этом фрагментация исчезает, а дисковое пространство освобождается с помощью сжатия страниц, используя заданное или существующее значение коэффициента заполнения, строки индекса упорядочиваются заново в смежных страницах (при необходимости выделяются новые страницы). Это может повысить быстродействие диска, уменьшая число чтений страниц, необходимое для получения запрошенных данных.

Сравнение реорганизации и перестройки индексов

Решение о том, реорганизовывать или перестраивать индекс для устранения дефрагментации, должно основываться на существующем уровне дефрагментации индекса, сообщаемого средой SQL Server Management Studio или процедурой **sys.dm_db_index_physical_stats**.

Рекомендации по оптимальному подходу к устранению дефрагментации различной степени:

avg_fragmentation_in_percent < = 30% : реорганизация;

avg_fragmentation_in_percent > 30% : требуется индекс перестроить.

Если фрагментация составляет меньше 30 процентов, но реорганизация индекса привела к незначительному улучшению, следует попытаться перестроить индекс.

Реорганизация индекса

Для дефрагментации индексов можно использовать предложение REORGANIZE инструкции ALTER INDEX. Инструкция ALTER INDEX с предложением REORGANIZE заменяет инструкцию DBCC INDEXDEFRAG в предыдущих версиях SQL Server.

В следующем примере кода показывается синтаксис инструкции ALTER INDEX при ее использовании для реорганизации индекса **AK_Product_Name** для таблицы **Production.Product**.

```
ALTER INDEX AK_Product_Name ON Production.Product
REORGANIZE
```

Можно также реорганизовать все индексы таблицы, как показано в следующем примере.

```
ALTER INDEX ALL ON Production.Product
REORGANIZE
```

Перестройка индекса

Для дефрагментации индексов можно использовать предложение REBUILD инструкции ALTER INDEX. Инструкция ALTER INDEX с предложением REBUILD заменяет инструкцию DBCC DBREINDEX в предыдущих версиях SQL Server.

В следующем примере показывается синтаксис инструкции ALTER INDEX при ее использовании для перестройки индекса **AK_Product_Name** для таблицы **Production.Product**.

```
ALTER INDEX AK_Product_Name ON Production.Product
REBUILD
```

5 ВНЕДРЕНИЕ ПРЕДСТАВЛЕНИЙ

Большинство баз данных содержат представления для обеспечения удобного способа доступа к данным с помощью предварительно определенного запроса. Умение создавать представления упростит пользователям доступ к необходимым данным и повысит эффективность и производительность их баз данных.

В разделе описаны представления, преимущества, которые они обеспечивают, создание представления с помощью Microsoft SQL Server Management Studio и Transact-SQL, параметры, доступные при создании представлений, и как осуществлять поиск сведений о представлениях. Также рассматриваются ограничения на изменение данных в представлениях и то, как представления могут повысить производительность базы данных.

5.1 Представления

Представление — это виртуальная таблица, чье содержимое определяется запросом. Как и реальная таблица, представление состоит из набора именованных столбцов и строк данных. Представление существует в базе данных в виде хранимого набора значений данных, только если оно проиндексировано. Используются строки и столбцы данных из таблиц, на которые выполняются ссылки в запросе, определяющем представление, и они создаются динамически при ссылке на представление. Таблицы, запрашиваемые в представлении, называются базовыми таблицами (см. рисунок 12) [6].

Типичные примеры представлений:

- подмножество строк или столбцов базовой таблицы;
- объединение одной или нескольких базовых таблиц;
- соединение одной или нескольких базовых таблиц;
- статистическая сводка базовой таблицы;
- подмножество другого представления или некоторая комбинация представлений и базовых таблиц.

Преставления обычно используются для:

- Предоставления пользователям возможности сосредоточиться на интересующих их данных и на конкретных задачах, за которые они ответственны. Оставление за пределами представления ненужных или конфиденциальных данных.
- Упрощения работы пользователей с данными. Можно определить часто используемые соединения, проекты, запросы UNION и SELECT как представления, чтобы пользователям не нужно было указывать все условия и квалификацию каждый раз при выполнении дополнительной операции с этими данными.

- Повышения безопасности благодаря предоставлению пользователям возможности обращаться к данным через представление без получения разрешения на прямой доступ к основным базовым таблицам представления.
- Обеспечения обратной совместимости путем определения эмуляции представлением таблицы, которая существовала ранее, но схема которой была изменена.
- Определения набора данных, которые пользователь может экспортировать и импортировать в SQL Server.
- Обеспечения консолидированного представления секционированных данных, т. е. сходных данных, которые хранятся в разных таблицах.

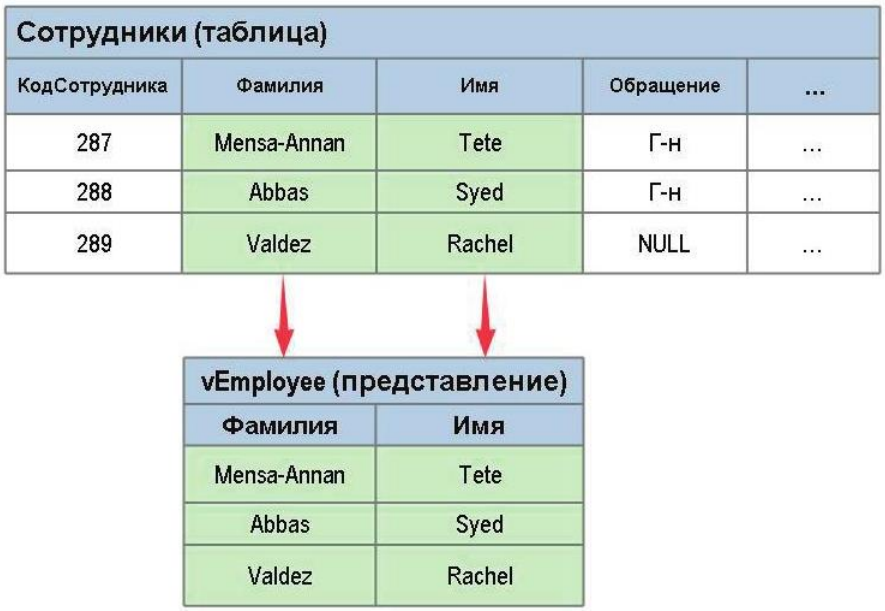

Рисунок 12 Представления

Типы представлений

В SQL Server возможно создание представлений трех типов:
- стандартные представления;
- индексированные представления;
- секционированные представления.

Стандартные представления

Стандартное представление объединяет данные из одной или нескольких базовых таблиц в новую виртуальную таблицу. Сохраняется только определение стандартного представления, а не строки представления. При каждой ссылке на это представление ядро базы данных создает данные для этого представления динамически. Стандартные представления — это тип представлений, наиболее часто используемый.

Индексированные представления

Индексированные представления — это представления, которые были материализованы, то есть, рассчитаны и сохранены. Представление индексируется с помощью создания уникального индекса представления. Индексированные представления существенно повышают производительность некоторых типов запросов. Индексированные представления хорошо работают для запросов, которые объединяют много строк, но они не подходят для базовых таблиц, которые часто обновляются.

Секционированные представления

Секционированные представления соединяют секционированные по горизонтали данные из одной или нескольких таблиц на одном или нескольких серверах. При этом данные из нескольких базовых таблиц выглядят так, как если бы они принадлежали одной таблице. Представление, которое соединяет таблицы одного и того же экземпляра SQL Server называется локальным секционированным представлением. Представление, соединяющее данные разных таблиц серверов, называется распределенным секционированным представлением.

Локальные секционированные представления включены в SQL Server только для обеспечения обратной совместимости; они устарели. Для локального секционирования данных предпочтительнее использовать секционированные таблицы.

Преимущества представлений

Представления обеспечивают следующие преимущества:

- **Сосредоточенность пользователя на данных**. Представления создают управляемую среду, которая разрешает доступ к одним данным, тогда как другие данные скрываются. Ненужные, конфиденциальные или неподходящие данные можно исключить из представления. Пользователи могут управлять отображением данных в представлении, как в таблице. Кроме того, с соответствующими разрешениями и некоторыми ограничениями пользователи могут изменять данные, создаваемые представлением.
- **Маскировка сложности базы данных**. Представления скрывают от пользователя сложность структуры базы данных. Это позволяет разработчикам изменять структуру, не влияя на взаимодействие пользователя с этой базой данных. Кроме того, пользователи могут видеть более удобную версию данных, поскольку представления могут быть созданы с именами, которые проще понять, чем зашифрованные названия, которые часто используются в базах данных. Сложные запросы, включая распределенные запросы неоднородных данных, также могут маскироваться с помощью представлений. Пользователь запрашивает

представление, вместо того, чтобы писать запрос или выполнять сценарий.

- **Упрощение управления пользовательскими разрешениями**. Вместо предоставления разрешений пользователям на запрос конкретных столбцов в базовых таблицах, владельцы баз данных могут предоставлять разрешения пользователям запрашивать данные только через представления. Это также защищает изменения в структуре основных базовых таблиц. Пользователи могут продолжать запрашивать представление без прерывания.
- **Повышение производительности**. Представления позволяют сохранять результаты сложных запросов. Эти сводные данные могут использоваться в других запросах. Представления также позволяют секционировать данные. Можно размещать индивидуальные разделы на разных компьютерах и без проблем объединять их для пользователя.
- **Организация данных для экспорта в другие приложения**. Можно создать представление на основе сложного запроса, соединяющее две и более таблицы, и экспортировать данные в другое приложение для дальнейшего анализа.

5.2 Создание представлений и управление ими

Рассмотрим процедуры создания, изменения и удаления представлений, поясним, как избежать разорванных цепочек владения, скрыть определения представления и получить сведения о представлениях в базе данных.

Синтаксис для создания представлений

Представление можно создать с помощью инструкции CREATE VIEW Transact-SQL или с помощью графического интерфейса структуры в SQL Server Management Studio. С помощью инструкции SELECT можно указать содержание представления, как части определения представления.

Чтобы отличать представления от таблиц, необходимо разработать согласованные правила именования. Можно, например, добавлять букву «v» или слово «view» в качестве суффикса имени каждого создаваемого представления. Этот подход позволяет легко различать таблицы и представления.

Создание представлений

Чтобы создать представление, используйте конструктор представлений в SQL Server Management Studio или инструкцию CREATE VIEW языка Transact-SQL. Чтобы открыть конструктор представлений в обозревателе объектов, разверните базу данных, с которой будете работать, щелкните правой кнопкой узел «Представление» и выберите «Новое представление». Затем можно конструировать свое представление, используя графический интерфейс, в

котором можно выбирать таблицы и столбцы для включения в представление, определять связи столбцов, ограничивать возвращаемые строки и настраивать такие параметры, как псевдонимы столбцов и порядок сортировки, которые используются для создания представления.

Инструкция CREATE VIEW имеет синтаксис.

```
CREATE VIEW [schema_name . ] view_name [(column [ ,...n ])]
[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]]
AS select_statement [ ; ]
[WITH CHECK OPTION]
```

Пример кода иллюстрирует создание представления **HumanResources.vEmployee**, которое состоит из набора столбцов из таблиц базы данных **AdventureWorks**.

```
CREATE VIEW [HumanResources].[vEmployee]
AS
SELECT
    e.[EmployeeID],c.[Title],c.[FirstName],c.[MiddleName],c.[Las
          tName]
    ,c.[Suffix],e.[Title] AS
          [JobTitle],c.[Phone],c.[EmailAddress]
    ,c.[EmailPromotion],a.[AddressLine1],a.[AddressLine2],a.[City]
    ,sp.[Name] AS [StateProvinceName],a.[PostalCode]
    ,cr.[Name] AS [CountryRegionName],c.[AdditionalContactInfo]
FROM [HumanResources].[Employee] e
    INNER JOIN [Person].[Contact] c
    ON c.[ContactID] = e.[ContactID]
    INNER JOIN [HumanResources].[EmployeeAddress] ea
    ON e.[EmployeeID] = ea.[EmployeeID]
    INNER JOIN [Person].[Address] a
    ON ea.[AddressID] = a.[AddressID]
    INNER JOIN [Person].[StateProvince] sp
    ON sp.[StateProvinceID] = a.[StateProvinceID]
    INNER JOIN [Person].[CountryRegion] cr
    ON cr.[CountryRegionCode] = sp.[CountryRegionCode]
```

Требования к созданию представлений

При создании представлений надо помнить следующее:

- Необходимо иметь роль **sysadmin**, **db_owner**, роль **db_ddladmin** или иметь разрешение CREATE VIEW в базе данных и разрешение ALTER SCHEMA в схеме, в которой должно быть создано представление. Также необходимо иметь разрешение SELECT во всех таблицах или представлениях, на которые есть ссылка в данном представлении.
- Создавать представления можно только в текущей базе данных.

- Имя создаваемого представления должно соответствовать правилам для идентификаторов и должно отличаться от имен других представлений или таблиц в базе данных.
- Можно создавать представления на основе других представлений. Вложения не должны превышать 32 уровня и могут также быть ограничены сложностью представления и доступным объемом памяти.
- Представление может содержать не более 1024 столбцов.
- Имена столбцов необходимо указать в следующих случаях:
  a Любой из столбцов в представлении является результатом арифметического выражения, встроенной функции или константой.
  b Любые столбцы в таблицах, которые будут соединены, имеют одинаковое имя.
- Невозможно создание временных представлений и создание представлений во временных таблицах.
- С представлением нельзя связать объекты Rule или Default.
- Нельзя связать с представлениями триггеры AFTER, можно только триггеры INSTEAD OF.
- Нельзя включить предложение COMPUTE или COMPUTE BY, а также ключевое слово INTO в запрос, который определяет представление.
- Нельзя включить предложение ORDER BY в запрос, который определяет представление, если только в списке выбора инструкции SELECT нет предложения TOP.
- Нельзя включить предложение OPTION, определяющее совет для запроса, в запрос, который определяет представление.
- Нельзя включить предложение TABLESAMPLE в запрос, который определяет представление.

Синтаксис, используемый для изменения и удаления представлений

Если необходимо изменить представление, можно изменить его с помощью SQL Server Management Studio или с помощью выполнения инструкции ALTER VIEW языка Transact-SQL.

Если представление больше не нужно, его можно удалить из базы данных с помощью SQL Server Management Studio или с помощью выполнения инструкции DROP VIEW языка Transact-SQL.

Определение представления можно изменить, открыв конструктор представлений в обозревателе объектов, или с помощью выполнения инструкции ALTER VIEW Transact-SQL. Можно изменять таблицы и столбцы, входящие в представление, изменять связи столбцов, ограничивать возвращаемые представлением строки и изменять такие параметры, как

псевдонимы столбцов и порядок сортировки, которые используются для создания представления

Инструкция ALTER VIEW изменяет определение представления, включая индексированные представления, не воздействуя на зависимые хранимые процедуры или триггеры. Это позволяет сохранять разрешения для представления. На эту инструкцию распространяются те же ограничения, что и на инструкцию CREATE VIEW. Инструкция ALTER VIEW имеет следующий синтаксис.

```
ALTER VIEW [schema_name . ] view_name [(column [,...n ])]
[WITH <view_attribute> [,...n]]
AS select_statement [ ; ]
[WITH CHECK OPTION]

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

Следующий код изменяет представление **HumanResources.vEmployee** базы данных **AdventureWorks** и удаляет все почтовые адреса из представления.

```
ALTER VIEW [HumanResources].[vEmployee]
AS
SELECT
    e.[EmployeeID]
    ,c.[Title]
    ,c.[FirstName]
    ,c.[MiddleName]
    ,c.[LastName]
    ,c.[Suffix]
    ,e.[Title] AS [JobTitle]
    ,c.[Phone]
    ,c.[EmailAddress]
FROM [HumanResources].[Employee] e
    INNER JOIN [Person].[Contact] c
    ON c.[ContactID] = e.[ContactID]
```

Удаление представления удаляет его определение и все назначенные ему разрешения. Кроме того, если пользователи запрашивают любое представление, которое ссылается на удаленное, они получают сообщение об ошибке. Однако удаление таблицы, на которую ссылается представление, не удаляет автоматически и представление. Представление следует удалить явно.

Удалить представление можно, удалив его в обозревателе объектов, или с помощью инструкции DROP VIEW. Инструкция DROP VIEW имеет следующий синтаксис.

```
DROP VIEW [ schema_name . ] view_name [ ...,n ] [ ; ]
```

Следующий код иллюстрирует инструкцию, используемую для удаления представления **HumanResources.vEmployee** из базы данных **AdventureWorks**.

```
DROP VIEW [HumanResources].[vEmployee]
```

Влияние цепочек владения на представления

Представления зависят от базовых представлений или таблиц. Когда SQL Server преобразует содержание представления, он проверяет иерархию таблицы и зависимости от представления, чтобы найти соответствующее содержание. Когда несколько объектов баз данных таким образом последовательно обращаются друг к другу, эта последовательность называется цепочкой. SQL Server оценивает разрешения в объектах цепочки не так, как если бы доступ к этим объектам осуществлялся по отдельности (см. рисунок 13) [6].

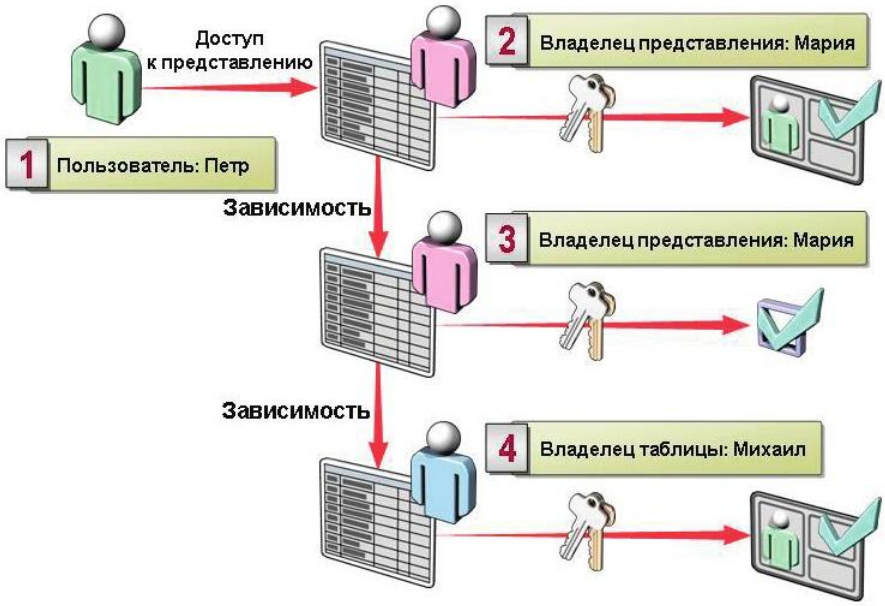

Рисунок 13 Цепочки владения

Можно настроить в SQL Server разрешение цепочек владения между указанными базами данных или между всеми базами данных в пределах одного экземпляра SQL Server. Цепочки владения между базами данных отключены по умолчанию, и не следует включать их, если это не требуется.

Проверка SQL Server разрешения в цепочке владения

Когда при доступе пользователя к объекту базы данных (например, представлению) к этому объекту обращается другой объект (например,

таблица), SQL Server не проверяет разрешения пользователя для второго объекта, если владелец обоих объектов один и тот же. Это означает, что когда пользователь имеет разрешение для доступа к представлению, SQL Server не оценивает разрешения пользователя для каждой базовой таблицы или представления, если их владелец является владельцем исходного представления. SQL Server оценивает разрешения пользователя для объекта только при доступе к объекту другого владельца.

Цепочка владения позволяет управлять доступом к нескольким объектам, например к нескольким таблицам, с помощью настройки разрешений для одного объекта, например для представления. Цепочка владения также обеспечивает небольшое увеличение производительности в сценариях, в которых разрешается пропускать проверки.

Чтобы избежать повреждения цепочек владения, убедитесь, что все представления и базовые таблицы, а также функции имеют одного и того же владельца.

Обратите внимание, что цепочки владения применимы и к другим типам объектов базы данных, а также к таблицам и представлениям.

Источники сведений о представлениях

Сведения о существующих представлениях могут потребоваться перед созданием, изменением или удалением представления. В SQL Server можно использовать следующие источники для получения сведений о представлениях:

- SQL Server Management Studio.
- представление каталога **sys.views**.
- системная хранимая процедура **sp_helptext**.
- представление каталога **sys.sql_dependencies**.

6 РЕАЛИЗАЦИЯ ХРАНИМЫХ ПРОЦЕДУР

При работе с базами данных требуется разрабатывать и реализовывать логику для установления бизнес-правил или обеспечения согласованности данных с помощью хранимых процедур, изменять и обслуживать существующие хранимые процедуры.

6.1 Основные сведения о хранимых процедурах

Хранимая процедура — это группа инструкций Transact-SQL, собранных в одном плане выполнения. С помощью хранимых процедур достигается согласованная реализация алгоритма в нескольких приложениях.

Хранимая процедура — это именованная коллекция инструкций Transact-SQL, которая хранится на сервере в самой базе данных. Хранимые процедуры представляют собой метод включения повторяющихся задач, они поддерживают объявленные пользователем переменные, условное выполнение и другие мощные функции программирования.

Хранимые процедуры в Microsoft SQL Server аналогичны процедурам в других языках программирования, в которых они могут:

- содержать инструкции, выполняющие операции в базе данных, включая возможность вызова других хранимых процедур;
- принимать входные параметры;
- возвращать значение состояния вызывающей хранимой процедуре или пакету, чтобы указать успешность выполнения или сбой;
- возвращать несколько значений вызывающей хранимой процедуре или клиентскому приложению в форме выходных параметров.

Преимущества хранимых процедур

Хранимые процедуры имеют много преимуществ по сравнению с выполнением нерегламентированных запросов Transact-SQL.

- **Включают бизнес-функциональность и создают многократно используемую логику приложений**. Бизнес-правила или политики, содержащиеся в хранимых процедурах, могут быть изменены в одном месте. Все клиенты могут использовать одни и те же хранимые процедуры для обеспечения согласованного доступа к данным и изменения их.
- **Защищают пользователей от необходимости вникать в подробности таблиц в базе данных**. Если набор хранимых процедур поддерживает все бизнес-функции, которые необходимо выполнять пользователям, пользователям никогда не потребуется напрямую обращаться к таблицам.
- **Обеспечивают механизмы защиты**. Пользователям может быть предоставлено разрешение на выполнение хранимой процедуры, даже

если они не имеют разрешения на доступ к таблицами или представлениям, на которые эта хранимая процедура ссылается.

- **Повышают производительность**. Хранимые процедуры реализуют многие задачи как последовательность инструкций Transact-SQL. К результатам первых инструкций Transact-SQL может быть применена условная логика для определения того, какие следующие инструкции Transact-SQL должны выполняться. Все эти инструкции Transact-SQL и условная логика становятся частью одного плана выполнения на сервере.
- **Уменьшают объем сетевого трафика**. Вместо отправки по сети сотен инструкций Transact-SQL пользователи могут выполнить комплексную операцию, отправив одну инструкцию, что сократит количество запросов, проходящих между клиентом и сервером.
- **Уменьшают уязвимость по отношению к атакам с использованием изменения SQL-запроса**. Использование явно определенных параметров в коде SQL уменьшает возможность отправки хакерами внедренных инструкций SQL в значения параметров.

Создание хранимых процедур

Для создания хранимых процедур используется инструкция CREATE PROCEDURE. Хранимые процедуры могут создаваться только в текущей базе данных, кроме временных хранимых процедур, которые всегда создаются в базе данных **tempdb**. Создание хранимой процедуры очень похоже на создание представления. Сначала напишите и протестируйте инструкции Transact-SQL, которые будут включены в хранимую процедуру. Затем, если будут получены ожидаемые результаты, создайте хранимую процедуру.

Инструкция CREATE PROCEDURE содержит много возможных параметров, как можно видеть из примера части синтаксиса [4].

```
CREATE {PROC | PROCEDURE} [schema_name.] procedure_name
    [{@parameter [type_schema_name.] data_type}
        [VARYING] [= default] [[OUT [PUT]]
    [,...n]
[WITH <procedure_option> [,...n]
AS sql_statement [;][ ...n]

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

Следующий пример иллюстрирует создание простой хранимой процедуры, которая возвращает набор строк для всех продуктов, для производства которых требуется больше одного дня.

```
CREATE PROC Production.LongLeadProducts
AS
    SELECT Name, ProductNumber
    FROM Production.Product
    WHERE DaysToManufacture >= 1
GO
```

В этом примере создается процедура с именем **LongLeadProducts** в схеме **Production**. Команда GO включена для того, чтобы подчеркнуть тот факт, что инструкции REATE PROCEDURE должны быть объявлены в одном пакете.

В примере показано, как вызывать хранимую процедуру **LongLeadProducts**.

```
EXEC Production.LongLeadProducts
```

При создании хранимых процедур следует придерживаться рекомендаций.

- Для указания имен объектов, на которые ссылается хранимая процедура, использовать имя соответствующей схемы. Это гарантирует доступность таблиц, представлений или других объектов из различных схем в рамках хранимой процедуры. Если имя объекта ссылки не указано, схема хранимой процедуры ищется по умолчанию.
- Разрабатывать каждую хранимую процедуру для выполнения одной задачи.
- Создать, протестировать и устранить неполадки процедуры на сервере, затем протестировать ее со стороны клиента.
- Избегать использования префикса **sp_** в именах локальных хранимых процедур, чтобы можно было легко отличать системные хранимые процедуры. Другая причина, по которой следует избегать использования префикса **sp_** для хранимых процедур в локальной базе данных — это исключение ненужного поиска в главной базе данных. Когда вызывается хранимая процедура с именем, которое начинается с **sp_**, SQL Server, прежде чем начать поиск в локальной базе данных, выполняет поиск в главной базе данных.
- Использовать одни и те же настройки подключения для всех хранимых процедур. При сохранении или изменении хранимой процедуры SQL Server сохраняет настройки обоих параметров, SET QUOTED_IDENTIFIER и SET ANSI_NULLS. Эти исходные настройки используются при выполнении хранимой процедуры. Поэтому во время выполнения хранимой процедуры никакие настройки клиентского сеанса для этих параметров SET не учитываются.
- Свести к минимуму использование временных хранимых процедур, чтобы избежать конфликтов в системных таблицах в базе данных **tempdb**, так как эта ситуация может оказать негативное влияние на производительность.

Изменение и удаление хранимых процедур

Часто бывает необходимо изменить хранимые процедуры в ответ на запросы пользователей или изменения в определениях основных таблиц. Чтобы изменить существующую хранимую процедуру и сохранить назначение разрешений, используйте инструкцию ALTER PROCEDURE. SQL Server заменяет предыдущее определение хранимой процедуры, когда она изменяется с использованием инструкции ALTER PROCEDURE [5].

При использовании инструкции ALTER PROCEDURE следует учитывать:

- Для изменения хранимой процедуры, созданной с параметром, например WITH ENCRYPTION, необходимо включить этот параметр в инструкцию ALTER PROCEDURE, чтобы сохранить предоставляемые им функциональные возможности.
- Инструкция ALTER PROCEDURE изменяет только одну процедуру. Если процедура вызывает другие хранимые процедуры, вложенные хранимые процедуры остаются неизменными.

В следующем примере хранимая процедура **LongLeadProducts** изменяется для выбора дополнительного столбца и сортировки набора результатов с использованием предложения ORDER BY.

```
ALTER PROC Production.LongLeadProducts
AS
SELECT  Name, ProductNumber, DaysToManufacture
FROMProduction.Product
WHERE  DaysToManufacture >= 1
ORDER BY DaysToManufacture DESC, Name
GO
```

Для удаления пользовательских хранимых процедур из текущей базы данных используется инструкция DROP PROCEDURE.

Прежде чем удалять хранимую процедуру, выполните хранимую процедуру **sp_depends**, чтобы определить, имеются ли объекты, зависимые от этой хранимой процедуры, как показано в следующем примере.

```
EXEC sp_depends @objname = N'Production.LongLeadProducts'
Следующий пример иллюстрирует удаление хранимой процедуры
LongLeadProducts.
DROP PROC Production.LongLeadProducts
```

6.2 Создание параметризованных хранимых процедур

Хранимые процедуры будут более гибкими, если параметры будут включены как часть определения процедуры, поскольку можно создавать более общую логику приложений.

Входные параметры

Хранимая процедура взаимодействует с программой, которая вызывает процедуру с помощью списка, содержащего до 2100 параметров. Входные параметры позволяют передавать информацию в хранимую процедуру; эти значения затем могут быть использованы как локальные переменные в этой процедуре.

Чтобы определить хранимую процедуру, которая принимает входные параметры, объявите одну или несколько переменных в качестве параметров в инструкции CREATE PROCEDURE. При использовании входных параметров следует учитывать.

- Где возможно, используйте стандартные значения параметров. Если стандартное значение определено, пользователь может выполнять хранимую процедуру, не указывая значение для этого параметра.
- Проверьте все значения входящих параметров в начале хранимой процедуры, чтобы быстро выявить отсутствующие и недопустимые значения. При этом можно проверить, не имеет ли параметр значение NULL.

В следующем примере параметр **@MinimumLength** добавляется к хранимой процедуре **LongLeadProducts**. Это делает предложение WHERE более гибким по сравнению с представленным ранее благодаря разрешению вызова приложения, чтобы определить, какое время производства считается приемлемым.

```
ALTER PROC Production.LongLeadProducts
@MinimumLength int = 1      -- default value
AS
IF (@MinimumLength < 0)     -- validate
    BEGIN
    RAISERROR('Invalid lead time.', 14, 1)
    RETURN
    END

SELECT Name, ProductNumber, DaysToManufacture
FROM Production.Product
WHERE DaysToManufacture >= @MinimumLength
ORDER BY DaysToManufacture DESC, Name
```

Хранимая процедура определяет значение стандартного параметра 1, поэтому вызывающие приложения могут выполнять эту процедуру без указания аргумента. Если значение передается в **@MinimumLength**, оно проверяется на соответствие цели инструкции SELECT. Если это значение меньше нуля, возникает ошибка, и хранимая процедура возвращается немедленно без выполнения инструкции SELECT.

Вызов параметризованных хранимых процедур

Значение параметра можно задать, передав хранимой процедуре это значение по имени параметра, либо по расположению. Указывая значения, не следует смешивать разные форматы.

Указание параметра в инструкции EXECUTE в формате **@parameter = value** называется **передачей параметра по имени**. При передаче по имени параметра значения могут указываться в любом порядке; параметры, разрешающие неопределенные или стандартные значения, можно пропускать.

В следующем примере вызывается хранимая процедура **LongLeadProducts** и указывается имя параметра.

```
EXEC Production.LongLeadProducts @MinimumLength=4
```

Передача только значений (без ссылки на имена параметров, которым они передаются) называется передачей значений по расположению. Когда указывается только значение, значения параметров должны перечисляться в том порядке, в котором они определены в инструкции CREATE PROCEDURE. При передаче значений по расположению можно пропускать параметры, для которых имеются стандартные значения, но нельзя нарушать последовательность. Например, если хранимая процедура имеет пять параметров, можно опустить четвертый и пятый параметры, но нельзя пропустить четвертый и указать пятый.

В следующем примере вызывается хранимая процедура LongLeadProducts и указывается параметр только по расположению.

```
EXEC Production.LongLeadProducts 4
```

Стандартное значение параметра, если оно определено для параметра в хранимой процедуре, используется , если при выполнении хранимой процедуры не указано значение параметра или в качестве значения параметра указано ключевое слово DEFAULT.

Выходные параметры и возвращаемые значения

С помощью выходных параметров и возвращаемых значений данные могут возвращаться хранимыми процедурами в вызывающую хранимую процедуру или клиенту.

Выходные параметры позволяют сохранить любые изменения параметра, явившиеся следствием выполнения хранимой процедуры даже после того, как хранимая процедура завершает выполнение.

Чтобы использовать выходной параметр в Transact-SQL, необходимо указать ключевое слово OUTPUT в инструкциях CREATE PROCEDURE и EXECUTE. Если при выполнении хранимой процедуры ключевое слово OUTPUT будет пропущено, хранимая процедура все равно будет выполняться, но не будет возвращать измененное значение. В большинстве клиентских языков программирования, например в Microsoft Visual C#, направление параметра по

умолчанию назначается как входящий, поэтому следует указать направление параметра в клиенте.

В примере создается хранимая процедура, которая добавляет новый отдел в таблицу **HumanResources.Department** базы данных **AdventureWorks**.

```
CREATE PROC HumanResources.AddDepartment
    @Name nvarchar(50), @GroupName nvarchar(50),
    @DeptID smallint OUTPUT
AS
INSERT INTO HumanResources.Department (Name, GroupName)
VALUES (@Name, @GroupName)
SET @DeptID = SCOPE_IDENTITY()
```

Выходной параметр **@DeptID** сохраняет удостоверение новой записи путем вызова функции **SCOPE_IDENTITY**, так что вызывающее приложение может немедленно обратиться к автоматически созданному номеру ID.

В примере показано, как вызывающее приложение может сохранять результаты выполнения хранимой процедуры с использованием локальной переменной **@dept**.

```
DECLARE @dept int
EXEC AddDepartment 'Refunds', '', @dept OUTPUT
SELECT @dept
```

Можно также возвращать информацию из хранимой процедуры с помощью инструкции RETURN. Этот метод более ограниченный, чем использование выходных параметров, поскольку он возвращает только одно целочисленное значение. Инструкция RETURN чаще всего используется для возврата результата состояния или кода ошибки из процедуры.

В примере изменяется хранимая процедура **AddDepartment** для возврата значения успешного выполнения или сбоя.

```
ALTER PROC HumanResources.AddDepartment
    @Name nvarchar(50), @GroupName nvarchar(50),
    @DeptID smallint OUTPUT
AS
IF ((@Name = '') OR (@GroupName = ''))
    RETURN -1
INSERT INTO HumanResources.Department (Name, GroupName)
VALUES (@Name, @GroupName)
SET @DeptID = SCOPE_IDENTITY()
RETURN 0
```

Если для любого из параметров **@Name** и **@GroupName** в процедуру передается пустая строка, возвращается значение **–1**, указывающее на сбой. Если инструкция INSERT выполняется успешно, возвращается значение **0**, указывающее на успех.

В примере показано, как вызывающее приложение может сохранять возвращаемое значение выполнения хранимой процедуры с использованием локальной переменной **@result**.

```
DECLARE @dept int, @result int
EXEC @result = AddDepartment 'Refunds', '', @dept OUTPUT
IF (@result = 0)
SELECT @dept
ELSE
SELECT 'Error during insert'
```

SQL Server автоматически возвращает значение 0 из хранимой процедуры, если не указано значение RETURN.

Компиляция запроса

Когда SQL Server получает запрос для выполнения, он сначала проверяет кэш процедур на наличие кэшированного плана выполнения. Если такой план находится, он используется для выполнения запроса. Если кэшированный плана выполнения не находится, то запрос компилируется, план выполнения кэшируется, а затем запрос выполняется [6].

Процесс компиляции состоит из следующих четырех стадий.

1. **Анализ**. SQL Server проверяет наличие ошибок в синтаксисе запроса и подготавливает его для оптимизации. На этой стадии SQL Server не проверяет объект или имена столбцов.
2. **Нормализация**. SQL Server проверяет правильность и допустимость всех объектов и имен столбцов в запросе. SQL Server проверяет также фактическую осмысленность запроса.
3. **Компиляция**. SQL Server создает план выполнения для запроса, создавая диаграммы запроса для любых инструкций DML. Эти диаграммы запроса используются оптимизатором для создания оптимизированного плана, который затем сохраняется в кэше процедур.
4. **Оптимизация**. Оптимизатор SQL Server использует подход на основе затрат, чтобы определить затраты на различные возможные планы выполнения и определить план с наименьшими затратами. Если имеется только один путь по инструкциям запроса, в оптимизации плана нет необходимости.

Выполнение повторной компиляции хранимой процедуры

Иногда для SQL Server требуется повторная оптимизация планов выполнения хранимой процедуры. Например, при добавлении новых индексов или изменения данных в индексированных столбцах. Для этого можно перекомпилировать хранимую процедуру. При перезапуске SQL Server эта оптимизация происходит автоматически, когда первый раз выполняется запрос. Эта оптимизация также происходит автоматически при изменении

любой из основных таблиц. Однако, если, например, добавляется новый индекс, хранимую процедуру необходимо перекомпилировать вручную.

Также может быть полезно выполнить повторную компиляцию хранимой процедуры для обхода функции «вынюхивания параметра» в компиляции хранимой процедуры. Наряду с оптимизированным планом запроса план выполнения содержит контекст выполнения, который включает значения параметров. Если эти значения не являются типичными для последующих вызовов, может пострадать производительность, и может быть полезно перекомпилировать хранимую процедуру.

Перекомпиляция хранимых процедур

Перекомпиляция хранимой процедуры в SQL Server может быть выполнена тремя способами.

- Системная хранимая процедура sp_recompile выполняет перекомпиляцию указанной хранимой процедуры при следующем ее выполнении. Можно указать хранимую процедуру, триггер, таблицу или представление в текущей базе данных.
- Параметр WITH RECOMPILE инструкции CREATE PROCEDURE указывает SQL Server не сохранять план выполнения для этой хранимой процедуры в кэше процедур. SQL Server перекомпилирует хранимую процедуру при каждом ее выполнении. Этот подход следует использовать, когда значения параметров хранимой процедуры значительно отличаются для разных выполнений.
- Параметр WITH RECOMPILE инструкции EXEC указывает SQL Server перекомпилировать хранимую процедуру первый раз при выполнении хранимой процедуры. Этот подход следует использовать, когда значения параметров данного выполнения значительно отличаются от обычных значений.

В примере показано, как перекомпилировать все хранимые процедуры, которые действуют в таблице **Клиенты**, с помощью системной хранимой процедуры **sp_recompile**.

```
USE AdventureWorks;
GO
EXEC sp_recompile N'Sales.Customer';
GO
```

6.3 Обработка ошибок

Обработка исключений является важным требованием для многих инструкций Transact-SQL, особенно для тех из них, которые связаны с транзакциями. Структурированная обработка исключений уменьшает объем работы, необходимой для обработки ошибок, и позволяет получить более надежный программный код.

Структурированная обработка исключений

Структурированная обработка исключений является обычным способом обработки исключений во многих популярных языках программирования, таких как Microsoft Visual Basic и Visual C#. SQL Server позволяет использовать структурированную обработку исключений в любой транзакционной ситуации, такой как хранимая процедура. Это делает код более надежным и более удобным в обслуживании.

Для структурированной обработки исключений используются блоки TRY…CATCH. Блок TRY содержит транзакционный код, который может быть не выполнен успешно. Блок CATCH содержит код, который выполняется, если в блоке TRY происходит ошибка.

Блоки TRY…CATCH имеют следующий синтаксис [4].

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    { sql_statement | statement_block }
END CATCH
```

Часть **sql_statement** or **statement_block** — это любая инструкция или группа инструкций Transact-SQL.

В примере хранимая процедура **AddData** пытается вставить два значения в таблицу **TestData**. Первый столбец таблицы **TestData** — это целочисленное ключевое поле, а второй столбец — целочисленный тип данных. Блок TRY…CATCH в хранимой процедуре **AddData** защищает инструкцию INSERT таблицы **TestData** и возвращает номер ошибки и сообщение об ошибке как часть логики блока CATCH с помощью функций **ERROR_NUMBER** и **ERROR_MESSAGE**.

```
CREATE TABLE dbo.TableWithKey (ColA int PRIMARY KEY, ColB int)
GO

CREATE PROCEDURE dbo.AddData @a int, @b int
AS
BEGIN TRY
    INSERT INTO TableWithKey VALUES (@a, @b)
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() ErrorNumber, ERROR_MESSAGE() [Message]
END CATCH
GO

EXEC dbo.AddData 1, 1
EXEC dbo.AddData 2, 2
EXEC dbo.AddData 1, 3       --violates the primary key
```

Инструкции по обработке ошибок

Создайте блок CATCH сразу же после инструкции END TRY с помощью инструкций BEGIN CATCH и END CATCH. Между инструкциями END TRY и BEGIN CATCH нельзя вставлять никакие другие инструкции.

Следующий пример не будет скомпилирован.

```
BEGIN TRY
        -- INSERT INTO ...
END TRY
SELECT * FROM TableWithKey -- NOT ALLOWED
BEGIN CATCH
        -- SELECT ERROR_NUMBER()
END CATCH
```

Откат ошибочных транзакций

Использование транзакций позволяет группировать несколько инструкций таким образом, чтобы или они все выполнялись успешно, или ни одна из них не выполнялась успешно. Изучите следующий пример, в котором транзакции не используются.

```
CREATE TABLE dbo.TableNoKey (ColA int, ColB int)
CREATE TABLE dbo.TableWithKey (ColA int PRIMARY KEY, ColB int)
GO

CREATE PROCEDURE dbo.AddData @a int, @b int
AS
BEGIN TRY
    INSERT dbo.TableNoKey VALUES (@a, @b)
    INSERT dbo.TableWithKey VALUES (@a, @b)
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() ErrorNumber, ERROR_MESSAGE() [Message]
END CATCH
GO

EXEC dbo.AddData 1, 1
EXEC dbo.AddData 2, 2
EXEC dbo.AddData 1, 3       --violates the primary key
```

В примере выполняются две последовательные вставки в две различные таблицы. Первая вставка всегда будет успешной, поскольку в таблице нет ограничения для ключевого поля. Вторая вставка будет завершаться ошибкой, когда бы не вставлялся дубликат значения **ColA**. Поскольку в этом примере не используются транзакции, первая вставка заканчивается успешно, даже в случае ошибки второй вставки, что может привести к неожиданным результатам.

В примере используются транзакции, чтобы гарантировать, что ни одна вставка не закончится успешно в случае ошибки одной из транзакций. Для этого используются инструкции BEGIN TRAN и COMMIT TRAN в блоке TRY и инструкция в блоке CATCH.

```
ALTER PROCEDURE dbo.AddData @a int, @b int
AS
BEGIN TRY
    BEGIN TRAN
    INSERT dbo.TableNoKey VALUES (@a, @b)
    INSERT dbo.TableWithKey VALUES (@a, @b)
    COMMIT TRAN
END TRY
BEGIN CATCH
ROLLBACK TRAN
  SELECT ERROR_NUMBER() ErrorNumber, ERROR_MESSAGE() [Message]
END CATCH
GO
```

Использование XACT_ABORT и XACT_STATE

Параметр XACT_ABORT указывает, будет ли SQL Server автоматически выполнять откат текущей транзакции, когда инструкция Transact-SQL устанавливает ошибку времени выполнения. Однако, если ошибка происходит в блоке TRY, не будет выполняться автоматический откат транзакции; вместо этого она становится нефиксируемой.

Код с блоком CATCH должен проверить состояние транзакции, используя функцию XACT_STATE. Функция XACT_STATE возвращает **–1**, если в текущем сеансе имеется нефиксируемая транзакция. Блок CATCH не должен пытаться зафиксировать транзакцию; следует выполнить откат транзакции вручную.

Возвращенное функцией XACT_STATE значение **1** означает, что имеется транзакция, которая может быть безопасно зафиксирована. Возвращенное значение **0** означает, что текущей транзакции нет.

В следующем примере для функции XACT_ABORT устанавливается значение ON и выполняется проверка состояния в блоке CATCH.

```
SET XACT_ABORT ON
BEGIN TRY
    BEGIN TRAN
    ...
    COMMIT TRAN
END TRY
BEGIN CATCH
    IF (XACT_STATE()) = -1        -- uncommittable
        ROLLBACK TRAN
```

```
    ELSE IF (XACT_STATE()) = 1          -- committable
        COMMIT TRAN
    END CATCH
```

Для получения дополнительных сведений об использовании функций XACT_ABORT и XACT_STATE см. раздел «Использование TRY...CATCH в Transact-SQL» в электронной документации по SQL Server.

Сохранение сведений об ошибке, если требуется

SQL Server предоставляет несколько связанных с ошибками функций, которые можно вызывать в блоке CATCH для записи сведений об ошибке. Например, можно вызвать эти методы и сохранить результаты в таблице журнала ошибок. В Таблице 8 перечислены функции ошибок.

Таблица 8. Функции ошибок

Функция	Описание
ERROR_LINE	Возвращает номер строки, в которой произошла ошибка, вызвавшая выполнение кода блока CATCH.
ERROR_MESSAGE	Возвращает диагностические сведения о причине ошибки. Многие сообщения об ошибках имеют переменные замещения, в которых помещаются такие сведения, как имя объекта, создающего ошибку.
ERROR_NUMBER	Возвращает уникальный номер ошибки.
ERROR_PROCEDURE	Возвращает имя хранимой процедуры или триггера, в котором произошла ошибка.
ERROR_SEVERITY	Возвращает значение, указывающее, насколько серьезной является ошибка. Ошибки с низкой серьезностью, например 1 или 2, являются информационными сообщениями или низкоуровневыми предупреждениями. Ошибки с высокой серьезностью определяют проблемы, которые нужно решить возможно более быстро.
ERROR_STATE	Возвращает значение состояния. Некоторые сообщения об ошибках могут быть установлены в нескольких точках кода для ядра базы данных. Каждое конкретное условие, которое устанавливает ошибку, назначает уникальный код состояния. Эти сведения могут быть полезны при работе со статьями базы знаний Майкрософт, чтобы определить, совпадает ли описанная проблема со встретившейся вам ошибкой.

7 РЕАЛИЗАЦИЯ ФУНКЦИЙ

При работе с базами данных требуется разрабатывать и реализовывать логику для установления бизнес-правил или обеспечения согласованности данных с помощью определяемых пользователем функций, изменять и обслуживать существующие хранимые процедуры.

Рассмотрим, как создавать определяемые пользователем функции и работать с контекстом выполнения.

7.1 Создание и использование функций

Функции — это процедуры, которые используются для включения часто выполняемой логики. Любой код, который должен выполнить логику, вместо повторения всей логики функции может вызвать функцию.

Рассмотрим синтаксис, используемый для создания функций, общий обзор функций и их применение.

Типы функций

Функции — это процедуры, состоящие из одной или нескольких инструкций Transact-SQL, которые могут использоваться для сохранения кода для повторного использования.

SQL Server содержит много встроенных функций, например **ABS**, **COS**, **SQUARE** и другие математические функции, а также **AVG**, **COUNT**, **SUM** и другие статистические функции. Можно также создавать собственные определяемые пользователем функции. Функция принимает ноль или большее количество входных параметров и возвращает или скалярное значение, или таблицу. Входными параметрами могут быть данные любого типа за исключением **timestamp**, **cursor** или **table**. Функции не поддерживают выходные параметры.

Скалярные функции

Скалярные функции возвращают одиночное значение данных с типом, определенным в операторе RETURNS. Функции этого типа синтаксически очень похожи на такие встроенные системные функции, как **COUNT** или **MAX**.

Встроенная функция, возвращающая табличное значение

Встроенная функция, возвращающая табличное значение, возвращает таблицу, которая является результатом одной инструкции SELECT. Она сходна с представлением, но обеспечивает большую гибкость, чем приложение, поскольку функции можно передавать параметры.

Многоинструкционные функции, возвращающие табличные значения

Многоинструкционная функция, возвращающая табличное значение, возвращает таблицу, созданную одной или несколькими инструкциями Transact-SQL; она сходна с хранимой процедурой. В отличие от хранимой процедуры, на многоинструкционную функцию, возвращающую табличное значение, можно ссылаться в предложении FROM инструкции SELECT, как если бы это было представление или таблица.

Скалярная функция

Скалярная функция возвращает одиночное значение данных с типом, определенным в операторе RETURNS. Тело функции, определяемое в блоке BEGIN…END, содержит последовательность инструкций Transact-SQL, которые возвращают значение.

В примере создается скалярная функция, которая суммирует все продажи для конкретного продукта в базе данных **AdventureWorks** и возвращает общее значение с типом **int** [6].

```
USE AdventureWorks
GO

CREATE FUNCTION Sales.SumSold(@ProductID int) RETURNS int
AS
BEGIN
DECLARE @ret int
SELECT @ret = SUM(OrderQty)
FROM Sales.SalesOrderDetail WHERE ProductID = @ProductID
IF (@ret IS NULL)
SET @ret =  0
RETURN @ret
END
```

При изменении или удалении функции используется синтаксис, сходный с синтаксисом для изменения или удаления других объектов базы данных. ALTER FUNCTION используется для изменения функций, а DROP FUNCTION — для удаления функций из базы данных.

Скалярная функция может быть вызвана в любом месте, где в инструкции Transact-SQL разрешено скалярное выражение того же типа данных. Для вызова скалярных функций следует использовать имя функции, состоящее по меньшей мере из двух частей (schema_name.object_name).

Таблица 9 содержит примеры того, где могут использоваться скалярные функции.

Таблица 9. Скалярная функция

Область	Пример
Запросы	Как выражение в select_list инструкции **SELECT**. Как выражение или string_expression в предложении WHERE или HAVING. Как group_by_expression в предложении GROUP BY. Как order_by_expression в предложении ORDER BY. Как выражение в предложении **SELECT** инструкции **UPDATE**. Как выражение в предложении **VALUES** инструкции **INSERT**.
Определение таблицы	Ограничения CHECK. Функции могут ссылаться только на столбцы в одной и той же таблице. Определения DEFAULT. Функции могут содержать только константы. **Вычисляемые ст**олбцы. Функции могут ссылаться только на столбцы в одной и той же таблице.
Инструкции Transact-SQL	В операторах назначения. В логических выражениях инструкций потока управления. В выражениях CASE. В инструкциях **PRINT** (только для функций, которые возвращают строку символов).
Функции и хранимые процедуры	Как аргументы функций. Как инструкция **RETURN** хранимой процедуры (только для скалярных функций, которые возвращают целое число). Как инструкция **RETURN** определенной пользователем функции, предоставленное значение, возвращенное вызванной определенной пользователем функции, может быть явно преобразовано в тип возвращаемых данных вызывающей функции.

В следующем примере выполняется инструкция SELECT, которая извлекает **ProductID**, **Name** и результат скалярной функции **SumSold** для каждой записи продукта в базе данных **AdventureWorks**.

```
USE AdventureWorks
GO

SELECT ProductID, Name, Sales.SumSold(ProductID) AS SumSold
FROM   Production.Product
```

Встроенная функция, возвращающая табличные значения

Можно использовать встроенные функции для достижения функциональности параметризированных представлений. Одно из ограничений представлений состоит в том, что не разрешается включать определенный пользователем параметр в представление, когда оно создается. Его можно легко разрешить, включив предложение WHERE при вызове представления. Однако этот подход может потребовать создания строки для динамического выполнения, что может увеличить сложность приложения. Разработчики могут реализовать

возможности параметризованного представления с помощью встроенной функции, возвращающей табличные значения.

Встроенные функции, возвращающие табличное значение, имеют следующие характеристики:

- Инструкция RETURNS указывает требуемый тип данных table.
- Набор результатов инструкции SELECT определяет формат возвращаемой переменной.
- Предложение RETURN содержит одну инструкцию SELECT в скобках. Инструкция SELECT, используемая как встроенная функция, подвергается действию тех же ограничений, что и инструкции SELECT в представлениях.
- Тело функции не должно быть заключено в блок BEGIN…END.

В примере создается встроенная функция, которая возвращает имена сотрудников для конкретного руководителя в базе данных **AdventureWorks**.

```
USE AdventureWorks
GO
CREATE FUNCTION HumanResources.EmployeesForManager
(@ManagerId int)
RETURNS TABLE
AS
RETURN (
SELECTFirstName, LastName
FROM   HumanResources.Employee Employee INNER JOIN
Person.Contact Contact
ON Employee.ContactID = Contact.ContactID
WHERE ManagerID = @ManagerId )
```

Вызов встроенной функции, возвращающей табличное значение

Встроенные функции, возвращающие табличное значение, используются там, где можно обычно использовать представление, например, в предложении FROM инструкции SELECT. В следующем примере извлекаются все имена сотрудников для двух руководителей.

```
USE AdventureWorks
GO
SELECT * FROM HumanResources.EmployeesForManager(3)
-- OR
SELECT * FROM HumanResources.EmployeesForManager(6)
```

Многоинструкционная функция, возвращающая табличные значения

Многоинструкционная функция, возвращающая табличные значения — это комбинация представления и хранимой процедуры. Можно использовать

пользовательские функции, возвращающие таблицы, для замены хранимых процедур или представлений.

Функция, возвращающая табличные значения (как хранимая процедура), может использовать сложную логику и несколько инструкций Transact-SQL для создания таблицы. Таким же образом, как используются представления, можно использовать и функцию, возвращающую табличные значения из предложения FROM инструкции Transact-SQL.

Многоинструкционные функции, возвращающие табличное значение, имеют следующие характеристики:

- Инструкция RETURNS указывает требуемый тип данных **table** и определяет имя для таблицы и формата.
- Блок BEGIN…END ограничивает тело функции.

В примере создается табличная переменная с именем **@tbl_Employees** с двумя столбцами. Второй столбец изменяется в зависимости от запрошенного значения параметра **@format.**

```
CREATE FUNCTION HumanResources.EmployeeNames
(@format nvarchar(9))
RETURNS @tbl_Employees TABLE
(EmployeeID int PRIMARY KEY, [Employee Name] nvarchar(100))
AS
BEGIN
IF (@format = 'SHORTNAME')
INSERT @tbl_Employees
SELECT EmployeeID, LastName
FROM HumanResources.vEmployee
ELSE IF (@format = 'LONGNAME')
INSERT @tbl_Employees
SELECT EmployeeID, (FirstName + ' ' + LastName)
FROM HumanResources.vEmployee
RETURN
END
```

Можно вызвать функцию из предложения FROM вместо того, чтобы использовать таблицу или представление. В следующем примере извлекаются имена сотрудников в длинной и короткой форме.

```
SELECT * FROM HumanResources.EmployeeNames('LONGNAME')
-- OR
SELECT * FROM HumanResources.EmployeeNames('SHORTNAME')
```

7.2 Работа с функциями

Рассмотрим вопросы создания функций, различие между детерминированными и недетерминированными функциями и влияние этого

различия на возможность индексирования результатов функции в SQL Server и переписывание хранимых процедур в виде функций.

Детерминированные и недетерминированные функции

Функции являются детерминированными или недетерминированными. Детерминированные функции каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений и использовать одно и то же состояние базы данных. Недетерминированные функции могут возвращать каждый раз разные результаты, если предоставлять им один и тот же набор входных значений и даже если использовать одно и то же состояние базы данных.

В следующем примере создается детерминированная функция, которая вычисляет площадь круга данного радиуса.

```
CREATE FUNCTION Math.CircleArea(@Radius float) RETURNS float
AS
BEGIN
    RETURN PI() * SQUARE(@Radius)
END
```

В следующем примере создается недетерминированная функция, которая генерирует случайное целое число от 0 до 10.

```
CREATE FUNCTION Math.RandomInt() RETURNS int
AS
BEGIN
    DECLARE @Rand float
    SET @Rand = RAND()
    RETURN @Rand * 10
END
```

Индексация результатов функции определяемой пользователем

У функций, создаваемых пользователем, есть несколько свойств, которые определяют возможность или невозможность индексации результатов в SQL Server. Одним из этих свойств является детерминизм. Например, нельзя создать кластеризованный индекс в представлении, которое ссылается на любые недетерминированные функции.

Детерминизм встроенных функций

Детерминизм встроенных функций нельзя изменить. Все встроенные статистические и строковые функции являются детерминированными. Другие встроенные функции, например ABS, DATEDIFF и ISNULL также являются детерминированными.

Полный список детерминированных встроенных функций см. в разделе «Детерминированные и недетерминированные функции» электронной документации по SQL Server.

В Таблице 10 перечислены функции, которые не всегда являются детерминированными, но могут быть использованы как детерминированные в определенных условиях.

Таблица 10. Функции, которые не всегда являются детерминированными

Функция	**Комментарии**
CAST	Недетерминированная при использовании с типами **datetime**, **smalldatetime** и **sql_variant**.
CONVERT	Недетерминированная в следующих условиях: исходный тип **sql_variant**; целевой тип **sql_variant** и исходный тип недетерминированный; исходный или целевой тип **datetime** или **smalldatetime**.
CHECKSUM	Недетерминированная, если CHECKSUM(*).
ISDATE	Недетерминированная за исключением случаев использования с функцией **CONVERT**.
RAND	Недетерминированная за исключением случаев использования с параметром seed.

Все функции конфигурации, курсора, метаданных, безопасности и системные статистические встроенные функции являются недетерминированными.

К другим недетерминированным функциям относятся **@@CONNECTIONS, @@TIMETICKS** и **GETDATE**.

Полный список недетерминированных встроенных функций см. в разделе «Детерминированные и недетерминированные функции» электронной документации по SQL Server.

Функции, которые вызывают расширенные хранимые процедуры, также являются недетерминированными, поскольку расширенные хранимые процедуры могут оказывать побочное действие на базу данных.

При создании определяемых пользователем функций следует придерживаться следующих рекомендаций.

- Определите и используйте тот тип функции, который в наибольшей степени соответствует вашим требованиям.
- Для указания имен объектов, на которые ссылается функция, используйте имя соответствующей схемы. Это гарантирует доступность (разрешенных на уровне безопасности) таблиц, представлений или других объектов из различных схем в рамках функции. Если имя объекта ссылки не указано, схема функции ищется по умолчанию.
- Создавайте каждую функцию для выполнения одной задачи. Это упрощает обслуживание функции и устранение любых проблем.

- Создайте, протестируйте и устраните неполадки функции на сервере разработки, затем протестируйте ее со стороны клиента.
- Учтите возможность SQL Server индексировать результаты функции. Такие факторы, как детерминизм, точность и доступ к данным влияют на возможность SQL Server индексировать результаты функции. Например, если необходимо включить функцию как индекс в вычисляемый столбец или как часть индексированного представления, необходимо убедиться, что функция является детерминированной.

Преобразование хранимых процедур в виде функций

Определяемую пользователем функцию можно использовать в выражении в предложениях SELECT или WHERE, когда нельзя просто использовать хранимую процедуру. Возможно, будет полезно преобразовать хранимую процедуру или отдельные части хранимой процедуры в виде одной или нескольких пользовательских функций, чтобы их можно было использовать в выражении.

Рефакторинг определенных хранимых процедур или их части, как определенных пользователем функций, также может улучшить повторное использование кода.

В целом, функции, возвращающие табличное значение, следует использовать, когда хранимая процедура возвращает один набор результатов, а скалярные функции – когда хранимая процедура возвращает одно скалярное значение.

Функции, возвращающие табличное значение

Хранимую процедуру имеет смысл переписать как функцию, возвращающую табличное значение, если она удовлетворяет следующим условиям.

- Логика может быть выращена как единая инструкция SELECT, но требует входных параметров.
- Хранимая процедура не выполняет никаких операций обновления.
- Хранимая процедура главным образом используется для получения промежуточных результатов, которые загружаются во временную таблицу для запроса инструкцией SELECT.

7.3 Контроль контекста выполнения

Рассмотрим контекст выполнения и его влияние на выполнение функций, а так же, как изменить контекст выполнения с помощью оператора EXECUTE AS и проблемы олицетворения при наличии нескольких баз данных.

Контекстом выполнения вводится удостоверение, по которому проверяются разрешения. Контекст выполнения обычно определяется пользователем или именем входа, используемым для вызова модуля, например хранимой процедуры или функции.

Контекст выполнения – это удостоверение, используемое кодом, когда он выполняется, и по умолчанию означающее вызывающего пользователя, которое может создавать проблемы, если разорвана цепочка владения, как на рисунке 14. Если пользователь по имени **John** является владельцем таблицы **Sales.Order**, и он предоставляет разрешения SELECT только пользователю **Pat**, то пользователь **Ted** не будет иметь доступа к этой таблице, как видно на рисунке. Если пользователь **Pat** является владельцем определяемой пользователем функции **GetOrders**, которая считывает данные из таблицы **Sales.Order**, **Pat** может выполнить эту функцию без проблем с безопасностью, поскольку **Pat** имеет разрешения SELECT в этой таблице [6].

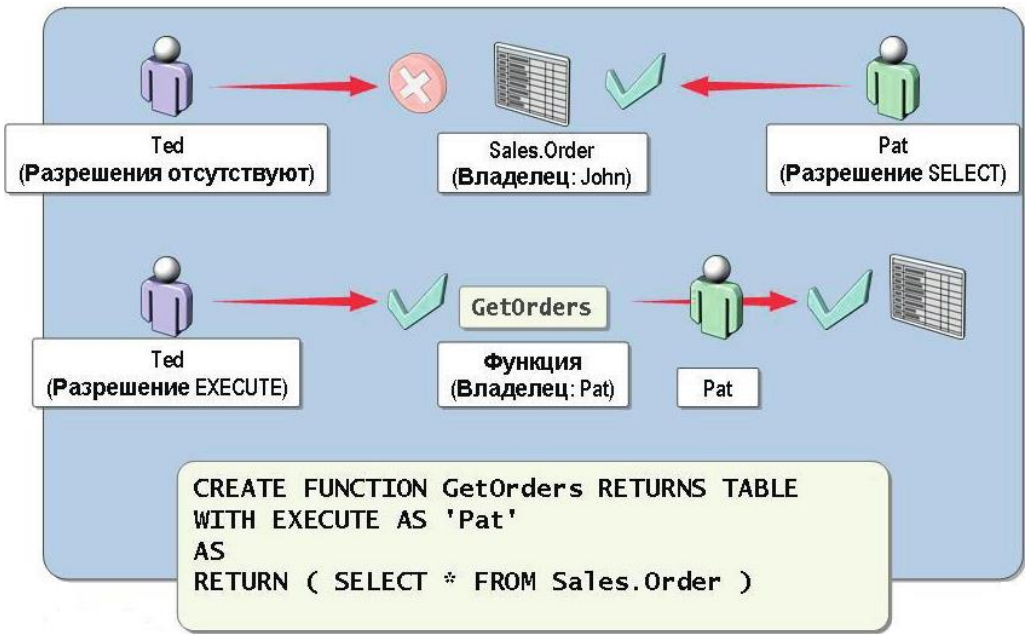

Рисунок 14 Пример контекста выполнения

Если **Pat** предоставляет разрешение EXECUTE на эту функцию пользователю **Ted**, то когда **Ted** попытается выполнить процедуру, это приведет к ошибке. Это объясняется тем, что по умолчанию функция выполняется как **Ted**, и он не имеет разрешения SELECT в таблице **Sales.Order**.

Чтобы разрешить пользователю **Ted** успешно выполнить определяемую пользователем функцию, контекст выполнения должен изменить значение на пользователя, который имеет необходимые разрешения. Можно использовать предложение EXECUTE AS, чтобы переключать контексты выполнения для функции, как описано в следующем разделе.

В этом примере также демонстрируется разорванная цепочка владения, поскольку таблицей Sales.Order и функцией GetOrders владеют разные пользователи.

Предложение EXECUTE AS

Предложение EXECUTE AS можно использовать в функции, чтобы задать удостоверение, используемое в ее контексте выполнения. Понимание того, как использовать предложение EXECUTE AS, может помочь в реализации

безопасности в сценариях, в которых необходим доступ к зависимым объектам, но нежелательно опираться на разорванные цепочки владения.

Параметры EXECUTE AS

Предложение EXECUTE AS можно использовать с любой инструкцией CREATE FUNCTION за исключением объявлений встроенных функций, возвращающих табличные значения. Синтаксис предложения EXECUTE AS приведен ниже.

```
EXECUTE AS { CALLER | SELF | OWNER | user_name }
```

Параметры, входящие в синтаксис предложения EXECUTE AS, перечислены в Таблице 11.

Таблица 11. Параметры синтаксиса предложения EXECUTE AS

Параметр	Описание
CALLER	Выполнение с использованием удостоверения вызывающего пользователя. Этот параметр установлен по умолчанию.
SELF	Выполнение с использованием удостоверения пользователя, который создает иди изменяет функцию or. Это значение остается незатронутым, если другой пользователь принимает владение модулем.
OWNER	Выполнение с использованием удостоверения владельца функции. Это значение изменяется, если другой пользователь принимает владение модулем.
user_name	Выполнение с использованием удостоверения указанного пользователя. Если user_name совпадает с именем пользователя, создающего или изменяющего модуль, это эквивалентно предложению EXECUTE AS SELF.

Чтобы решить проблему, изложенную в предыдущем разделе, указание user_name **Pat** разрешает пользователю **Ted** успешно выполнить определенную пользователем функцию, как показано в следующем примере.

```
CREATE FUNCTION GetOrders RETURNS TABLE
WITH EXECUTE AS 'Pat'
AS
BEGIN
    SELECT * FROM Sales.Order
END
```

Предложение EXECUTE AS можно также выполнить как самостоятельную инструкцию Transact-SQL, чтобы временно изменить текущий контекст выполнения любого соединения, пока не будет выпущена инструкция REVERT. Для получения дополнительных сведений см. статью «REVERT (Transact-SQL)» в электронной документации по SQL Server.

---

8 ОБЕСПЕЧЕНИЕ целостности данных. ТРИГГЕРЫ

Качество данных в базе данных в большой степени определяет эффективность работы приложений (и пользователей), связанных с этой базой данных, и может значительно повлиять на успех деятельности организации или хозяйствующего субъекта. Обеспечение целостности данных является важным направлением поддержания качества данных на высоком уровне.

Целостность данных следует поддерживать на всех уровнях приложения — от ввода или сбора сведений до их хранения. В Microsoft SQL Server доступен ряд возможностей, которые делают обеспечение целостности данных проще.

В разделе приводится описание возможностей, доступных в SQL Server для обеспечения целостности данных, рассматриваются ограничения, как одна из возможностей обеспечения целостности данных.

Целостность данных требует, чтобы каждое значение столбца являлось правильным значением данных, т. е. у него был правильный тип данных, и оно находилось в правильном домене. Ссылочная целостность означает, что между данными правильно установлены и правильно поддерживаются отношения. Одним из способов обеспечения целостности данных и ссылочной целостности является использование триггеров языка обработки данных (DML).

8.1 Обзор целостности данных

При планировании базы данных важно определить оптимальный способ обеспечения целостности данных. Под целостностью данных понимается согласованность и точность данных, хранящихся в базе данных. В этом разделе слушатели изучат различные типы целостности данных в реляционной базе данных, а также возможности, доступные в SQL Server для обеспечения целостности данных.

Типы целостности данных

Обеспечение целостности данных позволяет гарантировать высокое качество данных в базе данных.

Существует три следующих типа целостности данных (см. рисунок 15), которые необходимо учесть при планировании [6]:

- Доменная целостность (столбец).
- Объектная целостность (таблица).
- Ссылочная целостность.

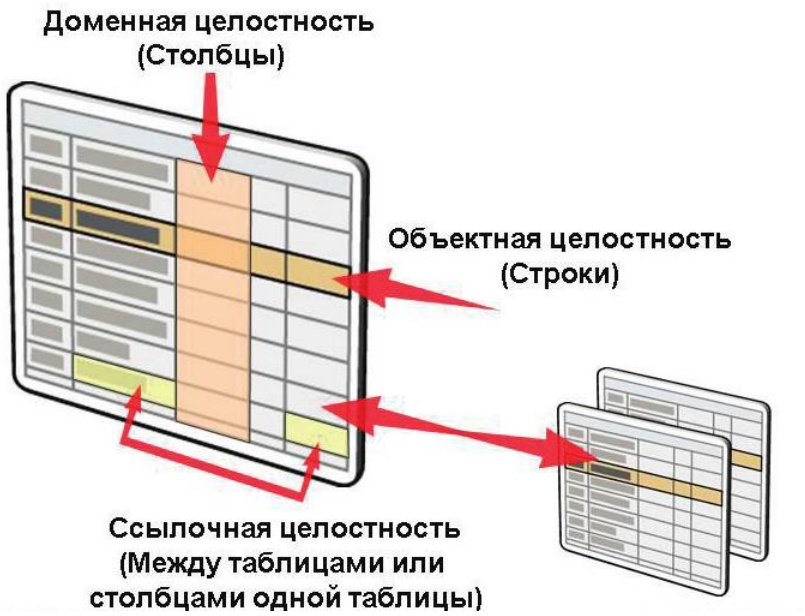

Рисунок 15 Типы целостности данных

Доменная целостность

Доменная целостность (для столбца) предусматривает определение набора значений данных, который является допустимым для этого столбца, а также возможности использовать пустые значения. Доменная целостность зачастую поддерживается с использованием проверки допустимости значений и может быть реализована путем ограничения допустимых для столбца значений по типу данных, формату или диапазону значений.

Объектная целостность

Для обеспечения объектной целостности (для таблицы) требуется, чтобы все строки таблицы имели уникальный код, называемый значением первичного ключа. Возможность изменения ключевого поля и удаления целой строки зависит от уровня целостности, который требуется обеспечить между ключевыми полями и другими таблицами.

Ссылочная целостность

Ссылочная целостность гарантирует сохранение связей между ключевыми полями (таблица, на которую указывают ссылки) и внешними ключами (в таблицах, которые содержат ссылки). Строку в таблице, на которую указывают ссылки, нельзя удалить. Кроме того, нельзя изменить ключевое поле строки, если на нее указывает внешний ключ и не разрешается выполнять каскадные действия. Отношения ссылочной целостности можно определить в рамках одной таблицы или между разными таблицами.

Способы обеспечения целостности данных

В Таблице 12 перечислены механизмы, доступные в SQL Server для обеспечения целостности данных.

Таблица 12. Механизмы обеспечения целостности данных

Механизм	Типы целостности	Описание
Типы данных	Доменная	Этот механизм предполагает введение фундаментальных ограничений по типам данных, хранящихся в каждом из столбцов. Назначение типов данных столбцам таблицы — первый шаг по обеспечению целостности данных. Однако одних ограничений по типу данных, без других механизмов, рассмотренных в данной таблице, недостаточно для обеспечения целостности данных в решениях для деловой деятельности. Например, если для столбца установлен тип данных int, это гарантирует, что в него могут быть введены только числовые целые значения, однако с помощью ограничений по типу данных нельзя указать, что допустимые значения находятся в диапазоне 0 — 1000.
Правила и значения по умолчанию	Доменная, объектная	С помощью правил определяются допустимые значения, которые может содержать столбец, а значения по умолчанию для столбца используются в том случае, если соответствующие значения не указаны. Важным является тот факт, что правила и значения по умолчанию являются независимыми объектами, которые могут быть привязаны к одному или нескольким столбцам или пользовательским типам данных, что позволяет определить их один раз и затем использовать на постоянной основе. Недостаток правил и значений по умолчанию состоит в том, что они несовместимы со стандартами ANSI. Вместо правил и значений по умолчанию следует использовать ограничения.
Ограничения	Доменная, объектная, ссылочная	С помощью ограничений можно определить порядок автоматического применения ядром базы данных SQL Server положений обеспечения целостности базы данных. Ограничения позволяют указать допустимые значения столбцов, а также определить стандартный механизм обеспечения целостности. Рекомендуется использовать ограничения вместо триггеров, правил и значений по умолчанию. В оптимизаторе запросов определения ограничений применяются для построения эффективных планов выполнения запросов.
Триггеры	Доменная, объектная, ссылочная	Триггеры являются особым видом хранимых процедур, которые выполняются при возникновении определенных DML-событий на сервере базы данных. К DML-событиям относятся инструкции UPDATE, INSERT или DELETE, выполняемые для таблицы или представления. Триггеры используются для применения бизнес-правил при изменении данных.
XML-схемы	Доменная (XML)	XML-схемы определяют пространство имен, структуру и допустимое содержимое документов XML. Применение XML-схемы для столбца таблицы, имеющего тип данных xml, позволяет ограничить структуру и содержимое XML-данных, хранимых в этом столбце.

8.2 Внедрение ограничений

Использование ограничений является рекомендуемым способом обеспечения целостности данных. Рассмотрим доступные типы ограничений, выбор типа ограничений в зависимости от потребностей, типы целостности данных, обеспечиваемые тем или иным ограничением, а также порядок определения ограничений. Если требуется дополнительная гибкость, доступны другие способы обеспечения целостности данных.

Что представляют собой ограничения

Ограничения являются соответствующим стандарту ANSI способом обеспечения целостности данных. Каждый тип целостности данных − доменная, объектная и ссылочная − предполагает использование соответствующих типов ограничений.

Ограничения обеспечивают ввод в столбцы допустимых значений и сохранение связей между таблицами. В Таблице 13 перечислены типы ограничений, доступные в SQL Server, а также указаны объекты, к которым они применяются — таблица или столбец.

Таблица 13. Типы ограничений

Тип целостности	Тип ограничения	Цель	Описание
Доменная	DEFAULT	Столбец	Указывает значение по умолчанию для столбца, если соответствующее значение не определено инструкцией INSERT. Ограничение DEFAULT рекомендуется использовать вместо объекта значения по умолчанию.
	CHECK	Столбец	Указывает значения данных, которые являются допустимыми для столбца. Ограничение CHECK рекомендуется использовать вместо объекта правила.
	FOREIGN KEY	Таблица	Указывает значения данных, доступные для обновления, на основе значений столбца в другой таблице.
	NULL	Столбец	Указывает, может ли столбец содержать значение NULL.
Объектная	PRIMARY KEY	Таблица	Присваивает каждой строке уникальный идентификатор, который гарантирует, что пользователи не смогут ввести дублирующие значения, создается индекс с целью повышения производительности. Пустые значения недопустимы.
	UNIQUE	Столбец	Предотвращает дублирование дополнительных (не основных) ключей, создается индекс с целью повышения производительности. Пустые значения допустимы.
Ссылочная	FOREIGN KEY	Таблица	Определяет столбец или сочетание столбцов со значениями, которые соответствуют ключевому полю этой же или другой таблицы.
	CHECK	Столбец	Определяет допустимые для столбца значения данных на основе значений в других столбцах той же таблицы.

Создание ограничений

Ограничения создаются при создании таблицы с использованием инструкции CREATE TABLE. Ограничения на уровне столбца применяются к отдельному столбцу и являются частью определения столбца. Ограничения на уровне таблицы применяются к одному или нескольким столбцам таблицы. Ограничения на уровне таблицы указываются в отдельном разделе после определения всех столбцов. Ограничения для существующей таблицы можно изменить с использованием инструкции ALTER TABLE. Можно добавить ограничения в таблицу с существующими данными либо установить ограничения для одного или нескольких столбцов.

Ниже следует упрощенное представление синтаксиса инструкции CREATE TABLE, демонстрирующее, где следует создавать различные ограничения на уровне столбца и на уровне таблицы.

```
CREATE TABLE table_name
    ({<column_definition> | <table_constraint>} [,...n])

<column_definition> ::=
    {column_name data_type}
    [{DEFAULT constant_expression | [IDENTITY [(seed,
          increment)]]}]
    [<column_constraint> [...n]]

<column_constraint > ::=
    [CONSTRAINT constraint_name]
    {[NULL | NOT NULL]
        | [PRIMARY KEY | UNIQUE]
        | REFERENCES ref_table [(ref_column)]
        [ON DELETE {CASCADE | NO ACTION}]
        [ON UPDATE {CASCADE | NO ACTION}]
    }

<table_constraint> ::=
    [CONSTRAINT constraint_name]
    {[{PRIMARY KEY | UNIQUE} {(column [,...n])}]
    | FOREIGN KEY (column [,...n])
        REFERENCES ref_table [(ref_column [,...n])]
        [ON DELETE {CASCADE | NO ACTION}]
        [ON UPDATE {CASCADE | NO ACTION}]
    }
```

Ограничения PRIMARY KEY

Ограничение PRIMARY KEY определяет один или несколько столбцов таблицы, которые образуют ключевое поле. Ключевое поле уникальным образом определяет строку таблицы и обеспечивает объектную целостность таблицы.

При внедрении ограничения PRIMARY KEY необходимо учитывать следующие условия:

- Для таблицы может быть установлено только одно ограничение PRIMARY KEY.
- Столбцы, включенные в ограничение PRIMARY KEY, не могут содержать пустые значения.
- Каждое из значений в столбцах, выбранных для ограничения PRIMARY KEY, должно быть уникальным. Если ограничение PRIMARY KEY включает несколько столбцов, дублирующие значения могут появляться в каждом из таких столбцов по отдельности, однако сочетание значений всех столбцов в определении ограничения PRIMARY KEY должно быть уникальным.
- Ограничение PRIMARY KEY создает уникальный индекс с указанными столбцами в качестве ключевого поля. Таким образом, в отношении столбцов, выбранных для ключевого поля, должны применяться правила создания уникальных индексов. Можно указать кластеризованный или некластеризованный индекс. (По умолчанию для новых индексов создается кластеризация). Не допускается удаление индекса, который поддерживает ограничение PRIMARY KEY. Индекс можно удалить только в том случае, если удалено ограничение PRIMARY KEY. Данный индекс также обеспечивает быстрый доступ к данным, если в запросах указано ключевое поле.

Ограничение PRIMARY KEY можно использовать в следующих случаях:

- Один или несколько столбцов таблицы должны уникальным образом идентифицировать каждую строку (сущность) в таблице.
- Один из столбцов в таблице является столбцом идентификаторов.

Создание ограничений PRIMARY KEY

Ограничения PRIMARY KEY создаются с использованием предложения уровня таблицы CONSTRAINT в инструкциях CREATE TABLE и ALTER TABLE. В следующем ниже примере демонстрируется инструкция Transact-SQL, используемая для создания таблицы HumanResources.Department базы данных AdventureWorks. Предложение CONSTRAINT определяет кластеризованное ограничение PRIMARY KEY с именем PK_Department_DepartmentID для столбца DepartmentID.

```
CREATE TABLE [HumanResources].[Department](
[DepartmentID] [smallint] IDENTITY(1,1) NOT NULL,
[Name] [dbo].[Name] NOT NULL,
[GroupName] [dbo].[Name] NOT NULL,
[ModifiedDate] [datetime] NOT NULL,
CONSTRAINT [PK_Department_DepartmentID] PRIMARY KEY CLUSTERED
    ([DepartmentID] ASC )WITH (IGNORE_DUP_KEY = OFF)
ON [PRIMARY] )
```

Ограничения DEFAULT

Ограничение DEFAULT вводит значение в столбец, если оно не указано в инструкции INSERT. Ограничения DEFAULT обеспечивают доменную целостность данных.

При внедрении ограничения DEFAULT необходимо учитывать следующие условия:

- Ограничение DEFAULT применяется только в отношении инструкций INSERT.
- Столбец может иметь только одно ограничение DEFAULT.
- Для столбцов, имеющих свойство IDENTITY или тип данных rowversion, нельзя устанавливать ограничение DEFAULT.
- Ограничение DEFAULT позволяет указать вместо пользовательских значений некоторые системные значения — USER, CURRENT_USER, SESSION_USER, SYSTEM_USER или CURRENT_TIMESTAMP.

Ограничение DEFAULT можно использовать в следующих случаях:

- Данные, хранящиеся в столбце, имеют явное значение по умолчанию.
- Столбец не может содержать пустые значения.
- В столбце не обеспечивается уникальность значений.

Создание ограничений DEFAULT

Ограничения DEFAULT создаются с использованием предложения уровня столбца CONSTRAINT в инструкциях CREATE TABLE и ALTER TABLE. В следующем ниже примере демонстрируется инструкция Transact-SQL, используемая для создания таблицы **Production.Location** базы данных **AdventureWorks**. В этом примере создаются ограничения DEFAULT для столбцов **CostRate**, **Availability** и **ModifiedDate**.

```
CREATE TABLE [Production].[Location](
[LocationID] [smallint] IDENTITY(1,1) NOT NULL,
[Name] [dbo].[Name] NOT NULL,
[CostRate] [smallmoney] NOT NULL CONSTRAINT
[DF_Location_CostRate]  DEFAULT ((0.00)),
[Availability] [decimal](8, 2) NOT NULL CONSTRAINT
[DF_Location_Availability]  DEFAULT ((0.00)),
[ModifiedDate] [datetime] NOT NULL CONSTRAINT
     [DF_Location_ModifiedDate]
DEFAULT (getdate())
)
```

Ограничения CHECK

Ограничение CHECK определяет диапазон значений, который пользователь может ввести в отдельном столбце с помощью инструкций INSERT и UPDATE. Проверочные ограничения устанавливаются на уровне столбца или

таблицы. Ограничения CHECK на уровне столбца определяют диапазон значений, которые могут храниться в этом столбце. Ограничения CHECK уровня таблицы могут действовать в отношении нескольких столбцов одной таблицы, что позволяет создавать перекрестные ссылки и сравнивать значения столбцов.

При внедрении ограничения CHECK необходимо учитывать следующие условия:

- Ограничение CHECK проверяет данные каждый раз при выполнении инструкции INSERT или UPDATE.
- Ограничение CHECK может представлять собой любое логическое выражение, которое возвращает значения true или false.
- Ограничение CHECK не может содержать вложенные запросы.
- В отношении одного столбца может действовать несколько ограничений CHECK.
- Ограничение CHECK не может быть установлено для столбцов с типом данных rowversion.
- Инструкция CHECKCONSTRAINTS, предназначенная для проверки согласованности базы данных (DBCC), возвращает все строки, данные в которых противоречат ограничению CHECK.

Ограничение CHECK можно использовать в следующих случаях:

- В соответствии с бизнес-логикой данные, хранящиеся в столбце, должны быть включены в конкретный набор или диапазон значений.
- Данные, хранящиеся в столбце, имеют предельные значения.
- Между столбцами таблицы существуют связи, которые ограничивают набор допустимых значений для столбца.

Создание ограничений CHECK

Ограничения CHECK создаются с использованием предложения уровня столбца или таблицы CONSTRAINT в инструкциях CREATE TABLE и ALTER TABLE.

В следующем ниже примере демонстрируется инструкция Transact-SQL, используемая для добавления инструкции CHECK в таблицу HumanResources.EmployeeDepartmentHistory базы данных AdventureWorks. В этом примере значение столбца EndDate обязательно должно быть не меньше значения столбца StartDate, если значение столбца EndDate не равняется NULL.

```
ALTER TABLE [HumanResources].[EmployeeDepartmentHistory] WITH
     CHECK
    ADD CONSTRAINT [CK_EmployeeDepartmentHistory_EndDate]
    CHECK (([EndDate]>=[StartDate] OR [EndDate] IS NULL))
```

Ограничения UNIQUE

Ограничение UNIQUE указывает, что в двух строках одного столбца не могут находиться одинаковые значения. Ограничение UNIQUE полезно, если при наличии ключевого поля, такого как номер сотрудника, нужно гарантировать уникальность других идентификаторов, например кода налогоплательщика для этого сотрудника.

При внедрении ограничения UNIQUE необходимо учитывать следующие условия:

- В столбце, для которого установлено ограничение UNIQUE, может находиться только одно пустое значение.
- Таблица может иметь несколько ограничений UNIQUE, в то время как ключевое поле только одно.
- Ограничение UNIQUE применяется путем создания уникального индекса для указанного столбца или столбцов. Из-за наличия этого индекса таблица не может превышать предел по некластеризованным индексам, который равен 249.
- При создании ограничения UNIQUE для столбца, содержащего данные с дублированными значениями, ядро базы данных возвратит ошибку.

Ограничение UNIQUE можно использовать в следующих случаях:

- Таблица содержит столбцы, которые не являются частью ключевого поля, однако должны содержать только значения, которые уникальны в рамках одного столбца или всей совокупности таких столбцов.
- В соответствии с бизнес-логикой данные, хранящиеся в столбце, должны быть уникальны.
- Данные, хранящиеся в столбце, по своей природе должны быть уникальны (например, номер налогоплательщика или номер паспорта).

Создание ограничения UNIQUE

Ограничения UNIQUE создаются с использованием предложения уровня столбца UNIQUE в инструкциях CREATE TABLE и ALTER TABLE. В следующем примере демонстрируется инструкция Transact-SQL, используемая для создания таблицы **HumanResources. Employee** базы данных **AdventureWorks**. В этом примере определяется ограничение UNIQUE для столбца **NationalIDNumber**, а также обеспечивается отсутствие пустых значений в этом столбце и использование некластеризованного индекса, созданного для поддержки ограничения UNIQUE.

```
CREATE TABLE [HumanResources].[Employee](
[EmployeeID] [int] IDENTITY(1,1) NOT NULL,
[NationalIDNumber] [nvarchar](15) NOT NULL UNIQUE NONCLUSTERED,
[ContactID] [int] NOT NULL,
    …
)
```

Ограничения FOREIGN KEY

Внешний ключ представляет собой столбец или сочетание столбцов, которое используется для обеспечения связи между данными двух таблиц. Ограничение FOREIGN KEY обеспечивает ссылочную целостность. Ограничение FOREIGN KEY определяет ссылку на столбец с ограничением PRIMARY KEY или UNIQUE в той же или в другой таблице. Значения в столбце внешнего ключа должны отображаться в столбце ключевого поля. При наличии ссылок на значения ключевого поля эти значения не могут быть изменены или удалены.

При внедрении ограничения FOREIGN KEY необходимо учитывать следующие условия:

- Ограничение FOREIGN KEY обеспечивает ссылочную целостность для одного или нескольких столбцов. Число столбцов и типы данных, указанные в инструкции FOREIGN KEY, должны соответствовать числу столбцов и типам данных в предложении REFERENCES.
- В отличие от ограничений PRIMARY KEY или UNIQUE ограничения FOREIGN KEY не создают индексы автоматически.
- Чтобы другой пользователь мог создать ограничение FOREIGN KEY для таблицы текущего пользователя, ему необходимо предоставить разрешение REFERENCES на доступ к этой таблице. Это гарантирует, что внешний пользователь не сможет ограничить действия с таблицей, создав ограничение FOREIGN KEY, которое ссылается на эту таблицу.
- Ограничение FOREIGN KEY, в котором используется только предложение REFERENCES без предложения FOREIGN KEY, ссылается на столбец в той же таблице.

Ограничение FOREIGN KEY можно использовать в следующих случаях:

- Данные в одном или нескольких столбцах могут содержать только значения, хранящиеся в определенных столбцах той же или другой таблицы.
- Строки таблицы не должны удаляться, если у них есть зависимые строки в другой таблице.

Создание ограничений FOREIGN KEY

Ограничения FOREIGN KEY создаются с использованием предложения уровня столбца или таблицы CONSTRAINT в инструкциях CREATE TABLE и ALTER TABLE. В следующем ниже примере демонстрируется инструкция Transact-SQL, используемая для добавления ограничения FOREIGN KEY в таблицу **Sales.SalesOrderHeader** базы данных **AdventureWorks**. В данном примере создается ограничение FOREIGN KEY с именем **FK_SalesOrderHeader_Customer_CustomerID**, которая устанавливает связь между столбцом **CustomerID** таблицы **Sales.SalesOrderHeader** и столбцом **CustomerID** таблицы **Sales.Customer**.

```
ALTER TABLE [Sales].[SalesOrderHeader] WITH CHECK ADD CONSTRAINT
[FK_SalesOrderHeader_Customer_CustomerID] FOREIGN
     KEY([CustomerID])
REFERENCES [Sales].[Customer] ([CustomerID])
```

Каскадная ссылочная целостность

Ограничение FOREIGN KEY имеет параметр CASCADE, с помощью которого можно распространять любое изменение значения столбца, определяющего ограничение UNIQUE или PRIMARY KEY, в любые значения внешнего ключа, ссылающиеся на это значение. Такое действие называется каскадной ссылочной целостностью.

Предложение REFERENCES инструкций CREATE TABLE и ALTER TABLE поддерживает предложения ON DELETE и ON UPDATE. Эти предложения позволяют указать поведение каскадной ссылочной целостности. Доступны значения NO ACTION, CASCADE, SET NULL и SET DEFAULT. По умолчанию используется значение NO ACTION.

Каскадные обновления

Предложение ON UPDATE ограничения FOREIGN KEY определяет поведение в том случае, если инструкция UPDATE предпринимает попытку изменить значение ключевого поля, на которое ссылаются значения внешнего ключа в других таблицах. В Таблице 14 приводится краткое описание возможных значений предложения ON UPDATE и последствий их использования.

Таблица 14. Возможные значения предложения ON UPDATE

Параметр	Поведение
NO ACTION	Указывает, что при попытке обновить ключевое поле в строке, на ключ которой ссылаются внешние ключи существующих строк других таблиц, произойдет ошибка и будет выполнен откат для инструкции UPDATE.
CASCADE	Указывает, что при попытке обновить значение ключевого поля в строке с ключом, на который ссылаются внешние ключи существующих строк других таблиц, все значения, образующие внешний ключ, также будут обновлены с использованием нового значения ключевого поля.
SET NULL	Указывает, что при попытке обновить строку с ключом, на который ссылаются внешние ключи существующих строк других таблиц, для всех значений, образующих внешний ключ, будет установлено значение NULL. Для выполнения данного ограничения необходимо, чтобы для всех столбцов внешнего ключа целевой таблицы разрешались пустые значения.
SET DEFAULT	Указывает, что при попытке обновить строку с ключом, на который ссылаются внешние ключи существующих строк других таблиц, для всех значений, образующих внешний ключ, будет установлено соответствующее значение по умолчанию. Для выполнения данного ограничения необходимо, чтобы все столбцы внешнего ключа целевой таблицы имели определения значений по умолчанию. Если для столбца допускаются пустые значения и не указан набор значений по умолчанию, значением по умолчанию для такого столбца неявно становится значение NULL. Чтобы обеспечить действие ограничения FOREIGN KEY, все непустые значения, заданные с параметром ON UPDATE SET DEFAULT, должны иметь соответствующие значения в основной таблице.

Каскадное удаление

Предложение ON DELETE ограничения FOREIGN KEY определяет поведение в том случае, если инструкция DELETE предпринимает попытку удалить значение ключевого поля, на которое ссылаются значения внешнего ключа в других таблицах. В Таблице 15 приводится краткое описание возможных значений предложения ON DELETE и последствий их использования.

Таблица 15. Возможных значений предложения ON DELETE

Параметр	Поведение
NO ACTION	Указывает, что при попытке удалить строку, на ключ которой ссылаются внешние ключи существующих строк других таблиц, произойдет ошибка и будет выполнен откат для инструкции **DELETE.**
CASCADE	Указывает, что при попытке удалить строку, на ключ которой ссылаются внешние ключи существующих строк других таблиц, будут также удалены все строки, содержащие такие внешние ключи.
SET NULL	Указывает, что при попытке удалить строку с ключом, на который ссылаются внешние ключи существующих строк других таблиц, для всех значений, образующих внешний ключ, будет установлено значение NULL. Для выполнения данного ограничения необходимо, чтобы для всех столбцов внешнего ключа целевой таблицы разрешались пустые значения.
SET DEFAULT	Указывает, что при попытке удалить строку с ключом, на который ссылаются внешние ключи существующих строк других таблиц, для всех значений, образующих внешний ключ, будет установлено соответствующее значение по умолчанию. Для выполнения данного ограничения необходимо, чтобы все столбцы внешнего ключа целевой таблицы имели определения значений по умолчанию. Если для столбца допускаются пустые значения и не указан набор значений по умолчанию, значением по умолчанию для такого столбца неявно становится значение NULL. Чтобы обеспечить действие ограничения FOREIGN KEY, все непустые значения, заданные с параметром ON DELETE SET DEFAULT, должны иметь соответствующие значения в основной таблице.

Проверка с использованием ограничений

При создании ограничений им следует присваивать понятные имена. Если не указано имя для ограничений, SQL Server подставляет сложное имя, создаваемое автоматически. Имена должны быть уникальными для владельца объекта базы данных и следовать правилам, установленным для идентификаторов SQL Server.

При внедрении и изменении ограничений необходимо учитывать следующие условия:

- Ограничения можно создавать, изменять и удалять, не прибегая к удалению и воссозданию таблиц.
- Необходимо встроить в приложения и транзакции логику проверки на наличие ошибок, чтобы проверять нарушение ограничений.

- При добавлении новых ограничений для существующих таблиц необходимо указать, должны ли применяться ограничения FOREIGN KEY и CHECK для существующих данных.

При работе с ограничениями, следует использовать системную хранимую процедуру **sp_helpconstraint** или **sp_help** либо воспользоваться представлениями схемы сведений о запросе, такими как **check_constraints**, **referential_constraints** и **table_constraints**. Определения ограничений хранятся в следующих системных таблицах: **syscomments**, **sysreferences** и **sysconstraints**.

Отключение ограничений

Можно отключать только ограничения CHECK и FOREIGN KEY. Другие ограничения необходимо удалить, а затем добавить вновь. Ограничения имеет смысл отключать в следующих случаях:

- Требуется выполнить объемное пакетное задание или импорт данных и нужно повысить производительность. Перед повторным включением ограничений необходимо иметь уверенность, что данные удовлетворяют условиям соответствующих ограничений, либо выполнить запросы в пакетном задании, чтобы убедиться, что данные верны.
- Ограничение определяется для таблицы, которая уже содержит данные. В результате каждая строка данных будет проверена ограничением при ее следующем изменении.

Отключение ограничений для таблицы не влияет на ограничения других таблиц, которые ссылаются на исходную таблицу. При обновлении таблицы могут возникать ошибки нарушения ограничения.

Порядок отключения ограничений

Чтобы отключить проверку с использованием ограничений при добавлении ограничения CHECK или FOREIGN KEY в таблицу, в которой содержатся данные, включите параметр WITH NOCHECK в инструкцию ALTER TABLE.

Существующие данные будут проверены только при последующем обновлении столбца с ограничениями.

В примере создается ограничение FOREIGN KEY для таблицы **Sales.SalesOrderHeader** с использованием параметра WITH NOCHECK для отключения проверки существующих данных.

```
ALTER TABLE [Sales].[SalesOrderHeader] WITH NOCHECK ADD
     CONSTRAINT
[FK_SalesOrderHeader_Customer_CustomerID] FOREIGN
     KEY([CustomerID])
REFERENCES [Sales].[Customer] ([CustomerID])
```

Можно отключить проверку с использованием ограничения для существующих ограничений CHECK и FOREIGN KEY, чтобы при добавлении данных в таблицу или их изменении эти данные не проверялись. Чтобы

отключить ограничение CHECK или FOREIGN KEY, воспользуйтесь инструкцией ALTER TABLE и укажите параметр NOCHECK. Чтобы включить отключенное ограничение, выполните другую инструкцию ALTER TABLE, на этот раз с параметром CHECK.

Пример отключения ограничения **FK_SalesOrderHeader_Customer_CustomerID**:

```
ALTER TABLE [Sales].[SalesOrderHeader]
NOCHECK
CONSTRAINT [FK_SalesOrderHeader_Customer_CustomerID]
```

Пример включения ограничения **FK_SalesOrderHeader_Customer_CustomerID**:

```
ALTER TABLE [Sales].[SalesOrderHeader]
CHECK
CONSTRAINT [FK_SalesOrderHeader_Customer_CustomerID]
```

Чтобы определить, включено или отключено ограничение для таблицы, следует выполнить системную хранимую процедуру **sp_help** или воспользоваться свойством **CnstIsDisabled** функции **OBJECTPROPERTY**.

8.3 Внедрение триггеров

Триггеры DML — мощное средство контроля целостности данных доменов, объектов и ссылочной целостности. Рассмотрим, что представляют собой триггеры DML, как с их помощью обеспечивается целостность данных, какие существуют типы триггеров и как определять триггеры в базе данных.

Представление триггеров

Триггер — это хранимая процедура особого вида, которая выполняется, когда инструкция INSERT, UPDATE или DELETE изменяет данные в указанной таблице.

Триггер может опрашивать другие таблицы и включать составные инструкции Transact-SQL. Триггеры часто создаются для обеспечения ссылочной целостности или согласованности логически связанных данных, хранящихся в разных таблицах. Пользователи не могут обходить триггеры, поэтому разработчик имеет возможность, используя средства Transact-SQL, реализовать с помощью триггеров сложную бизнес-логику, которую было бы трудно или вообще невозможно реализовать с помощью других механизмов контроля целостности данных.

Необходимо знать о триггерах следующее.

- Триггер и инициирующая его инструкция вместе рассматриваются как одна транзакция, допускающая откат из процедуры триггера. При

возникновении серьезной ошибки (например, при нехватке места на диске) производится автоматический откат всей транзакции.

- Триггеры могут инициировать каскадное распространение изменений по связанным таблицам базы данных, однако эти изменения можно выполнить более эффективно, воспользовавшись каскадными ограничениями ссылочной целостности.
- Триггеры могут служить защитой от неправильных или злонамеренных операций вставки, обновления и удаления, а также контролировать соблюдение других ограничений, более сложных, чем те, которые определяются на основе ограничений CHECK.
- Триггеры могут содержать ссылки на столбцы других таблиц, в отличие от ограничений CHECK. Например, в триггере можно использовать инструкцию SELECT для выбора из другой таблицы, чтобы выполнить сравнение с вставленными или обновленными данными или другие действия, например изменить данные или выдать сообщение об ошибке, определенное пользователем.
- Триггеры позволяют оценивать состояние таблицы до и после изменения данных и предпринимать те или иные действия в зависимости от обнаруженных различий.
- Применение к таблице нескольких триггеров одного типа (INSERT, UPDATE или DELETE) позволяет выполнить несколько различных действий в ответ на одну и ту же инструкцию изменения.

В SQL Server поддерживаются триггеры и других типов. Это триггеры языка определения данных (DDL) и управляемые триггеры.

Создание триггеров

Триггеры создаются с помощью инструкции Transact-SQL CREATE TRIGGER. Инструкция CREATE TRIGGER использует следующий синтаксис.

```
CREATE TRIGGER [schema_name . ] trigger_name
ON {table | view}
[WITH <dml_trigger_option> [,...n]]
{FOR | AFTER | INSTEAD OF}
{[INSERT] [,] [UPDATE] [,] [DELETE]}
[WITH APPEND]
[NOT FOR REPLICATION]
AS {sql_statement [;] [...n] | EXTERNAL NAME <method specifier
     [;]>}
```

Типы триггеров

Триггеры DML делятся на две категории [6]:

- **Триггеры AFTER**. Триггер AFTER выполняется после действия, заданного инструкцией INSERT, UPDATE или DELETE. Использование триггера AFTER эквивалентно заданию ключевого слова FOR; это единственная возможность, доступная в предыдущих версиях SQL Server. Триггеры AFTER можно определять только для таблиц.
- **Триггеры INSTEAD OF**. Триггер INSTEAD OF выполняется вместо обычного действия, для которого он задан. Триггеры INSTEAD OF можно также определять для представлений с одной или несколькими базовыми таблицами, расширяя типы обновлений, поддерживаемых в представлении.

Триггеры и ограничения

Основное преимущество триггеров состоит в том, что они могут включать сложную логику обработки, использующую код Transact-SQL. Поэтому набор функциональных возможностей, поддерживаемых триггерами, шире множества функциональных возможностей ограничений. Однако целостность данных всегда следует контролировать на как можно более низком уровне, обеспечиваемом использованием ограничений, при условии, что они удовлетворяют требованиям решения [3].

Триггеры наиболее полезны в случаях, когда возможности, предлагаемые ограничениями, не отвечают функциональным требованиям приложения. Например:

- Ограничение FOREIGN KEY позволяет проверить значение столбца только при точном совпадении со значением другого столбца, если только не задано предложение REFERENCES, определяющее каскадное ссылочное действие.
- Ограничения могут извещать об ошибках только с помощью стандартных системных сообщений об ошибках. Если для приложения требуется более сложная обработка ошибок, следует использовать триггер.

Триггеры могут инициировать каскадное распространение изменений по связанным таблицам базы данных, однако эти изменения можно выполнить более эффективно, воспользовавшись каскадными ограничениями ссылочной целостности:

- Триггеры позволяют выполнять откат изменений, нарушающих ссылочную целостность, или запрещать их, тем самым отменяя попытку изменения данных. Такой триггер может срабатывать при изменении внешнего ключа, когда новое значение не соответствует его первичному ключу. Однако для этих целей обычно используются ограничения FOREIGN KEY.

- Если в таблице триггера определены ограничения, они проверяются после выполнения триггера INSTEAD OF, но до выполнения триггера AFTER. Если ограничения нарушены, в случае триггера INSTEAD OF производится откат действий, а триггер AFTER не выполняется.

Триггер INSERT

Триггер INSERT выполняется каждый раз, когда инструкция INSERT вставляет данные в таблицу или представление, для которых он определен.

Действие триггера INSERT

При срабатывании триггера INSERT новые строки вставляются в таблицу триггера и в таблицу **inserted** (таблицу вставки). Таблица **inserted** — это логическая таблица, в которой хранится копия вставленных строк. Таблица **inserted** содержит записанные в журнал операции вставки, порожденные инструкцией INSERT. Таблица **inserted** позволяет обращаться к записанным данным из соответствующей инструкции INSERT. Триггер может проанализировать таблицу вставки и определить, следует ли выполнять действия триггера и как это делать. Строки таблицы **inserted** всегда представляют собой дубликаты одной или нескольких строк таблицы триггера.

Все операции изменения данных (инструкции INSERT, UPDATE и DELETE) записываются в журнал, однако содержимое журнала транзакций недоступно для чтения. Таблица **inserted** предоставляет доступ к записанным изменениям, вызванным инструкцией INSERT. Затем можно сравнить изменения со вставленными данными, чтобы проверить их или выполнить дополнительные действия. Кроме того, можно ссылаться на вставленные данные, не сохраняя их в переменных.

Создание триггера INSERT

В следующем примере кода создается триггер INSERT с именем **insrtWorkOrder** для таблицы **Production.WorkOrder** в базе данных **AdventureWorks**. Обратите внимание на использование таблицы **inserted** для работы со значениями, вызвавшими запуск триггера.

```
CREATE TRIGGER [insrtWorkOrder] ON [Production].[WorkOrder]
AFTER INSERT AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO [Production].[TransactionHistory](
        [ProductID],[ReferenceOrderID],[TransactionType],
        [TransactionDate],[Quantity],[ActualCost])
    SELECT inserted.[ProductID],inserted.[WorkOrderID],
        'W',GETDATE(),inserted.[OrderQty],0
    FROM inserted;
END;
```

Триггер DELETE

Триггер DELETE — это хранимая процедура особого вида, которая выполняется каждый раз, когда инструкция DELETE удаляет данные из таблицы или представления, для которых определен этот триггер (Рис. 6.3).

Действие триггера DELETE

При срабатывании триггера DELETE строки, удаленные из таблицы, помещаются в специальную таблицу **deleted** (таблицу удаления). Таблица **deleted** — это логическая таблица, в которой хранится копия удаленных строк. Таблица **deleted** позволяет обращаться к записанным данным из соответствующей инструкции DELETE.

При использовании триггера DELETE необходимо иметь в виду следующее.

- Если строка добавлена к таблице **deleted**, это означает, что ее больше нет в базе данных; следовательно, таблица **deleted** и таблицы базы данных не имеют общих строк.
- Для создания таблицы **deleted** выделяется область памяти. Таблица **deleted** всегда хранится в кэше.
- Триггер, определенный для действия DELETE, не выполняется для инструкции TRUNCATE TABLE, поскольку эта инструкция не записывается в журнал.

Создание триггера DELETE

Триггер, создаваемый в этом примере, обновляет столбец **Discontinued** в таблице **Products** при каждом удалении какой-либо категории (т. е. при удалении записи из таблицы **Categories**). Все продукты, затрагиваемые таким удалением, помечаются флагом 1 в знак того, что они изымаются из продажи.

```
CREATE TRIGGER [delCategory] ON [Categories]
AFTER DELETE AS
BEGIN
    UPDATE P SET [Discontinued] = 1
    FROM [Products] P INNER JOIN deleted as d
    ON P.[CategoryID] = d.[CategoryID]
END;
```

Триггер UPDATE

Триггер UPDATE выполняется каждый раз, когда инструкция UPDATE изменяет данные в таблице или представлении, для которых он определен.

Действие триггера UPDATE

Действие триггера UPDATE можно представить себе состоящим из двух шагов:

1. Шаг DELETE — записывается образ данных до обновления.
2. Шаг INSERT — записывается образ данных после обновления.

Когда в таблице, для которой определен триггер, выполняется инструкция UPDATE, первоначальные строки (образ «до») перемещаются в таблицу **deleted**, а обновленные строки (образ «после») вставляются в таблицу **inserted.**

Триггер может проверить таблицы **deleted** и **inserted** (равно как и таблицу **updated**), чтобы определить, не было ли обновлено несколько строк и как следует выполнять действия триггера.

Можно определить триггер для отслеживания обновлений данных в определенном столбце, воспользовавшись инструкцией IF UPDATE. Это позволяет легко изолировать операции, выполняемые над конкретным столбцом. Обнаружив, что данный столбец обновлен, триггер может выполнить соответствующее действие, например, выдать сообщение об ошибке, информирующее о невозможности обновления столбца, или обработать цепочку инструкций на основе нового значения столбца.

Создание триггера UPDATE

В примере создается триггер UPDATE с именем updtProductReview для таблицы Production.ProductReview в базе данных AdventureWorks.

```
CREATE TRIGGER [updtProductReview] ON
     [Production].[ProductReview]
AFTER UPDATE NOT FOR REPLICATION AS
BEGIN
SET NOCOUNT ON;
```

Триггер INSTEAD OF

Триггер INSTEAD OF выполняется вместо обычного запускающего его действия. Триггеры INSTEAD OF можно также определять для представлений с одной или несколькими базовыми таблицами, расширяя типы обновлений, поддерживаемых в представлении.

Действие триггера INSTEAD OF

Такой триггер выполняется вместо исходного действия, запускающего его. Использование триггеров INSTEAD OF расширяет диапазон типов обновлений, применимых к представлению. В каждой таблице или представлении можно использовать не более одного триггера INSTEAD OF для каждого соответствующего действия (INSERT, UPDATE или DELETE).

Триггер INSTEAD OF можно определять как для таблиц, так и для представлений. Нельзя создать триггер INSTEAD OF для представления, в котором определен раздел WITH CHECK OPTION.

Триггеры INSTEAD OF имеют следующие преимущества.

- В представлениях с несколькими базовыми таблицами обеспечивается поддержка операций вставки, обновления и удаления, ссылающихся на данные в этих таблицах.
- С помощью этих триггеров можно реализовать программным способом алгоритм, отклоняющий некоторые части выполняемого пакета и позволяющий продолжить обработку остальных его частей.
- Такой триггер позволяет определить альтернативное действие в базе данных при соблюдении заданных условий.

Создание триггера INSTEAD OF

В следующем примере кода создается триггер INSTEAD OF с именем **delEmployee** для таблицы **HumanResources.Employee** в базе данных **AdventureWorks**.

```
CREATE TRIGGER [delEmployee] ON [HumanResources].[Employee]
INSTEAD OF DELETE NOT FOR REPLICATION AS
BEGIN

    SET NOCOUNT ON;
    DECLARE @DeleteCount int;
    SELECT @DeleteCount = COUNT(*) FROM deleted;
    IF @DeleteCount > 0
    BEGIN
        RAISERROR
            (N'Employees cannot be deleted. They can only be
                marked as not current.', -- Message
            10, -- Severity.
            1); -- State.
        -- Roll back any active or uncommittable transactions
        IF @@TRANCOUNT > 0
        BEGIN
            ROLLBACK TRANSACTION;
        END
    END;
END;
```

Вложенные триггеры

В любой триггер можно включить инструкцию UPDATE, INSERT или DELETE, применяемую к другой таблице. Триггеры называются вложенными, если один триггер выполняет действие, запускающее другой триггер (см. рисунок 16).

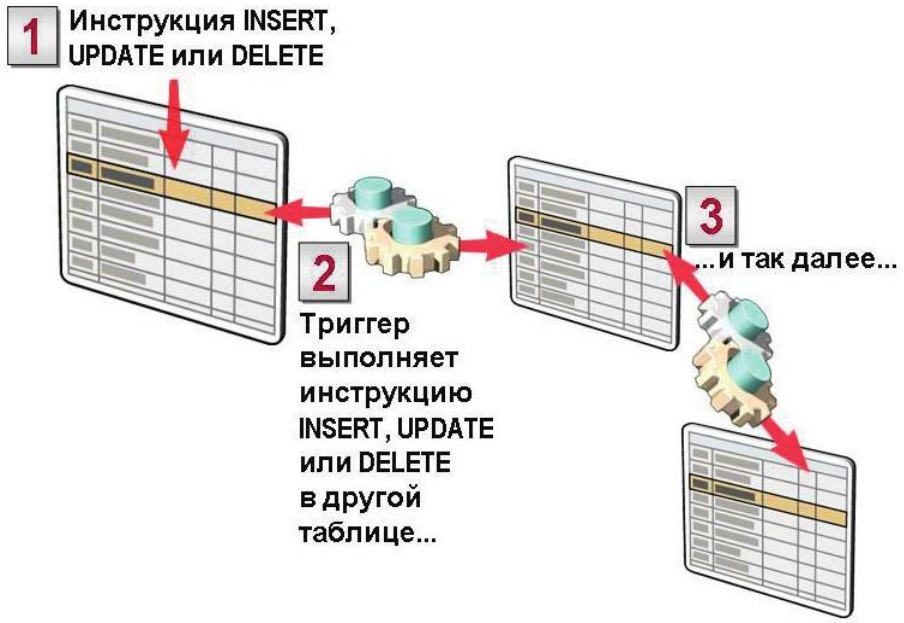

Рисунок 16 Вложенные триггеры

Действие вложенных триггеров

Вложение триггеров контролируется с помощью соответствующего параметра конфигурации сервера. Вложение включается при установке и устанавливается на уровне сервера, однако его можно отключить и затем вновь включить, воспользовавшись системной хранимой процедурой **sp_configure** [6].

Максимальная глубина вложенности триггеров составляет 32 уровня. Если один из триггеров в цепочке вложенных триггеров начинает бесконечный цикл, это приводит к нарушению ограничения уровней вложенности. В таком случае триггер прекращает обработку и выполняет откат транзакции. Вложенные триггеры можно использовать для выполнения таких функций, как резервное копирование строк, обрабатываемых предыдущим триггером.

При использовании вложенных триггеров необходимо иметь в виду следующее:

- По умолчанию параметр конфигурации, разрешающий вложенные триггеры, установлен.
- Вложенный триггер не срабатывает дважды в одной транзакции; триггер не вызывает сам себя в ответ на второе обновление той же таблицы в процедуре триггера. Однако если изменение таблицы триггером вызывает запуск другого триггера, который изменяет ту же таблицу, произойдет рекурсивный запуск первоначального триггера. Чтобы избежать косвенной рекурсии подобного рода, отключите поддержку вложенных триггеров.
- Поскольку триггер представляет собой транзакцию, сбой на любом уровне вложения триггеров приведет к отмене всей транзакции, с последующим откатом всех изменений данных. Поэтому при тестировании триггеров

рекомендуется использовать инструкции PRINT, чтобы можно было определить место сбоя.

Уровни вложенности

Уровень вложенности увеличивается при каждом срабатывании очередного вложенного триггера. SQL Server поддерживает до 32 уровней вложенности, однако число уровней можно дополнительно ограничить, чтобы не допустить превышения максимального предела вложенности. Для просмотра текущих уровней вложенности используется функция **@@NESTLEVEL.**

Отключение вложенных триггеров

Вложение триггеров является весьма полезным средством, используемым для обеспечения целостности данных во всей базе данных. Однако бывают случаи, когда необходимо отключить поддержку вложения. Если вложение отключено, триггер не сможет выполнять каскадирование (действие, запускающее другой триггер, который запускает еще один триггер и т. д.).

Необходимость отключения вложенных триггеров может быть вызвана следующими причинами:

- Вложенные триггеры требуют тщательной и сложной разработки. Каскадирование изменений может привести к нежелательным изменениям в данных.
- Изменение данных в любом месте цепочки вложенных триггеров приводит к запуску последовательности триггеров. Хотя это обеспечивает надежную защиту данных, могут возникнуть проблемы, если таблицы требуется обновлять в определенном порядке.

Для отключения вложенных триггеров используется следующая инструкция.

```
sp_configure 'nested triggers', 0
```

Рекурсивные триггеры

Рекурсивный триггер — это триггер, выполняющий действие, которое приводит (непосредственно или косвенно) к повторному срабатыванию того же триггера. Любой триггер может содержать инструкцию UPDATE, INSERT или DELETE, воздействующую на ту же или на другую таблицу. Если рекурсивные триггеры включены, то триггер, изменяющий данные в таблице, может снова активизировать сам себя рекурсивным путем.

Существует два вида рекурсии.

- **Прямая рекурсия**. Прямая рекурсия означает, что триггер при срабатывании выполняет действие над той же таблицей, которое вызывает повторное срабатывание того же триггера. Например, предположим, что в приложении обновляется таблица **T1**, что вызывает срабатывание

триггера **Trig1**. **Trig1** снова обновляет таблицу **T1**, что вызывает повторное срабатывание триггера **Trig1.**

- **Косвенная рекурсия**. Косвенная рекурсия имеет место, когда триггер при срабатывании выполняет действие, которое вызывает срабатывание другого триггера (в той же или другой таблице) и затем — обновление исходной таблицы. Это приводит к повторному срабатыванию первого триггера. Например, предположим, что приложение обновляет таблицу **T2**, что вызывает срабатывание триггера **Trig2**. **Trig2** обновляет таблицу **T3**, вызывая срабатывание триггера **Trig3**. **Trig3**, в свою очередь, обновляет таблицу **T2**, что вызывает повторное срабатывание триггера **Trig2**.

Управление рекурсивными вызовами

При создании базы данных поддержка рекурсивных триггеров по умолчанию отключается, но ее можно включить с помощью инструкции ALTER DATABASE. Триггер не сможет рекурсивно вызывать сам себя, если не установлен параметр базы данных RECURSIVE_TRIGGERS. Для включения рекурсивных триггеров используется следующая инструкция.

```
ALTER DATABASE AdventureWorks SET RECURSIVE_TRIGGERS ON
```

Если вложенные триггеры отключены, отключаются и рекурсивные триггеры, независимо от настройки параметра рекурсивных триггеров в базе данных.

Если вложенные триггеры включены, косвенная рекурсия будет возможна, хотя поддержка рекурсивных триггеров отключается. Параметр рекурсивных триггеров служит защитой только от прямой рекурсии.

В таблицах **inserted** и **deleted** для данного триггера будут содержаться только строки, соответствующие последней инструкции UPDATE, INSERT или DELETE, инициировавшей триггер.

Глубина рекурсии триггеров может составлять до 32 уровней. Если рекурсия триггеров вызывает бесконечный цикл, то после выхода за 32-й уровень триггер автоматически прекращает обработку, и выполняется откат транзакции.

Рекурсивные триггеры представляют собой мощный инструмент, позволяющий разрешать сложные отношения, например отношения, ссылающиеся сами на себя (так называемые транзитивные замыкания).

Прежде чем использовать рекурсивные триггеры, необходимо усвоить следующее.

- Рекурсивные триггеры отличаются высокой сложностью, их следует очень аккуратно разрабатывать и тщательно тестировать.

- Рекурсивные триггеры требуют программирования логики контролируемого цикла (проверки прекращения). В противном случае будет превышено 32-уровневое ограничение вложенности.
- Изменение данных в любой точке может инициировать последовательность триггеров. Хотя это позволяет обрабатывать сложные отношения, могут возникнуть проблемы, если таблицы требуется обновлять в определенном порядке.
- Аналогичные функции можно реализовать без помощи рекурсивных триггеров, однако это потребует существенного изменения структуры триггера. При разработке рекурсивных триггеров в каждый из них следует включать проверку некоторого условия, чтобы прекращать рекурсивную обработку при нарушении этого условия. Нерекурсивный триггер должен включать полные программные структуры циклов и соответствующие проверки.

Литература

1. Электронная документация по SQL Server 2014 - MSDN - Microsoft [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/ms130214(v=sql.120).aspx, свободный. — Яз. русский. (дата обращения: 01.11.2016).
2. Библиотека Microsoft SQL Server - MSDN [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/bb545450.aspx, свободный. — Яз. русский. (дата обращения: 01.11.2016).
3. Александр Бондарь. Microsoft SQL Server 2014. — СПб.: БХВ-Петербург, 2015. — 592 с.
4. Ицик Бен-Ган. Microsoft SQL Server 2012. Основы T-SQL. — М.: Эксмо, 2015. — 400 с.
5. #20461C Querying Microsoft® SQL Server®. Официальный курс Microsoft.
6. #20464C Developing Microsoft® SQL Server® Databases. Официальный курс Microsoft.

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

КАФЕДРА ПРОГРАММНЫХ СИСТЕМ

http://ps.ifmo.ru

Кафедра Программных систем входит в состав факультета Инфокоммуникационных технологий. На кафедре обеспечена возможность обучения студентов по современным образовательным стандартам в области программного обеспечения ИКТ. Для этого на кафедре работает высококвалифицированный преподавательский состав, имеется современная техническая база и специализированные лаборатории, оснащенные необходимым оборудованием и программным обеспечением; качественная методическая поддержка образовательных программ. Наши студенты принимают активное участие в российских и зарубежных исследованиях и конференциях, имеют возможность публикации результатов своих исследований в ведущих российских и зарубежных реферируемых изданиях. Существенным преимуществом является возможность выпускников продолжить научную деятельность в аспирантуре Университета ИТМО и в других передовых российских и зарубежных научных Центрах.

Кафедра обеспечивает подготовку бакалавров и магистров по образовательным программам:

- Интеллектуальные инфокоммуникационные системы - бакалавры;
- Программное обеспечение в инфокоммуникациях – магистры.

На кафедре реализуется международная образовательная программа DD Master Program, в рамках которой выпускники имеют возможность получить два диплома: Диплом Университета ИТМО с присвоением магистерской степени по направлению «Программное обеспечение в инфокоммуникациях» и Международный диплом - Master of Science in Technology Lappeenranta University of Technology in the field of Computer Science majoring in Software Engineering.

Выпускники кафедры обладают компетенциями:

- проектирования и создания рациональных структур ИКС;
- разработки алгоритмов решения задач и их программной реализации на основе современных платформ;

- моделирования процессов функционирования сложных систем;
  •обеспечения безопасности работы ИКС;
- реализации сетевых услуг и сервисов в ИКС;
- проектирования и разработки баз данных;
- разработки клиент-серверных приложений ИКС;
- проектирования, создания и поддержки Web-приложений;
- управления проектами перспективных направлений развития ИКС.

Трудоустройство выпускников кафедры возможно на предприятиях: ООО «Digital Design»; ООО «Аркадия»; ОАО «Ростелеком»; ООО «ЭПАМ Системз»; ООО «Т-Системс СиАйЭс» и многие другие.

Мы готовим квалифицированных инженеров в области инфокоммуникационных технологий с новыми знаниями, образом мышления и способностями быстрой адаптации к современным условиям труда.

Осетрова Ирина Станиславовна

Разработка баз данных в MS SQL Server 2014

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж 100 экз.
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49